9-21-1992

Introduction to Software Engineering, An

Harlan D. Mills

J. R. Newman

Charles B. Engle, Jr.

Luwana Clever

An Introduction to Software Engineering

Revised: September 21, 1992

By Harlan D. Mills, J. R. Newman,
Charles B. Engle, Jr., & Luwana Clever

Florida Tech

Table of Contents

Chapter 1 - Mathematical Foundations

Chapter 2 - Sequential Ada I

Chapter 3 - Program Behavior

Chapter 4 - Software Analysis

Chapter 5 - Sequential Ada II

Chapter 6 - Program Verification

Table of Contents

Chapter 7 - Software Design and Certification

Chapter 8 - Sequential Ada III

Table of Contents

Chapter 9 - Linked Lists

Chapter 10 - Ada Package Design with Box Structures

Syntax Definitions

Syntax Charts

Figures

Tables

Table of Contents

Chapter 11 - Sequential Ada IV

Chapter 12 - Software Design for Interactive Use

Syntax Definitions

Syntax Charts

Figures

Tables

Chapter 1

Mathematical Foundations

Computing and computer applications have become an integral part of modern society. The impact of computing is so widespread that it is taken for granted or even overlooked as both a positive and negative influence on everyday life. Although it is easy to see the impact on our lives, it is much more difficult to understand how this technology can be controlled and productively used. Even the simple application of writing a letter using a word processor can become a real challenge when the concept is expanded to include the production of a daily newspaper or a monthly magazine. To use a computer to balance a checkbook is relatively simple but when compared to balancing the profit and loss statement of a international corporation the computational complexity is significantly increased. Each problem that a computer is used to solve is made up of a complex set of smaller problems, some of which by themselves may be relatively straight forward. One of the major challenges in the world of computing is to recognize how the large problem can be broken down into the right set of small problems. Then no sooner is the correct set of components fit together to make a correct solution, than the person using the system inevitably uncovers many new useful functions that they want to be added to the system. Thus for every new computer application that becomes operational, many modifications or even totally new systems are proposed. As our society grows in understanding this technology, the demand continues to escalate for larger, more complex systems to enhance all aspects of our lives. The potential demand for computing appears limitless and every day our lives seem to become more tied to this technological marvel.

It is easy to see the impact computing has on our lives, but what is it that makes these sophisticated devices function? How can one computer perform so many useful, yet different actions? What knowledge is necessary to be able to control the actions of computers? For some situations it can be as easy as providing the computer with a list of names and a few simple instructions on what to do with them. On the other hand, to develop and correctly implement a large aggregation of the financial data that a large organization requires to function every day could require a team of several people working for many months. What happens when one of those involved in creating the computer solution makes a mistake? How will that error affect the entire system? What does it mean to have a computer failure? It is easy to point out a wide range of problems for which computers have been blamed. Is it the computer that malfunctions or the person that prescribed the actions the computer is to perform? Although at first it may appear to be fun to make the computer perform so many complex operations in such a short period of time, the potential exists for great harm and costly problems if the correct solution is not specified, designed, and implemented.

Those people who choose to make a profession in developing correct computer-based solutions will require an in-depth study of many aspects of computing. The foundations will come from applied mathematics, engineering, and the sciences. Understanding how computers work will require an understanding of the basic components of their operation in both an abstract and practical way. This chapter begins with some practical introduction to the way computers are used and the effort it requires to make them perform the desired actions in an exact and correct manner . Then the formal foundations of computing are introduced in an abstract representation of the actions that are to be performed. The process of understanding the logical actions computers perform begins with some basic but non-traditional mathematics. Not the usual algebra or geometry, but the mathematics of logic, relationships, number systems, and specifying operations. It is this *discrete mathematics* of computing elements that is needed to describe and predicate the behavior of each computer action.

The introduction to this form of mathematics covers topics that have an intuitive representation. For example, the words used to write this text, were all constructed on a computer using a word processor. To the computer, each character, including the blank space, is just a part of a formal *string* that has both specific physical and mathematical properties. These strings have been treated mathematically only recently, with the advent of computers. There are several classes of strings that are very common and very important, but the one that is of most interest in computing is *character strings*, in which the components are characters. Ordinary text, such as this very sentence, is a character string. Natural numbers given in place notation are character strings with only digits as components. Roman numerals are character strings with strong restrictions on the order of the roman numeral components in a string. But of special interest are the instructions of what actions a computer is to perform, which are often called a program, as they are represented by a character string.

Many character strings have a higher level structure of strings, as well. Ordinary text in a given language will also be a string of words, numbers, space, and punctuation characters which form a proper language string. Such a string will be a string of strings, such as words made up of several letters, numbers made up of several digits, space and punctuation characters that are regarded as strings of a single character. At higher levels, proper language text can be organized into lines, lines into pages, pages into books, and books into libraries.

Another part of the computing formalism is a grouping of items into *sets* . Sets as formal objects of mathematical study are also a recent development, beginning before computers, but still less than a hundred years old. Sets represent an abstraction of strings, listing their objects, but ignoring the order and duplication of such objects. While computer programs are character strings, the values that are provided as inputs are sets of character strings, each of which constitutes a single input unit. The outputs in response to each input make up another set of character strings. Such sets of input, output pairs that describe all possible program behavior are called *mathematical functions*. Each program has a *program function* that describes its behavior over all possible inputs. By generating this function as a mathematical object, where the behavior is a listing of the set of input, output pairs , it is possible to examine every action the program will perform. Program functions need not be numerical, but may describe any possible behavior, dealing with word processing or any other behavior that has been programmed. Although functions have been studied mathematically before the use of sets, their study has become more rigorous, yet simplified, with sets as their foundations. And these new foundations are especially relevant for computer programs.

Although the basic process of making the computer perform the right functions is straightforward, it quickly becomes complicated and requires a great deal of knowledge, skill, and discipline. The mastery of the process of developing full scale computer applications will require a different set of knowledge plus patience, diligence and hard work. As with other fields that result in the construction of useful devices, the term engineering can be applied to developing computer solutions to real world problems. The term Computer Engineer has developed a more specialized meaning as a field directly related to Electrical Engineering. The general term of Software Engineer is evolving as a description for those professionals that develop computer-based systems for a wide variety of applications.

1.1 What is Software Engineering?

1.1.1 Engineering Correct Software

The impact of the era of computing has reached almost every facet of our society. Some useful insight and a historical perspective can be gained through a quick look back into the development of the computing that is now taken for granted. The sheer speed of the numerical processing capability was enough to make the first primitive digital electronic computers very valuable. During the 1950's, one of the early designers of these basic computers was reported to have said that the commercial needs could be satisfied by building less than 10 operational computers. During the early years of computing, most applications were numeric and the developments were driven by the new electronic technologies. Improvements were realized by increasing the reliability, reducing the physical size, increasing the speed, and expanding the capacity, while lowering the cost per computation unit. These initial improvements focused primarily on the electromechanical components, where the financial investment was considered highest. The initial cost reduction based on improvement in these technologies established a trend where the price per computational unit dropped by half about every two years.

Hardware is the general term that describes the combination of the electrical and mechanical portion of a computer. If a particular computer is made up of only hardware, it is restricted to perform only the functions that are included in the physical logic and it will perform them in fixed sequence. The real flexibility of computers comes when the hardware is designed to determine which instructions and what sequences are to be performed, based on values that are found in specific storage locations, appropriately called memory. Thus, in most modern computers the basic functions (*i.e.*, addition, subtraction, and comparison) are built into hardware logic. The sequence of which action is to be executed next and the specific values to be used in each action (*i.e.*, the numbers to be added together) is determined by placing the appropriate values in the correct sequence in the form of instructions that are stored in the proper memory locations. The set of basic hardware functions that a specific computer can recognize and execute is called the hardware instruction set. These instructions can be grouped together to perform logical actions such as combining repetitive additions to make multiplication or repetitive subtraction to result in division. The ultimate results that the computer produces are thus determined by having the hardware execute a specific sequence of logical instructions that have been placed into specific memory locations. *Software* is the general term used to describe a specific set or sequence of logical instructions that can be placed in memory to control the hardware execution and thus the results the computer produces.

Software was initially developed to provide easier access to the hardware and more flexibility in the way it could be used. In addition to the early numerical problem solving, software was developed for common problems in data comparisons or manipulation, as well as combinations of sorting and searching. Software was also produced that could aid in developing other software. Another key factor in the development of modern software was the concept of storing the instructions used to control the computer in the same area that the data, that would be used in the computational or manipulation process, was stored. In this form the hardware does not distinguish between instruction or data, it only executes the next piece of data, as an instruction, in the sequential order in which they are stored. It is the responsibility of the software developer to insure that the proper instruction and the correct data is placed where it should be. This concept is known as the *stored program concept* and is credited to the computer pioneer John Von Neumann. By storing data and instructions together in logical groups called programs, a single computer could be made to perform many different tasks in almost any order.

Such programs could be created separately, then read into the computer's memory to be performed one after another.

The search for easy ways to specify what software instructions were to be combined to make specific programs lead to the use of software translators. These translators were just programs that were able to convert instructions given in a restricted form that was easy for humans to understand, into the correct instruction codes that the hardware could perform or execute. The term programmer has come to be used for those who specify programs using some form of a software translator or programming language interpreter. A programming language is a restricted set of rules for specifying the logical actions the computer will perform. The specific programming language syntax can provide the initial mechanism to check the correctness of instructions to be executed. The history of programming language translators represents a broad and diverse segment of the development of computers. Many languages have been developed for specialized, as well as generalized, use. The concept of a high order programming language (HOL) is to restrict the instruction specification by the programmer to an easy to use, useful, but less ambiguous, and more formal, format. Except for the inherent ambiguity, perhaps the easiest HOL to use would be a natural language such as English, but this is not practical for technical reasons. The restricted format of the common HOL's does provide some discipline to the way problems can be specified for translation into hardware instructions, but each has its own unique inherent problems.

Evolving much as a craft, or art, the skill of programming was applied to scientific, commercial, and governmental problems of all magnitudes. As programming languages and techniques developed, the complexity and size of individual programs grew astronomically. The range of potential problems for computer solutions expanded faster than our ability to generate correct solutions. The demand for skilled programmers was so high, and the breadth of applications so wide, that often unlimited experimentation was allowed in spite of catastrophic failures. As the percentage of time and cost associated with developing software became larger than the investment in hardware, the concern grew for control over the software development process. The role of a system analyst evolved to provide assistance in the initial stages of programming. Interfacing with the eventual user of a computer system, the analyst was involved in determining what the problem really consisted of and the feasibility of a computer solution. If it was determined that the problem could and should be solved using a computer, a blueprint or design was created. This design was to guide the programmer in developing each program unit. Again without much formal training the art of system analysis evolved from the good and bad experiences of those involved.

During this period of rapid development of computing technology, our society began to adopt a complete dependence on computing. Whereas a manual accounting system was expected to take weeks to determine the end-of-month balance, a computerized accounting system was expected to provide instantaneous results. There was total dependency on the availability of the computing system. Complicated computer-based medical equipment was expected to perform without flaw at all times in all conditions. The sheer size of the data that was expected to be processed eliminated any other solution. In spite of new and more sophisticated design and implementation techniques, software continued to contain a significant number of errors. At first the blame for these errors was placed on the computer itself, but no one ever took these excuses seriously, as it was always the human that created the flaw in the software. An additional and very alarming issue is the estimation of the number of software errors that still exist in many large scale operational software systems, but have yet to be uncovered.

The advent of the inexpensive personal computers and the associated easy-to-use software has opened the door to an even broader range of applications. Many who use these systems are not trained in the basic skills of logic and programming. It is true that for many simple applications, the available software tools are generally reliable and therefore many straightforward problems can be easily solved. It is precisely because these tools are so easy to use that the owners fall into the complexity trap. Here the ability of the untrained developers is quickly exceeded as they try to manage the problem complexity while learning the necessary programming skills for what initially seemed an easy problem.

The collective problems that have been summarized here constitute a crisis for those involved in the profession of software development as well as society in general. As the scope and complexity of development projects expand, and the ability of those involved fails to keep pace with the problems that are created, the significance of the software crisis continues to grow.

1.1.2 Dealing with Large Complex Software Solutions

To be aware of a crisis is not a sufficient condition for the solution. The awareness of this crisis is, however, an initial condition that can lead to identifying the causes and providing a beginning for developing potential solutions. The first cause to be examined is the size and complexity issue. If all the problems that were to be solved could be limited in scope and effort to the effort one individual could produce in a short period of time, say a few months, and assuming that the single individual understood or could master all the associated nuances of the problems involved, then a set of basic design skills and programming techniques could be provided to train programmers in solving these problems. This is in fact the way much of formal Computer Science education has taken place. The basic skills in problem solving, programming, and application techniques form the foundation for what is called *programming-in-the-small.* Such an education could include studying specific topics such as: programming languages, compiler construction, algorithm analysis, and data structures. This knowledge is combined with applications in Computer Graphics, Robotics, Data Base Systems, and Operating Systems to provide a basis for developing real world applications. This incremental approach provides a reasonable foundation depending on the quality of the instruction and the depth of knowledge that is obtained.

The major weakness of this approach is that it fails to recognize the problems that are introduced when the size and complexity are so large that it will require a team of experts working over an extended period of time to complete the full development. This team will spend months determining the scope and feasibility of a potential computer-based solution. If the problem is sufficiently ill defined, prototypes or simplified sample solutions will be developed just to be sure that the ultimate solution will fill the need of those who will use the system. After this, a major specification, design and implementation effort will be required. The development of systems of this magnitude require special management skills and a process that controls each phase or cycle. The evolution of a discipline for managing this complexity has led to the use of the term *programming-in-the-large.* In this environment the total lifecycle of a software system is managed and controlled based on specific systems development techniques.

The second cause for the software crisis is represented by the difficult task of maintaining existing software systems. As a software system is developed, conscious decisions are being made as to what should be included in the ultimate solution. For various reasons, certain functions may not be included in the initial development. As the system is finalized, external conditions may change, new requirements may be uncovered, and improvements may be recommended. Once the system becomes operational, a new phase of development begins that is traditionally called maintenance. This involves expanding and refining the software system utilizing many of the same techniques required for the initial development. This refinement process leads to the concept of developing software for reuse as part of other systems. Until the techniques that have been developed for maintenance and reuse are integrated into all aspects of software development, the software crisis will continue to grow.

The recognition of the need to manage the full lifecycle of a large-scale software development effort has led to an expanded study of each component of the life cycle. Although there are many models for the life cycle of software development, they all share the same goal of decomposing the process into understandable and manageable stages. The initial stage is the problem definition and gathering of the requirement for a solution. As easy as it may sound, the problems of determining exactly what the problem is and what it will take to solve it can consume a major portion of the full development effort. This is particularly true if the desired solution involves many units within an organization and many people with differing view points. As the problem is better understood, the user requirement can be converted into specifications that a software designer specialist can use to begin to develop a computerized solution.

Computer solutions to these large, complex, ill defined problems go through many iterations of refinement. One way to examine the initial approximations to a solution is to build a prototype that represents the general functions without requiring the full development of all components. After the prototype is evaluated by the user and developer, a second and then third version can be developed with more functionality and more completed components. As each of these components are systematically developed and tested they become part of the complete operational system. The skill and knowledge required to manage this full development process represents a significant challenge to the software development professionals.

Each organization has specific operational procedures that need to be considered in their software development process. There are many ways in which the development team can function to accomplish the same objectives. All large-scale software development efforts do not have the same characteristics, nor require the same management effort. Each organization must manage and mature their own style of software development. Only as the aggregate maturity of all organizations are substantially improved will we see the demise of the software crisis.

Perhaps the most fundamental and serious cause of the software crisis is the inability of all the individuals involved to develop each component of the software system error-free. It is widely recognized that errors can be introduced into all phases of the software development process. Incorrect understanding of the problem leads to erroneous specifications. Faulty specifications lead to inaccurate and inappropriate designs. Without correct designs, the resulting programs can not implement the correct solution. Beyond this proliferating sequence of errors, the potential for traditional problems of improper use of programming constructs, language syntax, and data structures is ever present. Recognizing that all aspects of software development are inexorably tied to human frailty, it is essential that a systematic and sound method be developed to test each component at each step in the development process.

1.1.3 Scope and Status of Software Engineering

The challenge presented by the current state of the art of developing correct software solutions is to discover simple yet powerful principles that can be applied in a disciplined and consistent manner throughout the entire process. The general term *software engineering* can be applied to this challenge. The definition includes a strong foundation in the formal methods for developing small error free programs and the associated testing techniques that are required to insure correctness. Software engineering involves the full life cycle of large scale systems development as well as the process that a team or organization utilizes. The software engineering concept embodies the maintenance of systems and reuse of components. A key component of this or any field of engineering is the recognition of the need for a controlling discipline. Such a discipline moves from a set of skills for a craft, to the formal foundation of a science and evolves into applied engineering principles.

Software engineering in the form described here is a new and evolving field. Many academic and professional disciplines have significantly influenced the current status of this field of engineering. Mathematics forms the foundation and provides many areas for application. More than just a formal way to think about computing concepts, mathematics provides the essential tools to decompose, analyze and improve key operational aspects of a computer's hardware and software. Depending on the specific academic definition and vantage point, Computer Science and Computer Engineering are integral parts of Software Engineering. From the basic electronic components to the basic software instructions, to the most complex algorithms and applications, these fields share many of the same concepts. The field of Information Systems shares the concern for organizational issues and the way they impact software development. The interrelationship of these three and Software Engineering will likely continue, with each emphasizing their own strength.

As the field of software engineering continues to develop an identity, it will be strongly influenced by current and future events. As organizations mature with respect to software engineering, the feedback will refine current techniques. As the academic world refines the formal education process, broad acceptance and certification will take place. Given the current bases and challenges, the outlook is both optimistic and exciting.

1.1.4 Exercises

1. What are some of the key dependencies our society has developed for computers?
2. What are some of the basic applications for personal computers? How are they different or similar to the basic applications of large scale computing systems?
3. Describe a critical application of computers you have encountered where a software error would cause catastrophic problems. Describe another application where minor errors could be tolerated.
4. Describe some of the ways that large complex software systems are different from small-scale computer-based solutions?
5. What particular challenge in current software development do you find most interesting? Can you suggest some potential solutions?
6. Banks are clearly major users of computers. Which has been most affected, bank customers or bank workers or both? Illustrate your conclusions with examples.
7. Hotel chains are clearly major users of computers. Which has been most effected, hotel customers or hotel workers, or both? Illustrate your conclusions with examples.

1.2 Developing Correct Software

1.2.1 Starting with the Correct Fundamentals

As with any field of knowledge, the fundamental principles of software development must first be understood and mastered. Computer software is a new form of applied mathematics, just as place notation and long division was a new kind of mathematics in the western world a thousand years ago. Computers follow their software exactly, whether the software is right or wrong. They behave with mathematical precision, but with no common sense. It is up to the people who create or modify the software to get it right. So the new form of mathematics in computer operations is in the software. It is logical, but not necessarily numerical mathematics. Tracking a satellite needs numbers, but also a good deal of logic in what to do with the numbers. Maintaining bank accounts needs numbers, but even more logic in keeping track of the customers and banking conditions. Dealing with a telephone switching station needs more logic than numbers. Tracking court cases, both civil and criminal, needs mostly logic in legal procedures and word processing rather than numbers.

Software engineering is needed to deal with the complex problems of developing correct computer software for important applications. Engineering discipline helps address the issue of getting millions of instructions right for a given problem. This engineering discipline is based on mathematics, namely the mathematics of computers with discrete memory and operations that are directed by billions of instructions every day. It is the discrete mathematics of strings and sets that is needed, that lead to mathematical relations, functions, and predicates in describing software and computer behavior. The precise mathematical description of the logical action the computer is to perform provides the foundation for specifying correct software solutions.

If a correct software solution is decomposed, several components can be identified. The first component is the logic that controls sequential flow of instructions. This can be abstractly described as an algorithm. Algorithms have the properties of precise sequential steps, that have a beginning, an ending and are successful in accomplishing the desired results. Algorithms can be studied for complexity, power as well as completeness and correctness. It is useful to understand many basic algorithms such as sorting and searching, as they will appear often in many software applications.

A second component of software is the internal representation of the data to be used. This data can be arranged in useful structures such as arrays, lists and strings. Using basic predefined operations on these data structures, such as insert, compare and delete, can provide powerful tools for solving complex problems in software development. Care must be exercised in choosing the correct data structure and each operation must be fully understood before it can be correctly used.

Other common components of software systems are predefined functions such as input, output and transfer of control. Although these functions could be defined and controlled by each programmer, they are so common as to warrant a uniform solution that is provided by the computer manufacturer or software vendor. For example, using a keyboard as input, a READ statement in one programming language performs much the same basic operation as in any other language. A write to an external disk storage unit, however, will require a more complete description of where and how to store the data. Again a full understanding of the specific implementation details is required if errors in usage are to be avoided.

A final component is the specification of the operational behavior of the software. When the algorithms, data structures and system functions are combined into an operational program, specific results will be generated. These actual results must be compared to the expected results to determine their correctness. If the initial specifications were defined properly and the software contains no errors, then the results match the specifications.

The complete mastery of many details is required to develop significant software systems. It is not enough to be approximately correct. It is not enough to be correct most of the time. It is totally inappropriate to learn the basic syntax of a simple programming language and through trial and error piece together an approximate solution. A comparison could be made with someone studying to be a mechanical engineer. They would not be given the pieces to an automobile on the first day of the first class and be asked to begin building a better car through trial and error. Nor would a medical student be asked to begin experimenting on patients by diagnosing minor illness in their first class. A significant level of knowledge is required in any field before major problems can be addressed.

1.2.2 Principles for Developing Software Systems

Although many very similar large and small problems have been solved with software many times, each new solution may have just enough of a variation to make it better or more useful. Even though on the surface a problem may appear to be similar to an earlier solution, it may harbor significant difference to require a totally different approach. Knowledge of a variety of previous solutions may not be useful when solving a totally new problem. All of these factors lead to the conclusion that computer-based problem solving is not a simple application of standard techniques. Yet the software engineer must be trained to approach each problem in a systematic way and use all the fundamental principles that apply.

In the past, many attempts to define a basic set of software development principles has often led to obvious buzz words that apply to common sense. By the same token there is no excuse for ignoring the obvious or not using everyday common sense approaches. Human problem solving is a divide and conquer process. There is also a natural tendency to work with the known and well understood problems first and postpone solving the ill-defined problems until later. A good plan is easier to change than a partially constructed building. Beyond these truisms there are some basic approaches that experience has shown to be successful in developing software systems.

A generally accepted approach to developing software systems is to first analyze the problem to determine if a computer solution is appropriate. Then a set of specifications can be created to describe the results that are to be produced. Given a correct set of specifications an abstract or concrete design can be created as a blue print for eventual implementation. The design can be converted into specific program units and tested both separately and as a combined system. The process of validation can be used to insure that the results match the initial specification. This development cycle can be used in the development of both large and small systems, but the approach will need to be modified based on numerous external factors and constraints.

The value of any overall approach is only as good as the correctness of each step used. Correct specifications require both the designer and user to understand what the final system is expected to produce. In turn each program unit must produce exactly the expected results every time. Thus each component of the development process must be shown to produce the exact expected results. It is important to have a mechanism for describing the behavior of each component of the system in a way that can be easily understood, yet include sufficient formality to be convinced of the correctness. It is this approach to the exact specification of the behavior of each component produced in the development process that makes it possible to ensure correctness.

1.2.3 Mathematics as the Basis for Program Behavior

In searching for a mechanism to describe the behavior of each component of a software system it is important to consider both the representation and communication aspects. Natural languages, such as English, are rich for communication, but very imprecise as a representational format and therefore difficult to use to precisely describe the complete behavior of a specific item. Mathematics provides a concise representation and in a restricted sense a powerful communication tool, provided the notation is completely understood. The manipulation and logical consistencies of a mathematical representation provide a powerful way to describe complete behavior. In an abstract specification, a precise representational form is not as important as the communication between the users and the engineers. In this situation English· or a pseudo-English is a valuable tool. The weakness of this form is the inability to demonstrate the completeness or correctness. For representing and communicating about program components, mathematics provides a simple but powerful way to describe all the actions of any given unit.

To adequately describe the behavior of programs, the sequence of instructions that make up that program can be grouped into logical units. The resulting action that each of the logical units of software are to perform can be described as the set of outputs resulting from a set of specific stimuli. In this way each input into a unit of software code will produce a specific output. The rules for describing the complete behavior of each logical unit can be found in the fundamental mathematics of functions and relations. Thus it will be important to understand the basic mathematical concepts of functions and relations.

By building on this rigorous mathematical foundation the complete behavior of any software component can be completely determined. If only correct software components are combined into program units and the behavior of each program is verified, then correct systems will be constructed. If the specifications have been reviewed with the same rigor and precision, then the results can be compared and validated. If the correct results meet the specified needs then the goal of software engineering has been accomplished. To build the basis for accomplishing *this goal a considerable effort will be required and involve several courses. To begin with, some* fundamental mathematics will be covered as a foundation for the rest of the course.

1.2.4 Exercises

1. What additional general problem solving skills have you learned? Do any of them apply specifically to the logical approach computer require?

2. A useful experiment in decomposing a problem is to examine an everyday activity and describe it as a sequence of basic steps. For example, if you have a checking account at a bank, you must reconcile the balance at the end of the month. In this process you make sure which of your checks have been cleared by the bank and which are still outstanding. Describe this process in as a sequence of single steps or actions to perform.

3. In balancing a check book and reconciling it at the end of the month, specific inputs are required and outputs or results are generated. Review the process you described in Exercise 2 and indicate what input is needed at each step and what out puts are possible for all conditions.

4. Our lives are filled with routine but complex processes that we take for granted each day. The route you take to work or to school each day may be simple to you, but if you consider every possible turn on every possible street it quickly becomes unmanageable. If you were to map each route you would find that many routs cross back over themselves. Try to write a set of input and output conditions that would guarantee you would only traverse each street once to find you way to work or school.

5. The choice of words to represent a logical sequence of actions is important if the instructions are to be followed correctly. Try to write using precise wording the instructions you would give to someone you were trying to teach to play the game of checker or tick-tac-toe.

1.3 Mathematical Notations and the Concepts of Strings

1.3.1 Strings and their Components

One of the main purposes of mathematics is to provide a clear and consistent notation to represent ideas. In its simplest form the concept of adding one plus one to get the result of two, can be represented as 1+1=2. Mathematics is much more than just a shorthand for natural languages, in that it provides a precise representation of rules and operations that lead to powerful conclusions. The notation that is used, however, must be thoroughly understood and internalized before the mathematical power becomes a useful tool. Thus the first topic to be discussed in this computer or discrete mathematics is the notation and basic operations using that notation. Once familiar with the basic notation and operations, the concepts can be pieced together to make a complete system for describing the fundamental principles that are necessary to prove the correctness of software.

One of the basic and perhaps most intuitive components of discrete math is that of a *string*. Strings are sequences of objects in which order and duplication are maintained. The objects in a string are called its *components*. The components of a string may be listed in order between angle brackets ⟨ ... ⟩, separated by blanks or commas. A *character string* can be listed directly as text without any separation marks. Angle brackets, *i.e.*, ⟨ and ⟩, may be used either as *metasymbols*[1] surrounding strings or as symbols within strings, depending on their context.

For example,

```
S = ⟨3, 7, 2, 2 ⟩
```

is a string of numbers called `S`, say the number of days it snowed each week in a given winter month. But this equation is itself a character string, called `C` say,

```
C = ⟨S = ⟨3, 7, 2, 2⟩⟩
```

in which the inner angle brackets are symbols, and the outer angle brackets are metasymbols in this context. This new equation is also a character string, of course, and so on. Please pay careful attention to the term used for `C`. It is specifically referred to as a character string as opposed to a string. Remember that a character string can be listed directly as text without any separation marks. This is the form used for the string `C`. An alternative listing of this same string would have shown the `S`, *blank*, =, *blank*, *etc.* as separate components of the string `C`, each separated

[1] A metasymbol is a special symbol that is used to show structure in a definition of another symbol.

by commas. However, since C has been represented as a character string the previous form was used rather than the latter. See Figure 1.1 for a representation of the meaning of these terms.

	1	2	3	4
S =	3	7	2	2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C =	S		=		⟨	3	,		7	,		2	,		2	⟩

String Component Representation
Figure 1.1

The relation between an component and a string is denoted by the symbol ∈ meaning "is an component of", for example

7 ∈ S.

The number of components in a string is called its length. This is represented by vertical bars surrounding the string name, as |S|. For example, the string S above has length 4, namely 3, 7, 2, and 2. Also, |C| = 16 indicates that the length of C is 16. Note that the metasymbols ⟨ and ⟩ do not count as components of S. These symbols are used merely to denote the beginning and end of the string components and thus are not components themselves. Similarly, character string C is of length 16 which is just the number of components in C. Don't forget that in this string there are five blank characters that are components of C and thus are counted in the total number of components. In the string C, the symbols ⟨ and ⟩ are used both as metasymbols, the outer most pair, and as string components, the internal pair. Thus, the inner pair are counted as components and contribute to the value of length, whereas the outer most pair are recognized as metasymbols and do not count as part of the string length.

Each component in a string has a numbered *place* in the string. For example, the place of 7 in S is 2, and places 3 and 4 both have 2 as their components. String components are denoted by the string name and parenthesized component place number. For example,

S(2) = 7, S(3) = 2

and in character string C,

C(4) = *blank*, C(5) = ⟨ ⟩

writing out and italicizing blank as a convention.

A string with 2 components is called a *2-string* or an *ordered pair*. A string with any integer number n components is called an *n-string*. The string S is a 4-string, character string C is a 16-string.

Components can be strings themselves. For example, the string

T = ⟨⟨1,3⟩, ⟨2,7⟩, ⟨3,2⟩, ⟨4,2⟩⟩

of 2-strings makes the connection between the weeks of the month and number of days of snow in them explicit. Components in a string are not necessarily the same kind of objects. For example, the string

U = ⟨3, ⟨1, 3⟩, ⟨3, 2⟩⟩

contains the number 3 and the components of S and T, where S is a string with components of 1 and 3 and T is a string with components 3 and 2. U(1) is not a string, but U(2) and U(3) are strings.

1.3.2 String Rules

Strings may be described by *explicit listings* as shown above. Strings may also be described by *implicit rules*. For example, let string P be the increasing prime numbers less than 20, described by the rule

P = ⟨n| n increasing, n is a prime number, and n < 20⟩

using a *dummy variable* n and the symbol | meaning *such that*. This can also be expressed as follows

P is the string of values n that are increasing, are prime numbers, and are less than 20.

This same string could also have been listed explicitly as

P = ⟨2, 3, 5, 7, 11, 13, 17, 19⟩.

There are many strings of prime numbers less than 20, but P is the only such string of increasing primes. For example, string Q, described as

Q = ⟨n| n decreasing, n is a prime number, and n < 20⟩

is the string

Q = ⟨19, 17, 13, 11, 7, 5, 3, 2⟩.

The dummy variable used in describing Q is completely independent of the dummy variable used in describing P. It has no meaning outside either description. Any other dummy variable will do as well. For example, string Q can be described as

Q = ⟨x| x decreasing, x is a prime number, and x < 20⟩

using dummy variable x instead of n. It is the same Q defined with a different dummy variable.

Although the dummy variables n and x above are single characters, it may not be necessary or even desirable in many cases. It may be useful to give the dummy variable a name for better understanding of a situation. For example, the string P can be described more explicitly as

```
P = ⟨number | number increasing, number is a prime,
    and number < 20⟩
```

using number as the dummy variable. In the same way, string names may be more descriptive when it is useful or to improve understanding of the meaning. For example, other names for P might be

```
Primes, Primes_Up_To_20, Ascending_Primes_Up_To_20
```

each of which is more descriptive. It is typically convenient to make names connected words, rather than phrases. This allows the names to be used in a different context. For example, they might be used in programs. In this case, underlines can be used to connect ordinary words into names.

1.3.3 Substrings and the Empty String

Strings have *substrings,* which are contiguous subsequences of components. For example,

```
⟨3⟩, ⟨7⟩, ⟨2⟩, ⟨3, 7⟩, ⟨7, 2⟩, ⟨2, 2⟩, ⟨3, 7, 2⟩, ⟨7, 2, 2⟩
```

are all substrings of S. But

```
3, 7, 2, ⟨3, 2⟩, ⟨7, 3⟩
```

are not substrings of S. In this case, 3, 7, 2 are components of S but not strings, and while ⟨ 3, 2 ⟩ and ⟨ 7, 3 ⟩ are strings with components of S, the components are not in contiguous order as found in S. The symbol for substring is δ, meaning "is a substring of", for example

```
⟨3,7⟩ δ S.
```

A string with no components is called the empty string and denoted ⟨ ⟩. At first glance, something with no components may not seem to deserve being called a string. With the number zero (which doesn't exist in roman numerals, for example), it will be useful to have the concept of an *empty string*. The empty string is a substring of every string, even the empty string, itself. In particular, ⟨ ⟩ is a substring of string S, that is

```
⟨ ⟩ δ S.
```

Note that ⟨ ⟩ is not a component of S. But ⟨ ⟩ can be a component of a string if it is explicitly included as a component. In this case, it is a component as well as a substring. Thus,

```
M = ⟨4, 7, ⟨2, 8⟩, ⟨ ⟩, ⟨6, 7⟩⟩
```

is a string with five components, namely 4, 7, the string ⟨ 2, 8 ⟩, the empty string ⟨ ⟩, and the string ⟨ 6, 7 ⟩. In this case the empty string is a component of M as well as a substring of M.

At first it may seem strange, but any string is a substring of itself. Its components in total make up a contiguous subsequence of its components. For example, S, or ⟨ 3, 7, 2, 2 ⟩ explicitly, is a substring of S.

1.3.4 String Operation of Catenate

One of the basic operations that can be performed on strings is *catenation*, also sometimes called *concatenation*. In this operation, two strings are combined together to form a new string. For example, given strings S1 = ⟨3, 7⟩, S2 = ⟨2, 2⟩, and the concatenation operator, &, then

```
S1 & S2 = ⟨3, 7⟩ & ⟨2, 2⟩
        = ⟨3, 7, 2, 2⟩
        = S3
```

Note that the catenation of

```
⟨3, 7⟩ and ⟨2, 2⟩
```

removes the interior string symbols to yield

```
⟨3, 7, 2, 2⟩.
```

Catenation can be shown directly on the interior strings with the catenation operator &, for example

```
⟨3, 7⟩ & ⟨2, 2⟩ = ⟨3, 7, 2, 2⟩,
⟨3⟩ & ⟨7⟩ & ⟨2⟩ & ⟨2⟩ = ⟨3, 7, 2, 2⟩.
```

Catenation can be applied to any string of strings to create a new string. For example

```
⟨3⟩ & ⟨3, 7⟩ & ⟨3, 7, 2⟩ = ⟨3, 3, 7, 3, 7, 2⟩.
```

1.3.5 String Operations of Decomposition and Composition

A non-empty string can be *decomposed* into its first component, called the *head* of the string, and the remainder of the string, called the *tail*.

For example, S = ⟨3, 7, 2, 2⟩ can be decomposed into its head and tail as follows, where ħ represents the head and *t* represents the tail,

```
ħ(S)    = ħ(⟨3, 7, 2, 2⟩)
        = 3
```

and

```
t(S)    = t(⟨3, 7, 2, 2⟩)
        = ⟨7, 2, 2⟩.
```

Let T = ⟨⟨1, 3⟩, ⟨2, 7⟩, ⟨3, 2⟩, ⟨4, 2⟩⟩. Since a component of a string may be another string, the head may be a string. For example

```
ħ(T) = ⟨1, 3⟩
t(T) = ⟨⟨2, 7⟩, ⟨3, 2⟩, ⟨4, 2⟩⟩.
```

In this case, since ħ(T) is a string, it has a head and tail, as well, namely

```
ħ(ħ(T)) = 1
t(ħ(T)) = ⟨3⟩.
```

Since $h(h(\mathtt{T}))$ is 1, and not a string, it has no head, more succinctly

$$h(h(h(\mathtt{T}))) \text{ is undefined.}$$

The tail of a non-empty string will always be a string, but may be the empty string. At that point the possibility for further decomposition stops. For example, every component of `S` can be separated out in a sequence of decompositions, as follows

$$\begin{aligned}
&h(\mathtt{S}) = 3\\
&h(t(\mathtt{S})) = 7\\
&h(t(t(\mathtt{S}))) = 2\\
&h(t(t(t(\mathtt{S})))) = 2
\end{aligned}$$

and at this final point

$$t(t(t(t(\mathtt{S})))) = \langle\ \rangle.$$

In the reverse direction, an object and string can be *composed* by adjoining the object to the front of the string, as the new head of the resulting string in an operation called ⊕. In composition, the string may be empty. For example, $h(\mathtt{S})$ and $t(\mathtt{S})$ composed back with the composition operator ⊕ "becomes first component of", which yields the original string `S`.

$$3 \oplus \langle 7,\ 2,\ 2\rangle = \langle 3,\ 7,\ 2,\ 2\rangle = \mathtt{S}.$$

That is, for any non-empty string, in this case `S`,

$$h(\mathtt{S}) \oplus t(\mathtt{S}) = \mathtt{S}.$$

Note that $h(\mathtt{S})$ also has the name `S(1)`, namely the head of any string is its first component, if any.

`S` can be composed out of its components, as shown in the last line of the sequence

$$\begin{aligned}
\mathtt{S} &= \langle 3,\ 7,\ 2,\ 2\rangle\\
&= 3 \oplus \langle\ 7,\ 2,\ 2\rangle\\
&= 3 \oplus (7 \oplus \langle 2,\ 2\rangle)\\
&= 3 \oplus (7 \oplus (2 \oplus \langle 2\rangle))\\
&= 3 \oplus (7 \oplus (2 \oplus (2 \oplus \langle\ \rangle))).
\end{aligned}$$

1.3.6 Exercises

1. Let Fx be the increasing string of prime factors of a natural number x, for example F15 = ⟨ 3, 5 ⟩. Determine the strings and lengths for F7, F12, F19, F32.

2. Describe an implicit rule for the decreasing string of perfect squares less than 50. Give the string explicitly as well.

3. Determine all substrings of the string ⟨1, 2, ⟨1, 2⟩⟩.

4. Give a formula for the last component of a string, say `S`, and for the next to last component. Are there conditions required for the existence of such components?

5. For an `n`-string, say `K`, provide a formula for component `i` of `reverse(K)`, where 1 <= `i` <= `n`.

1.4 Mathematical Concepts of Sets

1.4.1 Sets and their Members

Sets are collections of objects in which order and duplication play no role. Objects in a set are called its *members*. Sets are abstractions of strings, treating their components as collections of members in the sets.

Members of a set may be listed in any order between braces (also known as curly braces or curly brackets), {...}, separated by blanks or commas. For example, the collection of components of string S (described in Section 1.3) make up a set, say X where

```
X = {3, 7, 2}
```

in which the repeated component 2 in S appears only once in X. The members of set X may also be listed in any order. In particular, the members may be listed in ascending order, for example

```
X = {2, 3, 7}
```

but that is not required.

The relation between a member and a set is denoted by the membership symbol ∈ meaning "is a member of", for example

```
7 ∈ X.
```

The number of members in a set is called its *cardinality*. For example, the set X above has cardinality 3. While a string has a length, with a component at each place up to its length, the members of a set may be in any order and the collection has a cardinality.

Members of sets can be sets themselves. For example, the set

```
Y = {X, {3, 7}, {7, 2}, {2, 3}}
```

consists of X and some other sets made up of its members. Members in a set are not necessarily the same kind of objects. For example, the set

```
Z = {X, S} = { {3, 7, 2}, <3, 7, 2, 2>}
```

contains both the set X and the string S as members. In turn, strings can have sets as components.

1.4.2 Set Rules

Sets may be described by *explicit listings* as shown above. Sets may also be described by *implicit rules*. For example, let set R be the prime numbers less than 20, described as

```
R = {n| n is a prime number, and n < 20}
```

using a *dummy variable* n.

This set may also be listed explicitly as

```
R = {2, 3, 5, 7, 11, 13, 17, 19}.
```

The same set R may be listed as

```
R = {11, 13, 17, 19, 2, 3, 5, 7}
```

or with its members in any other order. R is of cardinality 8. As noted in the previous section, there are many strings of prime numbers less than 20, but R is the only such set of primes.

1.4.3 Subsets and the Empty Set

Sets have *subsets,* which are subcollections of members. For example,

```
{3}, {7}, {2}, {3, 7}, {7, 2}, {2, 3}
```

are all subsets of X. But

```
3, 7, 2, <3, 7>
```

are not subsets of X. In this case, 3, 7, 2 are members of X but not sets, and <3, 7> is a string, not a set, with members of X.

The *empty set* is a set with no members, denoted { }. The empty set { } is distinct from the empty string <>. The empty set is a subset of every set, even itself. In particular, { } is a subset of set X. Note that { } is not a member of X because it is not explicitly listed as a member. But { } can be a member of a set as well as a subset. For example, the set B

```
B = {2, 3, {}, {2, 3}}
```

contains as one of its members the empty set. Thus, the empty set is a member of B as well as a subset of B.

Any set is a subset of itself. For example, X, or {3, 7, 2} explicitly, is a subset of X.

1.4.4 Set Operations

There are several operations on sets. Three binary set operations are especially useful, namely *union, intersection,* and *difference.* The union of two sets, with operator denoted ∪ , is the set of members in either or both of the sets, namely, for any sets A and B,

$$A \cup B = \{z \mid z \in A \text{ and/or } z \in B\}.$$

For example, recall that X had been previously defined to be X = {3, 7, 2} and Z was defined to be Z = {X, S}, where X is the set X and S is the string defined in Section 1.3 to be S = ⟨ 3, 7, 2, 2 ⟩. Then

$$X \cup Z = \{3, 7, 2, X, S\}$$

since the members 3, 7, and 2 are members of X and the set X and the string S are members of Z.

The intersection of two sets, with operator denoted $\cap$, is the set of members that are in both sets, namely for any two sets A and B,

$$A \cap B = \{z | \ z \in A \text{ and } z \in B\}$$

For example, the set Y was previously defined to be Y = {X, {3, 7}, {7, 2}, {2, 3}}, and Z was defined as Z = {X, S}, so

$$Y \cap Z = \{X\}$$

since the set X is the only member that is common to both Y and Z.

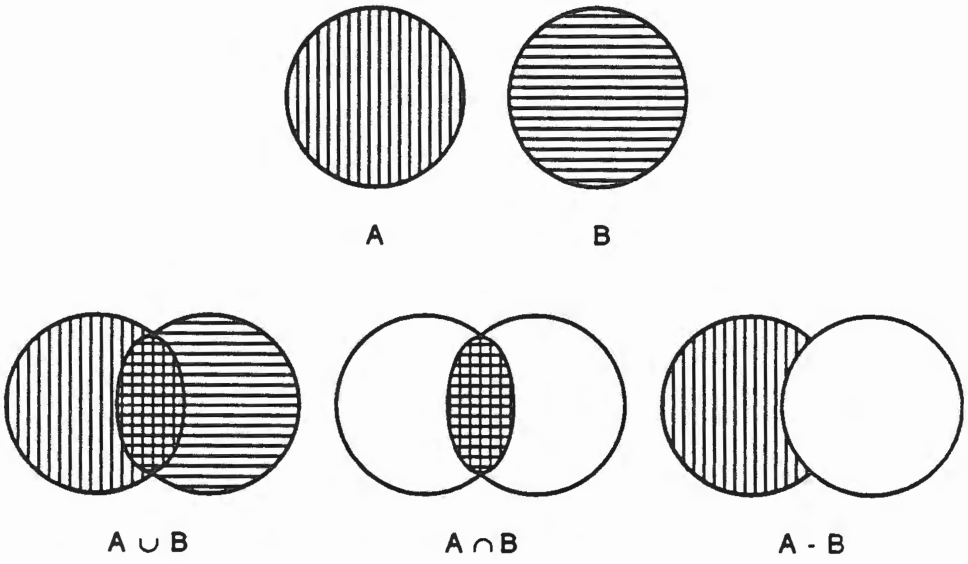
The difference of two sets, with operator denoted -, is the set of members of the first set that are not members of the second set, namely for any two sets A and B

$A - B = \{z | \ z \in A \text{ and } z \notin B\}$.[2],

For example

$$Z - Y = \{S\}$$

These set operations are easily visualized in Venn diagrams, as shown next.



Venn Diagrams
Figure 1.2

[2] The symbol ∉ means "not an element of." Note that in general the slash '/' over, or before, a symbol means not.

There are several set identities with these operations. For example, for any sets, A, B, C

$$\begin{aligned}
&A \cup A = A\\
&A \cup \{\} = A\\
&A \cup B = B \cup A
\end{aligned}$$

$$\begin{aligned}
&A \cap A = A\\
&A \cap \{\} = \{\}\\
&A \cap B = B \cap A
\end{aligned}$$

$$\begin{aligned}
&A - A = \{\}\\
&A - \{\} = A\\
&\{\} - A = \{\}
\end{aligned}$$

$$\begin{aligned}
&A \cup (B \cup C) = (A \cup B) \cup C\\
&A \cap (B \cap C) = (A \cap B) \cap C
\end{aligned}$$

$$\begin{aligned}
&A \cup (B \cap C) = (A \cup B) \cap (A \cup C)\\
&A \cap (B \cup C) = (A \cap B) \cup (A \cap C)
\end{aligned}$$

$$\begin{aligned}
&A \cup (B - C) = (A \cup B) - (A \cup C)\\
&A \cap (B - C) = (A \cap B) - (A \cap C)
\end{aligned}$$

1.4.5 Exercises

1. Are the sets below different or the same? Why or why not?

   a. $S = \{2, 5, 8\}$ and $T = \{5, 2, 8\}$
   b. $Q = \{2, 4, 7\}$ and $R = \{2, 4, 8\}$
   c. $C = \{\}$ and $D = \{n | n > 0 \text{ and } n = 0\}$
   d. $L = \{z | z < 10 \text{ and } z > 0\}$ and $M = \{x | x \text{ is non-negative}\}$

2. Given the sets below, perform the following set operations:

   $C = \{1, 2, 5, 8\}$ $\quad D = \{2, 4, 6, 8\}$
   $E = \{x | x < 0\}$ $\quad F = \{z | z > 0\}$
   $G = \{y | y > 0 \text{ and } y < 10\}$ $\quad H = \{a | a < 0 \text{ and } a > 10\}$

   a. $C \cup D$
   b. $G - H$
   c. $E \cap F$
   d. $E \cup F$
   e. $E - F$
   f. $C \cup E$
   g. $C \cap E$
   h. $H - F$
   i. $C \cap G$
   j. $D \cap F$

3. Enumerate the components of the set Q where

   $Q = \{x \mid x \text{ is one of the first five letters of the lower case alphabet}\}$

4. Write a set specification that describes the character set from which you name is a member This includes both upper and lower case characters.

5. Write a set specification for the set operation of the union of set A and set B in $A \cup B$.

1.5 Additional Mathematical Concepts

1.5.1 Mathematical Relations

A set of ordered pairs is called a *mathematical relation*, or more simply a *relation*. For example, consider the set r of ordered pairs ⟨ x, y ⟩ such that

```
r = {⟨x,y⟩| x > 0, y < 5, x < y}.
```

For example

```
⟨2, 4⟩, ⟨1, 2⟩
```

are members of r since these pairs represent values of x > 0 and y < 5 where x < y, while

```
⟨4, 2⟩, ⟨4, 5⟩
```

are not members of r. In the first case x = 4 is not less than y = 2, and in the second case, y is not less than 5.

The set of all first members of a relation is called its *domain*. For example, the possible values for x in r are

```
domain (r) = {1, 2, 3}
```

because the first member of r must be greater than zero and less than y which, in turn, must be less than 5. In order to be less than 5, y can be no larger than 4. Thus, values of x must be greater than zero and less than 4, leaving only the possible values given above as the domain of r.

The set of all second members of a relation is called its *range*. The possible values for y in r are

```
range (r) = {2, 3, 4}
```

because y must be greater than x, which we have previously determined can be no less than one. So y must be no less than 2 and, as determined above, must be no greater than 4, leaving only the values given above as possible values of y.

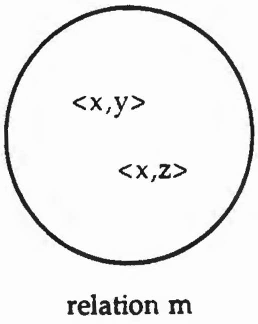
A member of the domain of a relation is called an *argument* of the relation. A member of the range of a relation is called a *value* of the relation. For example, 2 and 3 are both arguments and values of relation r, while 1 is strictly an argument of r and 4 is strictly a value of r.

The empty set is also the *empty relation*. For example, if r is generalized to a relation depending on parameters p and q written as

```
r(p, q) = {⟨x, y⟩| x > p, y < q, x < y}
```

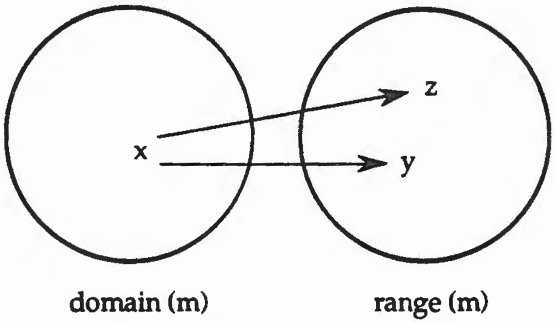
then the relation may be empty for some values of p and q. For example, if p = q, then r (p, q) will be the empty set.

Relations are easily visualized as shown next. First, a relation, say m, is a set of ordered pairs as shown in Figure 1.3.



Relation Set
Figure 1.3

Sometimes it is useful to separate the components of these pairs and show them connected by arrows, as in Figure 1.4, the first components making up the domain of the relation m, the second components making up the range.



A Relation Mapping
Figure 1.4

In general, there are four distinct cases where these components might lie in respect to the domain and range, shown in Figure 1.5, namely where

$$x \in \text{domain } (m) - \text{range } (m)$$

or

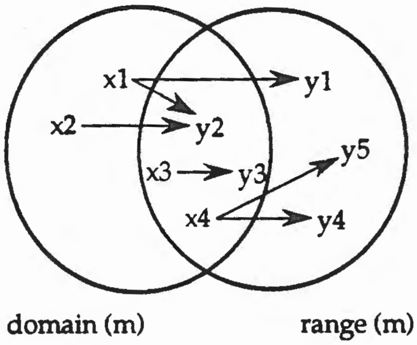
$$x \in \text{domain } (m) \cap \text{range } (m)$$

and where

$$y \in \text{range } (m) - \text{domain } (m)$$

or

$$y \in \text{range } (m) \cap \text{domain } (m).$$



Mappings of Additional Relations
Figure 1.5

Any one of these cases may be empty in a specific relation. For example, the relation defined by the simple inequality $x < y$ for x, y integers between 0 and 100, with domain (m) 0 to 99 and range (m) 1 to 100 has examples of all four cases. But the same simple inequality $x < y$ for domain (m) 0 to 10 and range (m) 90 to 100 is empty in the case domain (m) ∩ range (m) .

The domain or range of a relation may be a product set. For example

$$s = \{\langle\langle x, y\rangle, \langle u, v\rangle\rangle \mid x + y < u - v\}$$

is a relation where the strings $\langle x, y \rangle$ and $\langle u, v \rangle$ are 2-string components of the product set of two sets.

As sets, relations inherit all set properties and operations. In particular, relations are closed[3] under the three binary set operations of union, intersection, difference.

1.5.2 Mathematical Functions

A relation in which each argument has a unique value is called a *mathematical function*, or more simply, a *function*. Neither r nor s above are functions because arguments 1 and 2 both have multiple values. For example,

$$\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle$$

are all members of relation r. A simple subset of r, namely

$$f = \{\langle x, y\rangle \mid x > 0, y < 5, x = y - 1\}$$

is a function. In function f, $x < y$, but for each argument x only one value y will exist.

As for relations, the empty set is also the *empty function* and the domain or range of a function may be a product set. Functions inherit all set and relation properties and operations.

Functions are closed under the two binary set operations of intersection and difference, but not under union. However, functions with disjoint domains, *i.e.*, domains that have no values in

[3] A relation is said to be closed under a particular operation if the given operation, acting on elements of the set, always produces another element of the set.

common, are closed under union. In this case disjoint domains ensure that every argument has a unique value in the function.

A function in which each ordered pair member contains identical objects is called an *identity function*. For example,

```
    i = {⟨x,y⟩ | x = y}
```

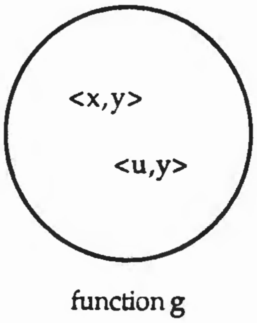
is an identity function because the two values x and y must always be the same.

A function whose range has cardinality 1 is called a *constant function*. Every argument in the domain has the same constant value. Thus,

```
    j = {⟨x,y⟩ | y = 2}
```

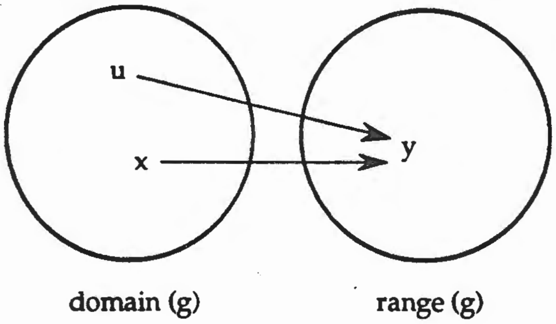
is a constant function because no matter what value is chosen from the domain of x, the value of y is always 2.

Functions are easily visualized as shown next. First, as a relation, a function, say g, is a set of ordered pairs, but with a unique second component for each first component as shown in Figure 1.6.



Function Set
Figure 1.6

Again, it is useful to separate the components of these pairs and show them connected by arrows, as in Figure 1.7, the first components making up the domain of the function g, the second components making up the range.



A Function Mapping
Figure 1.7

There are four distinct cases in which these components might lie in respect to the domain and range, shown in Figure 1.8, namely where

```
   x ∈ domain (g) - range (g)
```

or

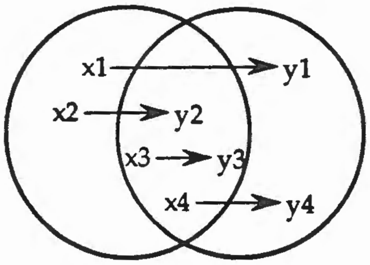
```
   x ∈ domain (g) ∩ range (g)
```

and where

```
   y ∈ range (g) - domain (g)
```

or

```
   y ∈ range (g) ∩ domain (g).
```



domain (g) range (g)

Mappings of Additional Functions
Figure 1.8

Any one of these cases may be empty in a specific function. For example, the function defined by the simple equation x + 1 for x, y integers, with domain (g) 0 to 99 and range (g) 1 to 100 has examples of only three cases, there being no case where x is in domain (g) - range (g) and y is in range (g) - domain (g).

A member ordered pair ⟨ x, y ⟩ of a function f may be given in *value notation*, y = f(x).
A function whose domain and/or range are product sets with members

```
   ⟨x1, x2, ..., xm⟩, ⟨y1, y2, ..., yn⟩
```

has value notation

```
   ⟨y1, y2, ..., yn⟩ = f(⟨x1, x2, ..., xm⟩)
```

where the string ⟨ x1, x2, ..., xm ⟩ is the argument to the function f yielding the value string ⟨ y1, y2, ..., yn ⟩.

As with sets, functions may be defined by explicit enumeration or by implicit rules.

The successive application of the values of one function f as arguments of another function g defines another function, called the *composition* of f and g, namely if

```
   y = f(x), z = g(y)
```

then

```
   z = g(f(x)).
```

The composition of functions f and g must follow what is possible, from an initial argument x to a value y of f which is used as an argument of g to reach a value z. In particular, the value y in range (f) must be a member of domain (g) to carry out the composition. It is possible for the composition of f or g to be an empty function, even though neither f or g are themselves empty. A simple example is when f is a numerical function, say y is the square of x, and g is a letter function, say an uppercase letter is converted to its lowercase letter and vice versa. The composition will be empty. Simpler restrictions of composition may simply reduce the domain of f to a smaller domain for the composition. For example, f may have a range of integers positive, zero or negative, but g only a domain of positive integers. The composition of f and g will be restricted to just what g can handle.

1.5.3 Predicates

A function whose range is a subset of the set {F, T} is called a *predicate.* The domain of a predicate may be a product set. Predicates inherit all set, relation, and function properties and operations. Predicates are closed under the two binary set operations of intersection, difference. As with all other functions, predicates with disjoint domains are closed under the binary set operation of union.

One unary and three binary operations among predicates are especially useful. The unary operation is *not*, the binary operations are *and*, *or*, and *xor*. Refer to Table 1.1 for the meaning of these operations as applied to predicates. From this table, sometimes called a *truth table*, it is easy to see that if p(x) = T and p(y) = F, then p(x) and p(y) = F, but p(x) or p(y) = T.

p(x)	p(y)	**not** p(x)	p(x) **and** p(y)	p(x) **or** p(y)	p(x) **xor** p(y)
F	F	T	F	F	F
F	T	T	F	T	T
T	F	F	F	T	T
T	T	F	T	T	F

Truth Table
Table 1.1

If multiple predicates are used with different logical operators, then there is an order which must be followed in evaluating the expression. All *not* operations are performed first, followed by the *and* operations, then last the *or* and *xor* operators are applied in a left to right order. Thus, if p(z) = F, and using p(x) = T, p(y) = F from above then

```
d = p(x) and p(y) and p(z)
  = F and p(z)
  = F
e = p(x) or p(y) and p(z)
  = p(x) or F
  = T
f = p(x) or not p(y) and p(z)
  = p(x) or T and p(z)
  = p(x) or F
  = T
```

Two quantifications of predicates define useful new predicates, namely *there exists*, denoted ∃, and *for all*, denoted ∀. The former is known as existential quantification and states a fact, namely that there exists an argument for which the predicate is true. The latter is known as universal quantification and states the fact that for every possible argument the predicate is true.

A member ordered pair ⟨ x, y ⟩ of a predicate p may be given in value notation y = p(x). A predicate whose domain is a product set with members

```
⟨x1, x2,..., xm⟩
```

has value notation

```
y = p(⟨x1, x2,..., xm⟩).
```

As sets, predicates may be defined by explicit enumeration or by implicit rules.

The successive application of the values of a function f as arguments of a predicate p defines another predicate, called the *composition* of f and p, namely if

```
y = f(x), z = p(y)
```

then

```
z = p(f(x)).
```

1.5.4 Exercises

1. Consider m = {⟨ q, r ⟩ | q < 5, r > 10}. Is m a relation? Is it a function? Explain your answer.

2. Write a function which maps all positive integers to their negative values, *e.g.*, given 5, the function returns -5.

3. Given the predicates p(x) = T, p(y) = F, and p(z) = T, what is the value of the following:

   a. p(x) **and** p(y) **and** p(z)
   b. **not** p(x) **xor** p(y)
   c. p(y) **or** p(z) **and** p(x)
   d. p(y) **and** p(z) **or** p(x)

4. Determine the composition of functions f and g where

   ```
   f = {⟨x,y⟩ | x >= 0 and y = x - 1000}
   g = {⟨y,z⟩ | y >= 0 and z = y - 100}
   ```

5. Let

   ```
   <y1,y2> = f(<x1,x2,x3>)
           = <x1 + x2, x2 - x3 | x1 > 0, x2 = 0, x3 <0>
   ```

   Determine explicitly the domain and range of f.

1.6 State Machines and Formal Grammars

1.6.1 State Machines

Sometimes in computing it is easier to describe how something is computed than to define it in any other way. A simple, universally understood means of doing so is via the use of a common notation that can be very powerful. This concept is known as a *state machine.*

A *state machine* is defined by an *initial state* `u0` and a *transition function* `f` whose domain is a set of ordered pairs

```
domain (f) = {<s, u> | <s, u> non terminal}
```

and whose range is another set of ordered pairs

```
range (f) = {<r, v> | <r, v> reachable}.
```

The arguments of a state machine transition function `<s, u>` consist of a *stimulus* `s` and a *state* `u`. The values of a state machine transition function `<r, v>` consist of a *response* `r` and *new state* `v`. For a state machine with transition function `f`, an initial state `u0`, and a sequence of stimuli

```
s1, s2, s3, ...
```

defines a sequence of transitions to states

```
u1, u2, u3, ...
```

and responses

```
r1, r2, r3, ...
```

such that

```
<r1, u1> = f(<s1, u0>),
<r2, u2> = f(<s2, u1>),
<r3, u3> = f(<s3, u2>), ... .
```

This simply means that if the state machine is started in state `u0` and receives stimulus `s1`, then the state machine will respond by providing response `r1` and moving to state `u1`. From state `u1`, with stimulus `s2`, the state machine responds by providing response `r2` and moving to state `u2`, *etc.*

A transition function `f` can be represented by a pair of functions `<g, h>`, a *response* function `g` and a *state transition* function `h` such that if

```
<r, v> = f(<s, u>),
```

then

```
r = g(<s, u>), v = h(<s, u>).
```

An initial state u0 and sequence of stimuli

```
   s1, ... sn
```

defines response rn without explicit calculation of intermediate states, in a derived function k of the form

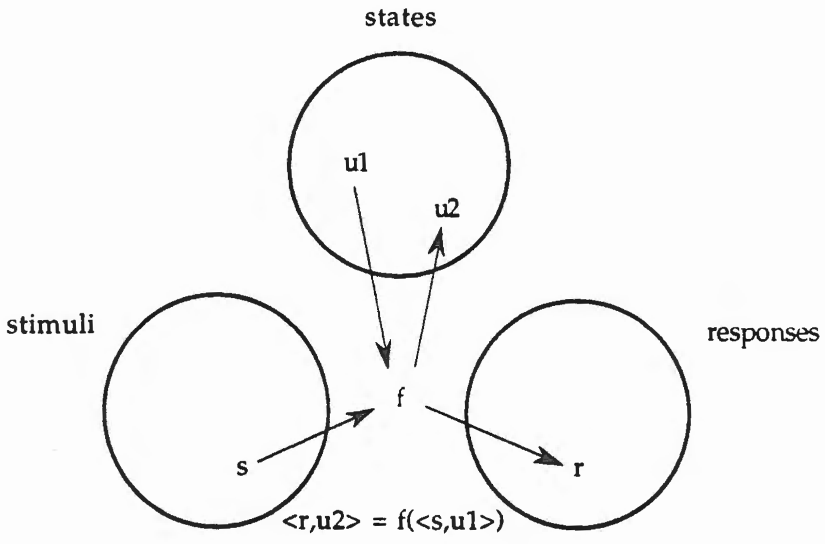
```
   rn = k(u0, s1,..., sn).
```

In particular,

```
   k(u0, s1,..., sn) = g(sn, h(sn-1, h(...))).
```

which simply says that if the state transition function is applied to each of the prior states up to state n-1, then the response function can be applied to stimulus sn and the result of all of the state transition function calls to provide the response for stimulus sn.

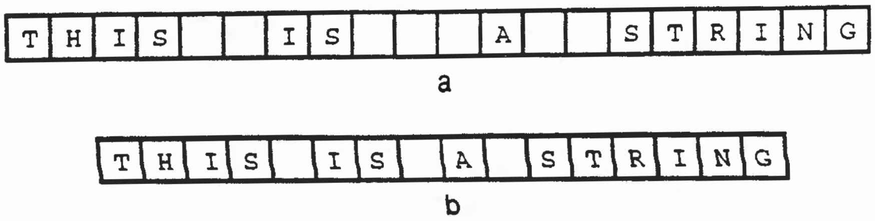
Figure 1.9 shows the transitions in a diagram with mappings from the sets of stimuli and states to responses and new states.



State Machine Diagram
Figure 1.9

State machines are useful in program design. For example, consider a character-by-character examination of a string for the purpose of removing excess blanks, so that on response all blank substrings have been reduced to a single member.

For example, Figure 1.10a shows a string with embedded excess blanks. Figure 1.10b shows the same message with the excess blanks removed.


T H I S   I S   A   S T R I N G
a
T H I S   I S   A   S T R I N G
b

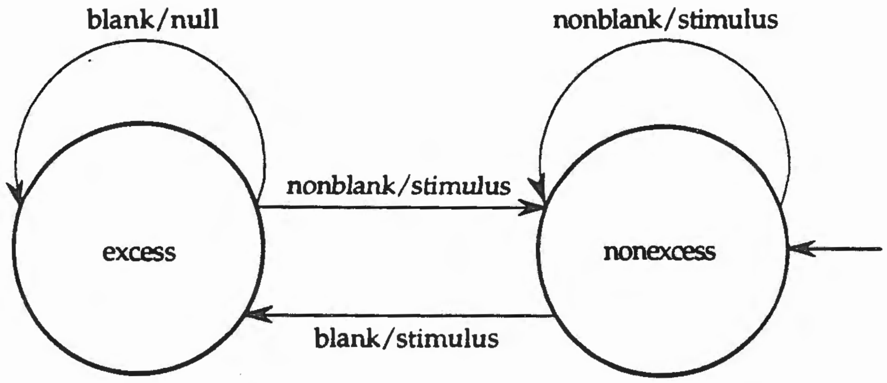
Excess Blanks Removal
Figure 1.10

A state machine for such a purpose can be enumerated in the following table with entries denoting ⟨ newstate, response ⟩.

	Stimulus	
State	Blank	Nonblank
excess	excess, null	nonexcess, stimulus
nonexcess	excess, stimulus	nonexcess, stimulus

Table 1.2
Transition Function for Excess Blanks Removal

Note that "null" means the empty response here. This state machine, initialized to state "nonexcess," will remove all excess initial and interior blanks by passing the first of each substring of blanks found, then ignoring the rest. This state machine can also be diagrammed as shown in Figure 1.10, in which circles denote states, called excess and nonexcess, and a directed line is labeled in the form stimulus/response, the line itself showing the state transformation.



State Machine for Excess Blanks Removal
Figure 1.11

In illustration, apply Figure 1.11 to the example of Figure 1.10a to generate Figure 1.10b.

1.6.2 Formal Grammars

Just as state machines provide a formal definition for specific operations that a computer can perform, there are other ways of describing the rules for correct logical operations. One such mechanism is the use of *formal grammars*. Formal grammars are sets of explicit rules for describing sets of character strings called formal languages. Grammar rules describe how grammar objects can be made up from other grammar objects and/or literal characters. Formal grammars are very useful in describing computer programs; particularly with high order languages where possible ambiguities exist.

1.6.2.1 Context-Free Grammars

Context-free grammars have rules that are independent of context, also called syntax rules. Such rules describe how syntactic categories are defined in terms of intermediate objects, also called non-terminals, and final literals, also called terminals, connected by metasymbols. Final literals are defined by character strings in a separate terminal font or given as terminal objects with separate definitions. Syntactic categories are described in our metalanguage as strings called syntax words consisting of one or more lower case letters, possibly with single underlines '_' between letters. For example,

```
    x, word, one_two, alpha_beta, alpha_beta_gamma
```

are syntax words, but

```
    x1, 1x, one__two, _word, Word
```

are not syntax words. In the example, `x1` is not a syntax word because it contains a non-letter character, and similarly for `1x`. There are two underlines between `one__two` and underlines are only allowed between letters, which is why this example is not a syntax word. Similarly, `_word` is not a syntax word because underlines are only allowed between letter and thus can never begin or end a syntax word. Finally, `Word` is not a syntax word because it is not made up exclusively from lower case letters.

Metasymbols are used to form syntactic assignments and expressions, and include

```
    ::=, |, {, }, [, ]
```

used for following purposes:

```
    intermediate_object ::= expression_in_objects
```

means the `intermediate_object` can be any possible value of the `expression_in_objects` (read `::=` as "is defined as")

```
    expression1 | expression2
```

means either `expression1` or `expression2`

```
    { expression }
```

means zero or more copies of expression

```
[ expression ]
```

means zero or one copy of expression.

1.6.2.2 Formal Grammars of Binary Numbers

In order to illustrate the foregoing concepts, consider a binary number grammar. For example,

```
binary_digit ::= 0 | 1

binary_number ::= binary_digit {binary_digit}
```

is a grammar of two syntax productions that defines all possible binary numbers. That is, the intermediate object binary_number is a string of one or more copies of binary_digit, which is a single binary_digit, followed by zero or more copies of binary_digit defined by {binary_digits}. This grammar for binary numbers is not unique. For example,

```
binary_digit ::= 0 | 1

binary_number ::= {binary_digit} binary_digit
```

also defines all possible binary numbers. Another example is

```
binary_digit ::= 0 | 1

binary_number ::= {binary_digit} binary_digit {binary_digit}
```

This example may not be so useful because {binary_digit} is used twice when it is only needed once, but it illustrates that any number of copies of {binary_digit} can be added without changing the definition of {binary_number}, namely one or more copies of {binary_digit}.

The grammars above allow leading zeros in any number. A different grammar can outlaw leading zeros for non-zero binary numbers. However, the zero binary number must start (and end with a leading zero. Such clean binary numbers can be defined in a syntax such as follows

```
binary_digit ::= 0 | 1

clean_binary_number ::= 0 | 1 {binary_digit}
```

which states that a clean_binary_number is either a zero or a one followed by zero or more copies of {binary_digit}.

1.6.2.3 Formal Grammars of Roman Numerals

Roman numerals can be defined in context-free syntax. Roman numerals are defined as strings of letters with rules about their possible order and values. The letters are I (1), V (5), X (10), D (50), C (100), L (500), and M (1000). In the simplest form of roman numerals, the letters must appear in order from largest value to smallest. There is no limit on the number of M's in a roman numeral, but there are limits for all other letters, namely at most one L, D, and V, and at most four C's, X's, and I's. In order to deal with a finite limit, such as four, we can extend the notation of {C}, which means a string of zero or more C's, to the notation of {C : 4} to mean a string of zero to four C's. Then, roman numerals in the simplest form can be given in a single syntax production as

```
roman_numeral ::= {M}[L]{C:4}[D]{X:4}[V]{I:4}
```

Later forms of roman numerals permit the representation of IIII by IV, VIIII by IX, XXXX by XD, DXXXX by XC, CCCC by CL, LCCCC by CM. In this case, the syntax is better defined in a set of syntax productions such as

```
later_roman_numeral ::= {M} c_alts x_alts i_alts
c_alts              ::= {C:4}| L {C:4}  | C L | C M
x_alts              ::= {X:4}| D {X:4}  | X D | X C
i_alts              ::= {I:4}| V {I:4}  | I V | I X
```

From this grammar it is possible to generate a correct roman numeral to represent any of the Arabic numbers we are familiar with. In the same fashion, the production rules of a specific formal grammar can be used to determine if a given string is a corrects string of that grammar. In Chapter 2, the grammar for the Ada programming language will be examined to recognize correct instructions for the computer to execute.

1.6.3 Exercises

1. Generalize the state machine for excess blanks removal in Figure 1.10 to handle cases where excess blanks may occur before or after the text as well as being embedded.

2. Describe a traffic control signal as a finite state machine with the colors red, yellow, and green controlling the action of wait, prepare to stop, and go.

3. Show the transition function for the traffic control signal of Exercise 2.

4. Describe in English what the syntax production `c_alts` in `letter_roman_numeral` means.

5. Are the following legal strings in the `later_roman_numeral` grammar? Show their Arabic equivalent and explain your answers.

   a. MMMCDII
   b. MXCVIII
   c. MLCCCDXXXXVIII

Chapter 2

Sequential Ada I

Now that you have seen the mathematical foundations that underlie the discipline of computing, you are ready to explore how these foundations can be used to instruct computers to perform the functions that users want them to do. In order to make the computer perform the operations that are needed in the sequence in which they must be performed to satisfy user needs, there must be a means to communicate with the computer. Communication requires that both sides of the communication understand what is meant by the other side. The mechanism by which humans communicate with the inanimate computer is via a computer language. In this course the language that will be used is Ada.

It is interesting to note that in the first fifty years of computers, the methods of describing programs, *i.e.*, the languages, have evolved in dramatic ways. In the first decade, there were no ways except to identify the operations wanted, then describe how to make these operations happen directly in the bare computer memory that would hold the program and its data. The next step was to invent symbolic names for the parts of words in the memory and operations, and to create a new kind of computer program that would translate a program written in the symbolic names into the bare computer memory. Since the first computers were designed to solve numerical problems in science and engineering, these very translations were new kinds of applications in non-numerical problems. These kinds of programs are called assembly programs, and are still used when high efficiency is required, or in special computers with no other program translators. The translator is called an assembler and the language in which the symbolic symbols are written is called an assembly language.

After a decade, another step was taken to create higher level programming languages, such as FORTRAN and COBOL, in which programs are stated in higher level operations which are then translated into bare computer operations and memory by program compilers.[1] After another decade, new programming languages appeared that supported structured programming, such as ALGOL 60 and Pascal. In the past decade, the Ada programming language has been developed for widespread use in industrial, engineering, and military applications.

The Ada programming language is large and certain aspects of it may seem complex, but it represents the best in the support of software engineering for the future. Its effective use will be studied in steps, in order to develop good understanding and engineering control over the programming process in Ada. In this text, sequential Ada will be introduced, by which we mean that part of the Ada programming language that can be used to control the sequential execution of a single computer. Ada also provides the capability for describing the concurrent execution of tasks in one or more computers simultaneously. This aspect of Ada will be treated later in Volume II.

[1] Compilation is the name given to the translation process mentioned earlier for high level languages. When the programming language is translated into a form that the machine can understand and execute, the high level language program is said to be compiled, and thus the process is called compilation.

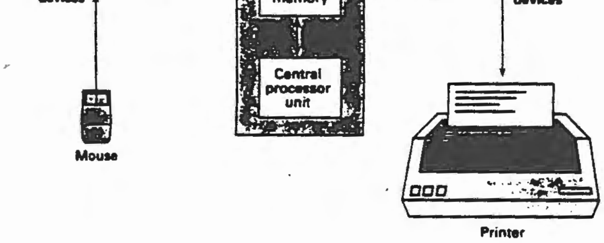
2.1 Introduction to Computers

Computers are wonderful inventions that have enormous power. Unlocking that power and controlling it are the very fundamentals of computer science and software engineering. We must understand what a computer is, at least from a rudimentary level, before we can begin to understand how to control and manage these devices. This section is designed to provide you with an brief introduction to the components of a computer and the process involved in making it do what you want it to do. It is not an in-depth treatment of this subject and many of the details will be deferred until later courses.

2.1.1 What is a Computer?

Before we can begin to explore the world of Ada, we must first understand some fundamental concepts for interacting with a computer. Computers come in many different sizes and have many different purposes. Some are general-purpose computers that can be programmed to do almost anything. Other computers are special-purpose computers that are specifically designed for a single purpose. In any case, a computer is an electronic device that interprets predefined instructions to accomplish a task. Note that the computer cannot "think" because it can only execute predefined instructions. In other words, a computer can do what you tell it to do, and it can do it very fast. However, remember that a computer can ONLY do what you tell it to do; it cannot do what you *mean* for it to do, unless you specifically tell it what to do and how to do it. If you want to learn more about the inner workings and hidden mechanisms of a computer you must wait until later in your course of study when you will be told about the details of how the hardware does what it does. This will include discussions of such computer components as registers, memory, busses, CPU, ALU, etc. For now, we need only to recognize that a computer is a device that we can instruct to do things for us.

The computers that we will be using consist of some basic parts that you should recognize since we will be mentioning them in our later discussions (See Figure 2.1). The first is the "box" that contains the CPU, the memory, and many other peripheral components of the computer. For our purposes we will refer to this "box" as the computer.



Components of a Computer
Figure 2.1

Next, you will see a television-like screen that is used for the computer to communicate with you and to echo back to you your commands to the computer. This device is called a monitor or a CRT (cathode ray tube). It is also sometimes referred to as the screen.

Finally, you should see two methods for you to provide commands to the computer. One is a keyboard which is very similar to the keyboards that typewriters have, but which contains some very special keys. The second device you should see is a mouse, which is the hand-manipulated device attached to the computer with buttons on it. Its purpose is to move a pointing indicator around the screen so that you can point to things that are displayed on the screen. When you are pointing to something that you want the computer to understand, you can depress the button or buttons on the mouse to let the computer know that you want to select the item that the indicator on the screen is pointing to.

These are the essential elements of the computer system that we shall refer to in this textbook. If you are not familiar with any of these items be sure to tell the lab instructor so that he or she can describe the item to you in more detail and answer your questions.

2.1.2 What is an Algorithm?

Computers seem to be very intelligent because they can compute very complex arithmetic expressions quickly, or look up some data from a table with blinding speed. In reality, computers are neither intelligent or dumb; they are, after all, only silicon (beach sand) and electronics and do not possess the attribute of intelligence. How, then, are they able to appear to be so smart? The answer is that they follow a sequence of explicit instructions that someone that *is* intelligent provides. The explicit instructions make up a solution to a problem that we want the computer to repeatedly solve for us. This solution to a known problem is called an *algorithm*.

Algorithms are not new to you, even if you have never heard this term before. When you prepare something in the kitchen and you follow a recipe, you are following an algorithm. An algorithm is nothing more than a step-by-step sequence of instructions that tell what to do in what order. The assembly instructions for a model airplane represent an algorithm, as does the registration instructions for a university or college. Any step-by-step set of instructions can be considered an algorithm.

In working with computers we get very familiar with algorithms because we must provide these step-by-step instructions to the computer in order for it to do anything. The computer cannot do anything by itself, so we must provide it with explicit, detailed instructions about what to do and when to do it. This sounds like it should not be too hard, but, surprisingly, it can be.

Consider an algorithm for adding two numbers together. The algorithm might be described as follows:

1. Align the rightmost digits of the numbers to be added.
2. Add leading zeros as necessary to make both numbers the same length.
3. Sum the digits in the rightmost column.
4. Record the rightmost digit of this sum.
5. If this sum is greater than 10, record the digit in the next to rightmost column of this sum as the carry.
6. Sum the digits in the next to rightmost column and add the carry if there is one.
7. Repeat steps 3 through 5 until there are no columns remaining to be summed.

This may seem very complex given the simplicity of the task, namely adding two numbers together. However, if you think about it, it is precisely what you do when you add two numbers together. Now, as complex as this may seem, it is still not detailed enough to be able to make the computer carry out these instructions. For example, what does it mean to "align the rightmost digits?" Further, what is a column?

It can be seen from this simple example that algorithms that do anything complex or interesting are themselves very complex and detailed. It is our job to reduce this complexity and make our algorithms as simple as possible.

2.1.3 What Is a Program?

In the previous section, we discussed the concept of an algorithm as a step-by-step sequence of instructions to solve a given problem. This sequence of instructions must be presented to the computer in a form that it can understand. The form of communication that we have with computers is a program. A program is a sequence of instructions presented in a specialized language, called a programming language. We use this language to explain to the computer how we want our algorithm executed.

Programs can be written in many languages if your computer can understand them. Languages exist in great diversity, and even within the same language there are often several dialects. A program is written in a language that your computer can understand. In this course we will be studying the Ada programming language. You will write programs that implement your algorithms that are, in turn, designed to solve your problem. In this manner you will solve your problems on the computer.

Programs are written in very special languages. Usually natural languages such as English, or Spanish, cannot be used to communicate with your computer, so a more structured, less free format language must be used. Very shortly we will see some Ada programs and the meaning of these ideas will become more clear.

2.1.4 What Is an Editor, a Compiler, and a Linker?

In the previous section, we learned that our algorithms are communicated to the computer by using a special language to write a program that the computer can understand. Unfortunately, while this is true, it is only part of the story. How do we get the program in a form that the computer can "read?" The answer is that we must first enter the program into the computer. We enter a program by creating a file, or sequence of characters stored in a form that the computer can understand. The file is created by a special program that you will have access to called an *editor*. The editor is a program that accepts your typed input, allowing you to make corrections as necessary, and creates a file that is able to be read by the computer. The details of how an editor is able to do this, indeed, even the details of what specific commands you use in the editor, are not important here. We want you to understand the concept of the editor. Specific instruction in the particular editor that you will use, how to invoke it, what its commands are, etc. will be explained to you in a laboratory.

Once you have entered the program into the computer using the editor, it still is usually in a form that is too high level to be understood by the computer. The reason for this is that the computer can only understand simple sequences of ones and zeros. While it is possible to program directly in the language of the computer, it is tedious and error prone. To reduce the chance for error, it is better to have the ability to express your program in a form that is closer to a natural language such as English. By using a restricted and structured set of English words, we can write a program and enter it into the computer. This is much less tedious and error prone than entering long sequences of ones and zeros, the computer "native tongue." Unfortunately, our program cannot be understood by the computer in this high level form. Therefore, we use another program, provided with your computer, to translate the high-level program that you wrote in the structured English-like form, into the sequence on ones and zeros that the computer can understand. The program that does this translation has a special name. It is called a *compiler.* Compilers are nothing more than translators, translating from the programming language in which you wrote the program into the language that the computer can understand and operate with. We say that the compiler *compiles* our program, by which we mean that it translates it into machine code. The name used for the program in its programming language form is *source code*; the name of the program after translation into machine code is *object code.*

Compilers are very useful to us in making our efforts more productive and more reliable. Their effect can be seen by comparing them to things that we can more easily understand. For example, suppose that you can only speak English. Suppose that you see someone that you desire to communicate with, but you find that this person only speaks Swahili. You cannot effectively communicate. Now imagine that you have a friend that reads English and writes Swahili. You could write a statement to your friend in English and your friend could restate your words in Swahili. You now have an effective one way communication. Now imagine that your computer speaks a special language called binary. You speak a language called Ada. You have a friend, the compiler, that can read your Ada and translate it into binary for the computer. This is precisely what occurs when we program in Ada using an Ada compiler.

Note that if we had a Pascal compiler, we could write our programs in Pascal and the compiler would translate it into the same binary language that the computer can understand. If we had a FORTRAN compiler, it could translate programs written in FORTRAN to binary. However, the Pascal compiler cannot translate a FORTRAN program and vice versa, since it does not "speak" that language.

Now that we have entered our program with an editor and have compiled it with a compiler, we still cannot just walk away. Instead, we must first *link* our program. When the compiler does the translation, it is convenient to take some routines that are used often, such as input/output statements, and not translate them completely. Instead, a reference is made to a library unit for this procedure; its like putting a bookmark in the code and saying I know what goes here and I will fill it in later. This speeds up the compilation time. However, when we have translated our source code to object code we still cannot execute it since we may have several of these reference marks throughout. Thus, we must perform another step called *linking.* This is the process whereby our object code is linked to library routines that the vendor supplies in object code format. In effect, the "bookmarks" are detected and the appropriate code from the library is substituted in its place. The result of linking is an *executable* program that we can directly run on our computer.

We must now test our executable program to ensure that it does what we want. One way to test the program is to supply it with some inputs for which we have already determined the appropriate outputs. We then verify that we get the appropriate, and presumably correct, outputs. Unfortunately, this is not always the case when we initially test our programs. Sometimes, there are errors in our algorithms or in our implementations of our algorithms.

These errors, for historical reasons, are known as *bugs*[2]. We would like to have a tool that would allow us to execute our program with certain controls so that we can see what it does and when it does it. Such a tool is just another program known as a *debugger.*[3] Debugging is a technique that allows us to uncover and eliminate defects in our software. The intent of our approach to software development is to create zero-defect software that is bug-free when it is developed. Practical applications of the cleanroom software engineering method have demonstrated that it is possible to create zero-defect software, and it is this method that we will be studying.

The complete cycle for the development of software can be shown graphically as in Figure 2.2. Note that the first step is understanding and defining the problem to be solved. This is a critical step and one that many students skip or place little importance upon, because it is not "fun." However, it is vital since all other steps in the cycle depend upon getting the problem definition correct. Next, we must design the algorithm to be used to solve the problem. In this step we cannot assume anything. We check that the algorithm is correct by desk checking it, which simply means applying the algorithm by hand to sample input data and watching the transformations on the data until the output is produced. If the output is correct for all of the sample data, then we may proceed to the next step which is program design. This step is where we determine how to implement the algorithm in a programming language. Now we enter the editor and type in our program. After saving this program as a file, we submit it to the compiler. If errors are found, then we reenter the editor and correct them. If no errors are found, we must then link the object code from our program to the vendor supplied library routines. If we discover errors in linking we must either reenter the editor and correct them, or resubmit it to the compiler, depending upon the error. If we are successful in linking we will have an executable program. This executable program can then be tested with test data to ensure that it produces correct results. If so, then we can hand in our program. If not, then we must reenter the development cycle at the appropriate point, usually at the editor step. In general, all programs are developed in this manner.

2.1.5 Exercises

1. What application of computers do you find most unique? Why?

2. Describe in your own words the difference between an algorithm and a program.

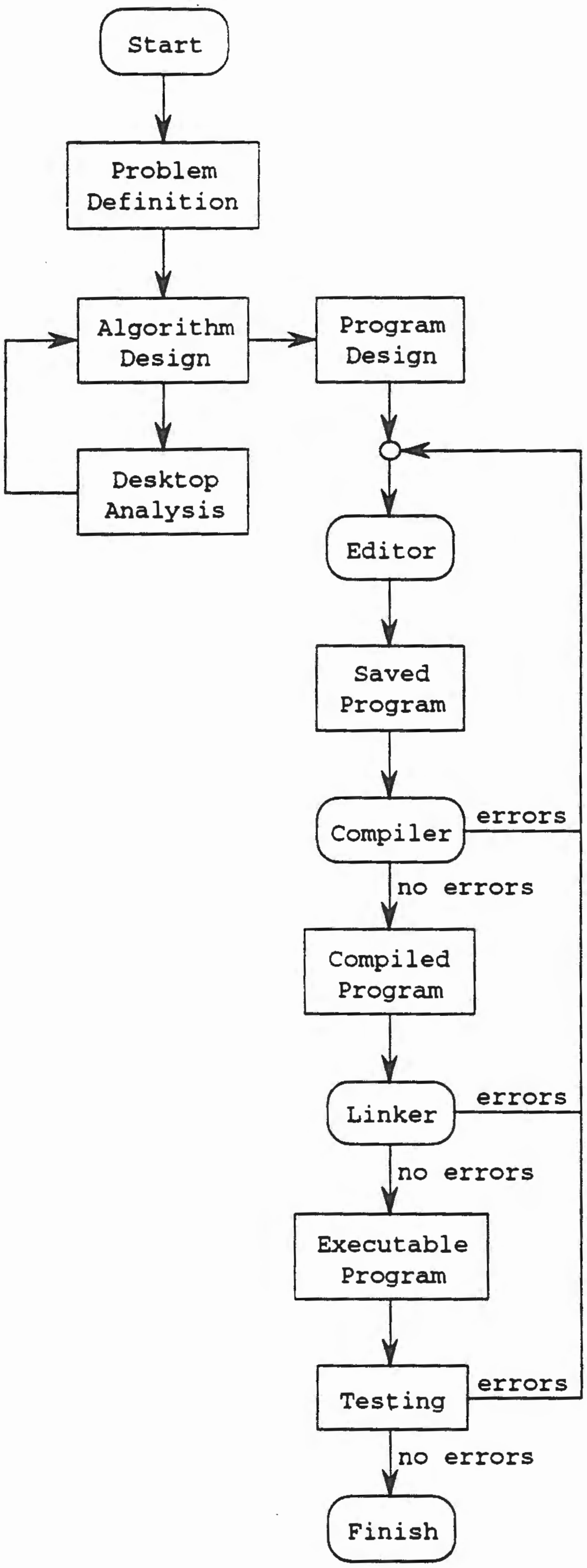
3. Write the complete algorithm, as detailed as you can, for opening a three number combination lock.

4. What is the purpose of a compiler?

5. Can an Ada compiler be used to compile a FORTRAN program? Why or why not?

[2] Rear Admiral Grace Murray Hopper was one of the original programmers on a large vacuum tube computer during World War II. One day the machine did not function properly and she tracked it down to a moth that had shorted out one of the vacuum tubes. She replaced the tube and taped the moth to the logbook, noting that it was a "bug" in the computer. To this day, when the computer is not acting properly we say that the computer, or more properly, our program, has bugs in it.

[3] A debugger is a vendor's program that allows us to single step through each line of our program to trace its actions. In this manner, we can often discover the source of defects, or bugs, and thus eliminate them. In the laboratory you will learn some special debugging techniques and be introduced more formally to the debugger.



Program Generation Flowchart
Figure 2.2

2.2 Introduction to Sequential Ada

In the early 1970's the US Government, particularly the Department of Defense (DoD), noted a disturbing trend. The increasing demand for computers was accelerating at a dizzying pace. Users wanted more computers and they wanted the computers to do more things. This combination served to increase the cost of computers, notably computer software, until a crisis point had been reached. The DoD selected a panel of experts to study this issue and recommend some solutions.

The preliminary study produced by the panel said that there were several things wrong with the way computers were designed, built, and used in the government, but that the most critical element was the software that controlled these computers. One of the causes of this crisis was that there was a proliferation of languages used throughout the DoD. This made it difficult to train people and even more difficult to get someone other than the original author to maintain or enhance software. The combination was driving the cost of computer system acquisition steadily higher at a non-linear pace.

The solution that was recommended by the panel of experts involved a three pronged approach. First, there needed to be a recognized discipline in the development of software throughout its life cycle. This was termed *software engineering*. Second, there needed to be a common environment in which to create software, consisting of at least an editor, a compiler, a debugger, and such other tools as are usually found at commercial software development sites. Lastly, there needed to be a single, common high order language throughout the DoD and preferably throughout the US Government.

The last item led to the engineering of a new language. It is interesting to note that Ada was the first, and only, language to be *engineered*, in the sense that the creation of the language followed the traditional engineering method. Thus, first the requirements for the language were generated. These were gathered over a one year period and consolidated into a requirements document, called the Strawman document. This document was refined in a series of increasingly more complete revisions into a Woodenman, Tinman, Ironman, and Steelman document. This final document, Steelman, served as the final requirements document for Ada and was more properly a specification for the proposed new language.

The DoD then asked for designs of a language that satisfied the Steelman requirements. Over 80 design teams submitted initial proposals. Four of them were selected to produce preliminary designs. In order to preserve the anonymity of the designers, the teams were given color codes, Red (Intermetrics), Blue (SofTech), Yellow (SRI), and Green (CII/Honeywell Bull). The preliminary designs were reviewed and two teams, Red and Green, were selected to prepare detailed designs. Finally, in 1979, the Green language design was selected to be the new DoD language. It was named Ada after Augusta Ada Lovelace, the daughter of the poet Lord Byron. She was, arguably, the world's first programmer.

The result of this engineering effort is a language suitable for a variety of applications. Ada has many advanced features that directly support the concepts of software engineering. In the remainder of this chapter, you will see how Ada can be used to solve simple problems on a computer.

In order for a computer to communicate its results to the external environment (usually the human user), and for the human user to get data into the computer, there needs to be a mechanism for communication. In most programming languages there are built-in commands for such communication with names such as PRINT, WRITE, or PUT[4] for output, *i.e.*, communication from the computer to the user, and READ or GET for input, *i.e.*, for communication from the user to the computer. These commands are collectively known as input-output, or I/O, statements. Ada is somewhat unique in that there are no built-in I/O statements in the language. Instead, the designers of Ada chose to provide the I/O capability by extending the language in the natural way for Ada, namely through the use of a package. Later, you will learn how to use a package to extend the language to perform other such specialized tasks to solve your problems. For now, however, suffice it to say that a package is merely a programming unit in Ada. Therefore, I/O in Ada is accomplished by the use of a predefined, standard set of I/O packages that are not part of the language, but are required to be supplied with every compiler.

The three well structured elementary Ada programs discussed next use character data to Put messages, possibly in response to input data.

2.2.1 Print_Message_1

The first of these programs, with its main procedure named `Print_Message_1`, only puts out a simple message "`Welcome Aboard`", as follows.

```
with TEXT_IO;
procedure Print_Message_1 is
begin
  TEXT_IO.Put (Item => "Welcome Aboard");
end Print_Message_1;
```

The first line begins with a with clause that makes a previously created, predefined Ada package called TEXT_IO available for use in the program. Package TEXT_IO is a predefined package that contains procedures that deal with the Standard Input and Output Files in the Ada environment. Usually these are the keyboard for Standard Input and the CRT, or terminal screen, for the Standard Output. Do not concern yourself with these any further at this point, but we will return to them later to discuss them in more detail. This with clause is

```
with TEXT_IO;
```

and includes the semicolon (;). The program continues on the next line as

```
procedure Print_Message_1 is
```

starting with the reserved word procedure then followed by the procedure name `Print_Message_1`, an identifier which is given in letters and digits, broken into words by the underlines.

The reserved word **is**, on the same line, connects the procedure name with the rest of the program.

The executable part of the program is found between the reserved words **begin** and **end**.

The next statement is a procedure call to `TEXT_IO.Put`, a predefined procedure in the package TEXT_IO, with the character string "`Welcome Aboard`" as its argument or parameter, *i.e.*, the string between the parentheses. The phrase "`Item =>`" identifies the argument following it.

[4] Put is the term Ada uses for output to a file or to the screen. In other high level languages, this may be called Write or Print.

You will note that the procedure name Put is attached to its parent package name TEXT_IO by a period in the full name `TEXT_IO.Put`. The semicolon (;) ends the statement.

```
TEXT_IO.Put (Item => "Welcome Aboard");
```

This statement will put the message

```
Welcome Aboard
```

to the Standard Output. Note that the double quotes in the `TEXT_IO.Put` statement are not part of the output message[5], but only contain between them the text to be written out. Thus, in this example the double quotes used in the `TEXT_IO.Put` statement define the limits of the output message. The semicolon (;) ends the statement again.

The procedure name `Print_Message_1` and semicolon (;) follow the end reserved word to complete the program. The semicolon is necessary to close the `begin...end` part and the procedure name `Print_Message_1` is optional between end and semicolon. While the procedure name is optional in Ada, in this text it will always be used. This is another instance of a good programming practice, where something is not required, but is desirable. Good programming practices may take a few minutes longer to write, but save hours in maintenance. Remember, you only write the code once, but you read it many times.

2.2.2 Print_Message_2

The second program, with main procedure named `Print_Message_2`, deals with both Standard Input and a Standard Output.

```
with TEXT_IO;
procedure Print_Message_2 is
  Choice : CHARACTER;
begin
  TEXT_IO.Put (Item => "Enter a capital letter => ");
  TEXT_IO.Get (Item => Choice);
  if Choice >= 'A' and Choice <= 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else
    TEXT_IO.Put (Item => "Not a capital letter!");
  end if;
  TEXT_IO.New_Line;
end Print_Message_2;
```

Program `Print_Message_2` expands on `Print_Message_1` by checking the state of Input and possibly putting one of two different messages. It begins with the with clause

```
with TEXT_IO;
```

as did `Print_Message_1`, but following reserved word is, the variable named `Choice` is defined to be of type CHARACTER in a declaration

[5] Double quotes can be output, but only by putting consecutive double quotes ("") within messages.

```
Choice : CHARACTER;
```

ahead of the executable part between **begin** and **end**. For now do not concern yourself with what this means beyond the following explanation. The identifier `Choice` names a storage location in memory that is limited, or constrained, to hold only values that are single characters. Later we will expand this definition, but for now this will suffice.

The executable part then begins with the `TEXT_IO.Put` procedure call, as did `Print_Message_1`. However, in this example there is a need for both input and output. Since the program will be requiring input from the user, it is customary and prudent to first write out a message, called a *prompt*, to let the user know that the program is awaiting input, and, more importantly, what kind of input is needed. Accordingly, the argument to `TEXT_IO.Put` is a message to the user asking for a capital letter. Thus,

```
TEXT_IO.Put (Item  => "Enter a capital letter => ");
```

specifies that the prompt to be written to the Standard Output is the message

```
Enter a capital letter =>
```

where the output appears without the enclosing quote marks, as in the previous example.

Next, the program waits for the user to enter a single character. This is caused by the use of the statement

```
TEXT_IO.Get (Item => Choice);
```

that "gets" or reads a character from the Standard Input. The user notifies the computer that he/she has entered the character requested by first typing the character to be entered and then pressing the return key. The return key is also called the enter key on some keyboards. The value that is obtained is placed in the variable `Choice`. Thus, after this statement, the character that the user entered in the Standard Input will be stored internally in the variable `Choice`.

Next is an **if** statement,

```
if Choice >= 'A' and Choice <= 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Not a capital letter!");
end if;
```

with three parts, the **if** condition part, the **then** part, and the **else** part. The if condition is

```
if Choice >= 'A' and Choice <= 'Z'
```

which checks to see if the value stored in the variable `Choice` is greater than or equal to a capital letter 'A' and less than or equal to a capital letter 'Z'. If these two conditions are both true, then the value in `Choice` is guaranteed to be a capital letter. This is true because in the character set used in Ada, all of the capital letters are contiguous, or in sequence.

An alternative way to write this condition is

```
if Choice in 'A' .. 'Z'
```

which makes use of range checking mechanism in Ada. The condition is examined to see if the value in `Choice` is an upper case letter. In Ada this can also be accomplished by checking to see if the value in the variable `Choice` is in the range 'A' to 'Z' inclusive, meaning all of the letters 'A', 'B', 'C' ... 'X', 'Y', and 'Z'. To avoid having to explicitly type in all of these letters Ada has a range notation symbolized by the '..' symbol. Note that this is two dots, not three like you might find in say an English paper. Thus, 'A' .. 'Z' can be used as a shorthand for the capital letters from 'A' to 'Z' inclusive. In later portions of this textbook this range notation feature of Ada will be used in other contexts, so it is important that you understand the concept now.

The **then** part is executed if the **if** condition is TRUE. If the user entered the letter 'A' (without the quotes) at the Get statement, then the simple statement

```
TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
```

will put out a message of two parts in a single line

```
Welcome Aboard, A
```

Note that the first message part ends with a space. This will cause a space to be put in the output file and prevent the two message parts from running together. For example, if the statement had been written as

```
TEXT_IO.Put (Item => "Welcome Aboard," & Choice);
                                     ↑
                               space missing
```

then the output written to the Standard Output with the same input character used in the previous explanation would be

```
Welcome Aboard,A
```

without a space between the comma and the letter A. Note that in Ada an ampersand (&) is used to catenate a string ("`Welcome Aboard, `") and a character (A, in our example). Refer to section 1.3.4 of Chapter 1 for a description of the catenation operation.

Notice that this program causes this to occur by catenating the character in `Choice` to the constant string "`Welcome Aboard, `" and then putting out the entire newly formed string. An alternative way of achieving this same effect would have been the following,

```
TEXT_IO.Put (Item => "Welcome Aboard, ");
TEXT_IO.Put (Item => Choice);
```

In this case, the string "`Welcome Aboard, `" would first be written to the Standard Output, followed immediately by putting the character in `Choice`. The effect on the output would be indistinguishable. Notice also that even though two calls have been made to the Put procedure, the output is still on the same line. Suppose you had wanted the string "`Welcome Aboard, `" to be on one line and the character in `Choice` to be written on the next line. One way to do this in Ada is the following,

```
TEXT_IO.Put (Item => "Welcome Aboard, ");
TEXT_IO.New_Line;
TEXT_IO.Put (Item => Choice);
```

where a call has been made to another procedure found in TEXT_IO, namely, New_Line that has been described earlier. This procedure causes the next Put statement to write its argument at the beginning of the next line. In the program segment above, the first Put statement will cause the string argument "Welcome Aboard, " to be written to the Output file. The call to New_Line will force the next Put statement to write its argument, namely the character in Choice, at the beginning of the next line, so the output in the Standard Output will be, assuming the 'X' is the character in Choice,

```
Welcome Aboard,
X
```

Thus, TEXT_IO has some line formatting control capabilities.

Back to our example, the else part is executed if the if condition is FALSE, being the simple statement,

```
TEXT_IO.Put (Item => "Not a capital letter!");
```

which produces the string

```
Not a capital letter!
```

in the Standard Output.

The if statement provides the programmer with the means to alter the normal sequential flow of control based upon the value of some condition. This is a very powerful tool in a programmer's toolkit.

Finally, the last procedure call

```
TEXT_IO.New_Line;
```

causes the program to reposition the cursor to the beginning of the next line. More properly, it causes a New_Line character to be placed in the Standard Output, but the effect is as previously described.

2.2.3 Print_Message_3

The next program, with main procedure named Print_Message_3, can look at additional characters from the Standard Input in a loop. It will continue to look at characters until a capital letter is found. This program is as follows:

```
with TEXT_IO;
procedure Print_Message_3 is
  Choice : CHARACTER;
--
begin
  Choice := ' ';
  while Choice not in 'A' .. 'Z' loop
    -- Better Look at Next CHARACTER
    TEXT_IO.Get (Item => Choice);
  end loop;
```

```
     if Choice in 'A' .. 'Z' then
       TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
     else
       TEXT_IO.Put (Item => "Try Again");
     end if;
   end Print_Message_3;
```

Program `Print_Message_3` extends the process one more step, by using a *while loop* statement to ask for more characters if no upper case letter is found so far. The variable `Choice` is given a value by assignment before the while loop. It also introduces a way to add comments for readers of the program. Comments have no effect on execution, but can make the program more understandable. Such comments begin with double hyphens, `--`, and continue only to the end of the line they begin on. Comment definers can also create empty lines for better readability as well.

`Print_Message_3` adds the assignment statement

```
   Choice := ' ';
```

which explicitly assigns the value of the space character, `' '`, to the variable `Choice`. The double character symbol `:=` designates that the value of the expression on the right side is to be assigned to the variable on the left side. This symbol may be read as "Choice is assigned (or gets) the value space"; alternatively, it may be read as "Choice becomes space." However, you should not read it as "Choice equals space." The reason for this will be made explicit later when we see Ada's relational operators.

Next, the while loop statement

```
     while Choice not in 'A' .. 'Z' loop
       -- Better Look at Next CHARACTER
       TEXT_IO.Get (Item => Choice);
     end loop;
```

specifies to keep looking for an upper case letter in the Standard Input until one is found, if available. The while condition

```
   Choice not in 'A' .. 'Z'
```

will be TRUE only if the clause is true, *i.e.,* `Choice` has no upper case letter value. That is, as long as no upper case letter has been found, the while condition specifies to continue looking in the while loop. Ultimately, if `Choice` has an upper case letter value, the while loop statement terminates and execution goes to the following if statement. Conversely, if Standard Input has no upper case value, the program terminates at the end of data as an exception, which will be explained later.

In summary, if the `loop...end` loop is entered, when execution reaches end loop, control returns to the while part to check the new value in `Choice`. The `loop...end` loop part will be executed zero or more times until `Choice` has an upper case letter value or it exhausts the data. It is clear that there is no way to determine in the general case how many times a while loop will execute. For this reason it is referred to as *indefinite iteration*. The condition in a while loop must be TRUE for the loop to be entered, and must remain TRUE each time the loop is entered. Thus, a while loop is exited normally only when the condition specified is determined to be FALSE.

Two comments are embedded in the program. First, an empty comment

```
--
```

is used to separate the declarations and the executable part of the program. Second, a comment is added to the line after the loop statement as

```
while Choice not in 'A' .. 'Z' loop
  -- Better Look at Next CHARACTER
```

to explain what follows in the program at that point.

These three examples of Ada programs show the main possibilities of Ada. An Ada program is always defined by a top level Ada procedure, such as `Print_Message_1`, possibly referring first to one or more Ada packages, such as TEXT_IO, by means of a with preface. TEXT_IO contains procedures Put and Get with full names `TEXT_IO.Put` and `TEXT_IO.Get` for putting data out and bringing data in from the Standard Output and the Standard Input. Variables to be used must be declared. The main body of a procedure will contain its declarations followed by a `begin...end` part within which its executable statements are found. Statements are executed in the sequence they appear unless they are part of an *if_statement* or a *loop_statement* which define alternation or iteration during execution respectively.

2.2.4 Exercises

1. Consider the program with main procedure `Print_Message_1` in which the Put statement

   ```
   TEXT_IO.Put (Item => "Welcome Aboard");
   ```

   has been replaced with two Put statements

   ```
   TEXT_IO.Put (Item => "Welcome");
   TEXT_IO.Put (Item => "Aboard");
   ```

   to produce a new program with main procedure `Print_Message_1a`

   ```
   with TEXT_IO;
   procedure Print_Message_1a is
   begin
     TEXT_IO.Put (Item => "Welcome");
     TEXT_IO.Put (Item => "Aboard");
   end Print_Message_1a;
   ```

   Determine the effect of `Print_Message_1a` and compare it with that of `Print_Message_1`.

2. Consider the program with main procedure `Print_Message_1` in which the Put statement

   ```
   TEXT_IO.Put (Item => "Welcome Aboard");
   ```

   has been replaced with two Put statements

   ```
   TEXT_IO.Put (Item => "Wel");
   TEXT_IO.Put (Item => "come Aboard");
   ```

to produce a new program with main procedure `Print_Message_1b`

```
with TEXT_IO;
procedure Print_Message_1b is
begin
  TEXT_IO.Put (Item => "Wel");
  TEXT_IO.Put (Item => "come Aboard");
end Print_Message_1b;
```

Determine the effect of `Print_Message_1b` and compare it with that of `Print_Message_1`.

3. Compare the following program with the program in section 2.2.2 with main procedure `Print_Message_2`. Is the output from each program the same or different? Explain.

```
with TEXT_IO;
procedure Print_Message_2 is
  Next_Item : CHARACTER;
begin
  TEXT_IO.Put (Item => "Enter a capital letter => ");
  TEXT_IO.Get (Item => Next_Item);
  if Next_Item not in 'A' .. 'Z' then
    TEXT_IO.Put (Item => "Not a capital letter!");
  else
    TEXT_IO.Put (Item => "Welcome Aboard, ");
    TEXT_IO.Put (Item =>  Next_Item);
  end if;
  TEXT_IO.New_Line;
end Print_Message_2;
```

4. Compare the following program with the one in section 2.2.3 with main procedure `Print_Message_3`. Is the output from each program the same or different? Explain.

```
with TEXT_IO;
procedure Print_Message_3a is
  Next_Item : CHARACTER;
--
begin
  Next_Item := ' ';
  while Next_Item not in 'A' .. 'Z' loop
      -- Better Look at Next CHARACTER
    TEXT_IO.Get (Item => Next_Item);
  end loop;
  if Next_Item not in 'A' .. 'Z' then
    TEXT_IO.Put (Item => "Try Again");
  else
    TEXT_IO.Put (Item => "Welcome Aboard, ");
    TEXT_IO.Put (Item =>  Next_Item);
  end if;
end Print_Message_3a;
```

5. Compare the following program to the one given in section 2.2.3 with main procedure `Print_Message_3`. Is the output of each of these programs the same? Explain.

```
with TEXT_IO;
procedure Print_Message_3b is
  Choice : CHARACTER;
--
begin
  while Choice not in 'A' .. 'Z' loop
    -- Better Look at Next CHARACTER
    TEXT_IO.Get (Item => Choice);
  end loop;
  if Choice in 'A' .. 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  end if;
end Print_Message_3b;
```

6. Compare the following program to the one given in section 2.2.3 with main procedure `Print_Message_3`. Is the output of each of these programs the same? Explain.

```
with TEXT_IO;
procedure Print_Message_3c is
  Choice : CHARACTER;
--
begin
  Choice := 'a';
  while Choice not in 'a' .. 'z' loop
    -- Better Look at Next CHARACTER
    TEXT_IO.Get (Item => Choice);
  end loop;
  if Choice in 'a' .. 'z' then
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  end if;
end Print_Message_3c;
```

2.3 The Formal Grammar of Ada

In Chapter 1, we introduced the concept of formal grammars to represent languages. Formal grammars can be used to uniquely and concisely define legal strings in any given language. Ada is just a programming language for which a grammar has been specified. In this section we will examine the Ada grammar to see how programs may be constructed.

A computer program in the programming language Ada is a string of lines. Each line is a character string that can be divided into space characters, identifiers, reserved words, literals, and delimiters. Identifiers are names that start with letters followed by zero or more additional letters and/or digits, possibly divided into subnames by underlines, '_' . Reserved words are 63 identifiers used exclusively for standard purposes. There are three kinds of literals, namely, numeric literals, character literals, and string literals. Delimiters are either one or two special characters. There are sixteen one character delimiters and ten two character delimiters. The delimiters and their names are listed in Table 2.2 and Table 2.3.

At higher levels are program parts called declarations and statements, which in turn will be strings of lines made up of the space characters, identifiers, reserved words, literals, and delimiters. Declarations define passive properties such as types of data to be used in the program, while statements define active program behavior that creates and alters data. At even higher levels, procedures, functions, and packages will contain strings of declarations and statements.

More specifically, in this text

a) Words of lower case letters, some containing embedded underlines, are used to denote syntactic categories, for example:

   statement
   simple_statement
   compound_statement

b) Boldface words are used to denote reserved words, for example:

   procedure
   with

c) Square brackets enclose optional items. For example, an `if_statement` may or may not contain an `else_part`:

```
if_statement ::= if condition then
                   sequence_of_statements
                 [else
                   sequence_of_statements]
                 end if;
```

d) Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right, for example:

```
sequence_of_statements ::= statement {statement}
identifier_list        ::= identifier {, identifier}
```

   In this case a `sequence_of_statements` must contain one or more statements. An `identifier_list` contains one or more identifiers separated by commas.

e) A vertical bar separates alternative terms unless it occurs immediately after an opening brace, in which case it stands for itself, for example:

```
letter_or_digit       ::= letter | digit
component_association ::= [choice {| choice} =>] expression
```

   The | in the syntax production for `letter_or_digit` is a metasymbol meaning or, but the | in the production for `component_association` is a literal Ada symbol that would appear in an Ada program.

f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example `type_name` and `task_name` are both equivalent to `name` alone, but convey some additional semantic information about what kind of name is needed.

g) The syntax rules describing structured constructs are presented in a form that corresponds to the recommended line structure and paragraphing. For example, a `loop_statement` with a **while** condition is defined as

```
loop_statement ::= while condition loop
                      sequence_of_statements
                   end loop;
```

Preferred places for other line breaks are after semicolons. More specific information regarding line structure and paragraphing (style) will be provided in the Ada Programming Style Guide that will be distributed in the laboratory.

2.3.1 Lexical Units

As illustrated above, Ada programs can be thought of as lines of character strings, including space characters used to separate basic substrings within the lines. Such basic substrings are called lexical units, and are made up of six basic types, namely

- identifiers
- numeric literals
- character literals
- string literals
- delimiters
- comments

Strings in the first five types can contain no unmeaningful space characters, but the comments type may contain space characters anywhere. Any lexical unit may be of any length that will fit on a single line.

2.3.2 Identifiers

As already seen above, identifiers start with a letter which may be followed by any number of letters or digits, possibly separated by underlines. For example,

```
Print_Message_2,
TEXT_IO,
Choice,
procedure,
begin,
New_Line,
with
```

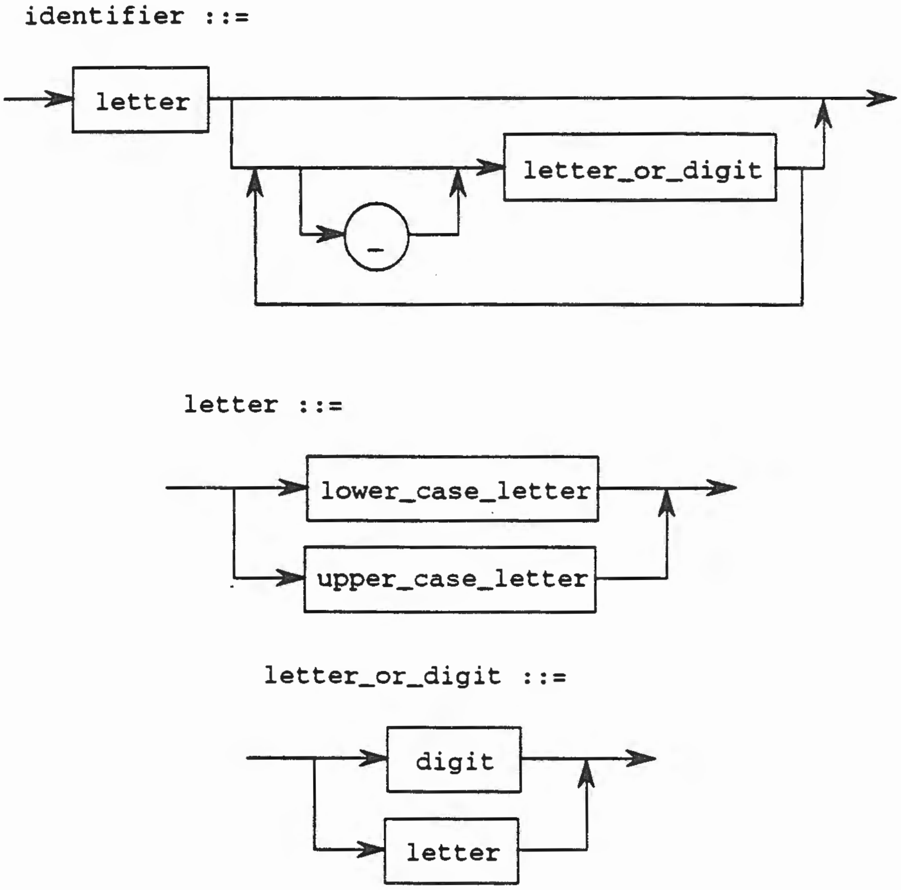
are all identifiers.

The formal syntax for an identifier in Ada is given in the three syntax productions shown next with four terminal words, namely `underline`, `upper_case_letter`, `lower_case_letter`, and `digit`.

```
identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit
```

Identifier
Syntax Definition 2.1

This syntax can also be shown in graphic form as follows.



Identifier
Syntax Chart 2.1

The fonts and upper or lower case of the letters in identifiers are ignored in Ada, but the underlines are not ignored. For example,

```
Print_Message_2, PRINT_MESSAGE_2, print_message_2,
```

are all the same identifier in Ada, but

```
PrintMessage2, PrintMessage_2, Print_Message_2
```

are different and distinct identifiers. As noted above, identifiers must fit on a single line, but no other limitation is imposed on their length.

As discussed in the last section, some of these identifiers are *reserved words*. For example, **procedure, begin,** and **with** are reserved words that are used in the programs above. Ada has 63 such reserved words, shown in Table 2.1.

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array			pragma	then
at	else		private	type
	elsif	limited	procedure	
	end	loop		
	entry		raise	use
begin	exception		range	
body	exit	mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

Table 2.1
Ada Reserved Words

2.3.3 Numeric Literals

Numeric literals can be integer literals and real literals. Real literals contain a decimal point while integers do not. We will not discuss real literals until later so that we can concentrate on integers. As in the case of identifiers, underlines may be used for readability, but unlike the case of identifiers, underlines do not change numeric values. For example,

```
1_234_567, 1_23_45_67, 1234567
```

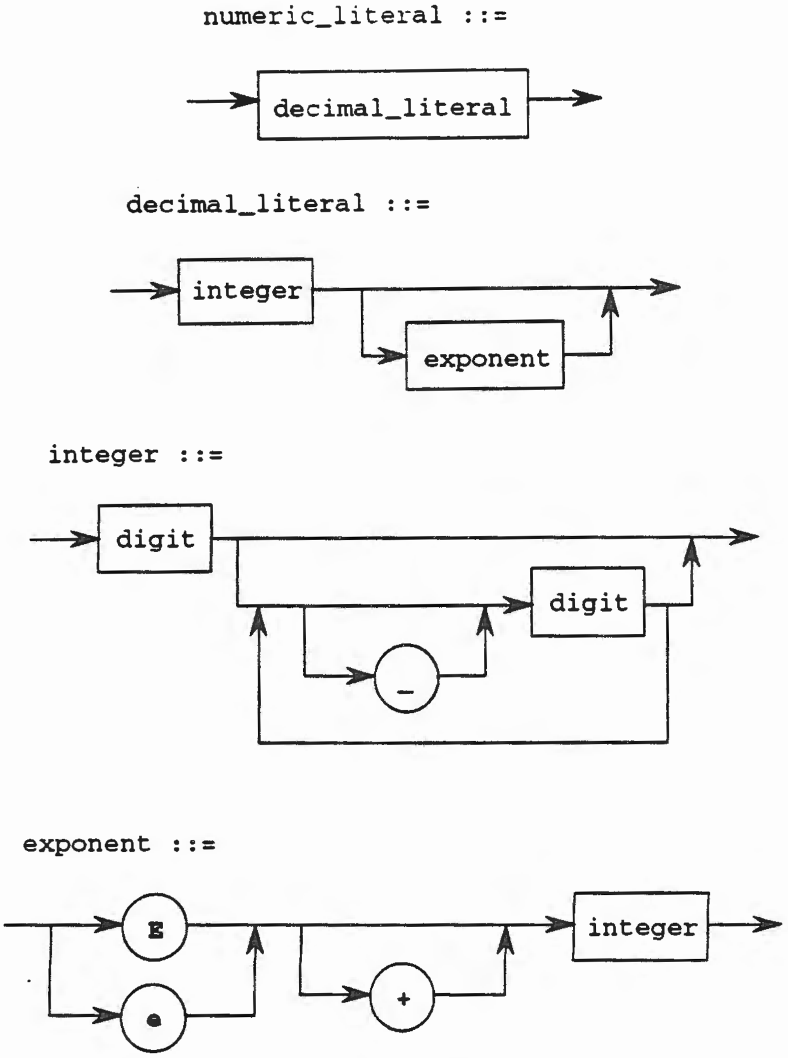
are integers that all have the same value. As with identifiers, numeric literals must be contained in a single line.

The syntax for integer literals in Ada includes numbers, possibly expressed in exponents. In the four syntax productions shown next rules for integers and real numbers are expressed with two terminal words, namely `underline`, and `digit`.

```
numeric_literal ::= decimal_literal

decimal_literal ::= integer [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | e [+] integer
```

Numeric Literal
Syntax Definition 2.2

This syntax can also be shown in graphic form as follows.



Numeric Literal
Syntax Chart 2.2

Some examples follow.

```
   Integer literals: 12,  0,  2E3 (which is 2 x 10³= 2000), 1_234_567
```

Leading zeros are allowed. No space is allowed in a numeric literal, not even in exponents. Exponents in integer literals must be zero or positive integers.

2.3.4 Character Literals

Character literals are the 95 ASCII characters enclosed between single quotes. These ASCII characters include the 26 upper and 26 lower case letters, the 10 digits, the space character, a set called "special characters"

```
" # ' ( ) * + , - . / : ; < = > _ | &
```

and a set called "other special characters"

```
! $ ? @ [ \ ] ` { } ~ ^ %
```

For example,

```
'A', ' ', '''
```

are characters A, space, and single quote, respectively.

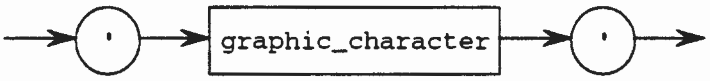
The syntax for character literals in Ada is shown next in three syntax productions with six terminal words, namely `upper_case_letter`, `lower_case_letter`, `digit`, `other_special_character`, `special_character`, and `space_character`.

```
character_literal ::= 'graphic_character'

graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
  upper_case_letter | digit
  | special_character | space_character
```

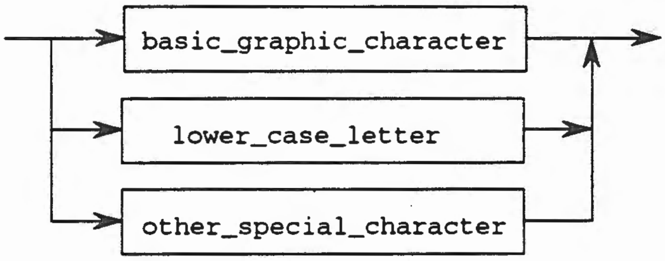
Character Literal
Syntax Definition 2.3

This syntax can also be shown in graphic form as follows.

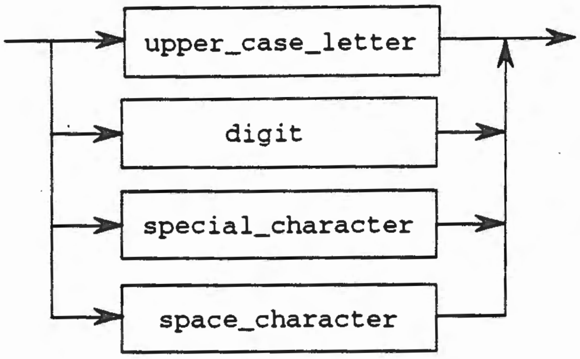
character_literal ::=



graphic_character ::=



```
basic_graphic_character ::=
```



Character Literal
Syntax Chart 2.3

2.3.5 String Literals

String literals are lists of zero or more characters enclosed within double quotes. For example,

```
"", "Welcome Aboard", """Welcome Aboard"""
```

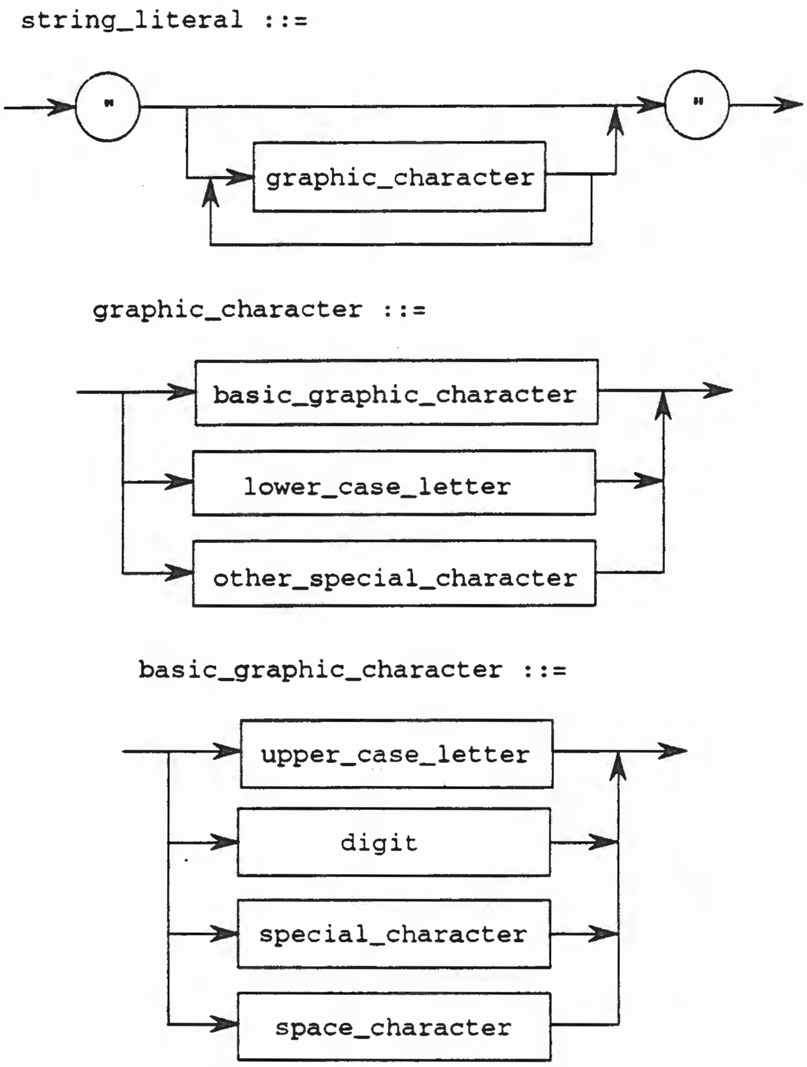
are strings, the first being a **null** (empty) string, the second containing the words `Welcome Aboard`, the third being the quoted words `"Welcome Aboard"`. Note the difference between the single quote as a character literal ' and the double quote as a string literal of length 1 but written as two double quotes `""`. Character literals are always single characters between single quotes, so a single quote is just one of the possible characters. For example the double quote is another possible character literal, written as `'"'`, using only one character. But string literals contain zero or more characters and a double quote in a string will end the string unless it is followed directly by another double quote.

The syntax for string literals in Ada is shown next in three syntax productions with six terminal words, namely `upper_case_letter`, `lower_case_letter`, `digit`, `other_special_character`, `special_character`, and `space_character` (the same terminal words as in the productions for character literals).

```
string_literal ::= "{graphic_character}"

graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
  upper_case_letter | digit
  | special_character | space_character
```

String Literal
Syntax Definition 2.4

This syntax can also be shown in graphic form as follows.



String Literal
Syntax Chart 2.4

2.3.6 Delimiters

Delimiters are composed of one or two special characters. They are used to convey special meanings for the program. They include the one character delimiters listed in Table 2.2 and the two character delimiters listed in Table 2.3.

Symbol	Name
'	apostrophe
(	left parenthesis
)	right parenthesis
*	star, multiply
+	plus
,	comma
-	hyphen, minus
.	dot, period, point
/	slash, divide
:	colon
;	semicolon
<	less than
=	equal
>	greater than
|	vertical bar
&	ampersand

Table 2.2
One Character Delimiters

Symbol	Name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: "becomes")
/=	inequality (pronounced: "not equal")
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

Table 2.3
Two Character Delimiters

The meanings of these delimiters depend on their context. Delimiters appear in many syntax productions, but are all terminal words, so no productions are shown for them.

2.3.7 Comments

Comments, as seen in programs in the last section, begin with double hyphens, --, and continue to the end of the line. An entire line can be devoted to a comment, which can be just an empty line to separate parts of a program, or a comment may be used to follow program text on the same line. If multi-line commentary is required, each line must begin with the double hyphens.

Comments should be liberally used in programs to convey information to the reader of the program. It is sometimes quite difficult to examine a complex section of code and determine why any particular statement is present. Comments can help to explain the reasons for certain design decisions or implementation decisions and thus, make the job of the reader significantly easier.

In this text, there are numerous examples of programs or program fragments which do not follow this guideline. Comments are sparse or non-existent. This is done deliberately because it is important for you to learn to read and understand programs without documentation. If the comments are present, then reading the program will be all that much easier. If the comments are not present, you will have learned to read and understand programs so as to fill in the comments. One of the goals of this text is to force you to do the analysis for each program, so to achieve that goal comments have been deliberately removed from some of the program examples.

2.3.8 Syntax for Ada Lexical Units

In summary, the syntax for identifiers, numerical literals, character literals, and string literals can be assembled into a single set of eleven syntax productions, eliminating the duplicated productions of the lexical units listed separately. This syntax is given in Table 2.4, Syntax for Ada Lexical Units.

```
identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit

numeric_literal ::= decimal_literal

decimal_literal ::= integer [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | e [+] integer

character_literal ::= 'graphic_character'

graphic_character ::= basic_graphic_character |
  lower_case_letter | other_special_character

basic_graphic_character ::=
  upper_case_letter | digit |
  special_character | space_character

string_literal ::= "{graphic_character}"
```

Table 2.4
Syntax for Ada Lexical Units

2.3.9 Exercises

1. a. Is the | in the following string in the grammar for the Ada programming language being used a metasymbol or a literal? Explain your answer.

```
exception_handler ::= when exception_choice
          {| exception_choice} =>
          sequence_of_statements
```

   b. What is the purpose of the italicized part of the following syntax statement?

      exception_choice ::= *exception*_name | others

2. Create a grammar that shows the relationship between the program parts procedures, functions, and packages, and declarations and statements, and space characters, identifiers, reserved words, literals and delimiters, as defined in this section.

3. a. Which of the following are identifiers?

```
TEXT_IO, TEXT_IO.Open_Files, Open_Files, A1, 1A
```

   b. Which of the following are numeric literals?

```
3.14159_26, 5e2, 5 e2, 5e+2, 5E0
```

   c. Which of the following are valid character literals?

```
'"', '', 'a', '=', '4'
```

   d. Which of the following are valid string literals?

```
"abc", "", """", "a", "1234", "'a'"
```

   e. Which of the following are valid comments in Ada?

```
i.    (* This is a comment *)
ii.   { Here is another comment }
iii.  -- This is a comment also
iv.   x := y + z;  -- This is an intermediate calculation
v.    A := B -- An integer -- + C -- Another integer -- ;
```

4. Which of the following are valid Ada reserved words?

```
begin, with, TEXT_IO, put, end, if
```

5. Use the syntax charts in Table 2.4 to determine if the following are valid lexical elements:

```
-312.34e+12, a_b_c, reserved_word_, .345, 2_54, 3,200.10
```

6. Decompose the following program into all of its lexical elements. List the lexical element and its name, *i.e.*, numeric literal, identifier, delimiter, *etc.*

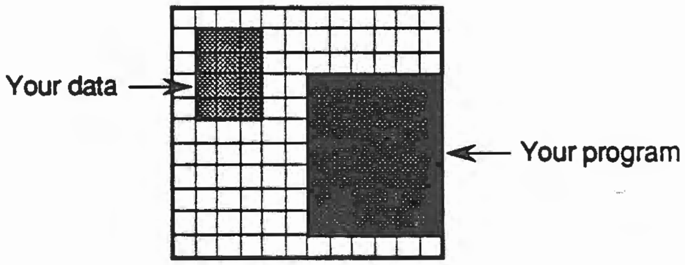
```
with TEXT_IO;
procedure Decompose_1 is
begin
  TEXT_IO.Put (Item => "Welcome Aboard");
end Decompose_1;
```

7. Decompose the following program into all of its lexical elements. List the lexical element and its name, *i.e.*, numeric literal, identifier, delimiter, *etc.*

```
with TEXT_IO;
procedure Decompose_2 is
  Choice : CHARACTER;
begin
    TEXT_IO.Get (Item => Choice);
    if Choice in 'A' .. 'Z' then
      TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
end Decompose;
```

2.4 Data Types and Objects

Data is stored and represented inside of the computer in different ways. The conceptual storage of data is described by a data type. An object of any given type represents an instance of the data type, and its storage will be handled as described by the type. The data to be stored may even be conceptualized further as an abstraction of some storage capability. In this case, it is called a data structure. For now, we will concentrate on data types and the objects declared from the data types. It is important to remember that although we will discuss data types in an abstract sense, they ultimately are nothing more than binary representations in memory. Your program and your data both are represented in the same manner, it is only the interpretation of the binary data that is unique. See Figure 2.3 for a graphic representation of memory and data and program storage.



Memory
Figure 2.3

2.4.1 Data Types

Data types identify several classes of data objects in Ada programs. Each data type defines a set of possible values for the type and the operations that can be applied to those values. Data types have attributes that describe properties of their values. For each data type defined, subtypes can also be defined as subsets of values with the same operations and attributes. At this time three kinds of data types and subtypes are introduced, namely

- Character Types and Subtypes
- Boolean Types and Subtypes
- Integer Types and Subtypes

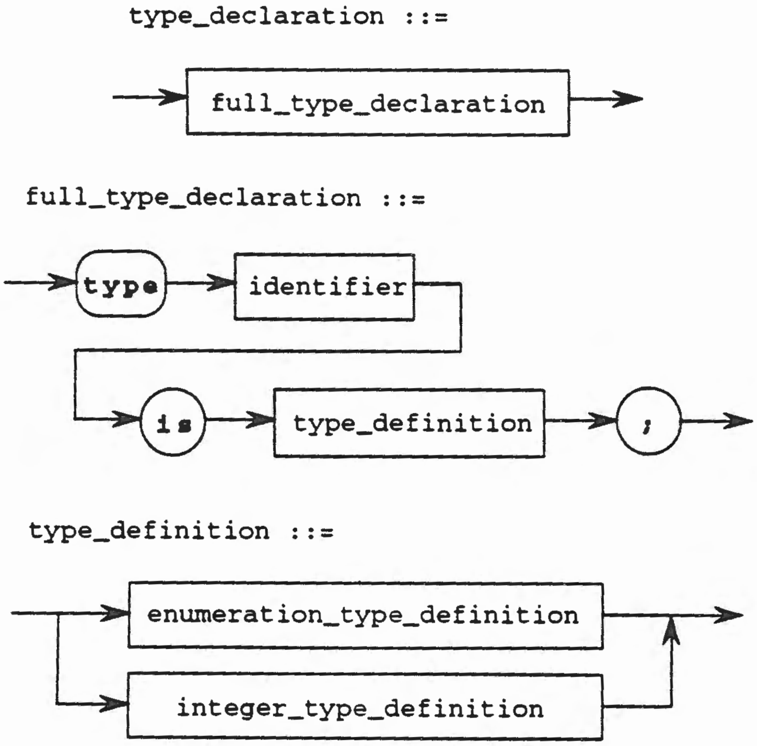
The formal syntax for type declarations in Ada is given in the three syntax productions, shown next with three intermediate words carried no further here, namely `identifier`, `enumeration_type_definition`, and `integer_type_definition`.

```
type_declaration ::=  full_type_declaration

full_type_declaration ::= type identifier is type_definition;

type_definition ::= enumeration_type_definition |
                    integer_type_definition
```

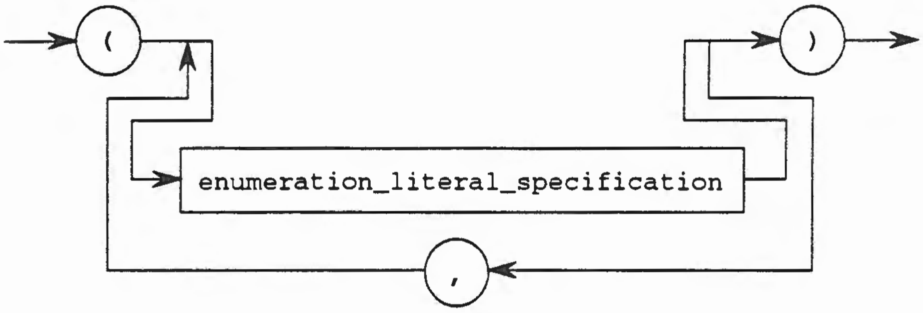
Type Declaration
Syntax Definition 2.5

Identifier has been defined previously; `enumeration_type_definition` and `integer_type_definition` will be defined below.

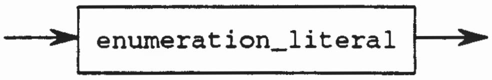
This syntax can also be shown in graphic form as follows:



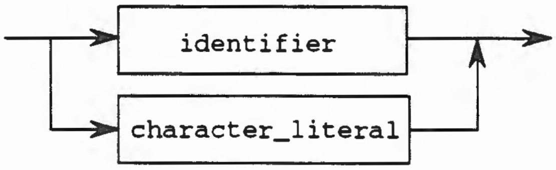
enumeration_type_definition ::=



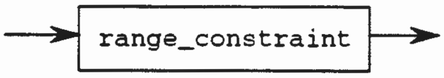
enumeration_literal_specification ::=



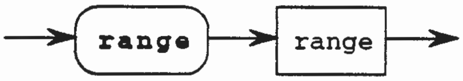
enumeration_literal ::=



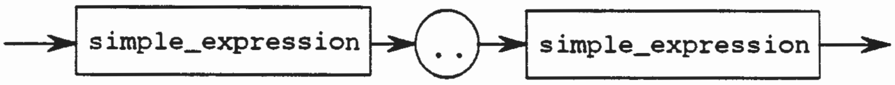
integer_type_definition ::=



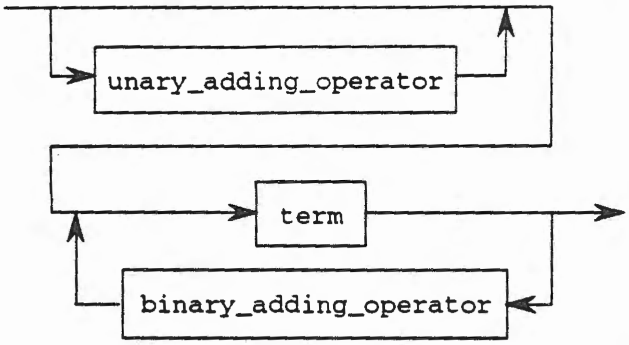
range_constraint ::=



range ::=



simple_expression ::=



Type Declaration
Syntax Chart 2.5

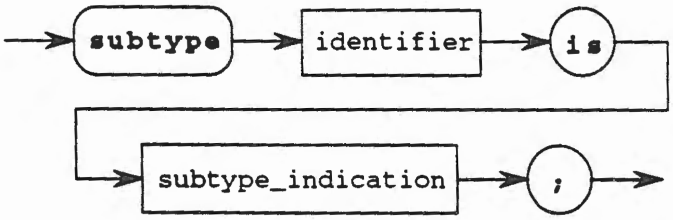
The formal syntax for a `subtype_declaration` is given next.

```
subtype_declaration ::= subtype identifier is
                        subtype_indication

subtype_indication ::= type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::= range_constraint
```

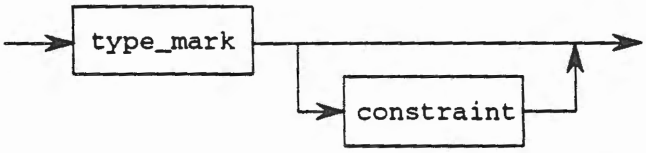
Subtype Declaration
Syntax Definition 2.6

This syntax can also be shown in graphic form as follows.

subtype_declaration ::=



subtype_indication ::=



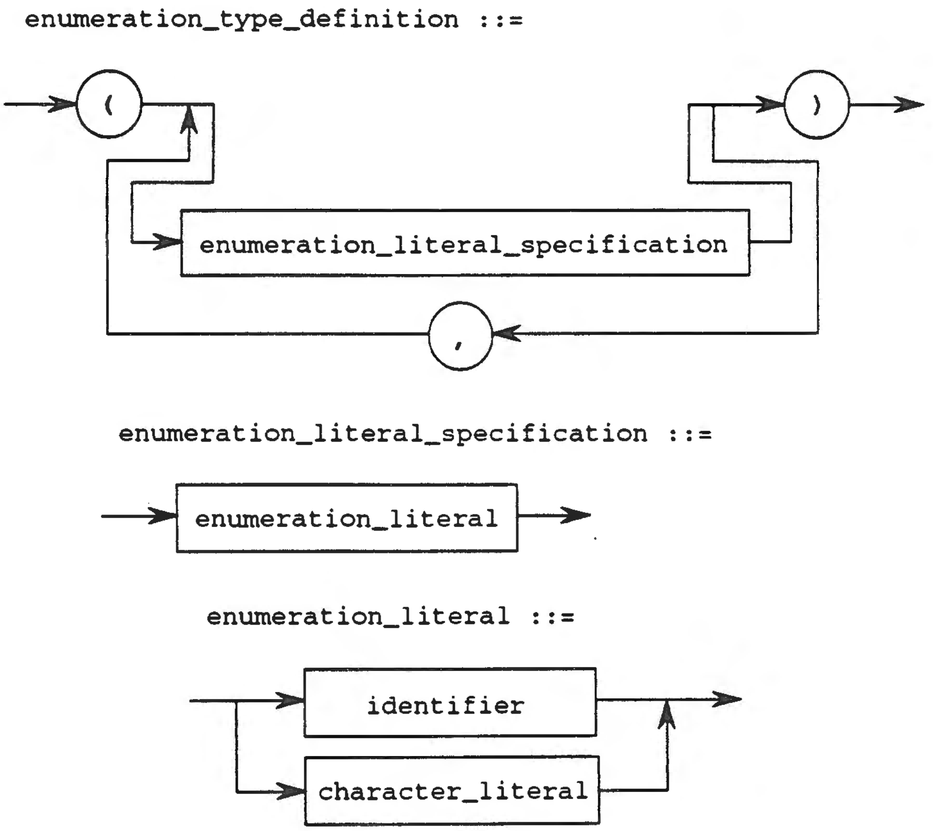
Character types of character literals can be defined in separate declarations using different character orderings. Character types are just forms of enumeration types, so the formal syntax for enumeration_type_definition is given next in three syntax productions with the terminal word character_literal.

```
enumeration_type_definition ::=
  (enumeration_literal_specification
    {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= character_literal | identifier
```

**Enumeration Type
Syntax Definition 2.7**

While this grammar states that an enumeration literal may be an identifier, in this section the discussion will be limited to the character_literal choice. Later the use of an identifier as a choice for enumeration_literal will be introduced and explained.

This syntax can also be shown in graphic form as follows.


enumeration_type_definition ::=
(
)
enumeration_literal_specification
,
enumeration_literal_specification ::=
enumeration_literal
enumeration_literal ::=
identifier
character_literal

**Enumeration Type
Syntax Chart 2.7**

Sample character type declarations follow. Note that semicolon (;) is part of the type declaration from the syntax above repeated here.

```
full_type_declaration ::= type identifier is type_definition;

type Decimal_Digit is ('0','1','2','3','4','5','6','7','8','9');

type Vowel is ('A', 'E', 'I', 'O', 'U', 'Y');
```

Next, sample subtype declarations follow based ultimately on type CHARACTER, with syntax repeated here.

```
subtype_declaration ::= subtype identifier is subtype_indication

subtype Upper_Case_Letter is CHARACTER range 'A' .. 'Z';

subtype Lower_Case_Letter is CHARACTER range 'a' .. 'z';

subtype Decimal_Digit is CHARACTER range '0' .. '9';

subtype Binary_Digit is Decimal_Digit range '0' ..'1';

subtype Octal_Digit is Decimal_Digit range '0' .. '7';
```

2.4.3 Boolean Types and Subtypes

BOOLEAN is a predefined Ada data type with values that are the two identifiers FALSE and TRUE, which are ordered in the relation FALSE < TRUE. A boolean subtype may be defined from the predefined type BOOLEAN. Boolean types are enumeration types, so the formal syntax for `enumeration_type_definition` as given in section 2.4.2 applies, where the choice for enumeration literal is chosen to be `identifier`, and the `identifier` is limited to the values FALSE and TRUE.

Sample BOOLEAN type declarations follow. Note that semicolon (;) is part of the type declaration from the syntax above repeated here.

```
full_type_declaration ::= type identifier is type_definition;

type Always_True is (TRUE);

type My_Boolean is (FALSE, TRUE);
```

Next, sample subtype declarations follow based ultimately on type BOOLEAN, with syntax repeated here.

```
subtype_declaration ::= subtype identifier is
  subtype_indication;

subtype_indication ::= type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::= range_constraint

range_constraint ::= range range
```

```
range ::= simple_expression .. simple_expression

simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}
```

Thus, a subtype of the predefined type BOOLEAN might be:

```
subtype Never_True is BOOLEAN range FALSE .. FALSE;
```

2.4.4 Integer Types and Subtypes

Ada views the integers as an unbounded, infinitely long set of the whole numbers, being infinite and unbounded in both the positive and negative directions on a number line. This data type is called universal integer. Clearly, this range of numbers is too large to be represented on any existing machine. Therefore, Ada allows a subset of this universal integer concept to be used on a specific machine. This subset is called the type INTEGER.

INTEGER is a predefined Ada data type with values from a range that is machine dependent, but which consists of consecutive integers, positive, zero, and negative. The number of positive integers will always be the same as the number of negative integers in the range, except that it is permissible to have an "extra" negative number. The user can declare a new integer type as a subtype of the predefined type INTEGER, defined by using the range reserved word. The formal syntax for an `integer type definition` is given next.

```
integer_type_definition :: = range_constraint
```

**Integer Type Definition
Syntax Definition 2.8**

The syntax is also shown in graphic form as follows.

`integer_type_definition ::=`

range_constraint

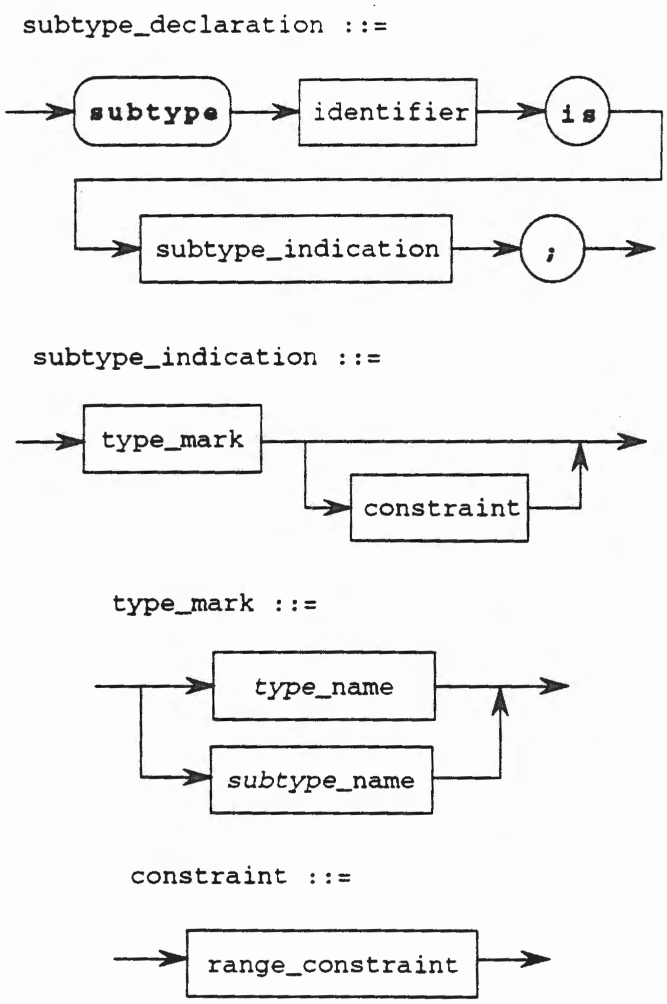
**Integer Type Definition
Syntax Chart 2.8**

The formal syntax for an `integer subtype declaration` is given next.

```
subtype_declaration ::= subtype identifier is
                        subtype_indication;

subtype_indication  ::= type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::= range_constraint
```

**Subtype Declaration
Syntax Definition 2.9**

This syntax can also be shown in graphic form as follows.



Subtype Declaration
Syntax Chart 2.9

Sample integer subtype declarations are

```
subtype Day_Of_Month is INTEGER range 1..31;

subtype INTEGER_6 is INTEGER range -1e6..1e6;

subtype Natural_6 is INTEGER range 0..999_999;

subtype Positive_3 is Natural_6 range 1..999;
```

Note again that the semicolon (;) is part of the declaration. In each case, the set of integers declared are all those between and including the two integer values following range, the smaller endpoint being listed first. Ada provides two predefined subtypes of the predefined type INTEGER, namely NATURAL and POSITIVE. All Ada implementations have these two subtypes defined in package Standard. Their definitions are provided below, but may also be found in Appendix F of the Language Reference Manual. Note that `INTEGER'LAST` merely means the largest representable value in the type INTEGER.

```
subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;

subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

2.4.5 Data Objects

Data objects are declared as identifiers that belong to given data types, either as variables or constants.

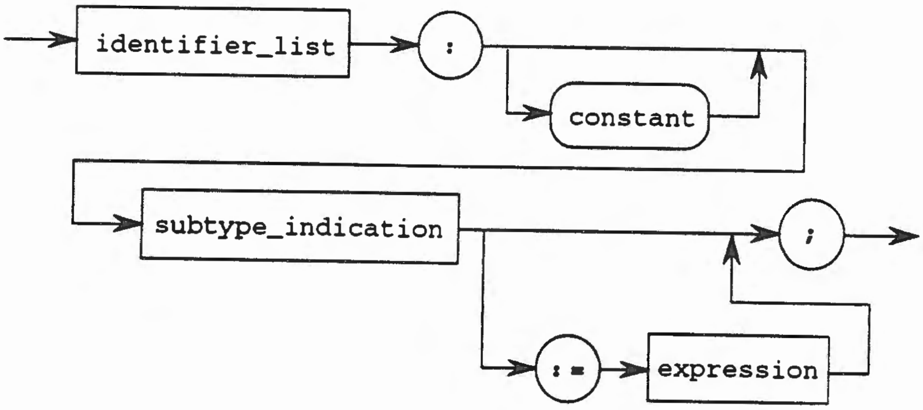
The formal syntax for an object in Ada is given in the two syntax productions shown next with two intermediate words subtype_indication, and identifier that were discussed above.

```
object_declaration ::= identifier_list : [constant]
             subtype_indication [:= expression];

identifier_list ::=  identifier {, identifier}

expression ::=  relation {and relation}
  | relation {or relation} | relation {xor relation}

relation ::=
  simple_expression [relational_operator simple_expression]
  | simple_expression [not] in range
  | simple_expression [not] in type_mark

relational_operator  ::=  = | /= | < | <= | > | >=
```

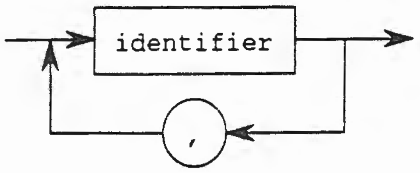
Object Declaration
Syntax Definition 2.10

This syntax can also be shown in graphic form as follows.

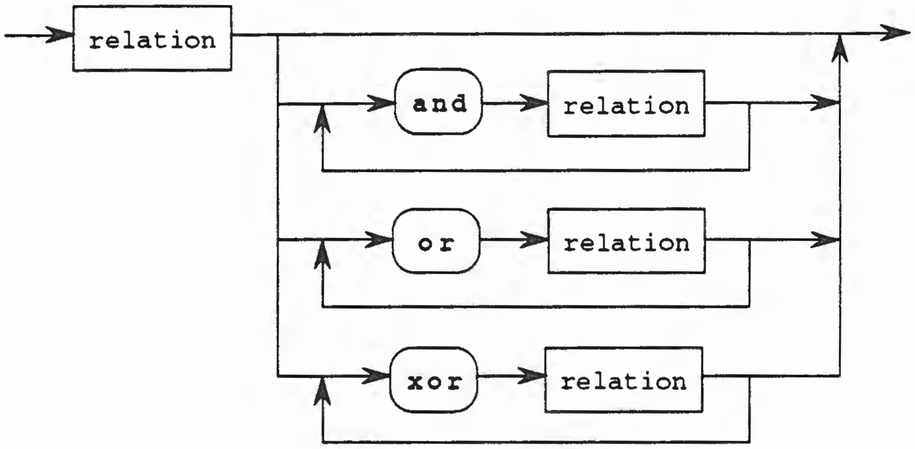
object_declaration ::=



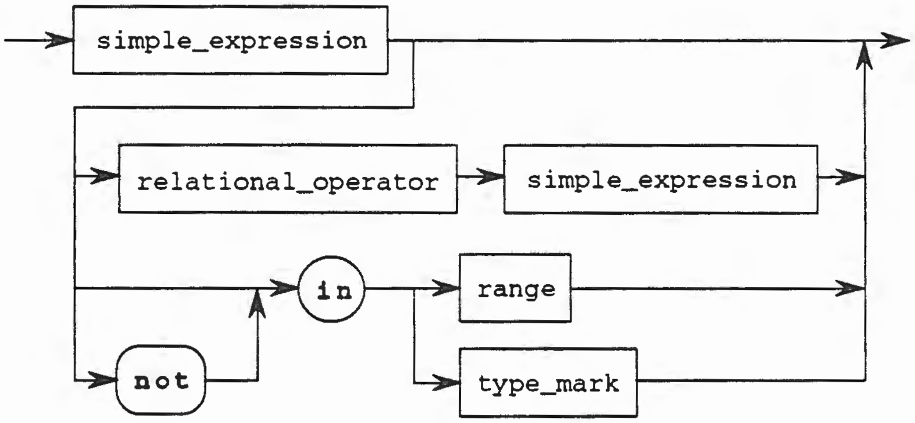
identifier_list ::=



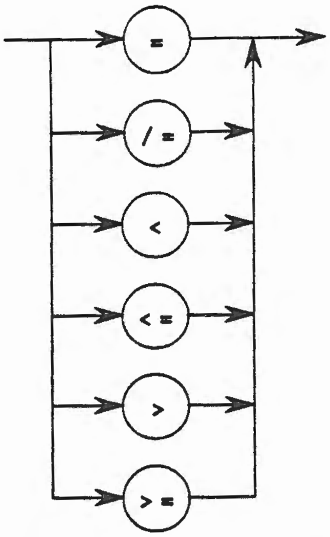
expression ::=



relation ::=



```
relational_operator ::=
```



Object Declaration
Syntax Chart 2.10

The objects are variables unless the optional term constant appears. As variables, the value stored in the location named by the identifier can be changed during the execution of the program. If the object is declared to be a constant, then its value may never change throughout the life of the program.

The operations possible among such declared identifiers depend on the types.

2.4.6 Character Type Objects

CHARACTER type objects must be declared before their references in the executable parts of programs. For example, in the program above with main procedure named `Print_Message_2`, character valued variable `Choice` was declared between the is and **begin** reserved words as

```
   Choice : CHARACTER;
```

which declares `Choice` to be an object that can hold CHARACTER values.

With character data types given above, other declarations possible include

```
   Digit : Decimal_Digit;

   Letter : Vowel;
```

which declare an object named `Digit` that can hold values of the type `Decimal_Digit`, and an object named `Letter` that can assume values of the type `Vowel`.

Operations

Operations for character types include the following.

Assignment	:=					
Membership	in	not in				
Relational	=	/=	<	<=	>	>=

If any initial value is not included in the character type or a result is not proper, a CONSTRAINT_ERROR exception will be raised in execution. This means that an attempt was made to assign a value to an object that it was not declared to be able to take on. For example, trying to assign the character 'Z' to an object that was declared to be of type `Vowel` would raise CONSTRAINT_ERROR because 'Z' is not a permissible value of the type `Vowel`.

Attributes

For any CHARACTER type, say `C`, attributes are automatically defined. They include the following:

FIRST the first value of type `C`, denoted `C'FIRST`
LAST the last value of type `C`, denoted `C'LAST`
PRED if value `X` is not `C'FIRST` then `C'PRED(X)` precedes `X`
else CONSTRAINT_ERROR raised
SUCC if value `X` is not `C'LAST` then `C'SUCC(X)` succeeds `X`
else CONSTRAINT_ERROR raised
POS the position of the character in the enumeration
VAL the value of the enumeration at the given position

For example, using the type `Vowel` defined above, the following equalities are true:

```
Vowel'FIRST = 'A'
Vowel'LAST  = 'Y'
Vowel'PRED('I') = 'E'
Vowel'SUCC('I') = 'O'
Vowel'VAL(3) = 'O'  --remember that the first literal is always zero
Vowel'POS('U') = 4
```

2.4.7 Boolean Type Objects

Boolean type objects must be declared before their references in the executable parts of programs. With boolean data types given above, other declarations possible include

```
I_Win   : Always_True;
You_Win : Never_True;
Rain_Today : My_Boolean := FALSE;
```

which declares an object named `I_Win` that can assume the value in `Always_True`, namely TRUE, and an object `You_Win` that has the value of the type `Never_True`, namely FALSE. Also, an object named `Rain_Today` has been declared which can assume values from the type `My_Boolean`, namely TRUE or FALSE. In this case, the object `Rain_Today` is provided with an initial value of FALSE.

Operations

Operations for boolean types include the following.

Assignment	:=					
Membership	in	not in				
Relational	=	/=	<	<=	>	>=

If any initial value is not included in the boolean type or a result is not proper a CONSTRAINT_ERROR exception will be raised in execution.

Attributes

A BOOLEAN type, say B, has attributes that include the following:

FIRST the first value of type B, denoted B'FIRST
LAST the last value of type B, denoted B'LAST
PRED if value X is not B'FIRST then B'PRED(X) precedes X
else CONSTRAINT_ERROR raised
SUCC if value X is not B'LAST then B'SUCC(X) succeeds X
else CONSTRAINT_ERROR raised
POS the position of the literal in the enumeration with the first one as zero
VAL the value of the enumeration at the given position

2.4.8 Integer Type Objects

INTEGER type objects must be declared before their references in the executable parts of programs. For example,

```
Date: Day_Of_Month := 1;

Population: Natural_6;
```

declare Date and Population as integers in different ranges. In this case, Date is initialized to 1 in the declaration.

Operations

Operations for integer types include the following.

Adding	+	-					
Assignment	:=						
Exponentiating	**						
Membership	in	not in					
Multiplying	*	/	rem				
Relational	=	/=	<	<=	>	>=	
Unary	+	-	abs				

If any initial value is not included in the integer type or a result is not proper a CONSTRAINT_ERROR exception will be raised in execution.

Attributes

For any INTEGER type, say I, attributes available to the user include the following:

FIRST	the first value of type I, denoted I'FIRST
LAST	the last value of type I, denoted I'LAST

PRED	if value X not I'FIRST then I'PRED(X) precedes X
	else CONSTRAINT_ERROR raised

SUCC	if value X not I'LAST then I'SUCC(X) succeeds X
	else CONSTRAINT_ERROR raised

2.4.9 Exercises

1. Declare a CHARACTER type called Possible_Grades that represents all of the capital letters from 'A' to 'F'.

2. Declare an INTEGER subtype called Weeks_In_Year with an appropriate range.

3. Declare a BOOLEAN subtype called Pass_Or_Fail with appropriate values.

4. Declare an object First_Week that can take on values of the type defined in Exercise 2 above.

5. Declare an object called My_Grade of the type defined in Exercise 3 and give it an initial value indicating that you passed.

6. Declare an object called Letter_Grade of the type defined in Exercise 1 and give it an initial value of E. Will this initialization be valid? Why or why not?

2.5 Ada Names and Expressions

2.5.1 Ada Names

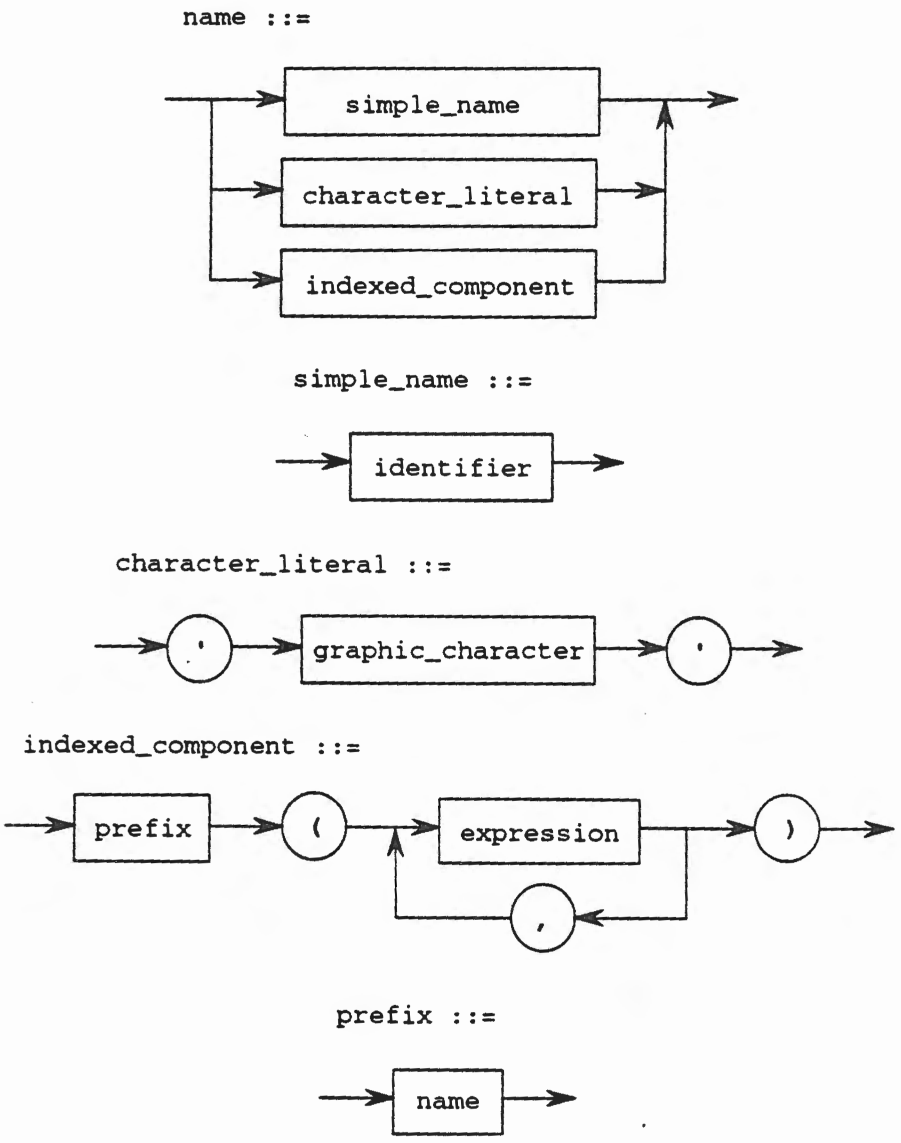
Ada names denote declared entities such as variables and constants. Names are either identifiers or literals, as given next in five syntax productions with identifier, graphic_character, and expression as terminal words since these have been previously provided.

```
name ::= simple_name  | character_literal  | indexed_component

simple_name ::= identifier

character_literal ::= 'graphic_character'
```

```
indexed_component ::= prefix (expression {, expression})

prefix ::= name
```

Name
Syntax Definition 2.11

This syntax can also be shown in graphic form as follows.



Name
Syntax Chart 2.11

For example, the following are all Ada names:

```
Heads_I_Win
'A'
'c'
```

These examples illustrate the use of an identifier and graphic characters as names. An example of an indexed component name will be provided later.

2.5.2 Ada Expressions

Given variables and constants, it is now possible to form expressions in building Ada programs. With one exception, expressions and their values must be formed within given types or closely related types, *i.e.*, if one variable is an INTEGER, then all of the other variables and constants must be INTEGER or subtypes of INTEGER. Further, operations must preserve the type such that if two INTEGER values are added, then the sum must be an INTEGER, and similarly for all other types. The one exception is that the relational operations in any type have boolean results.

2.5.3 Character Expressions

Aside from assignment statements and the previously mentioned attributes, the operations on character types are

Membership	in	not in				
Relational	=	/=	<	<=	>	>=

Membership and relational operations have boolean results, *i.e.*, the result of applying any of these operations to two or more CHARACTER values is either FALSE or TRUE.

Thus, the following CHARACTER expressions are valid and yield the results indicated,

```
A' in Vowel            TRUE    -- see section 2.4.2 for Vowel
z' in Decimal_Digit    FALSE   -- see same section
e' not in Vowel        TRUE    -- 'E' is, but 'e' isn't
X' < 'Z'               TRUE    -- since 'Z' comes after 'X'
A' > 'S'               FALSE   -- since 'S' comes after 'A'
A' /= 'a'              TRUE
```

2.5.4 Boolean Expressions

Aside from assignment statements and attributes, the operations on boolean types are

Membership	in	not in				
Relational	=	/=	<	<=	>	>=
Logical	and	or	xor			

Membership, relational, and logical operations have boolean results as discussed in Section 2.4.3.

The membership and relational operations have already been discussed for other types. The logical operators can only be applied to boolean expressions. These operators allow the programmer to connect boolean expressions to form more complex expressions. The meaning of the logical operators is given in Table 2.6, where A and B represent boolean expressions. In Chapter

1 you were already exposed to these concepts and the use of a truth table for predicates. This discussion is specific to Ada, but follows easily from the original generic discussion. The operators *and* and *or* have meanings that are probably intuitive from their English language equivalents. The meaning of xor, however, is likely to be new to you. For the logical expression containing xor to be TRUE, one or the other of the expressions on either side of the xor operator must be TRUE, but not both.

Another operation on BOOLEAN expressions is *not*. This operator inverts the logic of the expression to which it is applied, *e.g.*, if the argument to not is TRUE, then the result is FALSE.

A	B	A and B	A or B	A xor B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Table 2.6
Meaning of the Logical Operators

For example, the following are all valid boolean expressions with the results indicated:

```
TRUE in Never_True          FALSE  -- see section 2.4.3
FALSE not in Always_True    TRUE   -- see same section
FALSE < TRUE                TRUE   -- always!
TRUE /= TRUE                FALSE
TRUE or FALSE               TRUE   -- see Table 2.6
FALSE xor TRUE              TRUE
FALSE and FALSE             FALSE
not TRUE                    FALSE
```

2.5.5 Integer Expressions

Aside from assignment statements and attributes, the operations on integer types listed in the inverse order of their precedence (lowest to highest precedence) are:

Adding	+	-				
Exponentiating	**					
Membership	in	not in				
Multiplying	*	/	rem	mod		
Relational	=	/=	<	<=	>	>=
Unary	+	-	abs			

Membership and relational operations have boolean results, while the remainder of adding, exponentiating, multiplying, and unary operations have integer results. The adding and multiplying operators are the same as what is normal for integers. The multiplying operator / gives a result truncated toward zero. The operators *rem* and *mod* are probably new to you. The rem operator is used to obtain the remainder after division. For example, if we have the expression 5 rem 3, we observe that 5/3 is 1 with a remainder of 2. Thus, 5 rem 3 is 2. The mod operator works exactly like the rem operator when the operands are both positive, but produces different results when one or both of the operands is negative. For our purposes, we will always use the rem operator. The *abs* operator is also probably unfamiliar to you. It return the absolute value of its operand. Thus, abs -5 is 5, since the absolute value of the quantity -5 is five. Note that in Ada, abs is an operator and not a function call. Thus, it has an operand associated with it, not an argument or parameter. The meaning of this note will become more clear later in this course. These operations can be used and combined in more complex integer expressions.

For example, the following INTEGER expressions are all valid and yield the result indicated:

```
2 + 3                    5
5 in INTEGER             TRUE
33 not in Day_Of_Month   TRUE    -- see section 2.4.4
2 ** 3                   8       -- 23 = 8
7 / 2                    3       -- INTEGER has no fractional part
4 * 5                    20
abs -14                  14      -- absolute value
-6                       -6      -- unary minus
3 < 10                   TRUE
-4 > -2                  FALSE
abs -4 > abs -2          TRUE
7 rem 21                 0       -- the remainder after division
```

2.5.6 Precedence for Operations

Up until this point all of the expressions that have been illustrated contained only one operation (except for some abs expressions). This is the simple case. However, in many programs the operators are strung together to form more complex operations. When this happens, the order in which the operations are performed can affect the result obtained. For example, if variable `A1` contains the value 3, `B2` contains the value 4, and `C3` contains the value 5, then

```
D4 := A1 + B2 + C3;
```

trivially assigns 12 to `D4` since it makes no difference how the operands are grouped, that is,

```
D4 := A1 + (B2 + C3)  <==>  D4 := (A1 + B2) + C3
```

But consider the following expression,

```
D4 := A1 * B2 + C3
```

This expression can assign to `D4` the resultant value 17 (performing the multiplication first) or 27 (performing the addition first). That is,

```
D4 := (A1 * B2) + C3   π   D4 := A1 * (B2 + C3)
```

This situation is ambiguous and so rules must be provided that tell the user the order in which operations will be applied within expressions. This order is known as precedence. Table 2.7 shows the order in which operations are applied if operators of different precedence levels are mixed in an expression. Operators of the same precedence are always applied in a left to right order. Note that parentheses can override the highest precedence. This allows the programmer to force the expression to be evaluated in any order desired, overriding normal precedence rules. For example, in the expression

```
E5 := A1 * B2 + C3;
```

with the same values for `A1`, `B2`, and `C3` as before, Table 2.7 shows that the multiplication operator will be applied to `A1` and `B2`, then the addition operator will be applied to that product and `C3`. If the programmer wanted to force the expression to perform the addition first, this could be done as follows,

```
E5 := A1 * (B2 + C3);
```

where the subexpression inside of the parentheses is computed first, namely the addition of `B2` and `C3`, then this sum is used to multiply `A1`.

Highest Precedence	**	abs	not			
Multiplying	*	/	rem			
Unary	+	-				
Binary	+	-	&			
Relational	=	/=	<	<=	>	>=
Logical	and	or	xor			

Table 2.7
Precedence of Selected Operators

Two operators may never appear together so that the following expressions must be written as indicated,

```
4 + (-3)
5 - (-2)
```

Consider the following expression,

```
4 ** (13 - 2 * 5) < 100 and -5 > abs(7/2)
```

Examination of this complicated expression shows that there are two subexpressions inside of parentheses. These subexpressions are evaluated in a left to right order, so the leftmost subexpression is evaluated first. This expression consists of binary minus and multiplication operators. Table 2.7 ranks the multiplication operation higher than binary minus so that operation is performed first, using the operands 2 and 5, resulting in a value of 10. Next the binary minus is performed where the operands are this product and 13. The subtraction yields the result 3. Having determined the first subexpression's value, the second subexpression is evaluated. Here there is only a single operator so it is applied to the operands 7 and 2 to yield the result 3 (remember that these are integer values, so the result must be integer). The expression is now scanned to determine which operation to perform next. The highest precedence operator remaining is exponentiation (**) and abs. Since these are also at the same precedence level, they are performed in a left to right order. Thus, the exponentiation is performed on its operands, namely 4 and the result of the subexpression or 3. This operation yields the result 64. Next, the abs operation is performed on its operand (it only requires a single operand) yielding the result 3. At this point, the expression has been simplified to an equivalent one, namely

```
64 < 100 and -5 > 3
```

The next operation to be performed is the unary minus which takes a single operand, 5 in this case, and yields the value negative 5, or -5. The next precedence level is the two relational operators < and >. These are at the same precedence level so they are also performed in a left to right order, so that 64 < 100 is evaluated to TRUE and then -5 > 3 is evaluated and yields the result FALSE. Finally, there is only a single operator remaining since the expression has been evaluated down to the following,

```
TRUE and FALSE
```

The logical operator and is then applied and yields the final value of the whole expression, namely FALSE.

This is the manner in which you should evaluate complicated expressions. Follow the steps used in the evaluation of this expression which are:

1. Evaluate any parenthesized subexpressions first.
2. Perform the highest remaining operation next, doing so in a left to right order if there are operators of the same precedence level in the expression

2.5.7 Syntax for Ada Expressions

The syntax for Ada expressions is given next in terms of terminal words.

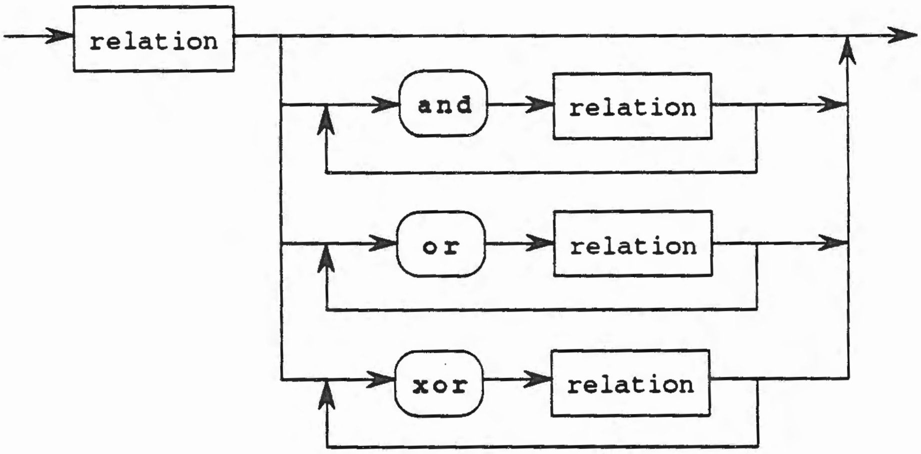
```
expression ::=  relation {and relation}
  | relation {or relation} | relation {xor relation}

relation ::=
  simple_expression [relational_operator simple_expression]
  | simple_expression [not] in range
  | simple_expression [not] in type_mark

simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}

relational_operator  ::=  = | /= | < | <= | > | >=

range ::=  simple_expression .. simple_expression

type_mark ::= type_name | subtype_name

unary_adding_operator  ::=  + | -

term ::= factor {multiplying_operator factor}

binary_adding_operator  ::=  + | - | &

name ::= simple_name  | character_literal

factor ::= primary | abs primary | not primary

multiplying_operator  ::=  * | / | mod | rem

simple_name ::= identifier

character_literal ::= 'graphic_character'
```

```
primary ::= null | string_literal | name | numeric_literal
            | (expression)

numeric_literal ::= decimal_literal

decimal_literal ::= integer [exponent]

exponent ::= E [+] integer | e [+] integer

identifier ::=  letter {[underline] letter_or_digit}

graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

string_literal ::= "{graphic_character}"

letter_or_digit ::= letter | digit

basic_graphic_character ::=
  upper_case_letter | digit
  | special_character | space_character

actual_parameter_part ::=
  (parameter_association {, parameter_association})

parameter_association ::=
  [formal_parameter =>] actual_parameter

formal_parameter ::= parameter_simple_name

actual_parameter ::=
  expression | variable_name | type_mark (variable_name)
```

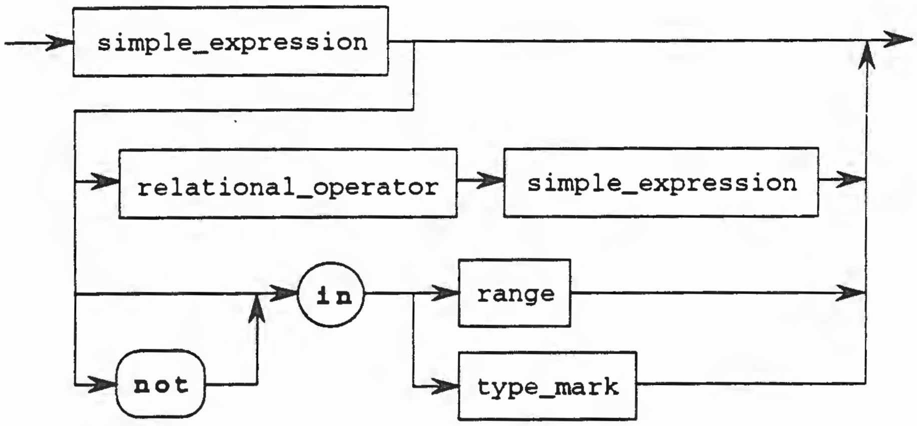
Ada Expressions
Syntax Definition 2.12

This syntax can also be shown in graphic form as follows.

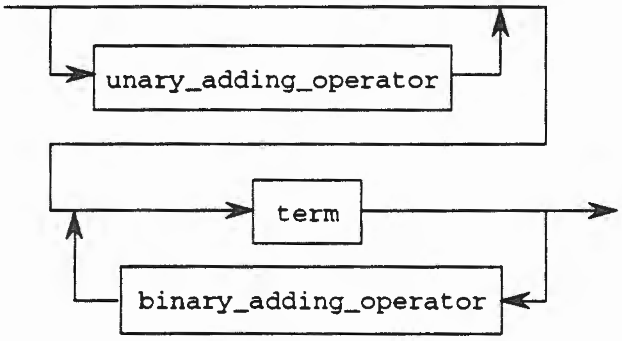
expression ::=



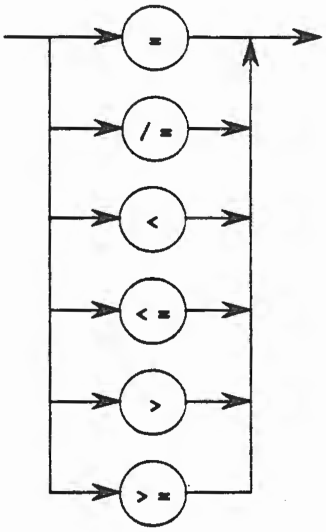
relation ::=



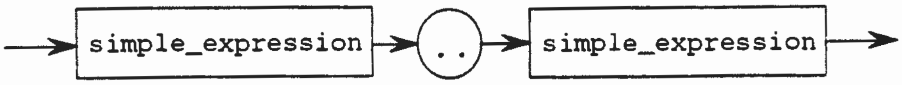
simple_expression ::=



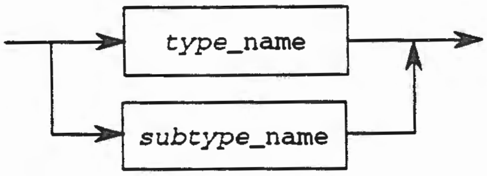
relational_operator ::=



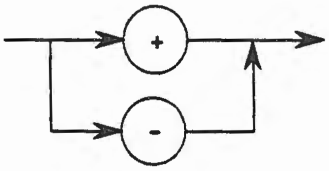
range ::=



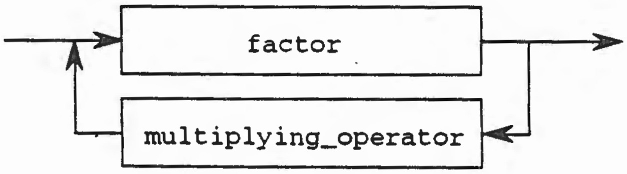
type_mark ::=



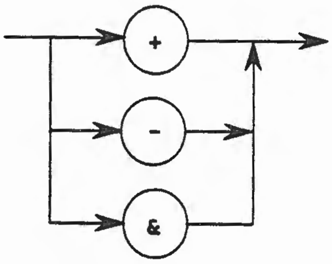
unary_adding_operator ::=



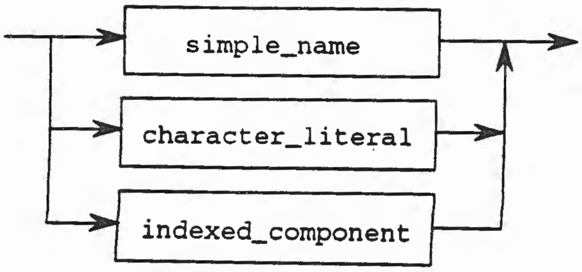
term ::=



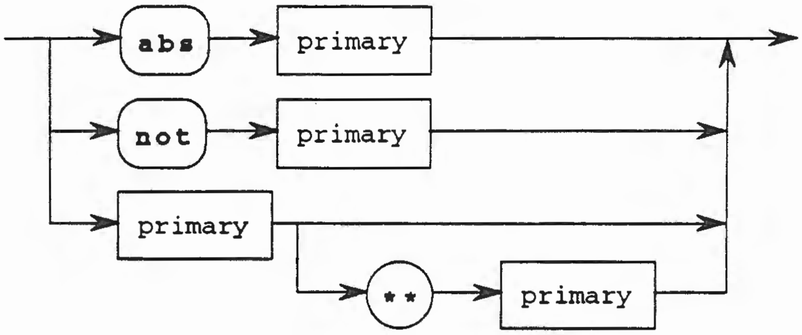
binary_adding_operator ::=



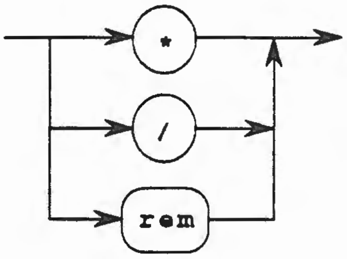
name ::=



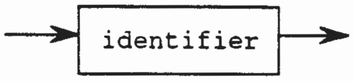
factor ::=



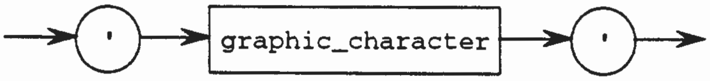
multiplying_operator ::=



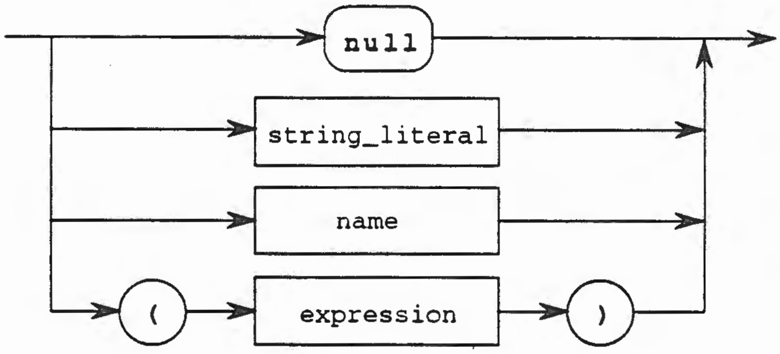
simple_name ::=



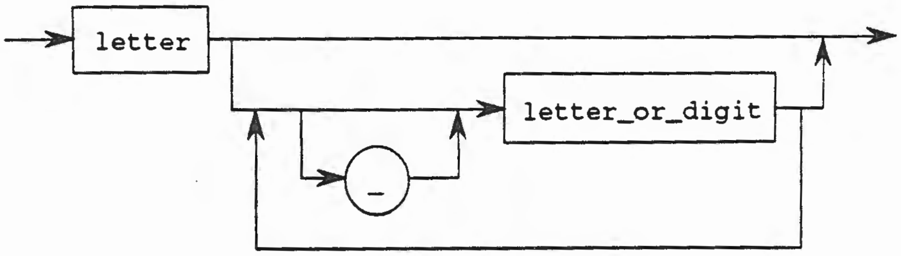
character_literal ::=



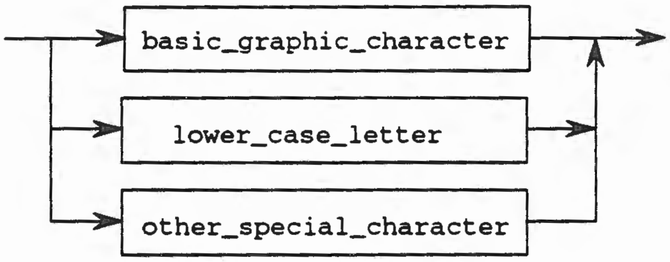
primary ::=



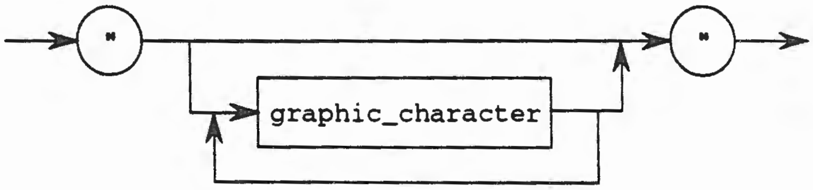
identifier ::=



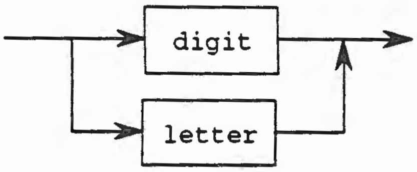
graphic_character ::=



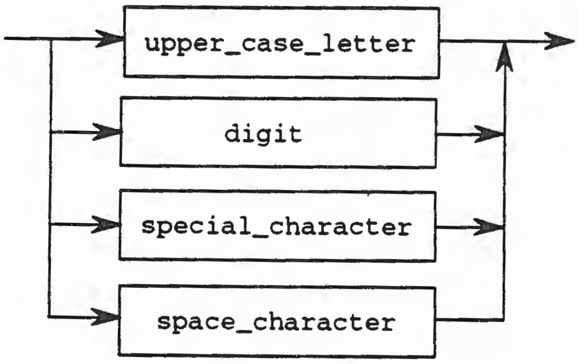
string_literal ::=

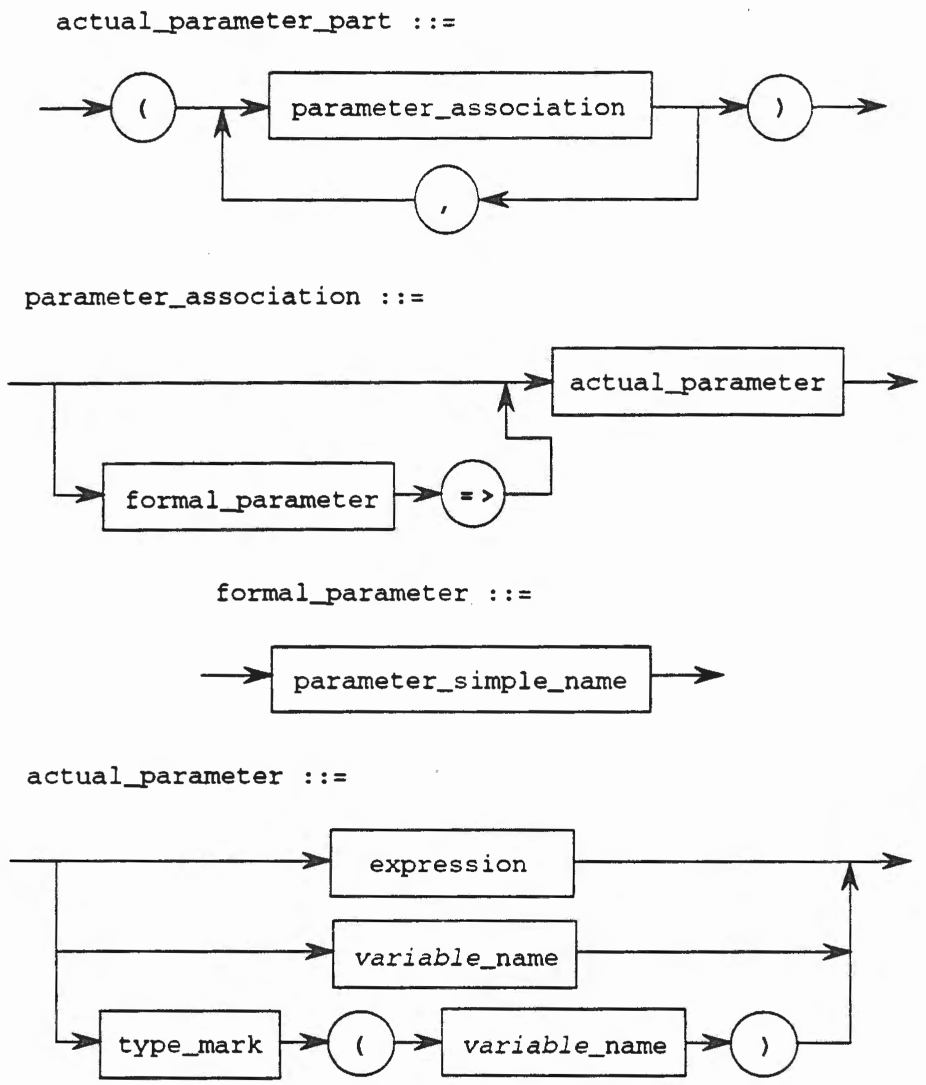


letter_or_digit ::=



basic_graphic_character ::=





Ada Expressions
Syntax Chart 2.12

2.5.8 Example Syntax Derivation of an Expression

In illustration of what we have been discussing, let's examine a top-down derivation of an expression to show how you might use the grammar of an Ada expression to determine if an expression is legal or not. Consider the following simple expression:

```
X + 3 > 10 and Y < 4
```

We can determine if this is a legal expression by first examining what an expression must be composed of as given by the grammar. We see that an `expression` is defined to be the following:

```
expression ::=  relation {and relation}
  | relation {or relation} | relation {xor relation}
```

We can choose the first production which says that an expression is a relation, followed by the reserved word and, followed by another relation. So our candidate expression matches this pattern if we let X + 3 > 10 be one relation and Y < 4 be another relation. Thus, we can conclude that this is a legal expression if we can show that these two parts are in fact relations.

A relation is defined to be as follows:

```
relation ::=
  simple_expression [relational_operator simple_expression]
  | simple_expression [not] in range
  | simple_expression [not] in type_mark
```

Now for X + 3 > 10, we determine the first production again matches our candidate expression if X + 3 is a simple_expression, and 10 is also a simple_expression, and further, that > is a relational operator. First we will establish that the relational operators are defined as:

```
relational_operator  ::=  = | /= | < | <= | > | >=
```

and we have confirmed that > is a legal relational operator. Next we examine the productions for a simple_expression given below:

```
simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}
```

and we find that X + 3 will fit this pattern if X is a term, 3 is a term, and + is a binary_adding_operator. Examining the binary_adding_operator syntax, we see the following production rule:

```
binary_adding_operator  ::=  + | - | &
```

and can confirm that + is a legal value for a binary_adding_operator. Now if we can show that X and 3 are both terms, then we will have confirmed that this portion of our original expression is well-formed and legal. Examining the syntax for a term we see:

```
term ::= factor {multiplying_operator factor}
```

and we can only confirm it by examining the syntax for a factor which follows:

```
factor ::= primary | abs primary | not primary
```

and this leads us to examine the syntax for a primary, given below:

```
primary ::= null | string_literal | name | numeric_literal
            | (expression)
```

Now we can establish that X is a term if we can show that it is a name and we can show that 3 is a term if we can show that it is a numeric_literal. The syntax for a numeric_literal follows:

```
numeric_literal ::= decimal_literal
```

and we know that we must show that 3 is a decimal_literal. This syntax rule is given next:

```
decimal_literal ::= integer [exponent]
```

and we now must show that 3 is an integer. This production is given below:

```
integer ::= digit { [underline] digit}
```

and finally, the following production for a digit:

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

allows us to confirm that indeed, 3 is a digit, implying that it is a derivable from the non-terminal term.

For X, we must show that it is also a term which we can do if we can show that it is a name. The syntax for a name is given below:

```
name ::= simple_name  | character_literal
```

and we can see that we now must show that X can be a simple_name. This production is given by the following:

```
simple_name ::= identifier
```

and now we must show that X is an identifier. This production rule is:

```
identifier ::=  letter {[underline] letter_or_digit}
```

The syntax for a letter is:

```
letter ::= lower_case_letter | upper_case_letter
```

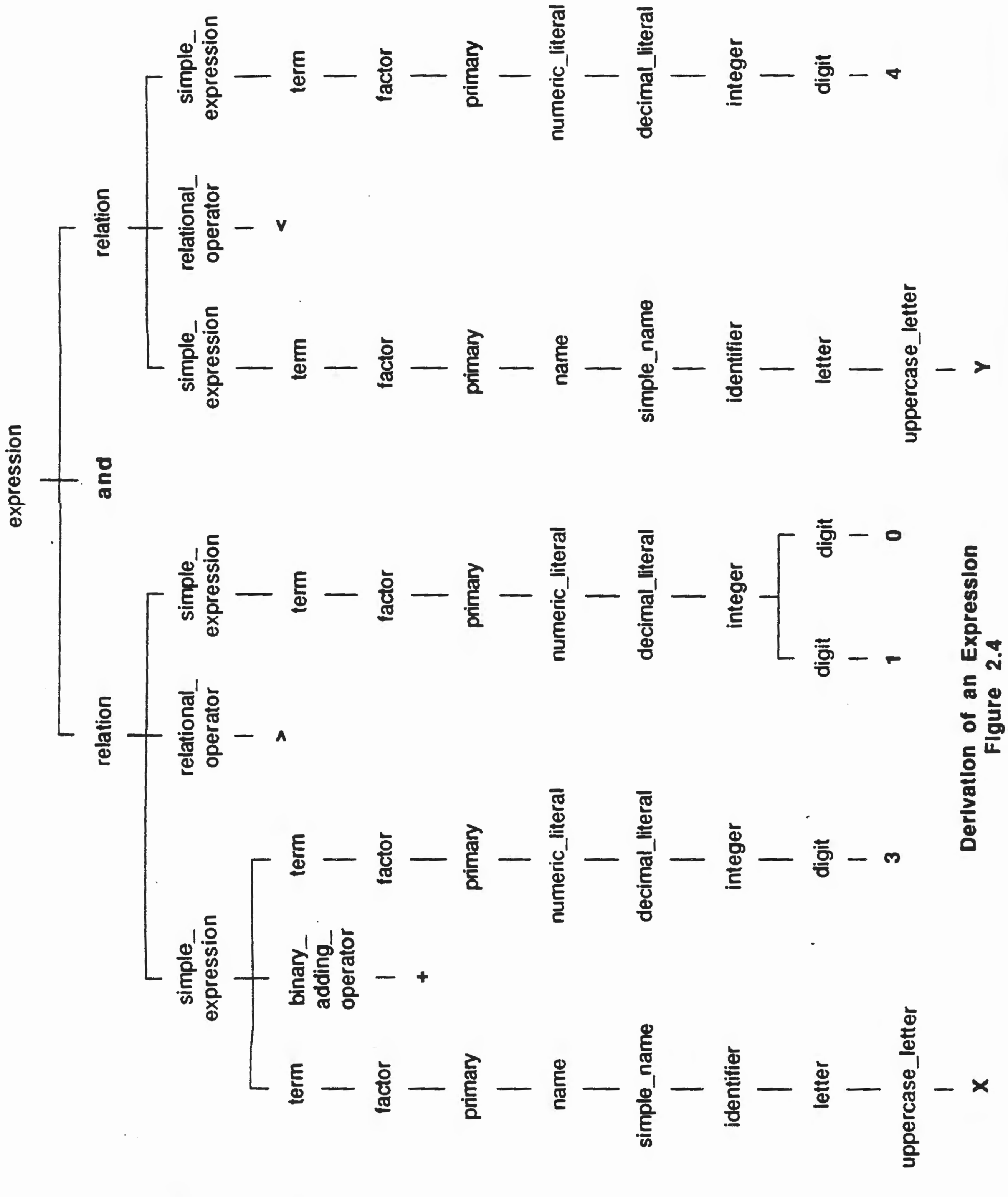
and an upper_case_letter is:

```
upper_case_letter ::= A | B | C | D | E | F | G | H | I | J | K | L |
                      M | N | O | P | Q | R | S | T | U | V | W | X |
                      Y | Z
```

and from this we can determine that indeed, X is an upper_case_letter and thus derivable from the non-terminal term. We have now shown that we can derive X + 3 as a simple_expression and a similar analysis will confirm that we can also derive 10 as a simple_expression. We can then conclude that X + 3 > 10 is derivable from the production rules as a relation. A similar analysis of Y < 4 will show that it also a relation and thus the expression

```
X + 3 > 10 and Y < 4
```

is indeed a valid expression in Ada. A pictorial representation of the complete derivation of the expression is shown in Figure 2.4.



Derivation of an Expression
Figure 2.4

Note that although this process is tedious, it is necessary in order to succinctly define what are legal strings in the language and what are not. Fortunately for us, the compiler does these checks for us automatically.

2.5.9 Exercises

1. Determine if the following are valid CHARACTER expressions. If so, then give the result; if not, explain why not.

   a. `'c' in CHARACTER`
   b. `"A" in CHARACTER`
   c. `'a' < 'A'`
   d. `'X' >= 'Y'`

2. Determine if the following are valid BOOLEAN expressions. If so, then give the result; if not, explain why not.

   a. `TRUE xor FALSE`
   b. `FALSE > TRUE`
   c. `TRUE in FALSE`
   d. `TRUE and FALSE or TRUE`

3. Determine if the following are valid INTEGER expressions. If so, then give the result; if not, explain why not.

   a. `3 + 2 and 7`
   b. `3 ** -2`
   c. `4 * (-9/4)`
   d. `7 * 3 / 10 + 2`

4. Compute the value of the following expressions.

   a. `TRUE and not FALSE`
   b. `not TRUE or not FALSE`
   c. `TRUE and FALSE or not FALSE and TRUE`
   d. `TRUE < FALSE and not FALSE > TRUE`

5. Compute the value of the following expressions.

   a. `'A' <= 'Z' and 'b' < 'v'`
   b. `'X' < 'Z' or 'Y' >= 'X' and FALSE`
   c. `'S' <= 'S' and 'S' > 'S'`
   d. `'H' < 'h' or 'h' < 'H'`

6. Compute the value of the following expressions.

   a. `2 < 10 * (-1) and 6 >= 2 * 3`
   b. `abs(-7) - 5 * 7 ** 2 / 3`
   c. `9 rem 4 * 3 / 7 + 4 ** 7 / 3`
   d. `12 rem 3 - (6 + 7 - (2 * 3 + (4 / 3)))`

7. Determine the value of the following expression showing the order in which each operator is applied. Include each new version of the simplified expression in your answer, *i.e.,* after each operator is applied, rewrite the expression substituting the result for the subexpression just computed. Show the result of each operation.

```
32 rem 3 - (22 + 7 - (19 / 3) * (7 ** abs(-4 * 2)))
```

2.6 Simple Ada Statements

2.6.1 An Overview of Statements

The declarations and statements that a programmer puts in a subprogram jointly determine its elaboration of declarations and execution of statements. The elaboration of declarations is that process by which each declaration achieves its desired effect. As shown above, variables can be initialized with values on declaration. The `sequence_of_statements` in the `begin...end` block of the body will be executed in sequence. Statements within the sequence may define branching or looping execution of other statements in their domain before returning to the main sequence, which continues until the final statement completes execution. For purposes of analysis, the entire sequence of statements finally executed needs to be understood, and their cumulative effects on the data of the program identified.

Ada statements of two types will be introduced, namely

- simple statements
- assignment
- null
- input/output

which will be described in this section, and

- compound statements
- sequence of statements
- if statements
- if then else
- loop statements
- while loop
- for loop

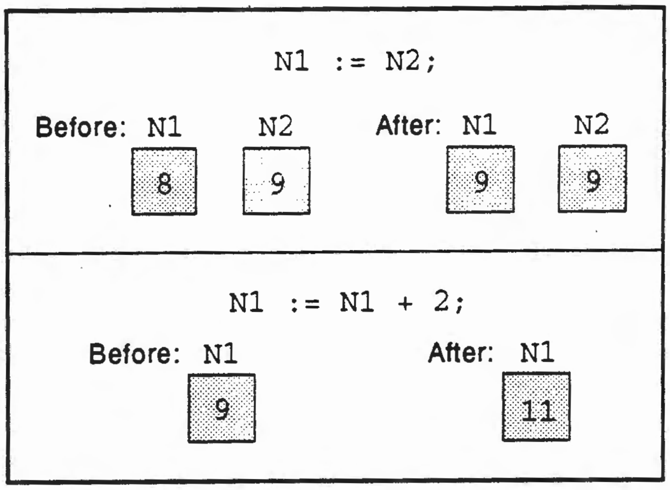
which will be described in the next section.

2.6.2 Assignment Statements

Assignment statements assign values to variables, in the form

```
   variable := expression; -- variable may be any object
```

The value of the expression on the right hand side of the assignment operator (`:=`) is determined and this value is then placed into the memory location whose name is given by variable on the left hand side of the assignment operator, provided that the types of the expression result and the variable are the same. (See Figure 2.5 for a graphical depiction of this process.)



```
N1 := N2;
```

```
N1 := N1 + 2;
```

The Assignment Operator
Figure 2.5

Thus, given the following assignment,

```
   My_Grade := 'A';
```

the object `My_Grade` will contain the value 'A' after the execution of this statement, as long as `My_Grade` has been previously declared to be of type CHARACTER. Similarly, the following assignment statement,

```
   Total_Value := 3 * 2 + 7;
```

will compute the value of the expression, namely 13, and then assign this value to `Total_Value` provided that `Total_Value` has been previously declared to be of type INTEGER.

Note that the assignment statement is not an equality test like X = 3, but is instead an assignment. In ordinary mathematics, a statement such as X = X + 1 would be nonsense since a quantity can never be equal to itself plus one. In a programming language such as Ada, the statement X := X + 1; is quite different. This means take the value currently stored in the location whose name is X, add one to it, and store this new result back in the same storage location, namely the one whose name is X. There is no confusion in this statement and it is perfectly legitimate to write such a statement.

2.6.3 Null Statement

A null statement consists of the reserved word **null** followed by a semicolon (;). This statement may seem a bit strange at first, because it causes the computer to do nothing! Why would a programmer want to tell the computer to explicitly do nothing? The answer is that there are several places in the Ada syntax where a statement is required, but where the programmer may not want anything to be done. In order to tell the computer that no operation is needed, a special statement is required, namely the **null** statement. An analogy is the page in a book which is labeled "This page intentionally left blank." What is the purpose of such a page? Simply to tell the reader of the book that even though there is no writing on this page, it was not a production error or a printing error. The author (or publisher) actually wanted this page to contain no text. For a similar reason, Ada provides the **null** statement. The **null** statement is written as follows,

```
null;
```

and is not very interesting out of context. Remembering the syntax of an **if** statement from earlier,

```
if X1 < Y2 then
  null;
else
  D3 := X1 + 2;
end if;
```

the use of a **null** statement can be illustrated. In this case, if X1 < Y2, then the programmer does not want anything to be done, otherwise the assignment is made to D3. This example is not very useful because the **if** statement could be rewritten as follows,

```
if not (X1 < Y2) then
  D3 := X1 + 2;
end if;
```

avoiding the use of the **null** statement. However, the example does show a use for the **null** statement and others will be illustrated later in the text.

2.6.4 Input/Output Statements

Input and Output statements are as varied as the data that we wish to input or output. For that reason, Ada treats each data type somewhat differently when it comes to input and output. We have already seen how some of the input/output statements work, such as for characters and strings. We will examine some of the different input/output statements in this section.

2.6.4.1 Integer Input/Output

Previously we had introduced the topic of integer numeric values. An integer is a whole counting number that ranges from negative infinity to positive infinity. Unfortunately, a machine cannot be made to represent this range of values. Thus, we have for each system a predefined INTEGER data type that represents the range of values permissible (representable) for the machine in use. We also discussed subranges of the predefined numeric type INTEGER such as POSITIVE (`range 1..INTEGER'LAST`) and NATURAL (`range 0..INTEGER'LAST`). In addition, we mentioned that the user is free to define new integer types and subtypes to best fit the application.

What has been missing until this point is the ability to use these types for input or output. CHARACTERS and STRINGS can be read from the keyboard and written to the screen without any necessary action from the software engineer other than to use the Put and Get procedures available in TEXT_IO. Unfortunately, this will not work for numeric values. The reasons for this have to do with the wide range of possible values, given the user-defined types that Ada allows. Accordingly, it was deemed prudent to limit the input/output operations to those specifically tailored for the data type that we desire to get or put.

Ada provides a very elegant capability for input/output of integer types. This allows a user to read or write integer types directly. However, for reasons that will be apparent later, this capability will not be fully discussed at this time. In future chapters, the capability to input/output integer types will be fully explained. However, for now a template will be provided to allow you to GET and PUT integer values for the integer types that you declare.

In the package TEXT_IO that has already been partially described for you, there is a nested package called INTEGER_IO. This package is actually not a "real" package at all. Instead, it is a template for creating specific instances of packages that will have the ability to read and write the enumerated literals that you declare. The details of how this is accomplished by the compiler are beyond the scope of our discussions here, other than to say that the process is not too terribly complex and will be explained to you in detail later in this course. For now, we are only concerned that you understand the global concepts of what is happening when an instance of this template is created and that you know how to create an instance for your use.

The global concept is that this INTEGER_IO package is a template for other packages much as your driver's license started out as a template. When you went to the Department of Motor Vehicles the driver's license form was blank and not useful. If you tried to drive using it and was stopped you would surely be told that this was an invalid license. What gave your license its value was when that template had the details filled in to describe you and to denote that it was now valid for use as a license. In the same manner, when INTEGER_IO is in TEXT_IO it is like a blank license form, without any value of its own. However, when we "fill it in" we produce a useful package that we can use for our purposes, namely, input and output of integer literals.

How do we "fill in" the template? The answer is that we must only supply it the name of the integer type that we desire to have available for input/output operations. For the integer type Day_Of_Month defined earlier, this is accomplished in the following manner,

```
   package Day_Of_Month_IO is new TEXT_IO.INTEGER_IO (Day_Of_Month);
```

This provides us with a new package, called Day_Of_Month_IO, that we can use like any other "normal" package, but this package's purpose is to provide us with the ability to perform input/output operations on the type Day_Of_Month. Again, for our purposes now, it is not imperative that you fully understand all of the hidden details of what is occurring here, only that you understand how to use this feature for your own Input/Output operations of your own integer types.

As another example, to be able to read and write the literals for the type Natural we would first need to have a line in our program such as,

```
   package Natural_IO is new TEXT_IO.INTEGER_IO (Natural);
```

which provides us a new package called Natural_IO that will give us the capability of reading and writing Natural literals directly.

Finally, if we had a data type such as,

```
   type Speed is range 0 .. 125;
```

then we could make put and get operations available for objects of this type by use of the following generic instantiation,

```
   package Speed_IO is new TEXT_IO.INTEGER_IO (Speed);
```

In this example, the name Speed_IO is any identifier that you choose. The rest of the statement is exactly as you will always declare it, except for the fact that you will substitute the name of the type that you want to get and put in the parenthesis where Speed appears in our example.

These new packages are used like any other package. It is impossible to detect that these packages, created by this mechanism, are any different than any other package that we might create by simply writing out the specification and body ourselves. This mechanism does have a name. We call it a *generic instantiation* because the package template, in our example INTEGER_IO, is called a *generic* package in Ada, and the process of creating an instance of the template is called *instantiation*. Thus, Natural_IO is a generic instantiation of the generic package TEXT_IO.INTEGER_IO.

How do we use a generic instantiation? In the same manner as any other package. Thus, to read an integer literal of the type Day_Of_Month, given the generic instantiation Day_Of_Month_IO, we would use

```
Day_Of_Month_IO.Get (Item => My_Day);
                                   -- assuming that we had previously
                                   -- declared My_Day : Day_Of_Month;
```

and similarly, we could write this literal as follows,

```
Day_Of_Month_IO.Put (Item => My_Day);
```

It should be pointed out that whereas we have chosen to use what we consider meaningful names to describe our input/output packages, there is no requirement to do so. Thus, to have a generic instantiation of INTEGER_IO for the type Day_Of_Month we could have written,

```
package Go_Ada is new TEXT_IO.INTEGER_IO (Day_Of_Month);
```

which would have produced the same generic instantiation, but with the name Go_Ada instead of Day_Of_Month_IO. This is one more example of a situation where you should pause to consider carefully the identifier name that you give to items that you create. It is very useful for maintaining your software to have descriptive, meaningful names.

2.6.4.2 Width Parameter for Integer Input/Output

There are a few differences, however, between the procedure Put defined for CHARACTERs and STRINGs, and the procedure Put defined for integer types. A Put procedure for integer types has two additional parameters, namely WIDTH and BASE. Normally, you do not need to concern yourself with them, because they have defaults that will serve your purposes in most cases. However, occasionally it is desirable to have more control over the format of your output. In this case, the WIDTH parameter is very useful. We will discuss it in this section. The BASE parameter is not as useful normally and we will defer a discussion of its purpose until later.

The WIDTH parameter allows the user to define the number of columns that will be used to display the number. For example, if we Put the integer 6 in a field width of 8 printing positions, we will see seven spaces and then the number 6. If we were to Put the value 128 in the same field width, we would see 5 spaces and then the number. Note that the number if always right justified, *i.e.*, always placed on the rightmost part of the field width specified.

Suppose that we were to specify a field width of 4 and then Put the value 128? We would have a single space before the number was printed. How do we tell the system that we want to change the value of the field width from the default that is used when we do not specify a field width? We just include the parameter WIDTH as one of the parameters to the Put statement. Thus, we would use the following statement to Put the value 128 in a field width of 5,

```
    Int_IO.Put (Item => 128, Width => 5);
```

Similarly, we would use the following Put statement to print the value 2,745 in a field width of 12,

```
    Int_IO.Put (Item => 2_745, Width => 12);
```

and we would see 8 leading spaces, followed by the digits 2, 7, 4 and 5. The underline in this number, 2_745, is allowed in numeric values in Ada for readability, as discussed in previously. It does not get stored internally and it is not part of the output. It does make the program more readable and assist in maintenance activities and so should be used in your programs.

We have used named notation for our parameters to introduce them to you. This is not actually necessary. We could have written each of the previous statements, respectively, as,

```
    Int_IO.Put (Item => 128,5);
    Int_IO.Put (Item => 2_745,12);
```

where the first parameter will be the value of the item to be printed and the second will always be the field width.

One additional wrinkle must be explained about this new capability. What happens if we want to output the value 27_381 but we specify a field width of 3? There are a number of possibilities. What would you think is the most reasonable? Output only the first 3 digits? The last three? Something else? The answer is that we will always get the value output that we desire to print, even if that means overriding, or ignoring, the field width specified in the Put statement. For example, given the following Put statement,

```
    Int_IO.Put (Item => 27_192, Width => 2);
```

the output would be 27_192, even though the field width is specified to be only two places. We say that the output field width is expanded to be exactly what is required (no extra spaces) when the field width specified is less than that needed to out put the value, otherwise, leading spaces are used to pad the field width and provide the value in the rightmost portion of the field specified.

In the case of a negative number, the minus sign (-) is treated as a printing position of the number just the same as the digits that make up the number. Thus, a Put statement of the form,

```
    Int_IO.Put (Item => -27_193, 7);
```

would cause the output to have a single leading space followed by the 6 printing positions that make up the number, *i.e.*, -, 2, 7, 1, 9, and 3. As before, the underline will not be part of the output.

One final item to note is that a field width that is specified to be zero will provide the output in exactly the number of spaces that are needed to write out the value, with the field width being expanded as described above. This is a very useful property that you should keep in your mental toolkit of little tricks of the trade. It makes it easy to have output of the form provided in the following statements,

```
TEXT_IO.Put (Item => "There are ");
Int_IO.Put  (Item => Number_of_Items, Width => 0);
TEXT_IO.Put (Item => " items in the toolkit.");
```

Here we can assume that there is some integer value in the object `Number_of_Items`, say 10. Then the output will look like,

```
There are 10 items in the toolkit.
```

where the spacing around the number 10 is achieved by leaving a space after the word are in the first Put statement and before the word items in the third Put statement. The field width of zero causes the exact number of spaces needed to write the 10 to be used without any leading spaces. Without this little trick, the code segment would be as follows,

```
TEXT_IO.Put (Item => "There are ");
Int_IO.Put  (Item => Number_of_Items);
TEXT_IO.Put (Item => " items in the toolkit.");
```

and the output would be as follows, assuming that the default field width is eight,

```
There are       10 items in the toolkit.
```

where the gap between the word are and the value 10 is caused by the leading spaces in the default field width. This illustrates the usefulness of having this feature in your repertoire.

By default, the system will normally use a field width of 8. You should check for yourself what size field width is used in your system.

From now on, it should be easy for you to define input/output operations on all of your integer numeric types.

2.6.4.3 Predefined Input/Output Procedures

The operations defined in the input/output packages that we have been using are generally procedures. A procedure is a unit of code statements that we use repeatedly. When we want to make use of a procedure, such as Put, we must invoke it, or call it. This causes the actions defined in the procedure to be performed. A procedure call statement consists of a procedure name, possibly with parameters, followed by a semicolon (;).

Several examples have already been seen of procedure call statements, such as

```
TEXT_IO.New_Line;
```

which is a call to the procedure `New_Line` in the package TEXT_IO. This is an example of a call without any parameters. It causes the output device to reposition itself to the next line so that any subsequent output will appear on a new line.

A procedure call with parameters consists of the procedure name followed by the parameters enclosed in parentheses, where each parameter is separated by commas. For example,

```
   TEXT_IO.Put (Item => "Welcome Aboard");
```

is a procedure call to the procedure Put in package TEXT_IO. It contains a single parameter whose formal name is Item and whose actual value is "Welcome Aboard". The action taken is to cause the string given as the parameter to be written to the output.

Another handy procedure to use is the procedure Put_Line, also found in the package TEXT_IO. This procedure causes the string provided as a parameter to be written to the output and then causes the output device to reposition itself to the next line. In effect, this single procedure is identical to calling the Put procedure and then immediately calling the New_Line procedure. For example, the procedure call

```
   TEXT_IO.Put_Line (Item => "Ada is fun.");
```

will produce the same result as the following sequence of statements:

```
   TEXT_IO.Put (Item => "Ada is fun.");
   TEXT_IO.New_Line;
```

From now on it will be useful to remember the Put_Line procedure and use it when it is appropriate.

2.7 Compound Statements

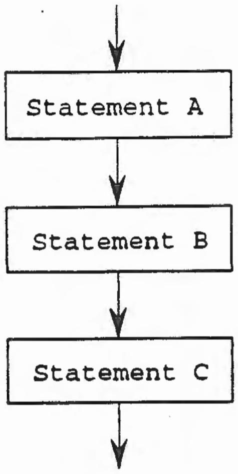
Compound statements are statements composed of potentially several other statements. They may have simple statements or other compound statements nested within them. This section will examine some of the compound statements that you are likely to see in Ada.

2.7.1 Sequences of Statements

Sequences of statements are executed in order. They consist of groups of other statements that are executed sequentially. For example,

```
   D4 := C3 + E5;
   F6 := 5 - D4;
   Draw_Rectangle (Side_1 => D4, Side_2 => F6);
```

is a sequence of simple statements that is executed in order from top to bottom. This is necessary since the value of D4 must be computed in the first statement before it can be used in the second statement. Most computers work by executing all statements sequentially unless the flow of control is altered explicitly by the programmer. Some of the other statements discussed later in this section will illustrate a non-sequential execution of program statements. Note that although this example of a sequence of statements used only simple statements, any type of statement mix is permissible. Thus, a mixture of simple and compound statements, or even all compound statements, listed sequentially are still considered sequences of statements. See Figure 2.6 for a pictorial representation of this concept.



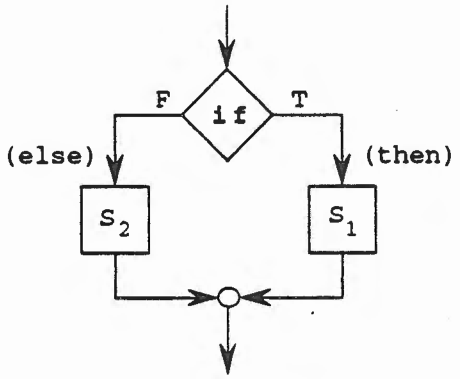
Sequence of Statements Flowchart
Figure 2.6

2.7.2 If Then Else Statements

The **if** statement selects up to one of at most two possible statements for execution. As has been explained previously, the **if** statement provides the programmer with the capability of altering the normal sequential flow of control based upon the value of some condition. For example, in the **if** statement

```
if X1 /= Y2 then
  X1 := Y2;
else
  X1 := X1 - Y2;
end if;
```

the condition X1 /= Y2 is evaluated to determine if it is TRUE or FALSE. If the condition is TRUE, then the sequence of statements after the **then** and before the **else** is executed, in this case only the assignment X1 := Y2. Control then passes to the statement after the **end if**. If the condition is FALSE, then the flow of control passes to the sequence of statements after the **else** and before the **end if**, skipping over the statements between the **then** and the else. In this case, the statement X1 := X1 - Y2 would be executed. The flow of control then continues with the statement after the **end if**. See Figure 2.7 for a graphical depiction of this flow of control.



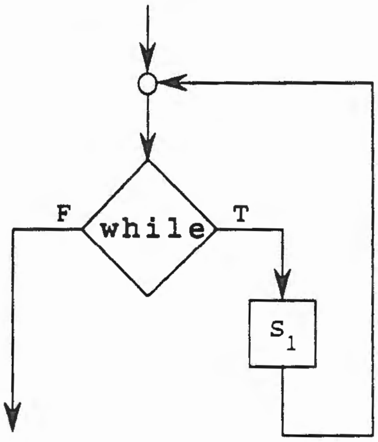
If Statement Flowchart
Figure 2.7

2.7.3 While Loop Statements

The *while loop* statement repeats a statement an indefinite, or variable, number of times. This may be zero or more times. The while loop is one form of iteration statement allowed in Ada. It is another statement that allows the programmer to alter the normally sequential flow of control. Given the while loop statement

```
while Next < 5 loop
  Next := Next + 1;
end loop;
```

the condition after the **while** is evaluated. If it is FALSE, then the entire while loop statement is skipped and the flow of control passes to the statement after the **end loop**. If the condition is TRUE, then the sequence of statements between the **loop** and **end loop** is executed. After completion of this execution, control does NOT pass to the statement after the end **loop**. Instead, the condition is re-evaluated and the flow of control continues as described above. In this case, if the value of `Next` is initially less than 5, then this loop will execute and the value of `Next` will be increased by one. Eventually, after some number of iterations depending on the initial value of `Next`, `Next` will be no longer less than 5 and the loop will terminate. See Figure 2.8 for a graphical depiction of this flow of control.



While Loop Flowchart
Figure 2.8

Any kind of loop statement in Ada can be given a name. This is accomplished merely by placing the name desired for the loop, which can be any legal identifier, before the loop, with a colon after the name. This same identifier must then be repeated after the reserved words end loop, just before the terminating semicolon. Thus, in the following example:

```
Checking_Valid_Number:
while Next < 5 loop
  Next := Next + 1;
end loop Checking_Valid_Number;
```

the loop is named `Checking_Valid_Number`. While Ada does not require that all loops be named, it is a good engineering practice because it can assist the maintenance engineer later in determining the purpose of the loop, assuming that a well chosen, meaningful identifier is used to name the loop.

The while loop is a very powerful statement for the programmer. However, like any other capability it can be misused and it may cause some difficulties. Consider what may happen with the following while loop

```
Checking_Now:
while Next < 5 loop
  Now := 1;
end loop Checking_Now;
```

Besides the fact that the body of the loop does not do much useful work, note that the value of `Next` is not changed within the loop. Thus, if the value of `Next` prior to this statement is, say 3, then the loop will be entered. After completion of the first iteration through the loop, the condition will be re-evaluated. Since the value in `Next` did not change, then `Next` must still be 3 and so the loop is re-entered. There is no way for the program to ever exit from this loop! This is an example of a common programming error and it even has a name; it is an *infinite loop*. An infinite loop can be avoided if the programmer writes all loops with this possibility in mind and therefore insures that all loops will eventually terminate.

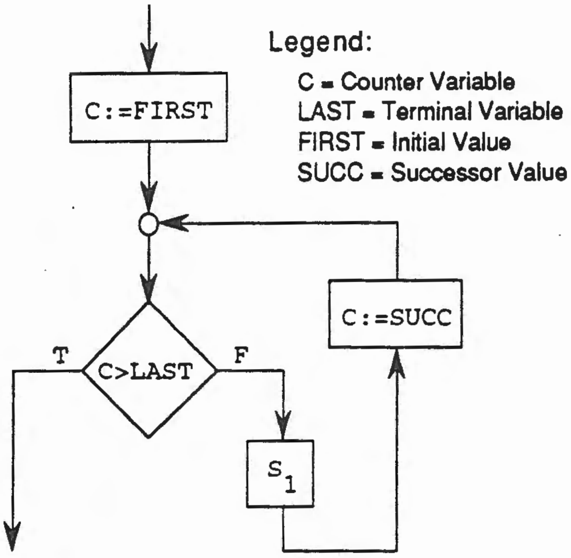
2.7.4 For Loop Statements

In all of the previous examples, whenever a looping structure was required an *indefinite* iteration mechanism, the while loop, was used. Recall that a while loop is a form of iteration that is used when the number of iterations is unknown. Rather, the iterations are continued until some condition is no longer true. In many circumstances that form of iteration is still the most appropriate. However, there are times when the number of iterations is known in advance or when the software engineer desires that a particular loop be executed precisely a known number of times. In such cases, a different form of iteration is used, namely *definite* iteration. In Ada, definite iteration is performed with a *for loop* statement.

A for loop is used when the number of iterations is determinable. This does not mean that the number of iterations must always be known in advance. For example, a simple for loop, when it is known that exactly ten iterations of the loop is desired, would be written as,

```
Put_One_To_Ten:
for Loop_Counter in 1 .. 10 loop
  INT_IO.Put (Item => Loop_Counter);
  -- assumes proper integer I/O instantiation called INT_IO
  -- is visible
end loop Put_One_To_Ten;
```

This for loop would set the value of `Loop_Counter` to 1 (the lower bound specified) and do one iteration. Upon completion of that iteration, the value of `Loop_Counter` would be incremented by one (actually the successor to the first value would be selected) to the next value (2 in this case) and another iteration would be completed since the new value is not greater than the upper bound (10 in this example). Similarly, each time the end of the loop is reached, `Loop_Counter` would be incremented and compared to the upper bound to see if its value was greater. If not, then one more iteration of the loop would take place. If `Loop_Counter` is greater than the upper bound, then the loop would terminate and the flow of control would pass to the statement following the for loop statement. See Figure 2.9 for a pictorial representation of this concept.



Legend:
C = Counter Variable
LAST = Terminal Variable
FIRST = Initial Value
SUCC = Successor Value

For Loop Flowchart
Figure 2.9

Note that the range must be determinable, not necessarily known in advance. For example, consider the following for loop, where the upper bound is obtained from a call to `INT_IO.Get`,

```
INT_IO.Get (Item => Upper_Bound);
  -- Upper_Bound is INTEGER
Weekend_Hooray:
for Counter in 1 .. Upper_Bound loop
  TEXT_IO.Put (Item => "Hooray for weekends!");
end loop Weekend_Hooray;
```

It is easy to see that while it is not known at the time that the program is written how many iterations will be made, it is completely determinable at the point that the for loop statement is encountered.

The range specified may be any discrete range of any discrete type. For example, consider the for loop below,

```
subtype Letters is CHARACTER range 'A' .. 'Z';

Loop_By_Letters:
for This_Letter in Letters loop
  TEXT_IO.Put_Line (Item => "Another neat letter => " & This_Letter);
end loop Loop_By_Letters;
```

In this example, the identifier `This_Letter` is initially given the value 'A', the first value in the subtype `Letters`. After one iteration of the loop, the successor to 'A' is assigned to `This_Letter`, which in this example is 'B'. Since this is not the last value in the subtype another iteration of the loop is executed. Now `This_Letter` is given the value of the successor to 'B', namely 'C' and another iteration of the loop is executed. Finally, when there is no successor to the value 'Z' in the subtype `Letters`, the loop is terminated normally. Thus, the loop identifier does not need to be an integer, but any discrete type is allowed.

Consider the loop identifier that appears after the reserved word **for**. This identifier is implicitly declared at the point that it appears. It does not need to be declared in the declarative section of the subprogram; in fact, it cannot be declared explicitly anywhere. It takes its type from the type of the `discrete_range` that must be provided. The scope of the identifier is solely within the loop. This means that after the loop has terminated, the identifier no longer exists and may not be referenced. Further, inside of the for loop, the value of this identifier may be read, but may not be updated, *i.e.*, the identifier may not be assigned a new value. Finally, it is not possible to iterate by more than the successor to the previous value. This means that it is not possible to iterate from 1 .. 10 by two's.

Sometimes it is desirable to iterate backwards, *i.e.*, from the upper bound down to the lower bound. This is accomplished in Ada by merely placing the reserved word **reverse** between the reserved word **in** and the discrete range. Thus, to count backwards from 50 to 25, a for loop statement would contain,

```
Backward_Example:
for My_Number in reverse 25 .. 50 loop
  TEXT_IO.Put ("Wow -- backwards!");
end loop Backward_Example;
```

This for loop would initialize `My_Number` to 50 and complete one iteration. It would then take the predecessor value, namely 49, and assign this to `My_Value` and complete another iteration. This would continue until `My_Value` was assigned the value 25. Then the loop would make one more iteration and terminate normally, since `My_Value` would be assigned the predecessor to 25 and that value is outside of the stated range. Note that the range is still written in the normal manner with the smaller value first and the larger value last. If the software engineer places these values in the other order, such as

```
for My_Number in reverse 50 .. 25 loop
```

it is not an error; it is a *null range*, *i.e.*, a range with no possible values. Thus, if the software engineer were to make the mistake of reversing the range bounds, the compiler would not detect an error, but no iterations would be executed.

In summary, a for loop allows the software engineer to provide definite iteration control to loops. The loop identifier is implicitly declared and must take on discrete values. The for loop can run backwards with the addition of the reserved word **reverse**.

2.7.5 Choosing the Appropriate Loop

A while loop is used when the number of iterations is indeterminable. It continues to execute as long as some specified condition is true. A for loop is used when the number of iterations is known or determinable. Thus, a while loop represents indefinite iteration, and a for loop represents definite iteration.

What does this mean? Consider the situation where the software engineer needs to write out the numbers 1 to 10 as headings for a table. Since the number of iterations (10) is known in advance, the appropriate looping mechanism for this situation is a for loop. Now suppose that the software engineer needs to read all of the numbers on one line of the input file. Since it is not known in advance how many numbers may be on any given line, a while loop is used, with the terminating condition being the detection of the end of the line (how this is done will be explained later).

In general, a for loop is safer than a while loop because a for loop will always execute for a specified number of iterations and then terminate. A while loop, on the other hand, is not guaranteed to terminate because it will continue to iterate until the condition in the loop becomes false. If the condition is never false, the while loop will never terminate. Sometimes this can be desirable; many real-time systems used in space exploration or radar tracking programs are designed precisely this way! Unfortunately, for our purposes this is almost always an error and will cause you to have infinite loops in your programs, thus preventing your programs from ever terminating.

When you decide which type of looping construct to use, first ask yourself if this loop will execute a specific number of times that can be determined at runtime when the loop is first entered. If so, then the for loop is what you want to use. If not, then you must ask yourself if the loop is dependent on a specific condition, that when the condition is not longer true, the loop may terminate. If so, then you want to use a while loop. Remember that you must consider the tradeoff that the loop indeterminacy has given you; it has made the possibility of an infinite loop a reality for you. You must be very careful to ensure that the loop condition will eventually become false to prevent the occurrence of an infinite loop.

2.8 Syntax for Ada Statements

The syntax for Ada statements follows next.

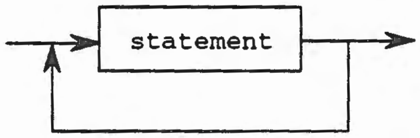
```
sequence_of_statements ::= statement {statement}

statement ::= simple_statement | compound_statement

simple_statement ::= null_statement
  | assignment_statement | procedure_call_statement

compound_statement ::= if_statement | loop_statement

null_statement ::= null;

assignment_statement ::= variable_name := expression;

procedure_call_statement ::=
  procedure_name [actual_parameter_part];

if_statement ::= if condition
          then
            sequence_of_statements
          [else
            sequence_of_statements]
          end if;

condition ::= boolean_expression

loop_statement ::=  [loop_simple_name :]
          [iteration_scheme] loop
            sequence_of_statements
          end loop;
```

```
iteration_scheme ::= while condition |
                     for loop_parameter_specification

loop_parameter_specification ::=
                  identifier in [reverse] discrete_range
```

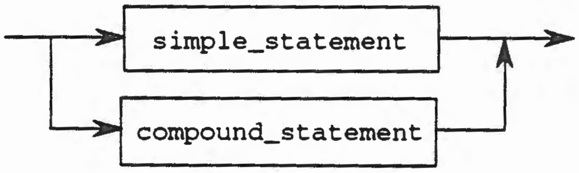
Sequence of Statements
Syntax Definition 2.14

This syntax can also be shown in graphic form as follows.

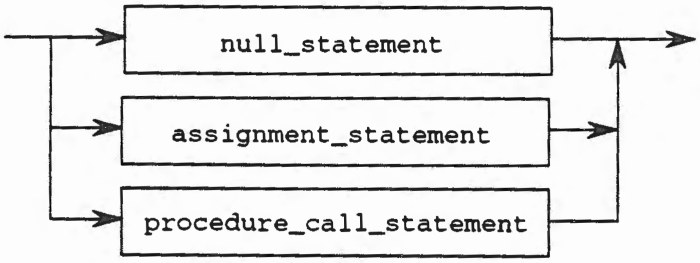
sequence_of_statements ::=



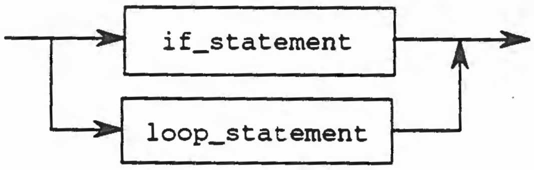
statement ::=



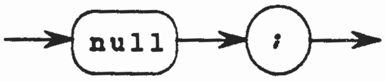
simple_statement ::=



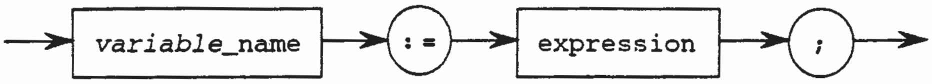
compound_statement ::=



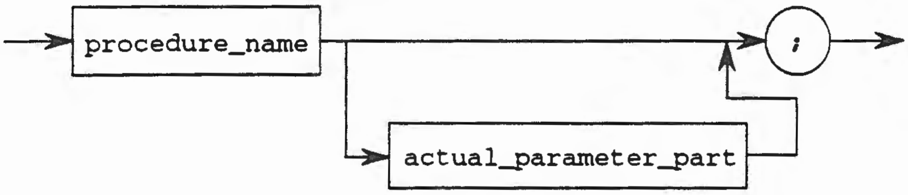
null_statement ::=



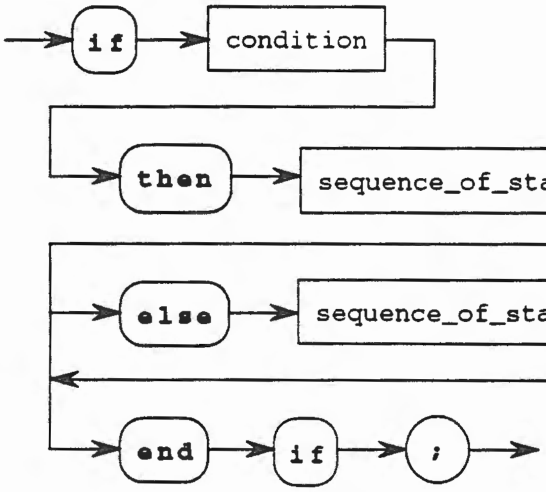
assignment_statement ::=



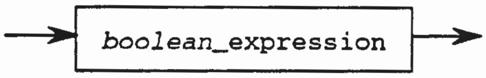
procedure_call_statement ::=



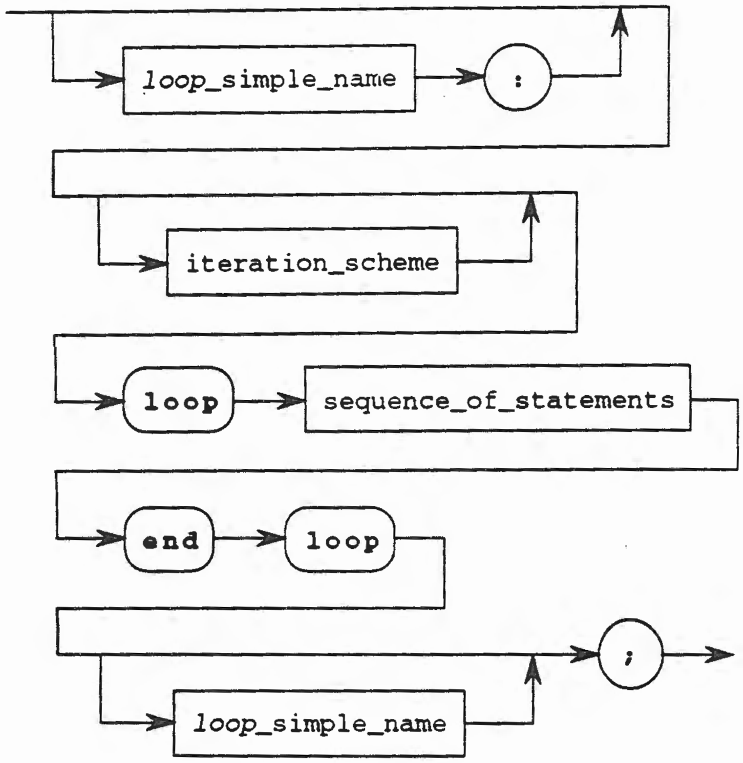
if_statement ::=



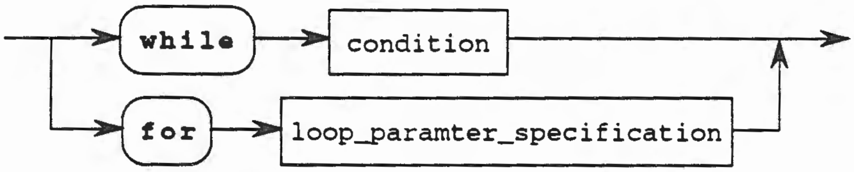
condition ::=



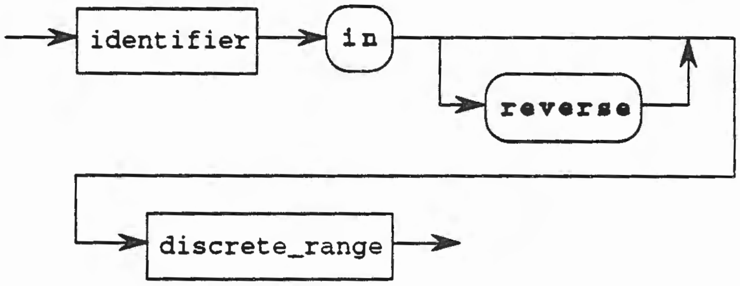
loop_statement ::=



iteration_scheme ::=



loop_parameter_specification ::=



Sequence of Statements
Syntax Chart 2.14

2.9 Example Derivation of a Statement

To illustrate the use of this grammar, we will derive a statement from the top-down. Consider the statement:

```
Value := X + 3 > 10 and Y < 4;
```

where `Value` is of type BOOLEAN. We would like to see if this statement is derivable from the grammar just given. We first examine the syntax for a `statement` and we see that a `statement` is:

```
statement ::= simple_statement | compound_statement
```

We next look at the production rule for a `simple_statement` and we see that it is given by the following rule:

```
simple_statement ::= null_statement
                   | assignment_statement | procedure_call_statement
```

We next examine the syntax for an `assignment_statement` and we see that it is defined to be:

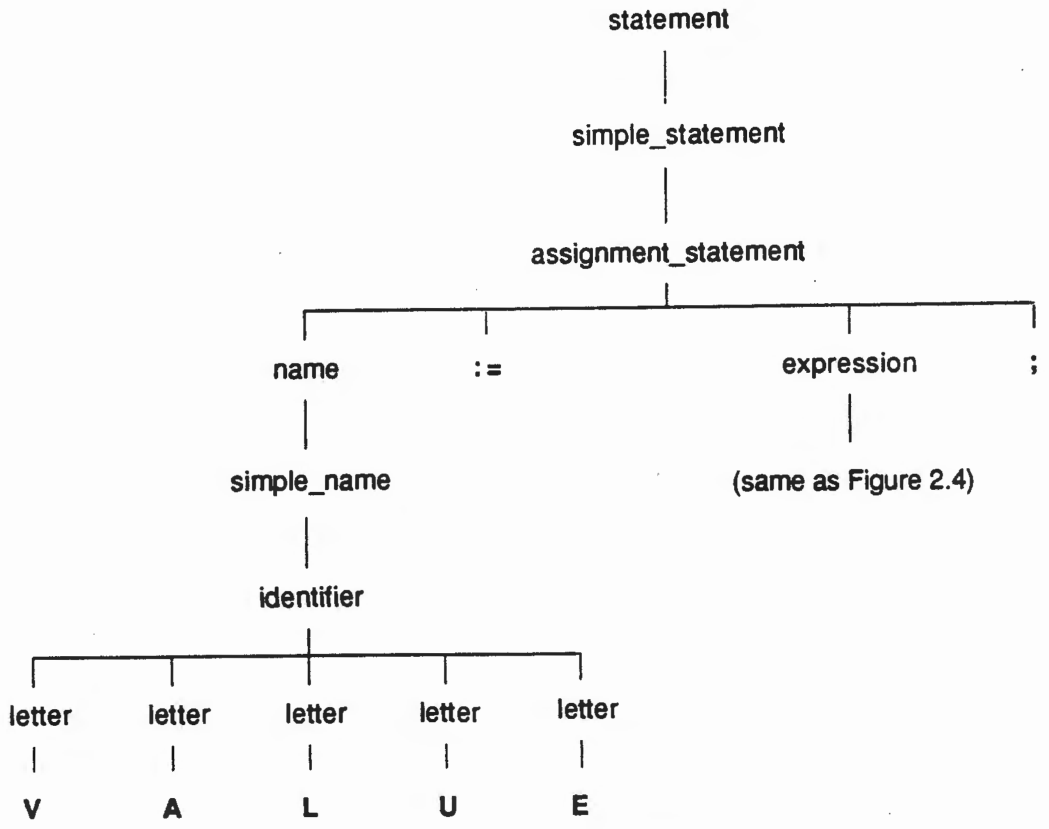
```
assignment_statement ::= name := expression;
```

Now if we can show that `Value` is a `name` we can conclude that this is a `statement`, since the := and the ; match and we have already determined that X + 3 > 10 and Y < 4 is an `expression` (See Section 2.5.8 for the derivation of this expression.)

Now the derivation for `name` is given by the following series of syntax rules:

```
name ::= simple_name

simple_name ::= identifier

identifier ::= letter {[underline] letter_or_digit}
```

where the complete derivation rules are shown in section 2.5.8. Since we find that `Value` is a `name`, the := and ; terminals are located in the proper position, and X + 3 > 10 and Y < 4 is an `expression`, we can conclude that this is a syntactically legal statement. Figure 2.10 shows a graphical depiction of this derivation.



Derivation of a Statement
Figure 2.10

As mentioned previously, we do not need to go through this tedium every time we write programs. The compiler will check the syntax to verify that we have not violated any rules, as part of its translation process. The key point here is that you should be able to manually trace through a derivation to determine if a statement is syntactically correct.

2.10 Exceptions

It would be an ideal world if you could anticipate all of the uses that would be made of your program. Similarly, it would be nice if you could anticipate all of the possible input that your program might have to accept. Unfortunately, this is rarely the case. Consequently, your program may fail for a variety of reasons, not all of which can be anticipated by you. In many languages, such a situation causes the computer to *"crash"*. This is a term used by computer professionals to indicate that the program was terminated abnormally. Usually the operating system can be counted upon to trap these errors and simply stop executing your program, indicating the error if it can. Other times, your entire system may *"lock up"* and need to be manually reset. Until now, this was just an accepted part of programming.

However, in Ada we have a built-in mechanism to detect and potentially correct errors while the program is still executing. Depending on what the software engineer wants to do, it is possible that such faults as would normally cause the program to *"crash"* can be detected, handled, and the program can continue in any manner that is desired. This facility in Ada is called an *exception handler* and the exceptional situation that caused a fault is called an *exception*.

Exceptions are both predefined and user-defined. Early in this text we will use the predefined exceptions as a convenience. Later, we will use both the predefined exceptions and the user-defined exceptions that we will define that are unique and specific to our problem domain.

A complete definition and explanation of exceptions will be deferred until later. For now, we will introduce a simple example of an exception handler. This mechanism should be a part of all of your programs from this point forward. Later we will see how to tailor this generalized exception handler to fit the specific problem that we are solving with our programming system.

```
with TEXT_IO;
procedure Demonstrate_Exception is
  Value : CHARACTER;
begin
  TEXT_IO.Put (Item => "Enter a value => ");
  TEXT_IO.Get (Item => Value);
exception
  when others => TEXT_IO.Put ("Oops - an error!");
end Demonstrate_Exception;
```

In this example, the part between the reserved words **begin** and **exception** are the normal programs executions statements that we have seen before. The part between the reserved words **exception** and **end** is called the exception handler. In this example, there is a single exception handler, designated by the **when others**. **When others** is a short cut means for saying that all exceptions are to be handled by this handler. When something goes wrong and an exception occurs, we say that it has been *raised*. Execution of the normal sequence of statements ceases and control passes to the exception handler, if there is one. If the proper exception is handled in that exception handler, or if there is a **when others**, the exception is lowered and the statements following the arrow (=>) are executed. When these statements have been executed, control returns to the calling procedure (if there is one) normally, *i.e.*, as if there had been no exception. The calling subprogram is not able to determine that an exception has ever been raised.

More on exceptions will be presented later. For now, use the mechanism shown in the demonstration program to embed an exception handler in all programs that you write.

2.11 Exercises

1. In what way are the following statements different,

```
Next  = Last + 10
Next := Last + 10
```

2. Given the following sequence of statements, what will be the outputs when `Able` and `Baker` have the values stated?

```
INT_IO.Put (Item => "Check on " & Able);
INT_IO.Get (Item => Baker);
Able := Able + Baker;
INT_IO.Put (Item => "New answer is " & Able);
```

a. Able = 10, Baker = 15
b. Able = -10, Baker = 15
c. Able = 10, Baker = -15
d. Able = -10, Baker = -15

3. Given the following if statement, what will be output when Now and Then have the values stated?

```
if Now > Then * 2 then
  INT_IO.Put (Item => Now);
else
  TEXT_IO.Put (Item => "Now <= Then * 2");
end if;
```

a. Now = 6, Then = 2
b. Now = 2, Then = 1
c. Now = 10, Then = 6
d. Now = 15, Then = 0

4. How many iterations of the following while loop will be executed when Now has the given values?

```
Iterating:
while Now < 10 loop
  Now := Now - 2;
end loop Iterating;
```

a. Now = 5
b. Now = 15
c. Now = 10
d. Now = 25

5. What is potentially wrong with the following while loop?

```
Checking:
while Now > Then loop
  Then := Then + 2;
  Now := Now + 2;
end loop Checking;
```

6. Write a while loop statement, controlled by the condition of Area being more than Side squared, whose body explicitly does nothing.

7. What will the following for loop do with the data given?

```
Trial_and_Error:
for Loop_Counter in 1 .. 5 loop
  if Loop_Counter = 1 or Loop_Counter = 10 then
    INT_IO.Put (Item => Loop_Counter);
  else
    INT_IO.Put (Item => Loop_Counter + Able);
end loop Trial_and_Error;
```

a. Let Able be 7.
b. Let Able be -7.

Chapter 3

Program Behavior

With a significant subset of the programming language Ada at our disposal, we are now interested in how programs and program parts can be constructed, analyzed, verified and tested. Any sequence of character strings may be regarded as an Ada program or program part, but it must pass syntax rules and semantic rules to be a legal such Ada program or part. Even then it may or may not exhibit the intended behavior when executed. For that reason, we introduce a formal definition for the *meaning* of any program or program part in terms of its *behavior*, namely the net effect of its execution on data.

In this Chapter 3, we first study Ada programs or program parts for their understanding in a general way. Next we give a formal definition for *program behavior*. Then we see the effects of specific program parts in behavior, including declarations as well as statements. Statements of interest here are ***simple statements***, namely procedure calls, assignments, and **null**, and ***complex statements***, namely sequence, **if** statements, and loop statements, both while loop and for loop. Complex statements may have simple statements or other smaller complex statements as components.

One special undeclared data is the Standard Input and Standard Output treated by the TEXT_IO package. The procedures and functions of TEXT_IO illustrate a collective behavior for a package used to simplify program design dealing with input and output.

A program can be defined in terms of mathematical objects with geometric forms that describe all possible behaviors. For example, sequence, **if**, and **for** statements can be used to describe all possible behaviors. But **while** statements, which are very powerful, can only describe a subset of all possible behaviors. For some data, a **while** statement will not loop at all, behaving like a **null** statement. For some data, a **while** statement may never terminate. But for some intended behavior, no **while** statement can be designed. That will be apparent with some mathematical treatment.

3.1 Understanding Ada Programs

The purpose of creating a program is to have the computer perform some specific function that solves a particular problem. If the solution is to be correct, then one must fully understand what the problem is and what the program does. Several ways could be invented to determine what a given program will do. One way would be to exhaustively try all possible inputs. This would include every character and number in every possible combination.

For programs that deal with small numbers of possibilities, this may be the right treatment. But most programs will have more possibilities than can be completely tested. For these programs, a systematic but incomplete process is needed to check their correctness. This process can include studying program behavior by means of

- analyzing input/output pairs that can occur,
- reading the code for what it does,
- reading the documentation and comparing with the code, and
- considering the program structure for the effect on behavior.

An obvious improvement to understanding what is transpiring within a program during execution is to read the program code itself. As straightforward as this seems it may not be easy if the software engineer did not create the code with this in mind. Reading program code is fundamental to understanding how programs work, yet making them readable requires skill and training. One way to make them readable is to physically structure them in a form that is easy to visualize. Another way for the author to transmit the rationale behind the program is to insert documentation specifically for that purpose.

3.1.1 Understanding Ada Programs from Author Documentation

To assist in understanding what the author intended the program to do, comments should be liberally embedded in each program. As described in Chapter 2, Ada provides a flexible mechanism to insert comments throughout any program. When learning a new programming language, comments can be used to describe in verbose terms the actions each statement causes. In `Print_Message_1` beginning Chapter 2, namely

```
with TEXT_IO;
procedure Print_Message_1 is
begin
  TEXT_IO.Put (Item => "Welcome Aboard");
end Print_Message_1;
```

each statement could be described in more detail with comments as

```
with TEXT_IO;
-- specifies input/output library package to be used
Procedure Print_Message_1 is
-- gives the procedure the name 'Print_Message_1'
-- connects 'Print_Message_1' with logic of procedure
begin
-- start of logical block of code
  TEXT_IO.Put (Item => "Welcome Aboard");
  -- creates a specific output
end Print_Message_1;
-- ends block associated with begin of Print_Message_1
```

Additional comments could be inserted if further explanation was desired. For example, the first with clause could be augmented as

```
with TEXT_IO;
-- specifies input/output library package to be used
-- uses Standard Input and Standard Output
```

At this level of specificity, the programs will become very long and cumbersome. Once the fundamental operations of each statement are understood many such comments will become superfluous. More useful comments can be inserted by the author to provide significant insight into the logic and rationale involved in creating the program. For example, the difference between procedure `Print_Message_2` and `Print_Message_1` could be indicated as

```
with TEXT_IO;
procedure Print_Message_2 is
-- expansion on Print_Message_1 to ask for capital letter and
-- return response depending on the user input
  Choice : CHARACTER; -- to hold capital letter if given
```

```
begin
  TEXT_IO.Put (Item => "Enter a capital letter => ");
  -- get character response
  TEXT_IO.Get (Item => Choice);
  -- check response
  if Choice >= 'A' and Choice <= 'Z' then
    -- have received capital letter as requested
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else -- did not receive capital letter as requested
    TEXT_IO.Put (Item => "Not a capital letter!");
  end if;
  TEXT_IO.New_Line;
end Print_Message_2;
```

The ability to create the appropriate level of comments is a skill that must be developed with practice and experience. A general rule of thumb is 'better too many than too few.' The author of a program must remember that the comments are inserted to help the author and others understand what should be going on. The following example shows how programs can be created with the appropriate level of comments.

3.1.2 A Program for Recognizing Roman Digits

Roman numerals are still used in numbering book chapters, as yearly dates on building fronts, and in other ways in artistic or historic objects. It is worth remembering that a thousand years ago roman numerals were used in business, engineering, and science for recording and arithmetic in Europe. While the place notation of the abacus was known and used in the Far East, it wasn't known in Europe. Arithmetic in roman numerals is possible, but much more difficult than arithmetic in place notation. Roman numerals do not permit negative numbers, nor even zero, as a valid number. The place ideas such as long division or step-by-step multiplication are not at all easy to discover in roman numerals, and little is recorded in arithmetic theory in roman numerals. With place notation and long division, grade school children today can solve problems in arithmetic that were beyond the practical capabilities of Euclid or Aristotle.

The following Ada program, named `Proper_Roman_Digit`, puts a message to the screen in response to a character found in the input as to whether that character is a proper roman digit for use in roman numerals or not.

```
with TEXT_IO; -- using Standard Input and Standard Output files
procedure Proper_Roman_Digit is
-- purpose is to determine if input character is a Roman digit
  -- first describe the data
  Digit : CHARACTER;
begin -- next describe sequential process on data
  TEXT_IO.Put (Item => "Enter Roman digit => ");
  TEXT_IO.Get (Item => Digit);
  TEXT_IO.Put (Item => "Character " & Digit);
  -- test to see if Roman digit was input
  if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
               or Digit = 'X' or Digit = 'V' or Digit = 'I' then
    -- Digit is a Roman digit
    TEXT_IO.Put (Item => " is");
  else -- Digit is not a Roman digit
    TEXT_IO.Put (Item => " is not");
  end if;
```

```
      -- complete the output
      TEXT_IO.Put (Item => " a Roman digit.");
    end Proper_Roman_Digit;
```

As discussed before, the program begins with a with clause that makes the predefined package TEXT_IO of Standard Input and Standard Output available for use in the program.

The **with** clause precedes the start of the actual program which begins on the next line as

```
procedure Proper_Roman_Digit is
-- purpose is to determine if input character is a Roman digit
```

The comment is provided to clarify the purpose of the procedure, namely to determine if the character input is one of the possible Roman digits.

In the next line an object named `Digit` is declared to be an object of type CHARACTER, and the object *declaration* is concluded with a semicolon (`;`).

The *executable* part of the program is found between the two separated reserved words **begin** and **end**. The **begin** line contains a comment to point out that a sequential process will follow. The **end** line provides the name of the program again. The **begin** is not followed by a semicolon, but the **end** is followed by a semicolon. In this case, it is the **begin** ... **end** pair that is followed by a semicolon. That is, in the **end** line, the semicolon follows **end**, and the name of the program is optionally inserted between **end** and its semicolon.

In this executable part of the program, the first statement to `Output`

```
TEXT_IO.Put (Item => "Enter Roman digit => ");
```

asks for a candidate Roman numeral. Next a value for `Digit` is obtained by the `TEXT_IO.Get` statement

```
TEXT_IO.Get (Item => Digit);
```

which makes use of the TEXT_IO facilities provided by the opening **with** clause. The Get procedure of TEXT_IO will get a character and give that value to the CHARACTER object named `Digit` and add it to the input file. In `Input`, this character will become part of a character string. If `Input` is empty before (which it is in this case), it becomes a one character string, not a CHARACTER.

In this case, just after obtaining the value of `Digit` in

```
TEXT_IO.Get (Item => Digit);
```

the next part of the message to `Output` is formed in the statement

```
TEXT_IO.Put (Item => "Character " & Digit);
```

which writes the string "`Character` " catenated with the value in the object `Digit` to the output file.

Next, the seven line **if** statement preceded by its comment

```
-- first test to see if a proper digit was input
if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
               or Digit = 'X' or Digit = 'V' or Digit = 'I' then
  -- Digit is a Roman digit
  TEXT_IO.Put (Item => " is");
else -- Digit is not a Roman digit
  TEXT_IO.Put (Item => " is not");
end if;
```

adds either “ is” or “ is not” to the message (note that either addition begins with a space to separate it from the value of `Digit` in the previous part of the message).

In more detail, the first two lines of the **if** statement

```
if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
               or Digit = 'X' or Digit = 'V' or Digit = 'I' then
```

ask the question “is the current value of the object Digit a roman digit?” If it is, say the roman digit ‘C’ to be specific, the **then** part of the **if** statement will be executed, namely

```
TEXT_IO.Put (Item => " is");
```

The comment after **then** states that if the **then** is reached, then the current value of `Digit` is indeed a valid roman digit.

On the other hand, if the current value of `Digit` is not a valid roman digit, say the value ‘A’ to be specific, the **else** part of the **if** statement will be executed, namely

```
else -- Digit is not a Roman digit
  TEXT_IO.Put (Item => " is not");
```

Finally, the **if** statement is completed by the line

```
end if;
```

which also ends with a semicolon. Then the statement

```
TEXT_IO.Put (Item => " a Roman digit.");
```

completes the message.

Note that the double quotes in the `TEXT_IO.Put` statements are not part of the output, only the text between them. Double quotes can be output as has been pointed out earlier, namely by putting consecutive double quotes (`""`) within messages, but these double quotes in this `TEXT_IO.Put` statement define the limits of this message and are not meant to be written to the output file. Recall also that, in Ada, single characters, such as ‘C’ or ‘A’ are framed by single quotation marks, while character strings of any length are framed by double quotation marks. Since a single character is also a character string of length 1, there may be an option of which to use in operations where both are legal. `TEXT_IO.Put` statements will put both characters and character strings, but other operations may recognize only one or the other, as will be seen later.

These TEXT_IO.Put statements are satisfactory in the creation of an output message. Typically, there are many alternative programs to meet an objective. There will, indeed, be better ones among them, in execution time, space required, understandability, and so on. For example, it is perfectly legal in Ada to use an TEXT_IO.Put statement to put out each individual character in the message. That as, in place of the statement above in Proper_Roman_Digit

```
   TEXT_IO.Put (Item => "Character " & Digit);
```

the following sequence of eleven statements would be equivalent

```
   TEXT_IO.Put (Item => "C");
   TEXT_IO.Put (Item => "h");
   TEXT_IO.Put (Item => "a");
   TEXT_IO.Put (Item => "r");
   TEXT_IO.Put (Item => "a");
   TEXT_IO.Put (Item => "c");
   TEXT_IO.Put (Item => "t");
   TEXT_IO.Put (Item => "e");
   TEXT_IO.Put (Item => "r");
   TEXT_IO.Put (Item => " ");
   TEXT_IO.Put (Item => Digit);
```

In this case “Character " is output as a sequence of ten character strings each of length 1. As already noted, single characters are also defined between single quotation marks, so another sequence of eleven statements will also be equivalent

```
   TEXT_IO.Put (Item => 'C');
   TEXT_IO.Put (Item => 'h');
   TEXT_IO.Put (Item => 'a');
   TEXT_IO.Put (Item => 'r');
   TEXT_IO.Put (Item => 'a');
   TEXT_IO.Put (Item => 'c');
   TEXT_IO.Put (Item => 't');
   TEXT_IO.Put (Item => 'e');
   TEXT_IO.Put (Item => 'r');
   TEXT_IO.Put (Item => ' ');
   TEXT_IO.Put (Item => Digit);
```

In either case, though proper and legal, the single TEXT_IO.Put statement is easier to understand and takes less space in the program than either of these alternative sequences. There will be occasions when such character by character output may be useful, but not here.

This example is designed to provide insight into understanding a program by both reading the Ada statements and the comments. An author must carefully choose the appropriate comments to convey the intended purpose of specific statements as well as the overall actions of the various program units. The level of specificity, the frequency of comments, and the complexity of the code all affect the quality of the documentation. Experimentation and experience will be the best tutor for good documentation.

3.1.3 Understanding Ada Programs from their Text Structures

The program examples shown thus far are set off in indented lines for easier reading. Such text structure is not necessary for machine execution. For example, program Print_Message_2, which appeared as

```
with TEXT_IO;
procedure Print_Message_2 is
-- expansion on Print_Message_1 to ask for capital letter and
-- return response depending on the user input
  Choice: CHARACTER; -- to hold capital letter if given
begin
  TEXT_IO.Put (Item => "Enter a capital letter => ");
  TEXT_IO.Get (Item => Choice);
  if Choice >= 'A' and Choice <= 'Z' then
    -- have received capital letter as requested
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else -- did not receive capital letter as requested
    TEXT_IO.Put (Item => "Not a capital letter!");
  end if;
  TEXT_IO.New_Line;
end Print_Message_2;
```

can also appear as an unstructured string of program text as follows (note all identifiers, comments, and other text objects must be on single lines)

```
with TEXT_IO;procedure Print_Message_2 is
-- expansion on Print_Message_1 to ask for capital letter and
-- return response depending on the user input
Choice:CHARACTER; -- to hold capital letter if given
begin TEXT_IO.Put(Item=>"Enter a capital letter => ");TEXT_IO.Get
(Item=>Choice);if Choice >= 'A' and Choice <= 'Z' then
-- have received capital letter as requested
TEXT_IO.Put(Item=>"Welcome Aboard, " & Choice);else
-- did not receive capital letter as requested
TEXT_IO.Put(Item=>"Not a capital letter!");end if;
TEXT_IO.New_Line;end Print_Message_2;
```

With its comments removed, it comes down even further in size as

```
with TEXT_IO;procedure Print_Message_2 is Choice:CHARACTER;begin
TEXT_IO.Put(Item=>"Enter a capital letter => ");TEXT_IO.Get
(Item=>Choice);if Choice >= 'A' and Choice <= 'Z' then
TEXT_IO.Put(Item=>"Welcome Aboard, " & Choice);else
TEXT_IO.Put(Item=>"Not a capital letter!");end if;
TEXT_IO.New_Line;end Print_Message_2;
```

The second program would be executed identically as the first, since the indented text structure is not used in execution. However, it is much more difficult for people to read and understand. Conversely, if such programs are encountered, they can be restructured into a standard form. The reserved words provide the key to such standards. For example, the reserved words **with**, **procedure**, **begin**, **end**, **if**, **else**, and while should all start lines, as well as certain sequences of reserved words such as **end if** and **end loop**. Text between **is** and **begin**, between **then**, **else**, and **end if**, and between **loop** and **end loop** should be indented. Declarations and procedure call statements should be started on new lines.

In illustration, consider a new Ada program as an unstructured string of program text such as

```
with TEXT_IO;procedure Mystery_Procedure is Chop:CHARACTER;begin
TEXT_IO.Put(Item=>"First Character is ");TEXT_IO.Get(Item=>Chop);
if Chop<'A' or Chop>'Z' then TEXT_IO.Put(Item=>"illegal");else
TEXT_IO.Put(Item=>"legal name "&Chop);end if;end Mystery_Procedure;
```

To begin, let every reserved word or reserved word sequence begin a new line as follows

```
with TEXT_IO;
procedure Mystery_Procedure is
Chop:CHARACTER;
begin TEXT_IO.Put(Item=>"First Character is ");TEXT_IO.Get(Item
=>Chop);
if Chop<'A' or Chop>'Z' then
TEXT_IO.Put(Item=>"illegal");
else TEXT_IO.Put(Item=>"legal name "&Chop);
end if;
end Mystery_Procedure;
```

Next, let declarations and statements also begin new lines as follows

```
with TEXT_IO;
procedure Mystery_Procedure is
Chop:CHARACTER;
begin
TEXT_IO.Put(Item=>"First Character is ");
TEXT_IO.Get(Item =>Chop);
if Chop<'A' or Chop>'Z' then
TEXT_IO.Put(Item=>"illegal");
else
TEXT_IO.Put(Item=>"legal name "&Chop);
end if;
end Mystery_Procedure;
```

Next, indent lines between reserved words as described above to obtain what follows

```
with TEXT_IO;
procedure Mystery_Procedure is
  Chop:CHARACTER;
begin
  TEXT_IO.Put(Item=>"First Character is ");
  TEXT_IO.Get(Item =>Chop);
  if Chop<'A' or Chop>'Z' then
    TEXT_IO.Put(Item=>"illegal");
  else
    TEXT_IO.Put(Item=>"legal name "&Chop);
  end if;
end Mystery_Procedure;
```

Finally, spaces around embedded delimiters are inserted to arrive at the final version which is in standard form

```
with TEXT_IO;
procedure Mystery_Procedure is
  Chop : CHARACTER;
begin
  TEXT_IO.Put (Item => "First Character is ");
  TEXT_IO.Get (Item => Chop);
  if Chop < 'A' or Chop > 'Z' then
    TEXT_IO.Put (Item => "illegal");
  else
    TEXT_IO.Put (Item => "legal name" & Chop);
  end if;
end Mystery_Procedure;
```

In this standard form it is much easier to see what `Mystery_Procedure` does. First, it declares a CHARACTER type variable `Chop`. Next it puts a message

```
First Character is
```

with a blank character at the end to the output file. Then it asks for a character for variable `Chop`. Next it checks the if condition

```
Chop < 'A' or Chop > 'Z'
```

to see if the value provided `Chop` is a capital letter or not. If the value is not a capital letter it completes the message as

```
First Character is illegal
```

while if value is a capital letter, say 'C', it completes the message as

```
First Character is legal name C
```

This analysis could be done directly on the unstructured program because it is small, but larger programs can't be treated so easily.

The creation of good programs is more than just providing the correct code. The software engineer must also provide a way to understand what actions were intended and why specific approaches were taken to solving the given problem. This requires skill in structuring and documenting the code. A specific discipline in the form of a standardized *style guide*, such as has been used in this text, can provide valuable insight into understanding programs.

3.1.4 Exercises

1. What would be an appropriate comment to describe the difference between the procedures `Print_Message_1` and `Print_Message_3` of Chapter 2?

2. What would be an appropriate comment to describe the difference between the procedures `Print_Message_2` and `Print_Message_3` of Chapter 2?

3. Provide the appropriate comments for `Mystery_Procedure` at those points in the program that will aid in understanding the way the program functions.

4. An alternative way to implement Proper_Roman_Digit is as follows.

```
with TEXT_IO; -- using named Input and Output Files
procedure Proper_Roman_Digit_2 is
-- Another way to determine if an input digit is a proper
-- Roman digit
  -- describe data first
  Digit : CHARACTER;
  Valid_Digit : BOOLEAN := FALSE;
begin -- describe sequential process on data second
  TEXT_IO.Put (Item => "Give input digit ");
  TEXT_IO.Get (Item => Digit);
  if Digit = 'I' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'V' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'X' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'L' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'C' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'D' then
    Valid_Digit := TRUE;
  end if;
  if Digit = 'M' then
    Valid_Digit := TRUE;
  end if;
  if Valid_Digit then
    -- Digit is a roman digit
    TEXT_IO.Put (Item => "Character " & Digit &
                         " is a roman digit.");
  else -- Digit is not a roman digit
    TEXT_IO.Put (Item => "Character " & Digit &
                         " is not a roman digit.");
  end if;
end Proper_Roman_Digit_2;
```

Does `Proper_Roman_Digit_2` indeed behave externally in every respect as `Proper_Roman_Digit`? Which of the two programs is easier to understand?

5. Another alternative way to implement `Proper_Roman_Digit` is as follows.

```
with TEXT_IO; -- using named Input and Output Files
procedure Proper_Roman_Digit_3 is
-- Still another way to determine if an input digit is a
-- proper Roman digit
  -- describe data first
  Digit : CHARACTER;
  Valid_Digit : BOOLEAN := FALSE;
begin -- describe sequential process on data second
  TEXT_IO.Put (Item => "Give input digit");
  TEXT_IO.Get (Item => Digit);
  if Digit = 'I' then
    Valid_Digit := TRUE;
  else
    if Digit = 'V' then
      Valid_Digit := TRUE;
    else
      if Digit = 'X' then
       Valid_Digit := TRUE;
      else
        if Digit = 'L' then
          Valid_Digit := TRUE;
        else
          if Digit = 'C' then
            Valid_Digit := TRUE;
          else
            if Digit = 'D' then
              Valid_Digit := TRUE;
            else
              if Digit = 'M' then
                Valid_Digit := TRUE;
              end if;
            end if;
          end if;
        end if;
      end if;
    end if;
  end if;
  if Valid_Digit then
    -- Digit is a roman digit
    TEXT_IO.Put (Item => "Character " & Digit &
                         " is a roman digit.");
  else -- Digit is not a roman digit
    TEXT_IO.Put (Item => "Character " & Digit &
                         " is not a roman digit.");
  end if;
end Proper_Roman_Digit_3;
```

Does `Proper_Roman_Digit_3` indeed behave externally in every respect as `Proper_Roman_Digit`? Which of the three programs is easier to understand? Which is the fastest in execution?

6. Consider the program given next in an unstructured form of program text. Work out the standard text structure and determine what it does.

```
with TEXT_IO;procedure Mystery_Program_2 is begin TEXT_IO.Put
(Item=>"This");TEXT_IO.Put(Item=>" is a ");TEXT_IO.Put
(Item=>"Test");end Mystery_Program_2;
```

7. Consider the program given next in an unstructured form of program text. Work out the standard text structure and determine what it does. Also provide the appropriate comments to explain the program actions.

```
with TEXT_IO;procedure Mystery_Program_3 is Chop:CHARACTER;
begin TEXT_IO.Put(Item=>"Return number");TEXT_IO.Get(Item=>Chop);
if '0'<=Chop and Chop<='9' then TEXT_IO.Put(Item=>
("First character is number "&Chop);else TEXT_IO.Get(Item=>Chop);
end if;if '0'<=Chop and Chop<='9' then TEXT_IO.Put(Item=>
"First or second character is number "&Chop);
else TEXT_IO.Put(Item=>"No number found, but "&Chop);
end if;end Mystery_Program_3;
```

8. Consider the program given next in an unstructured form of program text. Work out the standard text structure and determine what it does. Also provide the appropriate comments to explain the program actions.

```
with TEXT_IO;procedure Mystery_Program_4 is Foo,Bar:CHARACTER
:=' ';begin TEXT_IO.Put(Item=>"Return capital letters");
TEXT_IO.Get(Item=>Foo);while Foo<'A' or 'Z'<Foo loop
TEXT_IO.Put(Item=>"This character is "&Foo);TEXT_IO.New_Line;
TEXT_IO.Get(Item=>Bar);if Foo=Bar then null;else Foo:=Bar;
end if;end loop;end Mystery_Program_4;
```

3.2 Understanding Program Behavior

Improving the textual structure and inserting appropriate comments will provide insight into what the author of a program intended the program to do. Human fallibility and carelessness will, however, introduce unknown errors into even simple programs. With large and complex programs this error rate can become astounding. To counter this tendency, techniques have been developed that provide formal methods for understanding what a program will do. These methods are founded in formal mathematics so they can be scaled up as the programs become large and complex. They provide an understanding of what any given program will actually do, by analyzing the individual components of the program as mathematical entities. In so doing, all of the knowledge, formalism, and power of our understanding of mathematics can be brought to bear.

3.2.1 Program Behavior

The set of input, output pairs from all possible executions of an Ada program is called its *program behavior* which will be a *function* or a *relation*. This is so since they provide a mapping of all possible inputs into associated outputs. Ada programs with unique behavior in every possible execution are rules for functions. Ada programs with possible non-unique behavior in execution are rules for relations. For example, a program which makes use of declared data before it is assigned a value will exhibit non-unique behavior. Such a program is most likely a mistake, but it will be a legal program which behaves differently on separate executions. In illustration, the program with main procedure `Print_Message_2` is shown altered by removing the

```
   TEXT_IO.Get(Item => Choice);
```

statement, converting it into a comment for ease of analysis, and calling its main procedure `Print_Message_At_Random`.

```
with TEXT_IO;
procedure Print_Message_At_Random is
-- expansion on Print_Message_1 to ask for capital letter and
-- return response depending on the user input
  Choice : CHARACTER; -- to hold capital letter if given
begin
  TEXT_IO.Put (Item => "Enter a capital letter => ";
  -- TEXT_IO.Get (Item => Choice);
  if Choice >= 'A' and Choice <= 'Z' then
    -- have received capital letter as requested
    TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
  else -- did not receive capital letter as requested
    TEXT_IO.Put (Item => "Not a capital letter!");
  end if;
  TEXT_IO.New_Line;
end Print_Message_At_Random;
```

Since `Choice` is not assigned a value (because of the commented `TEXT_IO.Get`), the program will print whatever random character is found in `Choice`. This random value will be left behind from previous computer operations. Therefore, by mistake this program has a program relation, not a program function. For the remainder of this text, it will be assumed, unless explicitly stated otherwise, that programs under study will have program functions. For example, if `Choice` is initialized at declaration, such as

```
Choice : CHARACTER := '*';
```

which initializes `Choice` to one of the legal characters, namely the character '*', then

```
TEXT_IO.Put (Item => "Not a capital letter!");
```

will output

```
Not a capital letter!
```

3.2.2 Programs as Rules for Behavior

It is important to distinguish the program behavior from any of its member input, output pairs. Just as a mathematical function, say `f`, maps elements of the domain (`f`) to elements of the range (`f`), the program behavior maps any input to some output. For any particular input, a specific Ada program will deliver a particular output, and this input, output pair will be a member of its behavior.

The behavior of a program will be denoted by bracketing the program or its name with square brackets [,]. The program is a string of character strings of proper Ada. Thus a program behavior is a set of ordered pairs, each ordered pair being an input file, output file, themselves each a string (pages) of strings (lines) of character strings (characters).

For example, the program itself with main procedure named `Print_Message_2` could be named `Print_Message_2`, or some other name such as `Program_1`, say the latter. Then `Program_1` has program behavior denoted as

```
[Program_1]

= [with TEXT_IO;
  procedure Print_Message_2 is
  -- expansion on Print_Message_1 to ask for capital letter and
  -- return response depending on the user input
    Choice : CHARACTER; -- to hold capital letter if given
  begin
    TEXT_IO.Put (Item => "Enter a capital letter => ");
    TEXT_IO.Get (Item => Choice);
    if Choice >= 'A' and Choice <= 'Z' then
      -- have received capital letter as requested
      TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
    else -- did not receive capital letter as requested
      TEXT_IO.Put (Item => "Not a capital letter!");
    end if;
    TEXT_IO.New_Line;
  end Print_Message_2;]
```

The program behavior of `Program_1` is defined by the behavior `[Program_1]` from inputs to outputs, in this case from all possible input files to output files. As discussed less formally in the last chapter, this program behavior can be defined as follows,

```
[Program_1] = A ∪ B

where A = {⟨⟨Input, Output⟩, ⟨Input1, Output1⟩⟩ |
          ℏ(Input1) >= 'A' and ℏ(Input1) <= 'Z'
          and Output1 = "Enter a capital letter => " &
          "Welcome Aboard " & ℏ(Input1)}

and B = {⟨⟨Input, Output⟩, ⟨Input1, Output1⟩⟩ |
        ℏ(Input1) < 'A' or ℏ(Input1) > 'Z'
        and Output1 = "Enter a capital letter => " &
        "Not a capital letter!"}
```

which is the union of two simpler behaviors that are both functions. The domain and range are ⟨`Input`, `Output`⟩ pairs in which both `Input`, `Output` are originally empty strings. The results ⟨`Input1`, `Output1`⟩ depend on the input data provided by the user. `Input1` *will simply be a* single character received by the `TEXT_IO.Get` statement. `Output1` will depend on what the `Input1` data is, whether a capital letter—case A—or not a capital letter—case B.

Notice that `Input1` is a single character, since the `TEXT_IO.Get` statement reads and stores a single character. If the user enters more than one character, the first character is read while the remaining characters are stored in the input buffer in the order given. These remaining characters are then available for later Get statements.

In your mathematics classes, you use a notation f(x) = y to show that function f maps x to y. Similarly, to show that [Program_1] maps Input to Output, we write

```
[Program_1](Input) = Output
```

or for a particular input such as '1'

```
[Program_1]('1') = "Enter a capital letter/Not a capital letter!"
```

where "/" separates successive messages created by the program.

3.2.3 Behaviors of Programs that Fail

Although program behavior is predictable, it may not be what was intended. That is to say, a legal Ada program may execute correctly and produce output, however that output may not be what was expected. Another way to put this is that a program will do what it is *told* to do, which may not be what was *intended*. The program behavior will only indicate the program action for the program text that was specified.

As noted above, the procedure Print_Message_At_Random that makes line 9 a comment, remains a legal Ada program, as shown in Program_3 next. In this case

```
[Program_3]

= [with TEXT_IO;
  procedure Print_Message_At_Random is
  -- expansion on Print_Message_1 to ask for capital letter and
  -- return response depending on the user input
    Choice : CHARACTER; -- to hold capital letter if given
  begin
    TEXT_IO.Put (Item => "Enter a capital letter => ");
    -- TEXT_IO.Get (Item => Choice);
    if Choice >= 'A' and Choice <= 'Z' then
      -- have received capital letter as requested
      TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
    else -- did not receive capital letter as requested
      TEXT_IO.Put (Item => "Not a capital letter!");
    end if;
    TEXT_IO.New_Line;
  end Print_Message_At_Random;]
```

The program behaves as a relation depending on what data was left behind in the memory area used for Choice. In this case

```
[Program_3] = A ∪ B

where A = {⟨⟨Input, Output⟩, ⟨Input, Output1⟩⟩ |
          Output1 = "Enter a capital letter => " &
          "Welcome Aboard " & Some_Capital_Letter}

and B = {⟨⟨Input, Output⟩, ⟨Input, Output1⟩⟩ |
        Output1 = "Enter a capital letter => " &
        "Not a capital letter!"}
```

Note that Input is not used or altered by this program since no Get statements appear in it. Also notice that no reference is made to Choice since this is an internal variable. That is, it exists only during the execution of the program. Since we have no way of knowing the value of Choice, we can not predict which of the two behaviors, A or B, will be observed.

If the declaration for Choice included an initial value, such as

```
Choice : CHARACTER := ' '; -- to hold capital letter if given
```

the program would be a function. But of course it would not be the function intended.

Thus, the program may be proper, but not account for all possible cases. Such action is often called an error or a failure, when it may only be an oversight. In the Ada programming language, such an error or a failure is called an exception and will be handled either explicitly by the program or implicitly by the computing system. Whether error or oversight it can lead to serious consequences.

3.2.4 Program Behavior Based on Program Parts Behavior

Program behavior is determined by how the declarations and statements are carried out. For example the program Proper_Roman_Digit seen earlier this Chapter 2 has a program behavior defined as

```
[Proper_Roman_Digit]

= [with TEXT_IO; -- using Standard Input and Standard Output files
procedure Proper_Roman_Digit is
-- purpose is to determine if input character is a Roman digit
  -- first describe the data
  Digit : CHARACTER;
begin -- next describe sequential process on data
  TEXT_IO.Put (Item => "Enter Roman digit => ");
  TEXT_IO.Get (Item => Digit);
  TEXT_IO.Put (Item => "Character " & Digit);
  -- test to see if Roman digit was input
  if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
                 or Digit = 'X' or Digit = 'V' or Digit = 'I' then
    -- Digit is a Roman digit
    TEXT_IO.Put (Item => " is");
  else -- Digit is not a Roman digit
    TEXT_IO.Put (Item => " is not");
  end if;
  -- complete the output
  TEXT_IO.Put (Item => " a Roman digit.");
end Proper_Roman_Digit;]
```

which can be broken into steps for analysis. First, with TEXT_IO, Standard Input and Standard Output files are defined for use. Data for Standard Input will be provided by a user during execution and Standard Output starts as an empty file which will accept data as provided by TEXT_IO statements. Thus, after the initial with TEXT_IO; statement, the data available to the procedure goes from nothing to empty files Input, Output, in the behavior

```
[with TEXT_IO; -- using Standard Input and Standard Output files]

= {(⟨ ⟩, ⟨Input, Output⟩) | Input = Output = ⟨ ⟩}
```

Next, the declaration adds the variable `Digit` with CHARACTER values to the data, so it becomes

```
[Digit : CHARACTER;]

= {⟨⟨Input, Output⟩, ⟨Input, Output, Digit⟩⟩ |
    Digit any CHARACTER value}
```

With the declarations completed, execution of the procedure begin block starts. There are five statements to be executed in sequence, namely

```
TEXT_IO.Put (Item => "Enter Roman digit => ");
```

which sends the message "`Enter Roman digit => `" to the user.

```
TEXT_IO.Get (Item => Digit);
```

which accepts a user input, altering both `Input` and `Digit`.

```
TEXT_IO.Put (Item => "Character " & Digit);
```

which returns the message `"Character " & Digit` back to the user that includes the input just received from the user.

```
-- test to see if Roman digit was input
if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
               or Digit = 'X' or Digit = 'V' or Digit = 'I' then
  -- Digit is a Roman digit
  TEXT_IO.Put (Item => " is");
else -- Digit is not a Roman digit
  TEXT_IO.Put (Item => " is not");
end if;
```

which returns either "` is`" or "` is not`" to the user based on the value provided `Digit` by the user.

```
-- complete the output
TEXT_IO.Put (Item => " a Roman digit.");
```

which finally completes a sentence about the value of `Digit`.

Breaking these statements down into their behaviors, first

```
[TEXT_IO.Put (Item => "Enter Roman digit => ");]

= {⟨⟨Input, Output, Digit⟩, ⟨Input, Output1, Digit⟩⟩ |
    Output1 = Output & "Enter Roman digit => "}
```

in which only the value of `Output` is effected. Then

```
[TEXT_IO.Get (Item => Digit);]

= {⟨⟨Input, Output, Digit⟩, ⟨Input1, Output, Digit1⟩⟩ |
    Input1 = Input & Char and
    Digit1 = Char}
```

which accepts a user input, where `Digit1` is a character value provided by the user recorded also in `Input`. Then

```
[TEXT_IO.Put (Item => "Character " & Digit);]

= {⟨⟨Input, Output, Digit⟩, ⟨Input, Output1, Digit⟩⟩ |
    Output1 = Output & "Character " & Digit}
```

so that `"Character " & Digit` have been added to "`Enter Roman digit => `" and the initial contents of `Output` which in this case was empty, since there were no previous `Output` statements. Note also that the value of `Digit` is `Digit1` because of the previous statement, but again this depends on the sequence of these statements, not because of the statements themselves. Next, the **if** statement must be analyzed in two steps to get to the simple statements, as

```
[-- test to see if Roman digit was input
if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
               or Digit = 'X' or Digit = 'V' or Digit = 'I' then
  -- Digit is a Roman digit
  TEXT_IO.Put (Item => " is");
else -- Digit is not a Roman digit
  TEXT_IO.Put (Item => " is not");
end if;]

= {⟨⟨Input, Output, Digit⟩, ⟨Input, Output1, Digit⟩⟩ |
    Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
                or Digit = 'X' or Digit = 'V' or Digit = 'I'
    and Output1 = Output & " is"}
∪ {⟨⟨Input, Output, Digit⟩, ⟨Input, Output1, Digit⟩⟩ |
    not(Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D'
                    or Digit = 'X' or Digit = 'V' or Digit = 'I')
    and Output1 = Output & " is not"}
```

which ensures that either "` is`" or "` is not`" is added to `Output` as appropriate. Finally,

```
[-- complete the output
TEXT_IO.Put (Item => " a Roman digit.");]

= {⟨⟨Input, Output, Digit⟩, ⟨Input, Output1, Digit⟩⟩ |
    Output1 = Output & " a Roman digit."}
```

which completes a message in `Output`. Thus, for example, if 'C' was entered, `Output_1` would be

```
"Enter Roman digit => Character C is a Roman digit."
```

If 'A' was entered, `Output_1` would be

```
"Enter Roman digit => Character A is not a Roman digit."
```

3.2.5 Exercises

1. Examine the main procedure `Print_Message_1` from Chapter 2 and write the program behavior that describes this procedure.

2. What would be the expected behavior of the main procedure `Print_Message_2` if the statement

   ```
   TEXT_IO.Put (Item => "Enter a capital letter => ");
   ```

   was made into a comment?

3. Examine the main procedure `Print_Message_3` from Chapter 2 and write the program behavior that describes this procedure.

4. The program behavior of `Program_1` was written as

   ```
   [Program_1] = A ∪ B
   ```

   Another less formal form of `[Program_1]` would be

   ```
   [Program_1] = {⟨⟨Input, Output⟩, ⟨Input1, Output1⟩⟩ |
   Input1 is a sequence of single characters provided by user,
   Output1 begins with "Enter a capital letter => " then if ℎ(Input1)
   is a capital letter then Output1 is completed by "Welcome Aboard "
   & Input1 else Output1 is completed by
   "Not a capital letter!"}
   ```

   While less formal, all the information is there. Even less formal might be

   ```
   [Program_1] = {⟨⟨Input, Output⟩, ⟨Input1, Output1⟩⟩ |
   Input1 holds a single character, Output1 begins with
   "Enter a capital letter => " and if Input1 holds a capital
   letter then is completed by "Welcome Aboard " & Input1
   else completed by "Not a capital letter!"}
   ```

   Verify that this is also a correct behavior for `Program_1`.

5. If the main procedure `Print_Message_2` of Chapter 2 was made into `Program_5` and modified to replace the conditional statement

   ```
   if Choice >= 'A' and Choice <= 'Z'
   ```

   by substituting

   ```
   if Choice >= '0' and Choice <= '9'
   ```

   how would this affect the program behavior?

6. How could `Program_5`, as described above, be modified to check for a blank (or space) in the first position of input? How would this affect the program behavior?

7. Program_1 checks if the first character is in the subset of characters ranging from 'A' to 'Z.' Examine Program_6 to determine how the logic has changed and what the output will be.

```
[Program_6] =
  [with TEXT_IO;
  procedure Print_Message_2d is
  -- expansion on Print_Message_1 to ask for digit and
  -- return response depending on the user input
    Choice : CHARACTER; -- to hold digit if given
  begin
    TEXT_IO.Put (Item => "Enter a digit => ");
    TEXT_IO.Get (Item => Choice);
    if Choice in '0' .. '9' then
      -- have received digit as requested
      TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
    else -- did not receive digit as requested
      TEXT_IO.Put (Item => "Not a digit!");
    end if;
    TEXT_IO.New_Line;
  end Print_Message_2d;]
```

8. Write the program behavior that describes Program_6.

3.3 Program Parts as Rules for Behavior

3.3.1 Declaration Behavior

Simple intuitive forms of data such as characters or integers are easy to deal with in normal conversation. In communicating the desired operations to a computer program, the specific data types must first be declared. Once a data type is declared, then intuitive and explicit operations can be performed. Thus, data declarations define the data available. For example, the declaration in Program_1

```
Choice : CHARACTER;
```

defines the type of data that may be stored in Choice. Its *declaration behavior* is a *relation,* namely

```
[Choice : CHARACTER;] = {(⟨Input, Output⟩,
  ⟨Input, Output, Choice⟩) | Choice is an ASCII character}
```

that is, none of Input or Output is changed, but a new variable named Choice has been defined, whose value is restricted to be any one of the legal ASCII characters. Again in this case it is a relation, not a function, because there is not a unique mapping from the domain (⟨Input, Output⟩) to the range (⟨Input, Output, Choice⟩).

There is another form of declaration that includes an initialization, such as

```
Choice : CHARACTER := 'a';
```

which in fact creates a *declaration behavior* which is a *function*

```
[Choice : CHARACTER := 'a';] = {⟨⟨Input, Output⟩,
  ⟨Input, Output, Choice⟩⟩ | Choice = 'a'}
```

In this case, it is a function because the value of the new data ⟨Input, Output, Choice⟩ (which is now a single value in the range) is uniquely determined by the previous data ⟨Input, Output⟩ (as defined in the domain) and the assignment operator :=. Note that Choice = 'a' is part of the behavior definition, not part of the argument of the behavior.

Consider the following declarations.

```
type My_Integer is range -99..99;
First, Second : My_Integer := 0;
```

Here, two INTEGER variables First and Second are declared with range between -99 and 99, and initialized to 0. Any attempt to overflow those bounds will stop the execution. For example, if First = 75, Second = -50, the assignment

```
First := First - Second;
```

will overflow First and not be executed.

Another example of type BOOLEAN is

```
Equal : BOOLEAN := TRUE;
```

In this case, Equal is declared as BOOLEAN with initial value TRUE. For example

```
Equal := (First = Second)
```

will assign FALSE to Equal with the values for First, Second above.

3.3.2 Simple Statement Behavior

Simple statements include *procedure call* statements, *assignment* statements, and the **null** statement. For example, the simple procedure call statement

```
TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
```

in Program_1 will define a statement behavior denoted

```
[TEXT_IO.Put (Item => "Welcome Aboard, " & Choice );]
```

with value

```
{⟨⟨Input, Output, Choice⟩, ⟨Input, Output1, Choice⟩⟩ |
  Output1 = Output & "Welcome Aboard, " & Choice}
```

that is, a behavior that maps the initial state of all data, shown here as Input, Output and Choice, to a final state with a new value for Output and the same values for all other data. Note that the message put out is added to whatever is already in Output. In this program, the initial TEXT_IO.Put statement provides the first data to Output, but that fact is not contained in the TEXT_IO.Put statement itself. That is, this program part behavior describes its effect wherever it might appear in a program.

For example, given INTEGER variable `First`, as declared above with range -99 .. 99, the Get statement

```
Int_IO.Get (Item => First);
```

will execute properly only if an INTEGER value in the range is returned. For example, if INTEGER 75 is returned, the result is

```
{⟨⟨Input, Output, First⟩, ⟨Input1, Output, First1⟩⟩ |
  Input1 = Input & 75 and First1 = 75}
```

Again, the value entered is given to `First` and added to the string `Input`.

As another example, if INTEGER 150 is returned, the execution will be terminated because the value is not legal for the variable `First`. In still another case, if a non-INTEGER is returned, say Roman numeral XII, the execution will also be terminated.

The assignment statement

```
Choice := ' ';
```

puts the value of the space character into `Choice`, with statement behavior

```
[Choice := ' ';]

= {⟨⟨Input, Output, Choice⟩, ⟨Input, Output, Choice1⟩⟩ |
    Choice1 = ' '}
```

As might be expected, the **null** statement has no effect on data, with identity statement behavior

```
[null;]

= {⟨⟨Input, Output, Choice⟩, ⟨Input, Output, Choice⟩⟩}
```

3.3.3 Sequence Statement Behavior

As programs are analyzed, logical groupings of statements can be identified by language dependent constructs. The two basic constructs of sequential and conditional execution are represented in most languages. In Ada, the simplest way of representing normal sequential execution is through a series of consecutive statements. A *sequence* of two program parts has a program behavior which is the *composition* of the *part behaviors*. For example, the program part

```
with TEXT_IO;
procedure Print_Message_8 is
begin
  TEXT_IO.Put ("This is the first part; ");
  TEXT_IO.Put ("this is the second part; ");
  TEXT_IO.Put ("this is the last part.");
end Print_Message_8;
```

is the simple composition of each of the individual statements. The part behavior for the sequence of TEXT_IO.Put statements is

```
[TEXT_IO.Put ("This is the first part; ");
TEXT_IO.Put ("this is the second part; ");
TEXT_IO.Put ("this is the last part.");]

= {(⟨Input, Output⟩, ⟨Input, Output1⟩) |
    Output1 = Output & "This is the first part; this is the
    second part; this is the last part."}
```

As a second example, consider a sequence dealing with integer variables First, Second, Third, and Fourth declared as

```
type My_Integer is range -99..99;
First, Second, Third, Fourth : My_Integer := 0;
```

and found in a sequence

```
First := Third + Fourth;
Second := Third - Fourth;
```

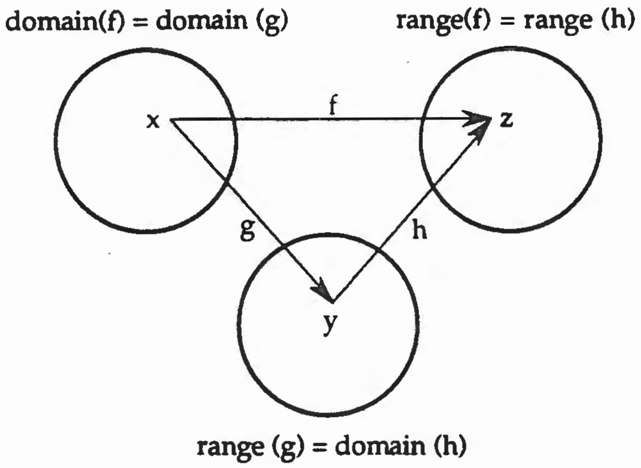
In this case, the values of both expressions Third + Fourth and Third - Fourth must lie in the interval -99 .. 99 for the assignments to be valid. The part function for the sequence is

```
[First := Third + Fourth;
Second := Third - Fourth;]

= {(⟨Input, Output, First, Second, Third, Fourth, ...⟩,
    ⟨Input, Output, First1, Second1, Third, Fourth, ...⟩) |
    First1 = Third + Fourth and Second1 = Third - Fourth and
    First1 in -99 .. 99 and Second1 in -99 .. 99}
∪ {(⟨Input, Output, First, Second, Third, Fourth, ...⟩,
    ⟨Input, Output⟩) | Third + Fourth not in -99 .. 99 or
    Third - Fourth not in -99 .. 99}
```

The behavior of sequences of statements can be described as the mathematical composition of the individual behaviors into the total behavior of the sequence. The underlying mathematics is very simple and straightforward, and can be visualized geometrically as shown next.

A composition of two behaviors into a third behavior can be pictured as in Figure 3.1 in terms of the mappings among their domains and ranges. For the composition of behaviors g and h into f, let g and h be behaviors such that the following relationships hold

domain(f) = domain(g)
range(f) = range(h)
range(g) = domain(h)

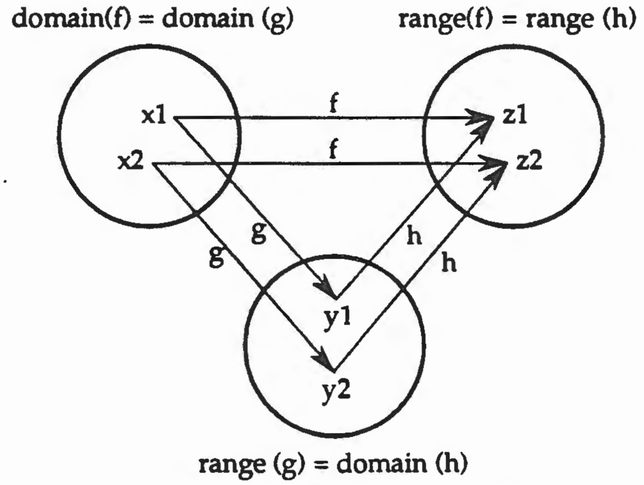


Behavior Composition Mapping
Figure 3.1

Then, for points x, y, z,

```
z = f(x) = h(y) = h(g(x))
```

Furthermore, if x1 and x2 map to separate values z1 and z2 under f, then x1 and x2 must map to separate values y1 and y2 under g as shown in Figure 3.2. Otherwise the behavior h could not map a single value y into separate values z1 and z2.



Distinct Composition Mappings
Figure 3.2

Note that the behaviors are defined so that the domains and ranges exactly fit. In particular, domain (g) = domain (f), is not a superset.

In illustration, let domain (f) be nonnegative integers and some behavior e be considered for the first composition behavior except it has domain of all integers. In this case the subset of e whose behavior domain is the nonnegative integers can be considered instead, calling this subset g. In general, any behavior has subsets which can exactly fit a domain condition.

3.3.4 If Statement Behavior

An if statement behaves as the disjoint *union* of the then part (if the condition is TRUE) and the else part (if the condition is FALSE). These are shown as the behavior given on the right with domain restricted by the condition on the left. The domain limiter (->) is a metasymbol used to show domain restriction, and is not an Ada delimiter.

For example, the part behavior of the if statement

```
if Choice in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Try Again");
end if;
```

is the union of the two disjoint behaviors, each shown with a domain limiter ->

```
Choice in 'A' .. 'Z' ->
[TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);]
```

and

```
Choice not in 'A' .. 'Z' ->
[TEXT_IO.Put (Item => "Try Again");]
```

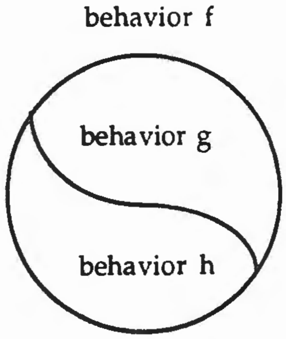
These two disjoint behaviors combine into the single behavior

```
[if Choice in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Try Again");
end if;]

= {⟨⟨Input, Output, Choice⟩
    ⟨Input, Output1, Choice⟩⟩ |
    Choice in 'A' .. 'Z' and Output1 = Output &
    "Welcome Aboard, " & Choice}
∪ {⟨⟨Input, Output, Choice⟩,
    ⟨Input, Output1, Choice⟩⟩ |
    Choice not in 'A' .. 'Z' and Output1 = Output & "Try Again"}
```

The union of two disjoint behaviors, say g and h, into f, can be pictured as a simple partition of the union. For example, let

```
f = g ∪ h when g ∩ h = {}
```

Then behaviors g and h are partitions of behavior f as pictured in Figure 3.3.



Disjoint Behavior Union
Figure 3.3

Note as before that any domain of a behavior can be restricted to a limited domain of a subset behavior. In particular, an **if** statement will define two specific behaviors, one for the **then** part, one for the **else** part, with domains limited to the cases **if** condition TRUE and **if** condition FALSE. The behavior of the statements within the **then** part and **else** part may be much more complicated than shown above. For example, the **if** statement

```
if First < Second then
  First := First + Second;
else
  Second := Second - First;
end if;
```

contains two assignment statements not restricted by the relationship between `First` and `Second`. However, in the **then** part of this **if** statement, it will always hold that `First < Second`, so only the restricted assignment statement

```
First < Second -> First := First + Second;
```

needs be considered, whose behavior is a limited subset of the simple assignment statement itself. Correspondingly, the **else** part defines a restricted assignment

```
First >= Second -> Second := Second - First;
```

In this spirit, the Disjoint Behavior Union of Figure 3.3 illustrates the use of behaviors with exactly the right domains.

3.3.5 Combination of Sequence and If Statement Behaviors

More complex program behaviors can be decomposed into the behaviors of the individual parts. For example, the program part

```
TEXT_IO.Get (Item => Choice);
if Choice in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Try Again");
end if;
```

is a sequence of the two smaller program parts already studied. Following the TEXT_IO.Get statement, the execution continues, with the if statement

```
if Choice in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Try Again");
end if;
```

as already discussed. In this case, the first statement as worked out previously has the part behavior

```
[TEXT_IO.Get (Item => Choice);]

= {(⟨Input, Output, Choice⟩, ⟨Input1, Output, Choice1⟩) |
      Input1 = Input & Char and
      Choice1 = Char}
```

where Char was provided in Input by the user.

The second statement has part behavior given above, that is, Output is augmented with either the string "Welcome Aboard, " & Choice or "Try Again" depending on the value of Choice.

The composition of these two part behaviors will provide the result of their sequential execution, namely

```
[TEXT_IO.Get (Item => Choice);
if Choice in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
else
  TEXT_IO.Put (Item => "Try Again");
end if;]

= {(⟨Input, Output, Choice⟩,
     ⟨Input1, Output1, Choice1⟩) |
     Char in 'A' .. 'Z', and
     Input1 = Input & Char, and
     Choice1 = Char, and
     Output1 = Output & "Welcome Aboard, " & Choice1}
∪ {(⟨Input, Output, Choice⟩,
     ⟨Input1, Output1, Choice1⟩) |
     Char not in 'A' .. 'Z', and
     Input1 = Input & Char, and
     Choice1 = Char, and
     Output1 = Output & "Try Again"}
```

This composition behavior is equivalent to that found before by direct examination. Although this behavioral decomposition requires some analysis and effort, it provides a mechanism by which programs can be understood using the rigor of mathematics. As programs grow, each program part can be analyzed using the same technique and the results combined into subprograms and the subprograms into programs. The level of effort required is dependent on the complexity of the problem, the experience of the author, or reader and the language used. The key point here is to continue this terminal process until the program actions are completely understood.

3.3.6 Exercises

1. Determine the part behavior for the sequence statement

```
TEXT_IO.Put (Item => "Welcome Aboard, " & Choice);
TEXT_IO.Put (Item => "Try Again");
```

2. Determine the part behavior for the sequence statement with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second, Third, Fourth : My_Integer := 0;
```

and found in a sequence

```
First := First + Third + Fourth;
Second := Second - Third - Fourth;
```

3. Determine the part behavior for the if statement with CHARACTER object declared as

```
Choice : CHARACTER := ' ';
```

and found in if statement

```
if Choice in 'A' .. 'Z' then
  Choice := 'A';
else
  TEXT_IO.Put (Item => "Try Again");
end if;
```

4. Determine the part behavior for the if statement with CHARACTER object declared as

```
Digit : CHARACTER := 'M';
```

and found in

```
if Digit = 'M' or Digit = 'L' or Digit = 'C' or Digit = 'D' then
  TEXT_IO.Put (Item => Digit & " is a large roman digit");
else
  if Digit = 'X' or Digit = 'V' or Digit = 'I' then
    TEXT_IO.Put (Item => Digit & " is a small roman digit");
  else
    TEXT_IO.Put (Item => Digit & " is not a roman digit");
  end if;
end if;
```

5. Determine the part behavior for the if statement with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second, Third, Fourth : My_Integer := 0;
```

and found in

```
if First < Second then
  Third := Fourth;
else
  if Third < Fourth then
    First := Second;
  else
    Third := First;
    Fourth := Second;
  end if;
end if;
```

6. Determine the part behavior for the sequence statement with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second, Third, Fourth : My_Integer := 0;
```

and found in

```
Third := First;
Fourth := Second;
if First < Second then
  Third := Fourth;
  Fourth := First;
else
  First := Fourth;
  Second := Third;
end if;
```

7. Determine the part behavior for the sequence statement with CHARACTER objects declared as

```
Digit, Big_Digit, Small_Digit : CHARACTER := 'M';
```

and found in

```
Big_Digit := 'M';
Small_Digit := 'I';
if Digit = Big_Digit then
  Digit := Small_Digit;
  Small_Digit := 'V';
else
  if Digit = Small_Digit then
    Digit := Big_Digit;
    Big_Digit := 'L';
  else
    Big_Digit := Digit;
    Small_Digit := Digit;
  end if;
end if;
```

8. Determine the part behavior for the sequence statement with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second, Third, Fourth, Fifth, Sixth : My_Integer := 0;
```

and found in

```
Third := First;
Fourth := Second;
if Fifth < Sixth then
  Third := Sixth;
  Fourth := Fifth;
  if First < Second then
    Fifth := Fourth;
    Sixth := First;
  else
    Sixth := Fourth;
    Fifth := Third;
  end if;
else
  First := Fourth;
  Second := Third;
end if;
```

3.4 For Loop Statement Behavior

Loop statements can be defined as either *for loops* or *while loops*. For loops define an exact number of iterations to be carried out, using a loop identifier which is defined only inside the **loop**. As noted in Chapter 2, this exact number of iterations can alter from use to use of the for loop, but is fixed and determined within each use. While loops define a variable number of iterations based on the data used. While loops can define an endless number of iterations. Although usually a mistake, there are programs that are intended to never terminate, like programs that control a telephone system.

For loops and while loops have different levels of theoretical difficulties. Since for loops define an exact number of iterations to be carried out, they define sequences of fixed length in a special form. There is no question of termination with for loops, even though the iterations may be complex to work out and analyze. On the other hand, while loops define sequences of variable length that are defined by the **while** condition. In summary, for loops always terminate as given explicitly in the **for** statement header, but while loops may or may not terminate.

In each case, we deal with how statement behavior is defined, then discuss properties of these respective loops. It will be noted that for loops are special forms for sequence statements, and while loops are more general. As a consequence, the behavior of for loops can be described in terms of the behavior of sequence statements. But the behavior of while loops are more general. The behavior of for loops can also be described in terms of the behavior of initialized while loops.

A for loop statement has program behavior closely related to a sequence of statements. This sequence of statements has two parts, first the sequence of statements that comprise the body of the loop and second the sequence representing the number of times the body of the loop is repeated. In this case the statements in the first sequence are all identical except for the value of the for loop identifier, which is incremented (positively or negatively) with each iteration.

For example, consider the following for loop.

```
Get_And_Put_Digits:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;
end loop Get_And_Put_Digits;
```

In this example, the first sequence has four statements, of which the third is an **if** statement. The second sequence has ten copies, one for each time through the loop. So the execution of `Get_And_Put_Digits` will execute forty first level statements. The for loop behavior is given by these forty first level statements executed in proper sequence. As a result, the behavior of

```
[Get_And_Put_Digits:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;
end loop Get_And_Put_Digits;] =

 [TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;

  -- eight more identical copies plus final copy below

  TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;]
```

This sequence logic has no for loop notation in it but will define the identical sequence of forty statements in execution. Each if statement identifies separately whether `Next_Character` is a digit or not and puts out an appropriate message. Of course it is longer, so the for loop pays off in that respect for reading and program storage.

In order to carry this out fully, we expand the forty statements directly. Each time through the for loop can be expressed in a single line of logic in an expression as follows. The 10 parts added to Input and Output are denoted I1, I2, ..., I10 and O1, O2, ... O10.

```
[Get_And_Put_Digits:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;
end loop Get_And_Put_Digits;]

= {⟨⟨Input, Output, Next_Character⟩,
    ⟨Input1, Output1, Next_Character1⟩⟩ |
    Input1 = Input & I1 & I2 & I3 & I4 & I5 & I6 & I7 & I8 & I9 & I10
    and I1, I2, I3, I4, I5, I6, I7, I8, I9, I10 are CHARACTERs
    and Next_Character1 = I10
    and Output1 = Output & O1 & O2 & O3 & O4 & O5 & O6 & O7 & O8 & O9
    & O10
    where
    O1 = I1 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O2 = I2 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O3 = I3 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O4 = I4 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O5 = I5 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O6 = I6 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O7 = I7 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O8 = I8 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O9 = I9 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    O10 = I10 & " is " & (⟨ ⟩, "not ") & "a digit" & New_Line
    and in each line defining O1, .. O10, ⟨ ⟩ or "not" is chosen on
    whether the CHARACTER in I1, I2, I3, I4, I5, I6, I7, I8, I9, I10
    is a digit or not}
```

Note that `Loop_Counter` does not appear in this final result, being an internal counter in the for loop. This expression is quite general for a for loop behavior. The for loop may be much longer, say a hundred or a thousand steps. But the result will be of the same structure. The body of the for loop may be much larger, say a hundred or a thousand simpler statements with many internal variables. So its analysis may take more time and effort. But some one must know what those internals are. So just as sequences and if statements define a standard means of analysis, so does the for loop.

3.4.1 Properties of For Loop Statements

As noted, for loop statements have properties that come exactly from the sequences of statements they define. In the example above, the internal counter Loop_Counter was used only to count progress through the iterations. But an internal counter can be sampled and used in each iteration, when useful. For example, consider a for loop that follows, whose objective is to find digits in Input and to ignore non digits.

```
Find_Digit_Locations:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter);
    TEXT_IO.New_Line;
  end if;
end loop Find_Digit_Locations;
```

In this case, a new line is produced if and only if Next_Character is a digit, which may happen any where between zero and ten times. Having stated that, is that really the case? First, note that only

```
TEXT_IO.Get (Item => Next_Character);
```

is outside the if statement, and will be exercised every time through the for loop. The three Put and New_Line statements are inside the if statement and are exercised only if Next_Character is a digit. Thus a digit value in Next_Character will create a new line.

As shown above, the effect of this for loop can be restated as a sequence. In this case counter Loop_Counter is reintroduced as an ordinary variable Loop_Counter_1 rather than the for loop counter because its value will be used in creating each line. (Recall that Loop_Counter cannot exist except in the for loop.)

```
[Find_Digit_Locations:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter);
    TEXT_IO.New_Line;
  end if;
end loop Find_Digit_Locations;] =

  [Loop_Counter_1 := 1;
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter_1);
    TEXT_IO.New_Line;
  end if;

  -- eight more copies incrementing Loop_Counter_1 and final copy
```

```
    Loop_Counter_1 := 10;
    TEXT_IO.Get (Item => Next_Character);
    if Next_Character >= '0' and Next_Character <= '9' then
      TEXT_IO.Put (Item => Next_Character);
      TEXT_IO.Put (Item => " is a digit in position ");
      INT_IO.Put (Item => Loop_Counter_1);
      TEXT_IO.New_Line;
    end if;]

  = {⟨⟨Input, Output, Next_Character⟩,
      ⟨Input1, Output1, Next_Character1⟩⟩|
      Input1 = Input & I1 & I2 & I3 & I4 & I5 & I6 & I7 & I8 & I9 & I10
      and I1, I2, I3, I4, I5, I6, I7, I8, I9, I10 are CHARACTERs
      and Next_Character1 = I10
      and Output1 = Output & O1 & O2 & O3 & O4 & O5 & O6 & O7 & O8 & O9
      & O10
      where
      O1 = (I1 & " is a digit in position 1" & New_Line, ⟨ ⟩)
      O2 = (I2 & " is a digit in position 2" & New_Line, ⟨ ⟩)
      O3 = (I3 & " is a digit in position 3" & New_Line, ⟨ ⟩)
      O4 = (I4 & " is a digit in position 4" & New_Line, ⟨ ⟩)
      O5 = (I5 & " is a digit in position 5" & New_Line, ⟨ ⟩)
      O6 = (I6 & " is a digit in position 6" & New_Line, ⟨ ⟩)
      O7 = (I7 & " is a digit in position 7" & New_Line, ⟨ ⟩)
      O8 = (I8 & " is a digit in position 8" & New_Line, ⟨ ⟩)
      O9 = (I9 & " is a digit in position 9" & New_Line, ⟨ ⟩)
      O10 = (I10 & " is a digit in position 10" & New_Line, ⟨ ⟩)
      and in each line the message or ⟨ ⟩ is chosen on whether the
      CHARACTER in I1, I2, I3, I4, I5, I6, I7, I8, I9, I10 is a digit
      or not}
```

This example is completely general as far as the for loop counter is concerned. As already noted in Chapter 2, the number of items processed is fixed in each execution of the for loop, but can be varied from use to use of the for loop. So any question about the counter can be resolved by mapping it as an ordinary variable into a sequence as shown above.

3.4.2 Exercises

1. Determine what the following variation of `Get_And_Put_Digits` does.

```
Get_And_Put_Digits_1:
for Loop_Counter in reverse 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Digit);
  TEXT_IO.Put (Item => Next_Digit);
  if Next_Digit in '0' .. '9' then
    TEXT_IO.Put (Item => " is a digit");
  else
    TEXT_IO.Put (Item => " is not a digit");
  end if;
  TEXT_IO.New_Line;
end loop Get_And_Put_Digits_1;
```

2. Determine what the following variation of `Get_And_Put_Digits` does.

```
Get_And_Put_Digits_2:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  TEXT_IO.Put (Item => Next_Character);
  if Next_Character < '0' or Next_Character > '9' then
    TEXT_IO.Put (Item => " is not a digit");
  else
    TEXT_IO.Put (Item => " is a digit");
  end if;
  TEXT_IO.New_Line;
end loop Get_And_Put_Digits_2;
```

3. Work out the behavior of `Get_And_Put_Digits_1`.

4. Work out the behavior of `Get_And_Put_Digits_2`.

5. Define `Find_Digit_Location_1` in which only one trial is made in the for loop, so the **for** line reads as

```
for Loop_Counter in 1 .. 1 loop
```

and determine its behavior.

6. Define `Find_Digit_Location_2` in which only one successful trial is made in the for loop, so the **for** line reads as

```
for Loop_Counter in 1 .. 10 loop
```

but an additional BOOLEAN variable called `Done` is used, initialized FALSE before the for loop, set TRUE if a digit is found, and printed only the first time a digit is found, if ever. Determine its behavior.

3.5 While Loop Statement Behavior

A while loop statement has a program behavior that is more complex than sequence, **if**, or **for** statements. The **loop** part may be executed zero or more times, depending on the data and the **while** condition. In some cases it may be readily seen just how many iterations will take place based on the data. If the **while** condition is FALSE for the data, the **loop** part will not be executed and the while loop statement will behave like a **null** statement. If the **while** condition is TRUE for all possible data, the while loop statement will never terminate and the statement behavior will be the empty set. If the **while** condition remains TRUE for all iterations from certain initial data, the while loop statement behavior will not be defined for that initial data. That is, the domain of a while loop statement function is the set of initial data states for which the loop terminates. For example, as already noted in Chapter 2, a while loop statement such as

```
Dumb_Loop:
while Value < 5 loop
  Result := 1;
end loop Dumb_Loop;
```

will never terminate if `Value < 5` is true initially. Usually, such a while loop statement is a mistake, but in more complex forms this error may not be so easy to discover.

For example, consider the following while loop statement

```
Character_Search:
while Choice not in 'A' .. 'Z' loop
  -- Better Look at Next Character
  TEXT_IO.Get (Item => Choice);
end loop Character_Search;
```

The while loop statement is looking for an upper case letter. But if `Choice` is an upper case letter to begin with the loop is never entered and the statement will terminate immediately. As a result the program part behavior becomes

```
[Character_Search:
while Choice not in 'A' .. 'Z' loop
  -- Better Look at Next Character
  TEXT_IO.Get (Item => Choice);
end loop Character_Search;]

= {⟨⟨Input, Output, Choice⟩, ⟨Input, Output, Choice⟩⟩ |
    Choice >= 'A' and Choice <= 'Z'}
∪ {⟨⟨Input, Output, Choice⟩, ⟨Input1, Output, Choice1⟩⟩ |
    Choice < 'A' or Choice > 'Z' and Input1 = Input & String and
    Choice1 = h(reverse(String)) and Choice1 >= 'A' and
    Choice1 <= 'Z' and Choice1 is first value to lie between
    'A' and 'Z' in String}
```

If no uppercase letter ever appears from the user, `Character_Search` never terminates. In this case, some natural language is used to describe the behavior. As behaviors become more complex, natural language is needed, but the underlying ideas of behaviors are still valid. Note that various options are available in representing data and conditions. For example, in Character_Search, the condition

```
Choice not in 'A'..'Z'
```

is also shown as

```
Choice < 'A' or Choice > 'Z'
```

in its behavior. This may or may not be a good idea depending on the purpose of the writing and the capabilities of the readers.

As another example, consider the for loop `Find_Digit_Locations` above repeated here.

```
Find_Digit_Locations:
for Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter);
    TEXT_IO.New_Line;
  end if;
end loop Find_Digit_Locations;
```

Counter `Loop_Counter` must be redefined as an ordinary variable, say as `Loop_Counter_1`, and initialized to start with, then incremented each time through the while loop. We also alter the loop name to `Find_Digit_Locations_1`. The initialized while loop form follows.

```
Loop_Counter_1 := 1;
Find_Digit_Locations_1:
while Loop_Counter_1 in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter_1);
    TEXT_IO.New_Line;
  end if;
  Loop_Counter_1 := Loop_Counter_1 + 1;
end loop Find_Digit_Locations_1;
```

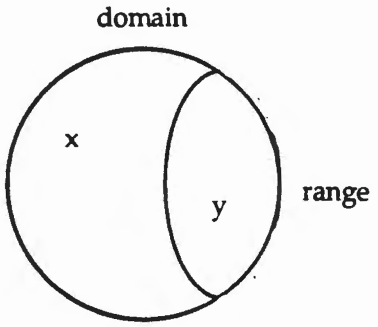
Note that the while loop condition is very different from the for loop counter, but very similar in form. The only difference between these two lines is for is replaced by while. The behavior of the initialized while loop is as follows.

```
[Loop_Counter := 1;
Find_Digit_Locations_1:
while Loop_Counter in 1 .. 10 loop
  TEXT_IO.Get (Item => Next_Character);
  if Next_Character >= '0' and Next_Character <= '9' then
    TEXT_IO.Put (Item => Next_Character);
    TEXT_IO.Put (Item => " is a digit in position ");
    INT_IO.Put (Item => Loop_Counter);
    TEXT_IO.New_Line;
  end if;
  Loop_Counter := Loop_Counter + 1;
end loop Find_Digit_Locations_1;]

= {⟨⟨Input, Output, Next_Character, Loop_Counter⟩,
    ⟨Input1, Output1, Next_Character1, Loop_Counter1⟩⟩|
    Input1 = Input & I1 & I2 & I3 & I4 & I5 & I6 & I7 & I8 & I9 & I10
    and I1, I2, I3, I4, I5, I6, I7, I8, I9, I10 are CHARACTERs
    and Next_Character1 = I10 and Loop_Counter1 = 11
    and Output1 = Output & O1 & O2 & O3 & O4 & O5 & O6 & O7 & O8 & O9
    & O10
    where
    O1 = (I1 & " is a digit in position 1" & New_Line, ⟨ ⟩)
    O2 = (I2 & " is a digit in position 2" & New_Line, ⟨ ⟩)
    O3 = (I3 & " is a digit in position 3" & New_Line, ⟨ ⟩)
    O4 = (I4 & " is a digit in position 4" & New_Line, ⟨ ⟩)
    O5 = (I5 & " is a digit in position 5" & New_Line, ⟨ ⟩)
    O6 = (I6 & " is a digit in position 6" & New_Line, ⟨ ⟩)
    O7 = (I7 & " is a digit in position 7" & New_Line, ⟨ ⟩)
    O8 = (I8 & " is a digit in position 8" & New_Line, ⟨ ⟩)
    O9 = (I9 & " is a digit in position 9" & New_Line, ⟨ ⟩)
    O10 = (I10 & " is a digit in position 10" & New_Line, ⟨ ⟩)
    and in each line the message or ⟨ ⟩ is chosen on whether the
    CHARACTER in I1, I2, I3, I4, I5, I6, I7, I8, I9, I10 is a digit
    or not}
```

For this simple problem, we have seen three possible solutions: a sequence, a for loop, and a while loop. Which is best? In this case, the for loop is most likely best, but in other cases any of the three may be preferred.

3.5.1 Properties of While Loop Statements

While loop statements and their statement behaviors have certain mathematical properties that are easy to see. The first property of the behavior is that the range of any while loop statement behavior is a subset of its domain. Such a relation is not generally true in behaviors, but is true in any while loop statement behavior. The reasoning behind this statement is easy and direct. Consider any state of data for which the **while** condition is FALSE. This state is in the range of the behavior because the statement will terminate from that state. This state is also in the domain of the behavior from that state as initial data because if execution starts in this state the statement will terminate immediately. In the other direction, any point in the range must have the **while** condition FALSE, because otherwise the while loop could not terminate with that data. This subset relation between the range and domain is shown next in Figure 3.4 with an initial state x in the domain and a final state y in the range.



While Loop Behavior Domain and Range
Figure 3.4

This property that the range is a subset of the domain of any while loop statement behavior helps define what behaviors are possible for while loop statements. In particular, if the range is not a subset of the domain of a behavior, then no while loop statement is possible for that behavior. For example, consider the behavior of incrementing an integer variable by one, say

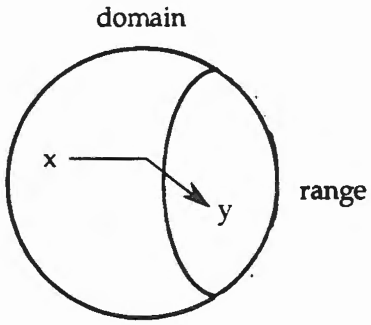
```
   Sum := Sum + 1;
```

where `Sum` has been declared

```
   Sum : INTEGER range 0 .. 100 := 0;
```

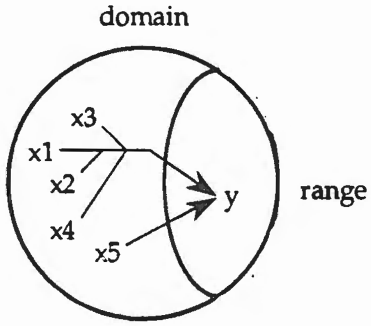
This behavior `[Sum := Sum + 1;]` has domain 0 .. 99 and range 1 .. 100, so the range is not a subset of the domain. This means that no while loop statement exists that can achieve this behavior. This is a simple example and a while loop statement is not really needed. But it illustrates a logical requirement on any behavior for a while loop statement to exist to achieve it.

A second property of a while loop statement is that no data in the domain of its behavior can be reached again in subsequent iterations. If such a state were reached more than once, the execution will never terminate, repeating its path through that state over and over. Stated positively, the **loop** part must make progress on each application to the final state in the range of the behavior, and never cycle to a previous state. This means that starting at any state of data in the domain of the while loop statement behavior, there is a unique, finite path to the range of the behavior. Every point on this path is also a possible initial state, including the final point in the range. And only the final point of the path is in the range, because the **while** condition becomes FALSE immediately on entering the range. As already noted, one possible path is of zero length, which starts in the range and remains there because the **while** condition is FALSE and the **loop** part is never entered. A general path from initial state x to final state y is shown next in Figure 3.5.



While Loop Behavior Path
Figure 3.5

Now, as noted, every point on the path from x to y is another potential initial state of data which the while loop statement will take to the same final state y. But there may be other paths that join this path at these points as well, making up a *computing tree* with its root at y in the range and with branches and leaves in the domain, as shown in Figure 3.6. Note, however, that while different paths may reach final state y, no branches in these paths can exist in the range. Each final step to y must be made from a point outside the range to a point inside the range.



While Loop Behavior Tree Branch
Figure 3.6

A third property of a while loop statement is that the behavior is always the union of two simpler behaviors, first, an identity behavior as noted above where the **while** condition is FALSE, and a second behavior with no identity parts from the set difference of domain – range to the range, whose values are computed by the **while** statement by going down the tree branches as noted above. The while loop behavior has no iteration, of course. The iteration of the while loop is simply part of one rule that describes the while loop behavior.

3.5.2 Exercises

1. Consider the while loop statement above with the **not** condition on the loop condition missing

```
Character_Search_1:
while Choice in 'A' .. 'Z' loop
  -- Better Look at Next Character
  TEXT_IO.Get (Item => Choice);
end loop Character_Search_1;
```

   What is its statement behavior?

2. Consider the while loop statement above with `Choice` initialized

```
Choice := ' ';
Character_Search:
while
loop  -- Better Look at Next Character
  TEXT_IO.Get (Item => Choice);
end loop Character_Search_2;
```

   What is its statement behavior?

3. Determine the while loop statement behavior with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second : My_Integer := 0;
```

   and found in a while loop statement

```
Adding:
while First < Second loop
  First := First + 1;
end loop Adding;
```

4. Determine the while loop statement behavior with INTEGER objects declared as

```
type My_Integer is range -99..99;
First, Second : My_Integer := 0;
```

   and found in a while loop statement

```
Adding_And_Subtracting:
while First < Second
loop
  First := First + 1;
  Second := Second - 1;
end loop Adding_And_Subtracting;
```

5. Which of the following behaviors can be realized by while loop statements?

   a. Finding two consecutive uppercase letters in `Input`.
   b. Exchanging values of INTEGER object `First` and `Second`.
   c. Adding the value of INTEGER object `First` to INTEGER object `Second` only if `First` is positive.
   d. Incrementing an object in a cycle, so the maximum value is transformed to the minimum value.

6. What are the domains and ranges for the part behaviors of Exercises 4 and 5 above? Are the ranges subsets of the domains?

Chapter 4

Software Analysis

As already noted, any Ada program is a rule for behavior as a function or relation. If the program determines a unique output for every input, it is a rule for a function. If the program does not determine a unique output for every input, say because an object is not initialized before using it, the program is a rule for a relation. For the moment, we will concentrate on programs that determine functions, and consider non-unique behavior as an exceptional case that probably needs fixing.

For small and simple programs, the rule for the behavior will be relatively easy to identify, and the behavior itself relatively easy to describe. But for large and complex programs, it will be more difficult to discover their behaviors. Of course any program itself defines its behavior, but it will usually be of interest to find another description of the behavior in order to understand independently what the program does.

In this chapter, we introduce techniques of recording and analysis for determining program behavior and program part behavior. For each kind of executing program part, simple assignments, procedure calls, sequences, if statements, for and while loops, we introduce specific methods of recording their sub-parts and determining their behavior. We introduce graphic and tabular forms to identify and carry out analyses that determine part behavior. We also introduce ways to combine the behaviors of various parts into the behaviors of complete programs.

4.1 Reading Programs for their Behaviors

No matter how large or complex a program may be, it can be read in small steps by determining lower level part functions and incorporating the results into the next level until the entire program is understood. However, for intellectual control a top down plan and recording is critical.

4.1.1 Hierarchical Structures of Ada Programs

Ada programs are made up of various parts, declarations and statements, the latter either simple or compound. Compound statements, in turn, are made from expressions and other statements, each either simple or compound. Each such part has a part behavior. The analysis of an entire program provides its program behavior.

In illustration, consider the following, perfectly legal Ada program, and the explicit discovery of its hierarchical structure.

```
with TEXT_IO;procedure Mystery_Program_5 is Choice:CHARACTER := ' ';
Tries:INTEGER := 0;begin Search_For_Upper_Case_Letter:while not
('A' <= Choice and Choice <= 'Z') and Tries <= 10 loop TEXT_IO.Get
(Item => Choice);Tries := Tries + 1;end loop
Search_For_Upper_Case_Letter;if 'A' <= Choice and Choice <= 'Z' then
TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
else if 'a' <= Choice and Choice <= 'z' then TEXT_IO.Put
(Item => "Lower Case Data Input " & Choice);else TEXT_IO.Put
(Item => "Try Again");end if;end if;end Mystery_Program_5;
```

Now, declarations are completed and no pointers remain there. In the executable part, two statements in sequence are given, one for a while loop statement, the other for an if statement. Next, adding the third level

```
with TEXT_IO;
procedure Mystery_Program_5 is
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10 loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z' then
    ...
    else
    ...
    end if;
  end if;
end Mystery_Program_5;
```

At this level, the while loop statement is completed, the **then** part of the **if** statement is completed, but the **else** part has another **if** statement framework with both then part and **else** part yet to be completed. Finally, the fourth level completes the hierarchy and recovers the original program.

```
with TEXT_IO;
procedure Mystery_Program_5 is
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10 loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z' then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
end Mystery_Program_5;
```

4.1.2 Bottom Up Analysis to Determine a Program Behavior

A hierarchical structure for an Ada program can be analyzed and low level part behaviors discovered and combined into larger part behaviors from the bottom up until the program behavior is discovered. For example, `Mystery_Program_5` has a low level if statement

```
if1 = if 'a' <= Choice and Choice <= 'z' then
        TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
      else
        TEXT_IO.Put (Item => "Try Again");
      end if;
```

whose part behavior can be identified as

```
[if1]

= {(⟨Input, Output, Choice, Tries, …⟩,
    ⟨Input, Output & "Lower Case Data Input " & Choice,
    Choice, Tries …⟩) |
    'a' <= Choice and Choice <= 'z'}
∪ {(⟨Input, Output, Choice, Tries, …⟩,
    ⟨Input, Output & "Try Again", Choice, Tries, …⟩) |
    not ('a' <= Choice and Choice <= 'z')}
```

or more simply,

```
[if1]

= {(⟨Input, Output, Choice, Tries, …⟩,
    ⟨Input, Output1, Choice, Tries, …⟩)}
```

where

```
if 'a' <= Choice and Choice <= 'z',
  Output1 = Output & "Lower Case Data Input " & Choice,

if not ('a' <= Choice and Choice <= 'z'),
  Output1 = Output & "Try Again"
```

Next, this if statement `if1` is part of a larger if statement `if2`, where

```
if2 = if 'A' <= Choice and Choice <= 'Z' then
        TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
      else
        if1
      end if;
```

whose part behavior can be identified as

```
[if2]

= {(⟨Input, Output, Choice, Tries, …⟩
    ⟨Input, Output2, Choice, Tries, …⟩)}
```

where

```
if 'A' <= Choice and Choice <= 'Z',
  Output2 = Output & "Welcome Aboard " & Choice

if not ('A' <= Choice and Choice <= 'Z') and
  'a' <= Choice and Choice <= 'z',
    Output2 = Output & "Lower Case Data Input " & Choice,

if not ('A' <= Choice and Choice <= 'Z') and
  not ('a' <= Choice and Choice <= 'z'),
    Output2 = Output & "Try Again"
```

The while loop statement just before the outer if statement `if2` can be analyzed next, namely

```
whl = Search_For_Upper_Case_Letter:
      while not ('A' <= Choice and Choice <= 'Z') and
          Tries <= 10 loop
        TEXT_IO.Get (Item => Choice);
        Tries := Tries + 1;
      end loop Search_For_Upper_Case_Letter;
```

With some analysis, it is clear what this initialized loop will do. It will get characters while necessary up to ten tries for `Input` and `Choice`, looking for an upper case letter. If it finds a `Choice` such that `('A' <= Choice and Choice <= 'Z')`, it will terminate. If it finds no such value in ten tries, it will terminate. The initialization of `Choice` at declaration as ' ' means the loop will seek an upper case letter. As a result, the part behavior for `whl` can be worked out as follows

```
[whl]

= {⟨⟨Input, Output, …, Choice, Tries, …⟩,
    ⟨Input1, Output, …, Choice1, Tries1, …⟩⟩}
```

where

```
if ft(reverse Input1)) = Char in 'A' .. 'Z',
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char, 1 <= Tries1 <= 10

if ft(reverse Input1)) = Char in 'a' .. 'z' and Tries1 = 10,
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char,

if ft(reverse Input1)) = Char not in 'a' .. 'z' and Tries1 = 10,
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char,
```

Finally, the entire executable part of the program is the sequence

```
seq = begin
        whl
        if2
      end Mystery_Program_5;
```

and

```
[seq]

= {⟨⟨Input, Output, …, Choice, Tries …⟩,
    ⟨Input1, Output2, …, Choice1, Tries1,…⟩⟩}
```

which will be worked out next.

First, whl, as noted above, will define an intermediate state of data for Input, Choice, and Tries with the values repeated from above as

```
[whl]

= {⟨⟨Input, Output, …, Choice, Tries, …⟩,
    ⟨Input1, Output, …, Choice1, Tries1, …⟩⟩}
```

where

```
if ℏ(reverse Input1)) = Char in 'A' .. 'Z',
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char, 1 <= Tries1 <= 10

if ℏ(reverse Input1)) = Char in 'a' .. 'z' and Tries1 = 10,
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char,

if ℏ(reverse Input1)) = Char not in 'a' .. 'z' and Tries1 = 10,
no preceding member of Input1 in 'A' .. 'Z',
  Choice1 = Char,
```

Next, this data is treated by if2. Note in working out this next step on the effect of if2 on the current data.

```
[if2]

= {⟨⟨Input, Output, Choice1, Tries1 …⟩
    ⟨Input, Output2, Choice1, Tries1 …⟩⟩}
```

where

```
if 'A' <= Choice1 and Choice1 <= 'Z',
  Output2 = Output & "Welcome Aboard " & Choice1

if not ('A' <= Choice1 and Choice1 <= 'Z') and
  'a' <= Choice1 and Choice1 <= 'z',
    Output2 = Output & "Lower Case Data Input" & Choice1,

if not ('A' <= Choice1 and Choice1 <= 'Z') and
  not ('a' <= Choice1 and Choice1 <= 'z'),
    Output2 = Output & "Try Again"
```

In summary, the program behavior of the entire program, including the with clause and declaration, both used implicitly in the derivation of the sequence part behavior, is

```
[Mystery_Program_5]

= {⟨⟨Input, Output⟩, ⟨Input1, Output2⟩⟩}
```

where

```
if ℏ(reverse (Input1)) in 'A' .. 'Z')
length (Input1) <= 10,
no preceding member of Input in 'A' .. 'Z'
  Output2 = Output & "Welcome Aboard " & ℏ(reverse (Input1))

if ℏ(reverse (Input1)) in 'a' .. 'z'
length (Input1) = 10
no preceding member of Input1 in 'A' .. 'Z',
  Output2 = Output & "Lower Case Data Input" & ℏ(reverse (Input1))

if ℏ(reverse (Input1)) not in 'a' .. 'z'
length (Input) = 10
no preceding member of Input1 in 'A' .. 'Z',
  Output2 = Output & "Try Again"
```

In this case, there is no reference to `Choice` or `Tries`.

4.1.3 Program Behaviors that Include Exception Handling

When program execution encounters exceptions, for example in Get operations or in arithmetic overflow, the program may terminate, but still has a program behavior. For example, `Mystery_Program_5` can be altered to limit the possible size of `Tries` to at most 5, calling the modified program `Mystery_Program_5a`. It will have a different program behavior as worked out next.

```
with TEXT_IO;
procedure Mystery_Program_5a is
  Choice: CHARACTER := ' ';
  type My_Integer is range 0..5;
  Tries: My_Integer := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10 loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z' then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
end Mystery_Program_5a;
```

In this case, the only difference in this new program from the old one is in the while loop, say `whla`

```
Search_For_Upper_Case_Letter:
while not ('A' <= Choice and Choice <= 'Z') and
    Tries <= 10 loop
  TEXT_IO.Get (Item => Choice);
  Tries := Tries + 1;
end loop Search_For_Upper_Case_Letter;
```

Although the while text is identical as before, so `Tries` can seem to go up to the value 10 in the while loop, the program will fail if an attempt to increase it past 5 because of the changed declaration. As it is written, the program behavior is quite different than for `Mystery_Program_5`. In particular, only one of the three possible outputs is possible, namely the first in finding a capital letter, but in five tries instead of ten. If the while loop attempts to go past 5 for `Tries`, the program terminates right there and does not finish, for example does not enter the if part. The program does terminate, just does not complete the normal execution in this case. As a result, the program behavior is as follows.

```
[Mystery_Program_5a]

= {⟨⟨Input, Output⟩, ⟨Input1, Output2⟩⟩}
```

where

```
if ℏ(reverse (Input1)) in 'A' .. 'Z')
length (Input1) <= 5,
no preceding member of Input1 in 'A' .. 'Z'
  Output2 = Output & "Welcome Aboard " & ℏ(reverse (Input))
```

In all other cases there will be no output, nor even finishing the normal execution of the program.

As introduced in Chapter 2, exceptions can be introduced in the program, in this case as follows in `Mystery_Program_5b`.

```
with TEXT_IO;
procedure Mystery_Program_5b is
  Choice: CHARACTER := ' ';
  type My_Integer is range 0..5;
  Tries: My_Integer := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10 loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z' then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z' then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
```

```
exception
  when others => TEXT_IO.Put (Item =>
    "Execution Halted, too many Tries.");
end Mystery_Program_5b;
```

Now, if more than 5 `Tries` are attempted, a message is returned before terminating the program normally.

4.1.4 Assignment Behaviors

Assignment statements have part behaviors that effect the objects on the left side only. But the expressions on the right side can be of any level of complexity. Expressions in INTEGER types may also overflow and lead to NUMERIC_ERROR or CONSTRAINT_ERROR exceptions and termination of execution. For example, let objects `X`, `Y`, `Z` be declared as INTEGERS as follows.

```
type My_Integer is INTEGER range 0..999;
X, Y, Z: My_Integer;
```

with a sequence of assignments

```
X := 500;
Y := 300;
Z := X + 2 * Y;
```

which, of course, leads to an overflow and NUMERIC_ERROR or CONSTRAINT_ERROR exception and possibly termination of execution (if no exception handler exists). As a result, the assignment statements in INTEGER types will lead to behaviors dealing with both normal assignments and NUMERIC_ERROR or CONSTRAINT_ERROR exceptions. For example, if no such exception handler exists,

```
[Z := X + 2 * Y;]

= {(⟨Input,Output, ..., X, Y, Z⟩, ⟨Input,Output, ..., X, Y, Z1⟩) |
     Z1 = X + 2 * Y and 0 <= X + 2 * Y <= 999}
∪ {(⟨Input,Output, ..., X, Y, Z⟩, ⟨Input, Output⟩) |
     not (0 <= X + 2 * Y <= 999)}
```

4.1.5 Exercises

1. In `Mystery_Program_5`, the **while** condition is changed to remove the first not in

   ```
   while not ('A' <= Choice and Choice <= 'Z') and
         Tries <= 10
   ```

   more specifically to

   ```
   while ('A' <= Choice and Choice <= 'Z') and
         Tries <= 10
   ```

   Determine program behavior [`Mystery_Program_5c`].

2. In Mystery_Program_5, Choice is initialized in its declaration, so the assignment

```
Choice := ' ';
```

is not necessary. But if a reader does not notice this in looking at the while loop whl, what might the writer have done to help?

3. Given the following program in compact form, reformat into proper lines and indentation, then provide comments in the procedure to describe its behavior.

```
with TEXT_IO; procedure Triangles is type My_Integer is range
0..1_000; Side_1, Side_2, Side_3, Temp: My_Integer; package Int_IO
is new TEXT_IO.INTEGER_IO (My_Integer); begin Temp := 5;
Search_Input_For_Triangles: while Temp > 0 loop Int_IO.Get
(Item => Side_1); Int_IO.Get (Item => Side_2); Int_IO.Get
(Item => Side_3); TEXT_IO.New_Line; TEXT_IO.Put (Item =>
"Sides:"); Int_IO.Put (Item => Side_1); Int_IO.Put (Item =>
Side_2); Int_IO.Put (Item => Side_3); if Side_2 >= Side_1 and
Side_2 >= Side_3 then Temp := Side_2; Side_2 := Side_1;
Side_1 := Temp; else if Side_3 >= Side_1 and Side_3 >= Side_2
then Temp := Side_3; Side_3 := Side_1; Side_1 := Temp; end if;
end if; if Side_1 > Side_2 + Side_3 then TEXT_IO.Put (Item =>
" do not make up a triangle."); else if Side_1**2 = Side_2**2 +
Side_3**2 then TEXT_IO.Put (Item => " make up a right triangle.");
else if Side_1**2 < Side_2**2 + Side_3**2 then TEXT_IO.Put
(Item => " make up an acute triangle."); else TEXT_IO.Put (Item =>
" make up an obtuse triangle."); end if; end if; end if;
TEXT_IO.Get (Item => Temp); end loop Search_Input_For_Triangles;
end Triangles;
```

4. Determine the program behavior [Triangles] under the condition that Input contains

```
3   4   5
2
9  10  11
0
```

5. Determine the program behavior [Triangles] under the condition that Input contains

```
3   4   5
2   9  10
```

6. Given the following program in compact form, reformat into proper lines and indentation.

```
with TEXT_IO; procedure XYZ is X, Y, Z, T : CHARACTER; begin
TEXT_IO.Get (Item => X); TEXT_IO.Get (Item => Y); TEXT_IO.Get
(Item => Z); if Y >= X and Y >= Z then T := Y; Y := X; X := T;
else if Z >= X and Z >= Y then T := Z; Z := X; X := T; end if;
end if; if Z > Y then T := Y; Y := Z; Z := T; end if; Wonder_What:
while T > 0 loop TEXT_IO.Get (Item => T); if T > X then Z := Y;
Y := X; X := T; else if T > Y then Z := Y; Y:= T; else if T > Z
then Z := T; end if; end if; TEXT_IO.Get (Item => T); end if;
end loop Wonder_What; TEXT_IO.Put (Item => X); TEXT_IO.Put
(Item => Y); TEXT_IO.Put (Item => Z); end XYZ;
```

7. Determine the program behavior [XYZ] in problem 6.

8. Revise program XYZ with more descriptive identifiers and comments that describe the program behavior.

4.2 Determining Sequence Statement Behaviors

4.2.1 Trace Tables

Sequence statement behaviors, as discussed in the previous chapter, can be discovered directly. But as sequence statements become more complex, their behaviors cannot be discovered so easily or accurately. For that reason, a new method of step by step discovery and recording is given next, in the form of a *trace table*. A trace table defines a column for each object that can be declared or altered in the sequence, and a row for each declaration or statement in the sequence. For example, consider the sequence of statements below, as part of a program:

```
X := Y;
Y := Z;
Z := X;
```

The trace table will have a column for each of the objects X, Y, Z being altered, and a row for each of the statements of the sequence above. At the heads of the columns will be identifiers for the objects. The first row within the table will contain the initial values of the objects. Then, in each place within the remaining rows, a new value will appear for the object of that column following the execution of the statement of that row. These values will be of the form of the subscripted identifiers given in equations with values from the previous row. In this case the trace table will be as follows.

Statements	X	Y	Z
	X_0	Y_0	Z_0
X := Y;	$X_1 = Z_0$	$Y_1 = Y_0$	$Z_1 = Z_0$
Y := Z;	$X_2 = X_1$	$Y_2 = Z_1$	$Z_2 = Z_1$
Z := X;	$X_3 = X_2$	$Y_3 = Y_2$	$Z_3 = X_2$

Trace Table 4.1

That is, each value of each object is given, step by step, in terms of values in the row above. In each row, only one object is assigned a new value, so the other two objects are unchanged. Now the equations in the table defining new values in terms of preceding values can be solved directly to find the final values in terms of the initial values. For example

$X_3 = X_2 = X_1 = Y_0$
$Y_3 = Y_2 = Z_1 = Z_0$
$Z_3 = X_2 = X_1 = Y_0$

That is, this part behavior is

```
[X := Y; Y := Z; Z := X;]

= {⟨⟨..., X, Y, Z, ...⟩, ⟨..., Y, Z, Y, ...⟩⟩},
```

or more simply, X, Z are assigned the initial value of Y, and Y is assigned the initial value of Z.

This final result can be described in the *concurrent assignment* already defined, which is *not* a legal Ada statement, but describes simultaneous changes to several objects. In this case, the concurrent assignment

```
X, Y, Z <- Y, Z, Y;
```

assigns the initial values of Y, Z, Y as final values for X, Y, Z.

Such trace tables can be abbreviated in two ways. First, records of values of objects unchanged in a statement can be left blank. The trace table above then becomes

Statements	X	Y	Z
	X_0	Y_0	Z_0
X := Y;	$X_1 = Y_0$		
Y := Z;		$Y_2 = Z_1$	
Z := X;			$Z_3 = X_2$

Trace Table 4.2

In this case, the missing equations must be supplied in working out the results. For example, Z is not found in the abbreviated table, but can be calculated as follows

$Z = X_2$ (X unchanged in step 3)

$\quad = X_1$ (X unchanged in step 2)

$\quad = Y_0$ (X changed in the table)

Second, the left side of each equation in the trace table can be determined from the row and column in which it appears, so only the right side is strictly necessary. The initial values are also unnecessary. The trace table in minimum form then becomes

Statements	X	Y	Z
X := Y;	Y_0		
Y := Z;		Z_1	
Z := X;			X_2

Trace Table 4.3

In this case, more information must be supplied in the minimum table, for example for Y_3 as follows

$Y_3 = Y_2$ (Y unchanged in step 3)

$\quad = Z_1$ (Y changed in the table)

$\quad = Z_0$ (Z unchanged in step 1)

Notice here that in the final calculation dealing with step 1, Z is the object of interest because Y was set to the value of Z in step 2.

4.2.2 Trace Tables with Concurrent Assignments

A sequence of concurrent statements, while not legal Ada, can describe a design later converted into Ada in a powerful way. For example, consider a sequence of concurrent statements

```
X, Y, Z <- Y, Z, W;
Y, Z, W <- Z, W, X;
Z, W, X <- W, X, Y;
W, X, Y <- X, Y, Z;
```

each of which can be expanded into legal Ada. In this case the trace table in minimum form is

Statements	X	Y	Z	W
X, Y, Z <- Y, Z, W;	Y_0	Z_0	W_0	
Y, Z, W <- Z, W, X;		Z_1	W_1	X_1
Z, W, X <- W, X, Y;	Y_2		W_2	X_2
W, X, Y <- X, Y, Z;	Y_3	Z_3		X_3

Trace Table 4.4

The determinations of values from this table are

$$X_4 = Y_3 = Y_2 = Z_1 = W_0$$
$$Y_4 = Z_3 = W_2 = X_1 = Y_0$$
$$Z_4 = Z_3 = W_2 = X_1 = Y_0$$
$$W_4 = X_3 = Y_2 = Z_1 = W_0$$

That is

```
[X, Y, Z <- Y, Z, W;
 Y, Z, W <- Z, W, X;
 Z, W, X <- W, X, Y;
 W, X, Y <- X, Y, Z;]

= {⟨⟨X, Y, Z, W⟩, ⟨W, Y, Y, W⟩⟩},
= (X, Y, Z, W <- W, Y, Y, W)
```

using the concurrent assignment.

4.2.3 Trace Tables with TEXT_IO Procedure Calls

As another example, consider the program part

```
TEXT_IO.Get (Item => Letter);
Term := Letter;
TEXT_IO.Put (Item => Term);
```

which has a trace table in a condensed form, abbreviating TEXT_IO as T

Statements	Input	Output	Letter	Term
`T.Get (Item => Letter);`	$\mathtt{Input}_1$		$\mathtt{Letter}_1$	
`Term := Letter;`				$\mathtt{Term}_2$
`T.Put (Item => Term);`		$\mathtt{Output}_3$		

Trace Table 4.5

where user input is called `Inchar`

$$\begin{aligned}
\mathtt{Input}_1 &= \mathtt{Input}_0 \ \& \ \mathtt{Inchar}_0 \\
\mathtt{Letter}_1 &= \mathtt{Inchar}_0 \\
\mathtt{Term}_2 &= \mathtt{Letter}_1 \\
\mathtt{Output}_3 &= \mathtt{Output}_2 \ \& \ \mathtt{Term}_2
\end{aligned}$$

In this case, the final values for all objects become

$$\begin{aligned}
\mathtt{Input}_3 &= \mathtt{Input}_2 \\
&= \mathtt{Input}_1 \\
&= \mathtt{Input}_0 \ \& \ \mathtt{Inchar}_0
\end{aligned}$$

$$\begin{aligned}
\mathtt{Output}_3 &= \mathtt{Output}_2 \ \& \ \mathtt{Term}_2 \\
&= \mathtt{Output}_1 \ \& \ \mathtt{Letter}_1 \\
&= \mathtt{Output}_0 \ \& \ \mathtt{Inchar}_0
\end{aligned}$$

$$\begin{aligned}
\mathtt{Letter}_3 &= \mathtt{Letter}_2 \\
&= \mathtt{Letter}_1 \\
&= \mathtt{Inchar}_0
\end{aligned}$$

$$\begin{aligned}
\mathtt{Term}_3 &= \mathtt{Term}_2 \\
&= \mathtt{Letter}_1 \\
&= \mathtt{Inchar}_0
\end{aligned}$$

so that the part behavior is

```
[TEXT_IO.Get (Item => Letter); Term := Letter;
 TEXT_IO.Put (Item => Term);]

= {⟨⟨Input, Output, Letter, Term⟩,
    ⟨Input & Inchar, Output & Inchar,
    Inchar, Inchar⟩⟩}

= Input, Output, Letter, Term <-
  Input & Inchar, Output & Inchar, Inchar, Inchar
```

using the concurrent assignment.

4.2.4 Trace Tables with Integer Arithmetic

With sequences involving integer arithmetic, overflow must be considered, for example, in the sequence

```
X := Y + Z;
Y := Z - W;
Z := W + X;
W := X - Y;
```

the trace table in condensed form will be

Statements	X	Y	Z	W
X := Y + Z;	$Y_0 + Z_0$			
Y := Z - W;		$Z_1 - W_1$		
Z := W + X;			$W_2 + X_2$	
W := X - Y;				$X_3 - Y_3$

Trace Table 4.6

with values as follows.

$$
\begin{aligned}
X_4 &= X_3 = X_2 = X_1 = Y_0 + Z_0\\
Y_4 &= Y_3 = Y_2 = Z_1 - W_1 = Z_0 - W_0\\
Z_4 &= Z_3 = W_2 + X_2 = W_1 + X_1 = W_0 + Y_0 + Z_0\\
W_4 &= X_3 - Y_3 = X_2 - Y_2 = X_1 - Z_1 + W_1 = Y_0 + Z_0 - Z_0 + W_0
\end{aligned}
$$

where Z_0 cancels out in the final value for W_4. The resulting part behavior is

```
[X := Y + Z; Y := Z - W; Z := W + X; W := X - Y;]

= {(⟨Input, Output, …, X, Y, Z, W⟩,
    ⟨Input, Output, …, Y + Z, Z - W, Y + Z + W, Y + W⟩) |
    no overflow has occurred}
∪ {(⟨Input, Output, …, X, Y, Z, W⟩, ⟨Input, Output⟩) |
    overflow has occurred and execution terminated}
```

when the overflow condition is explicitly

```
0 <= Y + Z <= 999,
0 <= Z - W <= 999,
0 <= Y + Z + W <= 999,
0 <= Y + W <= 999
```

The part behavior is also given in the concurrent assignment

```
X, Y, Z, W <- Y + Z, Z - W, Y + Z + W, Y + W
```

when no overflow has occurred and termination to ⟨Input, Output⟩ otherwise.

4.2.5 Exercises

1. Create a trace table and find the part behavior for the sequence below with

```
X, Y, Z: CHARACTER;

Y := Z;
Z := X;
X := Y;
```

2. Create a trace table and find the part behavior for the sequence below with

```
X, Y, Z: CHARACTER;

X := Y;
Y := Z;
Z := X;
X := Y;
```

3. Create a trace table and find the part behavior for the sequence with

```
type My_Integer is INTEGER range 0..999;
X, Y, Z, W: My_Integer;

Y := Z - X;
Z := Y + W;
W := X - Y;
```

4. Create a trace table and find the part behavior for the sequence with objects declared in Exercise 3

```
X := Y + Z;
Y := Z - X;
Z := Y + W;
W := X - Y;
Y := Z + W;
```

5. Create a trace table and find the part behavior for the sequence with

```
X, Y, Z, W: BOOLEAN;

X := Y;    Y := Z;
Z := W;    Z := Y;
Y := Z;    W := X;
W := X;    Y := Z;
X := Y;    Z := W;
```

6. Create a trace table and find the part behavior for the sequence with objects declared in Exercise 3

```
X := Y + Z;    Z := Y + W;
Y := Z + W;    W := X - Z;
Y := Y - W;    X := Y + Z;
Z := Z - X;    W := X - Y;
```

7. Create a trace table and find the part behavior for the sequence in which `Term` and `Letter` are declared CHARACTER

```
Term := Letter;
TEXT_IO.Put (Item => Term);
TEXT_IO.Get (Item => Letter);
Term := Letter;
TEXT_IO.Put (Item => Term);
```

4.3 Determining If Statement Behaviors

4.3.1 Conditional Trace Tables

Trace tables can be expanded to deal with **if** statements by adding a single new kind of column called a *condition column* to create a *conditional trace table.*

Consider the **if** statement

```
if X < Y then
  Y := Z;
  Z := X;
else
  X := Z;
  Y := X;
end if;
```

A conditional trace table will exist for each possible value of the **if** condition `X < Y`, which will contain the **then** part or the **else** part as appropriate. In this case, the first conditional trace table in abbreviated form is

Statements	Condition	Y	Z
X < Y	$X_0 < Y_0$		
Y := Z;		$Y_1 = Z_0$	
Z := X;			$Z_2 = X_1$

Trace Table 4.7

and the second conditional trace table is

Statements	Condition	X	Y
X >= Y	$X_0 >= Y_0$		
X := Z;		$X_1 = Z_0$	
Y := X;			$Y_2 = X_1$

Trace Table 4.8

In the first conditional trace table, the condition is already stated in initial values, while the final values of the objects need working out, as in ordinary trace tables as follows

$$X_0 < Y_0 \qquad Y_2 = Y_1 = Z_0 \qquad Z_2 = X_1 = X_0$$

and the second conditional trace table works out as follows

X_0 >= Y_0 $X_2 = X_1 = Z_0$ $Y_2 = X_1 = Z_0$

In this case, the if statement behavior is

```
[if X < Y then Y := Z; Z := X;
          else X := Z; Y := X; end if;]

= {(⟨...X, Y, Z, ...⟩, ⟨...X, Z, X, ...⟩) | X < Y}
  {(⟨...X, Y, Z, ...⟩, ⟨...Z, Z, Z, ...⟩) | X >= Y}
```

This program part can also be described in a *conditional assignment*, which is not a legal Ada statement, but describes conditional and simultaneous changes to several objects. A conditional assignment is defined by a sequence of conditions, each followed by a simple, concurrent, or another conditional assignment. In this case, the conditional assignment is

```
(X < Y -> Y, Z <- Z, X;
 X >= Y -> X, Y <- Z, Z;)
```

which is described with two conditions and a concurrent assignment for each condition.

The effect of a conditional assignment is defined by examining the conditions in the sequence in which they appear. If a condition is TRUE, the following concurrent or conditional assignment is used and the remainder of the conditions and assignments are ignored. If the condition is FALSE, the next condition, if any, is examined.

4.3.2 Conditional Trace Tables with Sequences

In dealing with more general circumstances of conditional program behavior, consider the sequence including an if statement, such as below, as part of a program:

```
X := U;
Y := V;
if X < Y then
  X := Y;
  Y := Z;
  Z := X;
else
  Y := Z;
  Z := X;
  X := Y;
end if;
```

This sequence with an if statement will define two conditional trace tables, depending on the value of the condition in the if statement, namely

Statements	Condition	X	Y	Z
X := U;		$X_1 = U$		
Y := V;			$Y_2 = V$	
X < Y	$X_2 < Y_2$			
X := Y;		$X_3 = Y_2$		
Y := Z;			$Y_4 = Z_3$	
Z := X;				$Z_5 = X_4$

Trace Table 4.9

and

Statements	Condition	X	Y	Z
X := U;		$X_1 = U$		
Y := V;			$Y_2 = V$	
X >= Y	$X_2 >= Y_2$			
Y := Z;			$Y_3 = Z_2$	
Z := X;				$Z_4 = X_3$
X := Y;		$X_5 = Y_4$		

Trace Table 4.10

Now in both conditional trace tables, the conditions must be worked back to initial values as must be the final values. For the first conditional trace table, the derivations are

$$X_2 < Y_2 \quad \text{or} \quad X_1 < V \quad \text{or} \quad U < V$$
$$X_5 = X_4 = X_3 = Y_2 = V$$
$$Y_5 = Y_4 = Z_3 = Z_2 = Z_1 = Z_0$$
$$Z_5 = X_4 = X_3 = Y_2 = V$$

with the conditional assignment

```
(U < V -> X, Y, Z <- V, Z, V)
```

and

$$X_2 >= Y_2 \quad \text{or} \quad X_1 >= V \quad \text{or} \quad U >= V$$
$$X_5 = Y_4 = Y_3 = Z_2 = Z_1 = Z_0$$
$$Y_5 = Y_4 = Y_3 = Z_2 = Z_1 = Z_0$$
$$Z_5 = Z_4 = X_3 = X_2 = X_1 = U$$

with the conditional assignment

```
(U >= V -> X, Y, Z <- Z, Z, U)
```

which combine into the entire conditional assignment

```
(U < V -> X, Y, Z <- V, Z, V;
 U >= V -> X, Y, Z <- Z, Z, U)
```

Note that although objects X and Y appear in the if condition, it is the initial values of U and V that define the conditions in the conditional assignment here.

Note also that the conditional assignment above has identical behavior as the conditional assignment

```
(U < V -> X, Y, Z <- V, Z, V;
  TRUE -> X, Y, Z <- Z, Z, U)
```

because whenever the first condition U < V fails, both second conditions, U >= V and TRUE evaluate TRUE and the component concurrent assignments are identical.

4.3.3 Conditional Assignments for Procedure Calls

Consider the if statement

```
if Next in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "The next item is " & Next);
else
  TEXT_IO.Get (Item => Next);
  if Next in 'A' .. 'Z' then
    TEXT_IO.Put (Item => "Next item in Input is " & Next);
  else
    TEXT_IO.Put (Item => "Next item in Input is improper");
  end if;
end if;
```

There will be three possible outcomes, namely 1) `Next` holds an upper case letter, 2) next character from user is an upper case letter, and 3) next character from user is *not* an upper case letter. There will be three corresponding conditional trace tables, as shown below, with the following abbreviations.

`FALSE`	`F`
`TRUE`	`T`
`Input`	`In`
`Output`	`Ou`
`Next in 'A' .. 'Z'`	`Next in 'A'..`
`TEXT_IO.Get (Item => Next);`	`TGet(Next);`
`TEXT_IO.Put (Item => "The next item is " & Next );`	`TPut("The…");`
`TEXT_IO.Put (Item => "Next item in Input is " & Next);`	`TPut("Next…" & Next);`
`TEXT_IO.Put (Item => "Next item in Input is improper");`	`TPut("Next…improper");`

We also use the name `Char` for the next character returned by the user, if called for.

The first conditional trace table is given next.

Statements	Condition	Output
Next in 'A' ..	Next$_0$ in 'A'..	
TPut("The…");		Ou$_0$&"The…"

Trace Table 4.11

In this case, the condition (abbreviated in the table) is

Next$_0$ in 'A' .. 'Z' is TRUE

and the next statement (abbreviated in the table) is

```
TEXT_IO.Put (Item => "The next item is " & Next );
```

in which the only object affected is `Output`, which becomes

Output = Output$_0$ & "The next item is " & Next

These combine into the conditional assignment

(Next in 'A' .. 'Z' is TRUE ->
 Output <- Output$_0$ & "The next item is " & Next ;)

The second table is as follows.

Statements	Condition	Input	Output	Next
Next in 'A'..	Next$_0$ in 'A'.. is F			
TGet(Next);		In$_0$&Char		Char
Next in 'A' .. 'Z'	Next$_1$ in 'A'.. is T			
TPut("Next..." & Next);			Ou$_0$&"Next..." & Next$_2$	

Trace Table 4.12

The first condition is

Next$_0$ in 'A' .. 'Z' is FALSE

and the next statement is

```
TEXT_IO.Get (Item => Next);
```

which effects two objects, namely `Input` and `Next` as follows

Input$_1$ = Input$_0$ & Char
Next$_1$ = Char

then the next condition is

Next$_1$ in 'A' .. 'Z' is TRUE

and the last statement effects only the object `Output` as

Output$_2$ = Output$_0$ & "Next item in Input is " & Next$_2$

First, the two conditions can be reduced to initial values as follows

Next$_0$ in 'A' .. 'Z' is FALSE (already in initial values)

Next$_1$ in 'A' .. 'Z' is TRUE
 = $\hbar$(Input$_1$) in 'A' .. 'Z' is TRUE

and can be combined into the initial condition as

```
Next in 'A' .. 'Z' is FALSE and
Char in 'A' .. 'Z' is TRUE
```

The objects can also be reduced to initial values as follows

$\texttt{Input}_2 = \texttt{Input}_1 = \texttt{Input}_0$ & Char

$\texttt{Output}_2 = \texttt{Output}_0$ & "Next item in Input is " & $\texttt{Next}_2$

$= \texttt{Output}_0$ & "Next item in Input is " & Char

$\texttt{Next}_2 = \texttt{Next}_1$ = Char

These conditions and object values combine into the conditional assignment

```
(Next in 'A' .. 'Z' is FALSE and
 Char in 'A' .. 'Z' is TRUE ->
   Input, Output, Next <- Input & Char,
   Output & "Next item in Input is " & Char, Char)
```

The third table is next.

Statements	Condition	Input	Output	Next
Next in 'A'..	$\texttt{Next}_0$ in 'A'.. is F			
TGet(Next);		$\texttt{In}_0$&Char		Char
Next in 'A' .. 'Z'	$\texttt{Next}_1$ in 'A'.. is F			
TPut("Next… improper");			$\texttt{Ou}_0$&"Next… improper"	

Trace Table 4.13

The first condition is

$\texttt{Next}_0$ in 'A' .. 'Z' is FALSE

and the next statement is

```
TEXT_IO.Get (Item => Next);
```

which effects two objects, namely `Input` and `Next` as in the second case and repeated here

$\texttt{Input}_1 = \texttt{Input}_0$ & Char

$\texttt{Next}_1$ = Char

then the next condition is

$\texttt{Next}_1$ in 'A' .. 'Z' is FALSE

and the last statement effects only the object `Output` as

$\texttt{Output}_2 = \texttt{Output}_1$ & "Next item in Input is improper"

First, the two conditions can be reduced to initial values as follows

$\texttt{Next}_0$ in 'A' .. 'Z' is FALSE (already in initial values)

$\texttt{Next}_1$ in 'A' .. 'Z' is FALSE

= Char in 'A' .. 'Z' is FALSE

and can be combined into the initial condition as

$Next_0$ in 'A' .. 'Z' is FALSE and
Char in 'A' .. 'Z' is FALSE

The objects can also be reduced to initial values as follows

$Input_2$ = $Input_1$ = $Input_0$ & Char
$Output_2$ = $Output_1$ & "Next item in Input is improper"
= $Output_0$ & "Next item in Input is improper"
$Next_2$ = $Next_1$ = Char

These conditions and object values combine into the conditional assignment

```
(Next in 'A' .. 'Z' is FALSE and
 Char in 'A' .. 'Z' is FALSE ->
   Input, Output, Next <- Input & Char,
   Output & "Next item in Input is improper", Char)
```

The conditional assignment for the entire if statement can now be assembled from the three cases handled above as

```
[if Next in 'A' .. 'Z' then
  TEXT_IO.Put (Item => "The next item is " & Next );
else
  TEXT_IO.Get (Item => Next);
  if Next in 'A' .. 'Z' then
    TEXT_IO.Put (Item => "Next item in Input is " & Next);
  else
    TEXT_IO.Put (Item => "Next item in Input is improper");
  end if;
end if;] =

(Next in 'A' .. 'Z' is TRUE ->
 Output <- Output & "The next item is " & Next)

(Next in 'A' .. 'Z' is FALSE and
 Char in 'A' .. 'Z' is TRUE ->
   Input, Output, Next <- Input & Char,
   Output & "Next item in Input is " & Char, Char)

(Next in 'A' .. 'Z' is FALSE and
 Char in 'A' .. 'Z' is FALSE ->
   Input, Output, Next <- Input & Char,
   Output & "Next item in Input is improper", Char)
```

4.3.4 Exercises

1. Create the conditional trace tables and part behavior for the **if** statement with

```
X, Y, Z: CHARACTER;

if Z < Y then
  X := Y;
  Z := X;
else
  X := Z;
  Y := X;
end if;
```

2. Create the conditional trace tables and part behavior for the **if** statement with objects declared as in Exercise 1

```
if X < Y then
  TEXT_IO.Get (Item => Z);
  TEXT_IO.Put (Item => X);
else
  TEXT_IO.Put (Item => Y);
  TEXT_IO.Get (Item => X);
end if;
```

3. Create the conditional trace tables and part behavior for the **if** statement with objects declared as in Exercise 1

```
if Z < X and Z < Y then
  X := Z;
  Z := Y;
  Y := X;
else
  Y := Z;
  Z := X;
  X := Y;
end if;
```

4. Create the conditional trace tables and part behavior for the **if** statement with objects declared as in Exercise 1

```
if X < Z then
  X := Y;
  Y := Z;
  Z := X;
else
  TEXT_IO.Get (Item => X);
  Y := X;
  Z := Y;
end if;
```

5. Create the conditional trace tables and part behavior for the if statement with objects declared as in Exercise 1

```
if Z < X and Z < Y then
  X := Z;
  Y := X;
  Z := Y;
  X := Z;
  Y := X;
  Z := Y;
else
  Y := Z;
  Z := X;
  X := Y;
  Y := Z;
end if;
```

6. Create the conditional trace tables and part behavior for the sequence statement with

```
type My_Integer is range 0..999;
X, Y, Z, U, V: My_Integer;

X := U - V;
Y := V + U;
if X < Y then
  X := Y - Z;
  Y := Z + X;
  Z := X - Y;
else
  Y := Z + X;
  Z := X - Y;
  X := Y + Z;
end if;
```

4.4 Determining For Statement Behaviors

4.4.1 Trace Tables with For Loops

For loops, define sequences with a special variable that is automatically incremented each time through the for loop. For example, the for loop

```
type My_Integer is INTEGER range 0..999;
X, Y: My_Integer;

Keeping_Track:
for Inc in 1..3 loop
  X := X + Inc;
  Y := Y + X;
end loop Keeping_Track;
```

means the sequence

```
Inc := 1;
X   := X + Inc;
Y   := Y + X;
Inc := 2;
X   := X + Inc;
Y   := Y + X;
Inc := 3;
X   := X + Inc;
Y   := Y + X;
```

In this case, the trace table in condensed form will be

Statements	Inc	X	Y
Inc := 1;	1		
X := X + Inc;		X_1 + 1	
Y := Y + X;			Y_2 + X_2
Inc := 2;	2		
X := X + Inc;		X_4 + 2	
Y := Y + X;			Y_5 + X_5
Inc := 3;	3		
X := X + Inc;		X_7 + 3	
Y := Y + X;			Y_8 + X_8

Trace Table 4.14

with values as follows

$$X_9 = X_7 + 3 = X_4 + 5 = X_1 + 6 = X_0 + 6$$

$$\begin{aligned} Y_9 &= Y_8 + X_8 \\ &= Y_7 + X_7 + 3 \\ &= Y_5 + X_5 + X_5 + 3 = Y_5 + 2 * X_5 + 3 \\ &= Y_4 + 2 * (X_4 + 2) + 3 = Y_4 + 2 * X_4 + 7 \\ &= Y_2 + X_2 + 2 * X_2 + 7 = Y_2 + 3 * X_2 + 7 \\ &= Y_1 + 3 * (X_1 + 1) + 7 = Y_1 + 3 * X_1 + 10 \\ &= Y_0 + 3 * X_0 + 10 \end{aligned}$$

Of course, the variable Inc has no value at the conclusion of the for loop, and is shown in the trace table during the execution of the for loop to help determine its behavior. Again, there are overflow conditions with each calculation of X and Y. If X and Y are declared as above in the range 0..999, the overflow conditions are explicitly

```
0 <= X + 1 <= X + 3 <= X + 6 <= 999
0 <= Y + X + 1 <= 999
0 <= Y + 2 * X + 3 <= 999
0 <= Y + 3 * X + 10 <= 999
```

Note that while the calculations for X, Y are computed backwards from X_9, Y_9 back down to X_0, Y_0, the overflow must be determined forwards because that is the sequence in which the values are determined.

With for loops that increment through many more values, the trace tables will be longer and require more analysis to determine the behavior of the loops. But if the behavior cannot be analyzed completely to determine the outcome, the for loop should be redesigned so that it can.

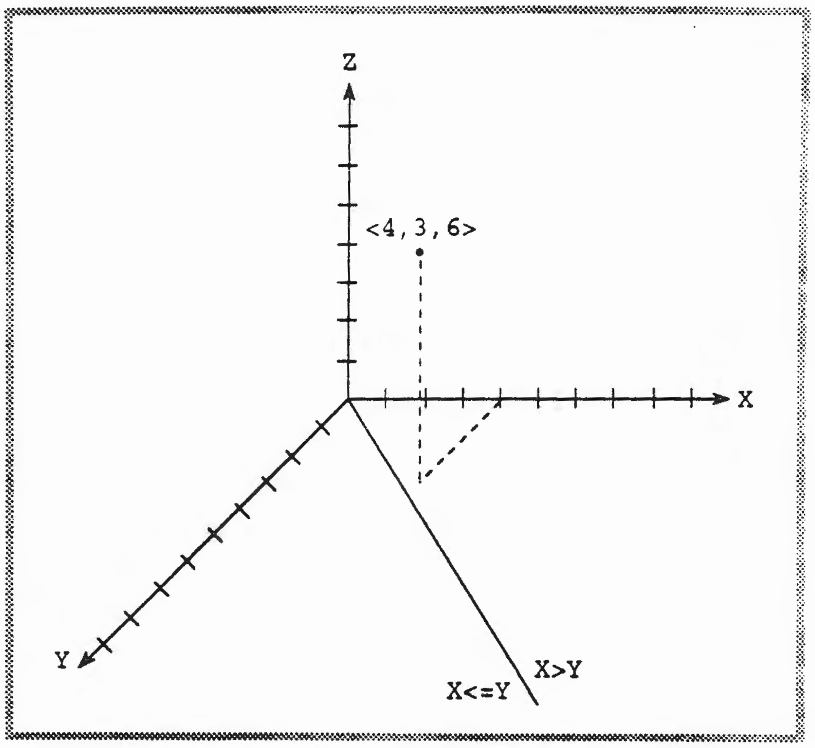
4.4.2 Trace Tables for Large For Loops

Consider a for loop with more iterations. For example, the for loop called `Large_Loop`

```
type My_Integer is range -1_000_000..1_000_000;
X, Y, Z : My_Integer;

Large_Loop:
for Inc in 1 .. 1_000 loop
  INT_IO.Put (Item => Inc);
  TEXT_IO.New_Line;
  if X > Y then
    Z := X - Y + Inc;
    Y := Z - X + Inc;
    X := Y - Z + Inc;
  else
    Z := Y - X - Inc;
    X := Z - Y - Inc;
    Y := X - Z - Inc;
  end if;
  INT_IO.Put (Item => X);
  INT_IO.Put (Item => Y);
  INT_IO.Put (Item => Z);
  TEXT_IO.New_Line;
  if Y > Z then
    X := Y - Z + Inc;
    Z := X - Y + Inc;
    Y := Z - X + Inc;
  else
    X := Z - Y - Inc;
    Y := X - Z - Inc;
    Z := Y - X - Inc;
  end if;
  INT_IO.Put (Item => X);
  INT_IO.Put (Item => Y);
  INT_IO.Put (Item => Z);
  TEXT_IO.New_Line;
  if Z > X then
    Y := Z - X + Inc;
    X := Y - Z + Inc;
    Z := X - Y + Inc;
  else
    Y := X - Z - Inc;
    Z := Y - X - Inc;
    X := Z - Y - Inc;
  end if;
  INT_IO.Put (Item => X);
  INT_IO.Put (Item => Y);
  INT_IO.Put (Item => Z);
  TEXT_IO.New_Line;
end loop Large_Loop;
```

means one of eight sequences on each iteration, or one of eight to the power of a thousand sequences altogether. So writing all the possibilities down will not be practical. Instead, we need to understand a single iteration in a more powerful form in order to put them together.

To begin with, consider a graph for the point X = 4, Y = 3, Z = 6 in the three dimensional graph as follows.



Initial Input
Figure 4.1

Note in this case that the then part of the first if statement would apply, namely where X > Y.

To begin with, consider the low level triples of three assignments, such as the then part of the first if statement in the for loop body

```
Z := X - Y + Inc;
Y := Z - X + Inc;
X := Y - Z + Inc;
```

In the three dimensional space of X, Y, Z, a single point is being moved to a new point. Then a trace table for these three assignments is

Statements	Inc	X	Y	Z
Z := X - Y + Inc;				$X_0-Y_0+Inc_0$
Y := Z - X + Inc;			$Z_1-X_1+Inc_1$	
X := Y - Z + Inc;		$Y_2-Z_2+Inc_2$		

Trace Table 4.15

For example, continue the case where

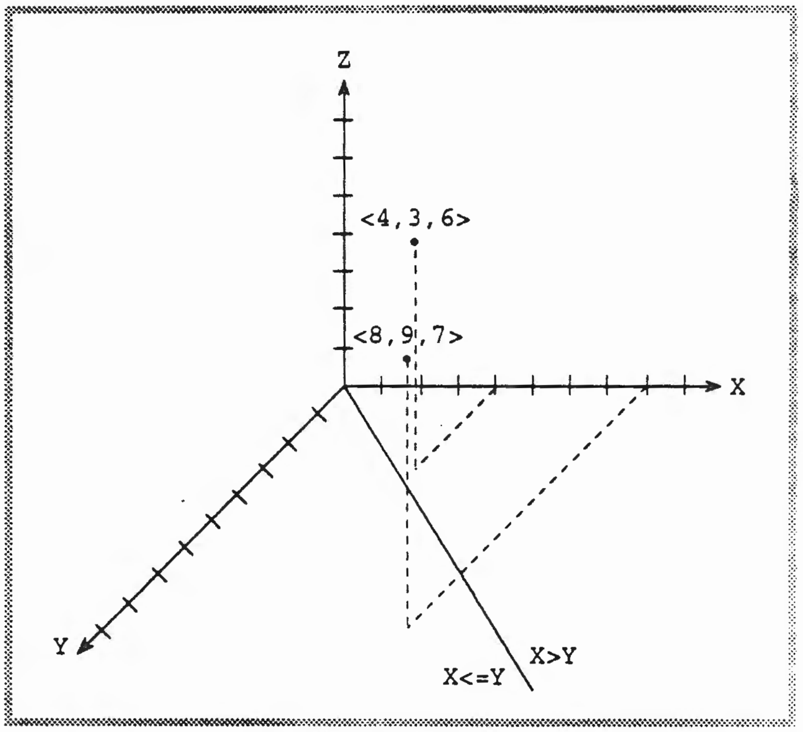
```
X = 4, Y = 3, Z = 6, Inc = 6.
```

Then, the values in the trace table become

Statements	Inc 6	X 4	Y 3	Z 6
`Z := X - Y + Inc;`				7
`Y := Z - X + Inc;`			9	
`X := Y - Z + Inc;`		8		
final values	6	8	9	7

Trace Table 4.16

which can be pictured as in Figure 4.2. In such pictures, remember the **if** conditions, in this case that X > Y to begin with. Therefore, only half of the three dimensional space is available.



First Step
Figure 4.2

In the following if statement the condition is Y > Z, which in this case is also TRUE so the then part will be used again. In this case, the assignments following the **then** applies

```
X := Y - Z + Inc;
Z := X - Y + Inc;
Y := Z - X + Inc;
```

This condition will define the following trace table

Statements	Inc	X	Y	Z
X := Y - Z + Inc;		Y_0-Z_0+Inc_0		
Z := X - Y + Inc;				X_1-Y_1+Inc_1
Y := Z - X + Inc;			Z_2-X_2+Inc_2	

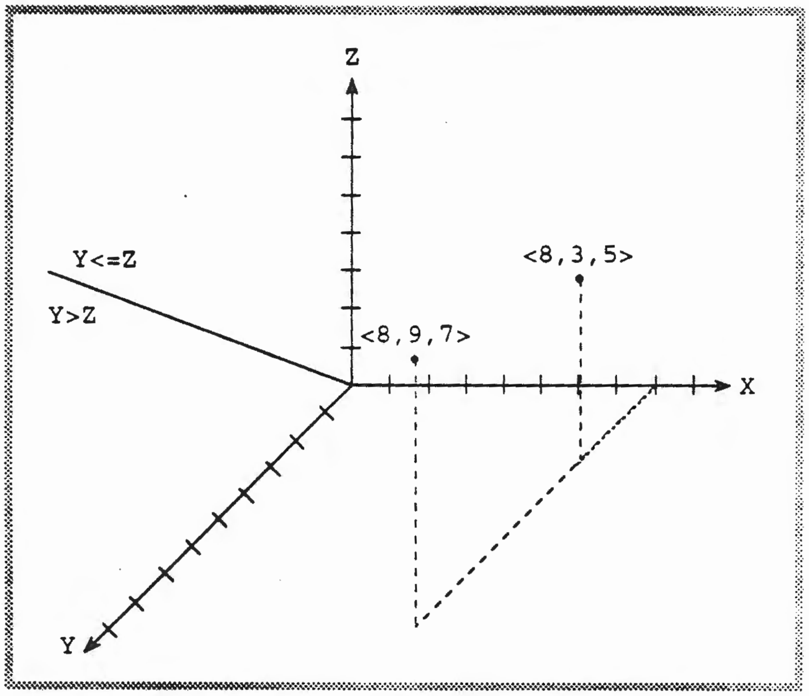
Trace Table 4.17

If we use our results from before, we have

Statements	Inc	X	Y	Z
	6	8	9	7
X := Y - Z + Inc;		8		
Z := X - Y + Inc;				5
Y := Z - X + Inc;			3	
final values	6	8	3	5

Trace Table 4.18

which can be pictured as in Figure 4.3. In such pictures, remember the if conditions, in this case that Y > Z to begin with. Again, only half of the three dimensional space is available.



Second Step
Figure 4.3

Finally, the third if statement has condition Z > X, so the else part will be used.

```
Y := X - Z - Inc;
Z := Y - X - Inc;
X := Z - Y - Inc;
```

This section will define the following trace table

Statements	Inc	X	Y	Z
Y := X - Z - Inc;			X_0-Z_0-Inc_0	
Z := Y - X - Inc;				Y_1-X_1-Inc_1
X := Z - Y - Inc;		Z_2-Y_2-Inc_2		

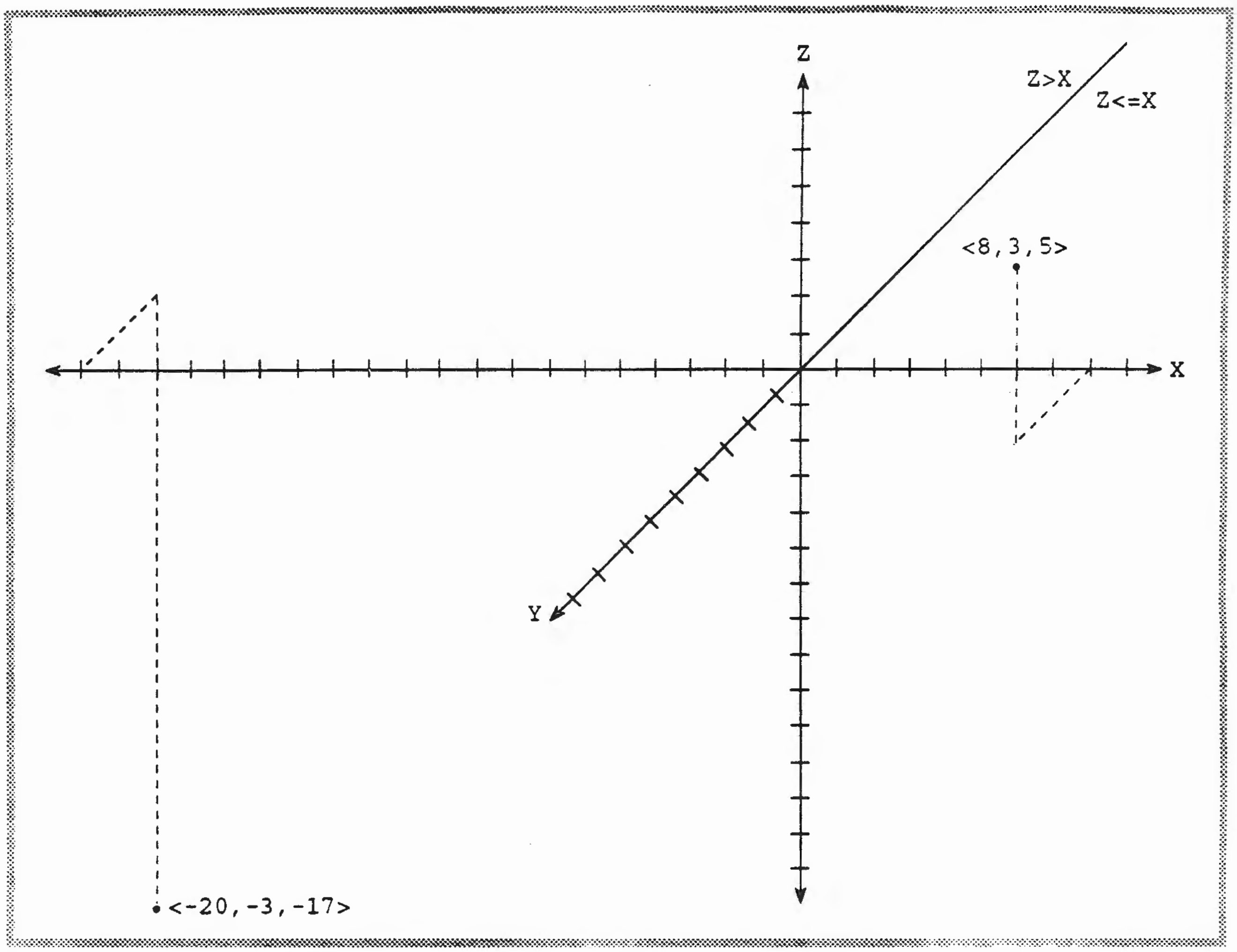
Trace Table 4.19

In this example, the specific values apply.

Statements	Inc	X	Y	Z
	6	8	3	5
Y := X - Z - Inc;			-3	
Z := Y - X - Inc;				-17
X := Z - Y - Inc;		-20		
final values	6	-20	-3	-17

Trace Table 4.20

which can be pictured as in Figure 4.4. In such pictures, remember the if conditions that hold, in this case that Z <= X to begin with. Again, only half of the three dimensional space is available.



Third Step
Figure 4.4

Finally, the for loop increment of Inc will take place to complete this loop increment as follows.

Statements	Inc	X	Y	Z
Inc := Inc + 1;	7			

Trace Table 4.21

Note this assignment to Inc is implied by the for loop, not explicitly defined. But it is handy to use the common notation for Inc as with other explicit assignments.

In this case four trace tables, namely 4.15, 4.17, 4.19, 4.21 have made up one increment. Starting with the values

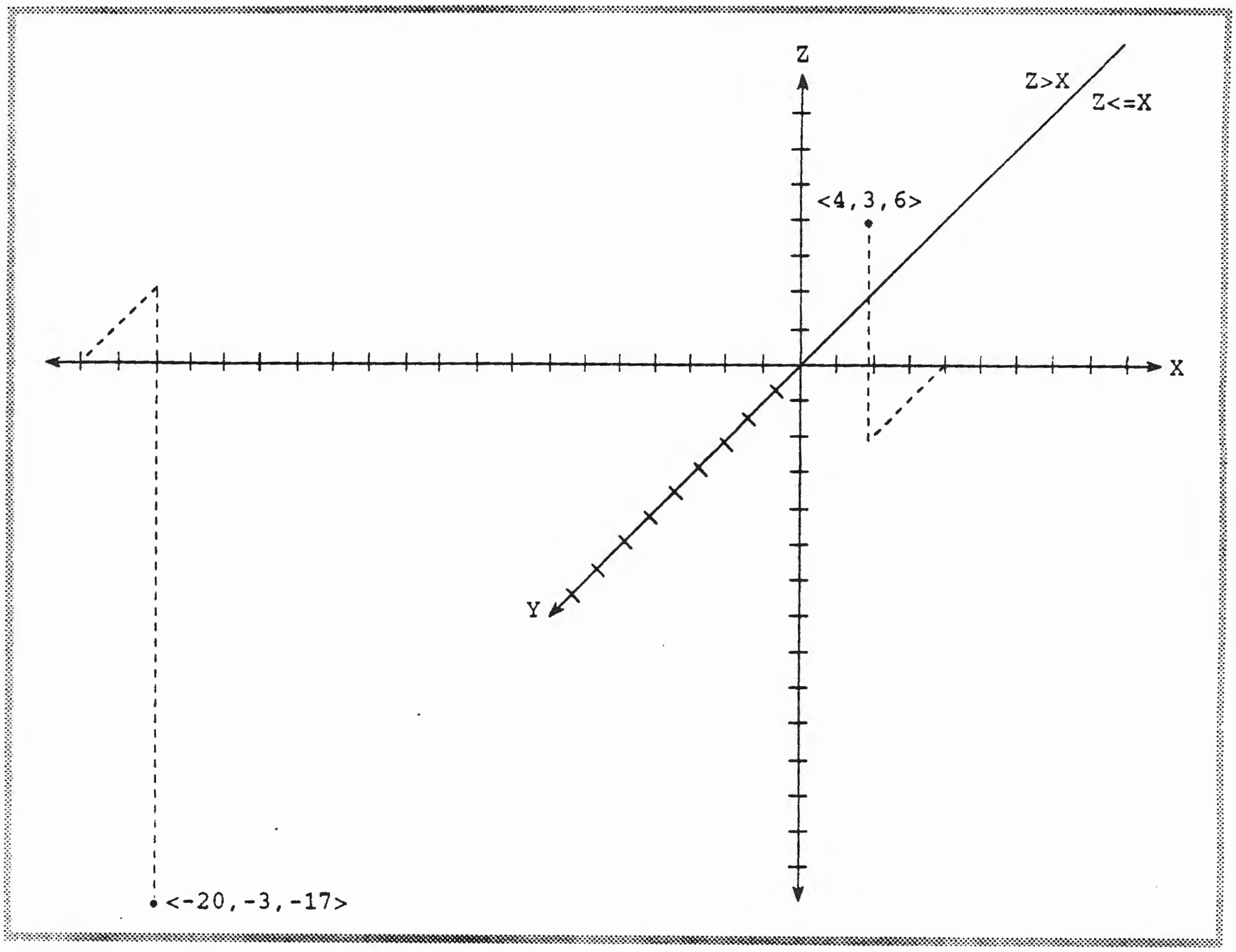
```
X = 4, Y = 3, Z = 6, Inc = 6,
```

a single increment ended up with the values

```
X = -20, Y = -3, Z = -17, Inc = 7.
```

These relatively small integers are a happy accident for drawing graphs. But with Inc taking on larger and larger values, X, Y, Z will also take on larger positive or negative values. In fact, X, Y, Z can start with large positive or negative values as well.

The effect of the one increment, combining the three if statements, can be given as follows.



Final Step
Figure 4.5

Now that a single increment of the for loop is better understood, we observe that the entire for loop execution can be pictured in this X, Y, Z space as a thousand steps. The value of Inc will increase in each step from 1 to 1_000, *and may be added or subtracted,* depending on which half of the space the point X, Y, Z is in. But aside from the iteration, no new ideas are involved in the calculation.

As noted at the outset, there are exactly eight of the three then/else sequences possible, of which one was shown above. The if conditions for each of the eight sequences can be worked out in terms of data at the start of the sequence. The first if condition is the original data. But the second and third if conditions result from assignments made in the first and second then or else statements. In illustration on the sequence analyzed above, the following hypothesis that two then conditions and one else condition was required.

```
if X > Y then
  Z := X - Y + Inc;
  Y := Z - X + Inc;
  X := Y - Z + Inc;
else
...
end if;
if Y > Z then
  X := Y - Z + Inc;
  Z := X - Y + Inc;
  Y := Z - X + Inc;
else
...
end if;
if Z > X then
...
else
  Y := X - Z - Inc;
  Z := Y - X - Inc;
  X := Z - Y - Inc;
end if;
```

Now, the second if statement is applied to the results of the first if statement. For example the second if condition `Y > Z` is applied to the `Y` and `Z` resulting from the first if statement. In trace tables above, each triple of assignments was analyzed. Now we want to analyze the entire set of conditions and assignments for a single increment through the for loop. The trace table for this full increment, except for the IO, follows.

Statements	Condition	Inc	X	Y	Z
X>Y	$X_0>Y_0$				
Z:=X-Y+Inc;					$X_0-Y_0+Inc_0$
Y:=Z-X+Inc;				$Z_1-X_1+Inc_1$	
X:=Y-Z+Inc;			$Y_2-Z_2+Inc_2$		
Y>Z	$Y_3>Z_3$				
X:=Y-Z+Inc;			$Y_4-Z_4+Inc_4$		
Z:=X-Y+Inc;					$X_5-Y_5+Inc_5$
Y:=Z-X+Inc;				$Z_6-X_6+Inc_6$	
Z>=X	$Z_7>=X_7$				
Y:=X-Z-Inc;				$X_8-Z_8-Inc_8$	
Z:=Y-X-Inc;					$Y_9-X_9-Inc_9$
X:=Z-Y-Inc;			$Z_{10}-Y_{10}-Inc_{10}$		
Inc:=Inc+1;		$Inc_{11}+1$			

Trace Table 4.22

Now, we need to convert these relations and equations to data at the beginning of the table, namely to X_0, Y_0, Z_0, Inc_0. This means making the calculations in the table, as follows.

Statements	Condition	Inc	X	Y	Z
X>Y	$X_0>Y_0$				
Z:=X-Y+Inc;					$X_1-Y_1+Inc_1=$ $X_0-Y_0+Inc_0$
Y:=Z-X+Inc;				$Z_2-X_2+Inc_2=$ $X_0-Y_0+Inc_0-X_0+Inc_0=$ $-Y_0+2*Inc_0$	
X:=Y-Z+Inc;			$Y_3-Z_3+Inc_3=$ $-Y_0+2*Inc_0-(X_0-Y_0+Inc_0)+Inc_0=$ $-X_0+3*Inc_0$		
Y>Z	$Y_4>Z_4=$ $-Y_0+2*Inc_0>X_0-Y_0+Inc_0=$ $Inc_0>X_0$				
X:=Y-Z+Inc;			$Y_5-Z_5+Inc_5=$ $-Y_0+2*Inc_0-(X_0-Y_0+Inc_0)+Inc_0=$ $-X_0+2*Inc_0$		
Z:=X-Y+Inc;					$X_6-Y_6+Inc_6=$ $-X_0+2*Inc_0-(Y_0+2*Inc_0)+Inc_0=$ $-X_0+Y_0+Inc_0$
Y:=Z-X+Inc;				$Z_7-X_7+Inc_7=$ $-X_0+Y_0+Inc_0-(-X_0+2*Inc_0)+Inc_0=$ Y_0	
Z<=X	$Z_8<=X_8=$ $-X_0+Y_0+Inc_0<=-X_0+2*Inc_0=$ $Y_0<=Inc_0$				
Y:=X-Z+Inc;				$X_9-Z_9-Inc_9=$ $-X_0+2*Inc_0-(-X_0+Y_0+Inc_0)-Inc_0=$ $-Y_0$	
Z:=Y-X+Inc;					$Y_{10}-X_{10}-Inc_{10}=$ $-Y_0-(-X_0+2*Inc_0)-Inc_0=$ $-Y_0+X_0-3*Inc_0$
X:=Z-Y+Inc;			$Z_{11}-Y_{11}-Inc_{11}=$ $-Y_0+X_0-3*Inc_0+Y_0-Inc_0=$ $X_0-4*Inc_0$		
Inc:=Inc+1;		Inc_0+1			

Trace Table 4.23

With one more step we show only the results, not their derivations as follows.

Statements	Condition	Inc	X	Y	Z
X>Y	$X_0>Y_0$				
Z:=X-Y+Inc;					$X_0-Y_0+Inc_0$
Y:=Z-X+Inc;				$-Y_0+2*Inc_0$	
X:=Y-Z+Inc;			$-X_0+3*Inc_0$		
Y>Z	$Inc_0>X_0$				
X:=Y-Z+Inc;			$-X_0+2*Inc_0$		
Z:=X-Y+Inc;					$-X_0+Y_0+Inc_0$
Y:=Z-X+Inc;				Y_0	
Z<=X	$Y_0<=Inc_0$				
Y:=X-Z+Inc;				$-Y_0$	
Z:=Y-X+Inc;					$-Y_0+X_0-3*Inc_0$
X:=Z-Y+Inc;			$X_0-4*Inc_0$		
Inc:=Inc+1;		Inc_0+1			

Trace Table 4.24

Carrying this trace table out on the example used above, we find.

Statements	Condition	Inc	X	Y	Z
		6	4	3	6
X>Y	4>3				
Z:=X-Y+Inc;					7
Y:=Z-X+Inc;				9	
X:=Y-Z+Inc;			8		
Y>Z	6>4				
X:=Y-Z+Inc;			8		
Z:=X-Y+Inc;					5
Y:=Z-X+Inc;				3	
Z<=X	3<=6				
Y:=X-Z+Inc;				-3	
Z:=Y-X+Inc;					-17
X:=Z-Y+Inc;			-20		
Inc:=Inc+1;		7			

Trace Table 4.25

The other seven cases will go just as this example. All eight cases are handled in the same way as shown in this example.

4.4.3 Exercises

1. Create a trace table and find the part behavior with variables X, Y declared as INTEGERS in the range 0..99 for the for loop

```
Update:
for Term in 1..4 loop
  X := X + Term;
  Y := Y - Term;
end loop Update;
```

2. Express the for loop of Exercise 1 as a direct sequence with no **for** logic.

3. Express the for loop of Exercise 1 as a direct sequence with only two assignments.

4. Create an alternative for loop to that of Exercise 1 with as few iterations as possible.

5. Repeating its form, `Large_Loop` is as follows.

```
TYPE My_Integer is INTEGER range -1_000_000, 1_000_000;
X, Y, Z : My_Integer;
...
Large_Loop:
for Inc in 1 .. 1_000 loop
  INT_IO.Put (Item => Inc);
  TEXT_IO.New_Line;
  if X > Y then
    Z := X - Y + Inc;
    Y := Z - X + Inc;
    X := Y - Z + Inc;
  else
    Z := Y - X - Inc;
    X := Z - Y - Inc;
    Y := X - Z - Inc;
  end if;
  INT_IO.Put (Item => X, Y, Z);
  TEXT_IO.New_Line;
  if Y > Z then
    X := Y - Z + Inc;
    Z := X - Y + Inc;
    Y := Z - X + Inc;
  else
    X := Z - Y - Inc;
    Y := X - Z - Inc;
    Z := Y - X - Inc;
  end if;
  INT_IO.Put (Item => X, Y, Z);
  TEXT_IO.New_Line;
  if Z > X then
    Y := Z - X + Inc;
    X := Y - Z + Inc;
    Z := X - Y + Inc;
  else
    Y := X - Z - Inc;
    Z := Y - X - Inc;
    X := Z - Y - Inc;
  end if;
  INT_IO.Put (Item => X, Y, Z);
  TEXT_IO.New_Line;
end loop Large_Loop;
```

Identify the eight possible sequences on each loop and the four Put messages.

6. Find the initial conditions required on X, Y, Z in each of the eight possible sequences of Exercise 5. Note that each **if** statement divides each space so far identified into two more parts.

4.5 Determining Loop Statement Behaviors

Sequence, **for**, and **if** statement behaviors can be discovered directly, but while loop statement behaviors require more analysis. It may be possible to determine the number of iterations that will occur directly from the initial data, and therefore describe the statement behavior directly. The number of iterations will be an object that depends on the initial data. A while loop statement will have a conditional trace table that describes continued looping. The conditional trace table will go on indefinitely, but the **while** condition may lead to necessary termination or the looping may become periodic after a certain point so that termination will not be possible. In either case, the conditional trace table can provide the while loop statement behavior as a conditional assignment statement.

4.5.1 While Loop Analysis with Conditional Trace Tables

For example, consider the while loop statement

```
Mystery_Loop:
while X < Y loop
  X := U;
  Y := Z;
  Z := X;
  X := V;
end loop Mystery_Loop;
```

The conditional trace table for this **loop** statement starts out as follows.

Statements	Condition	X	Y	Z
X < Y	$X_0 < Y_0$			
X := U;		$X_1 = U$		
Y := Z;			$Y_2 = Z_1$	
Z := X;				$Z_3 = X_2$
X := V;		$X_4 = V$		
X < Y	$X_4 < Y_4$			
X := U;		$X_5 = U$		
Y := Z;			$Y_6 = Z_5$	
Z := X;				$Z_7 = X_6$
X := V;		$X_8 = V$		
...				

Trace Table 4.26

When the number of statements executed is finite and known, the method of *backward analysis* used above is very straightforward. But in some cases, the number of statements to be executed will be data dependent. Therefore a *forward analysis* can show when loops are to be terminated. That is, statement by statement, the values can be determined entirely in terms of the initial values. When these values are determined in a forward way, the conditional trace table becomes as follows.

Statements	Condition	X	Y	Z
X < Y	$X_0 < Y_0$			
X := U;		$X_1 = U$		
Y := Z;			$Y_2 = Z_0$	
Z := X;				$Z_3 = U$
X := V;		$X_4 = V$		
X < Y	$V < Z_0$			
X := U;		$X_5 = U$		
Y := Z;			$Y_6 = U$	
Z := X;				$Z_7 = U$
X := V;		$X_8 = V$		
X < Y	V < U			
X := U;		$X_9 = U$		
Y := Z;			$Y_{10} = U$	
Z := X;				$Z_{11} = U$
X := V;		$X_{12} = V$		
X < Y	V < U			
...				

Trace Table 4.27

At this point, it can be seen that the last five lines of the conditional trace table will be repeated over and over, with no further change in the values of the objects. Thus the while loop statement part behavior can be deduced directly from this conditional trace table. In order for the while loop statement to terminate, one of the conditions in the conditional trace table must be false. The while loop statement will terminate when the first such condition is false. Therefore, the conditions of the conditional trace table can be analyzed for termination and the values of the objects determined at those points. In this case, the conditions of the conditional trace table are

$X_0 < Y_0$
$V < Z_0$
$V < U$

so the conditions for termination will be their negations, namely

$X_0 >= Y_0$
$V >= Z_0$
$V >= U$

The termination condition $X_0 >= Y_0$ is derived from the condition in line 1 of the conditional trace table in the condition column and none of the objects X, Y, or Z has been altered, so this corresponds to a conditional assignment

```
(X >= Y -> null)
```

The termination condition $V >= Z_0$ is derived from line 6 and the values of X_4, Y_4, Z_4 valid in line 6 can be determined in terms of initial values directly as

$X_4 = V$
$Y_4 = Y_3 = Y_2 = Z_0$
$Z_4 = Z_3 = U$

which corresponds to a conditional assignment

```
(V >= Z -> X, Y, Z <- V, Z, U)
```

The termination condition `V >= U` is derived from line 11 and the values of X_8, Y_8, Z_8 valid in line 11 can be determined in terms of initial values directly as

$$X_8 = V$$
$$Y_8 = Y_7 = Y_6 = U$$
$$Z_8 = Z_7 = U$$

which corresponds to a conditional assignment

```
(V >= U -> X, Y, Z <- V, U, U)
```

These three conditional assignments make up the entire conditional assignment for the `Mystery_Loop`, as shown next.

```
[Mystery_Loop:
while X < Y loop
  X := U;
  Y := Z;
  Z := X;
  X := V;
end loop Mystery_Loop;]

= (X >= Y -> null;
   V >= Z -> X, Y, Z <- V, Z, U;
   V >= U -> X, Y, Z <- V, U, U)
```

In this case, the while loop statement terminates in exactly the three cases

```
X >= Y, V >= Z, V >= U,
```

and no more, and the loop behavior is defined only in the domain given by these three conditions. In particular, the while loop statement will not terminate when

```
X < Y and V < Z and V < U
```

4.5.2 While Loop Analysis Ensuring Termination

This failure in termination may have been unintentional, but can be determined by analysis rather than a disaster from a hung program during execution. For example, augmenting `Mystery_Loop` with an iteration counter, initialized before entering, incremented each loop, and tested at each entry to the loop, will ensure termination, and looks as follows.

```
Iteration_Counter := 0;
Terminating_Mystery_Loop:
while X < Y and
  Iteration_Counter < 3 loop
  X := U;
  Y := Z;
  Z := X;
  X := V;
  Iteration_Counter := Iteration_Counter + 1;
end loop Terminating_Mystery_Loop;
```

This program may or may not do what is desired, but it will terminate with no more than 3 iterations. Abbreviating Iteration_Counter to I_C, and including the initialization, the conditional trace table becomes as follows.

Statements	Condition	X	Y	Z	I_C
I_C := 0;					I_C_0=0
X < Y and I_C < 3	$X_0 < Y_0$ $I\_C_1 < 3$				
X := U;		X_2=U			
Y := Z;			Y_3=Z_0		
Z := X;				Z_4=U	
X := V;		X_5=V			
I_C := I_C+1;					I_C_6=1
X < Y and I_C < 3	$V < Z_0$ $I\_C_7 < 3$				
X := U;		X_8=U			
Y := Z;			Y_9=U		
Z := X;				Z_{10}=U	
X := V;		X_{11}=V			
I_C := I_C+1;					I_C_{12}=2
X < Y and I_C < 3	$V < U$ $I\_C_{13} < 3$				
X := U;		X_{14}=U			
Y := Z;			Y_{15}=U		
Z := X;				Z_{16}=U	
X := V;		X_{17}=V			
I_C := I_C+1;					I_C_{18}=3
X < Y and I_C < 3	$V < U$ $I\_C_{19} < 3$				

Trace Table 4.28

The termination condition X_0 >= Y_0 is derived from the condition in line 2 of the conditional trace table in the condition column and none of the objects X, Y, or Z has been altered, but I_C has been assigned 0, so this corresponds to a conditional assignment

```
(X >= Y -> I_C <- 0)
```

The termination condition V >= Z_0 is derived from line 9 and the values of I_C_6, X_6, Y_6, Z_6 valid in line 9 can be determined in terms of initial values directly as

$$\begin{aligned}
&\mathtt{I\_C}_6 = 1\\
&X_6 = X_5 = V\\
&Y_6 = Y_5 = Y_4 = Y_3 = Z_0\\
&Z_6 = Z_5 = Z_4 = U
\end{aligned}$$

which corresponds to a conditional assignment

```
(V >= Z -> I_C, X, Y, Z <- 1, V, Z, U)
```

The termination condition `V >= U` is derived from line 16 and the values of $\mathtt{I\_C}_{12}$, X_{12}, Y_{12}, Z_{12} valid in line 16 can be determined in terms of initial values directly as

$$\begin{aligned}
&\mathtt{I\_C}_{12} = 2\\
&X_{12} = X_{11} = V\\
&Y_{12} = Y_{11} = Y_{10} = Y_9 = U\\
&Z_{12} = Z_{11} = Z_{10} = U
\end{aligned}$$

which corresponds to a conditional assignment

```
(V >= U -> I_C, X, Y, Z <- 2, V, U, U)
```

The termination condition $\mathtt{I\_C}_{19}$ `>= 3` is derived from line 24 and the values of $\mathtt{I\_C}_{18}$, X_{18}, Y_{18}, Z_{18} valid in line 24 can be determined in terms of initial values directly as

$$\begin{aligned}
&\mathtt{I\_C}_{18} = 3\\
&X_{18} = X_{17} = V\\
&Y_{18} = Y_{17} = Y_{16} = Y_{15} = U\\
&Z_{18} = Z_{17} = Z_{16} = U
\end{aligned}$$

which corresponds to a conditional assignment

```
(I_C >= 3 -> I_C, X, Y, Z <- 3, V, U, U)
```

These four conditional assignments make up the entire conditional assignment for the initialized `Terminating_Mystery_Loop`, as shown next.

```
[Iteration_Counter := 0;
Terminating_Mystery_Loop:
while X < Y and
  Iteration_Counter < 3 loop
  X := U;
  Y := Z;
  Z := X;
  X := V;
  Iteration_Counter := Iteration_Counter + 1;
end loop Terminating_Mystery_Loop;]

= (X >= Y -> I_C <- 0;
   V >= Z -> I_C, X, Y, Z <- 1, V, Z, U;
   V >= U -> I_C, X, Y, Z <- 2, V, U, U;
   I_C >= 3 -> I_C, X, Y, Z <- 3, V, U, U)
```

As already noted, it can be shown that this conditional assignment will be defined for all initial values of `I_C`, `X`, `Y`, `Z`. The proof is based on the value of `I_C` exceeding 2 in a finite number of loop iterations, being initialized at `0`, and incremented by `1` each iteration.

4.5.3 While Loop Analysis with Loop Behaviors

The foregoing analysis of `Mystery_Loop` can be done in two steps for better efficiency, by first analyzing the sequence of four statements that make up the loop body as a concurrent assignment, then using this concurrent assignment in the while loop statement conditional trace table. First, the trace table for the loop body using a forward analysis is

Statements	X	Y	Z
X := U;	X_1 = U		
Y := Z;		Y_2 = Z_0	
Z := X;			Z_3 = U
X := V;	X_4 = V		

Trace Table 4.29

with

$X_4 = V$
$Y_4 = Y_3 = Y_2 = Z_0$
$Z_4 = Z_3 = U$

and the concurrent assignment is

```
X, Y, Z <- V, Z, U;
```

So the while loop statement can be restated in the non-legal Ada, but simpler form with a concurrent assignment, as

```
Mystery_Loop_2:
while X < Y loop
  X, Y, Z <- V, Z, U;
end loop Mystery_Loop_2;
```

Now, the conditional trace table for this while loop statement in minimal form becomes

Statements	Condition	X	Y	Z
X < Y	X_0 < Y_0			
X, Y, Z <- V, Z, U;		V	Z_0	U
X < Y	V < Z_0			
X, Y, Z <- V, Z, U;		V	U	U
X < Y	V < U			
X, Y, Z <- V, Z, U;		V	U	U
...				

Trace Table 4.30

and the last two lines are repeated and will continue to be repeated indefinitely. As before, the conditional assignment statement can be derived directly from this condensed conditional trace table as follows:

```
(X >= Y -> null;
 V >= Z -> X, Y, Z <- V, Z, U;
 V >= U -> X, Y, Z <- V, U, U;)
```

4.5.4 While Loop Analysis with Integer Data

The previous examples have been based on simple assignments between variables within the loops. When integer data is subject to arithmetic operations within loops, the operations must be checked for satisfactory results in each step of every loop. For example, consider integer data declared as

```
type Small_Integer is range -999..999;
X, Y, Z: Small Integer := 0;
```

in the while loop statement

```
Additive_Loop:
while X < Y loop
  X := X + Z;
  Y := Y - Z;
end loop Additive_Loop;
```

and analyze whether underflow or overflow is possible in its execution. In general, X is increased by value Z and Y decreased by the same amount until X becomes larger than Y. But is that always the case? For the case

```
X = 5, Y = 10, Z = 2,
```

X becomes 7, Y becomes 8, then X becomes 9, Y becomes 6, so the loop is terminated because X > Y. But for the case

```
X = 900, Y = 950, Z = 100,
```

X goes out of its bound to 1000, so the program is terminated. As another case

```
X = 5, Y = 10, Z = -2
```

X heads for -999, Y exceeds 999 even faster, so the program is terminated again. Thus, it is apparent that the while loop Additive_Loop does not always terminate successfully. That may still be satisfactory if the while loop is used only if it terminates correctly, not for all possible data. Some analysis can determine the conditions necessary on X, Y, Z > 0 for correct termination. They are that for minimum integer n > 0 and for X < Y

```
999 >= X + Z * n >= Y - Z * n >= -999.
```

The inner inequality can be reformulated as

```
0 < Y - X <= 2 * Z * n,
```

and the value of n becomes

```
min n >= (Y - X) / (2 * Z)
```

or

```
n = min ((Y - X) / (2 * Z))
```

Now, this value for n can be substituted in the original string of inequalities, dividing out `Z` in the fraction to obtain

```
999 >= X + min (Y - X)/2 >= Y - min (Y - X)/2 >= -999.
```

Thus, if `Y - X` is even, `min (Y - X)/2 = Y/2 - X/2`, and

$$
\begin{aligned}
X_{final} &= X + Y/2 - X/2 = X/2 + Y/2 \\
Y_{final} &= Y - Y - Y/2 + X/2 = X/2 + Y/2
\end{aligned}
$$

and if `Y - X` is odd, `min (Y - X)/2 = Y/2 - X/2 + 1/2`, and

$$
\begin{aligned}
X_{final} &= X + Y/2 - X/2 + 1/2 = X/2 + Y/2 + 1/2 \\
Y_{final} &= Y - Y/2 + X/2 - 1/2 + Y/2 - 1/2
\end{aligned}
$$

assuming, of course, that the intermediate and final calculations for `X` and `Y` are all contained in the declared range -999..999. In this case, `X + Z, X + 2 * Z, ...` must all be <= 999 and `Y - Z, Y - 2 * Z, ...` must all be <= 999. To fail, for some n,

```
X + (n-1) * Z < Y - (n-1) * Z
X + n * Z > 999 or Y - n * Z < -999.
```

In the second example above, `X = 900, Y = 950, Z = 100, n = 1,` where the first condition holds, namely

```
X + 0 * Z < Y - 0 * Z,
```

but the first part of the second condition fails, namely

```
X + 1 * Z > 999.
```

4.5.5 Exercises

1. Consider Mystery_Loop_3

   ```
   Mystery_Loop_3:
   while X < Y loop
     X := U;
     Y := Z;
     Z := V;
   end loop Mystery_Loop_3;
   ```

   and determine its conditional trace table and part behavior.

2. Add iteration counter I_C to ensure termination of Mystery_Loop_3 as

   ```
   I_C := 0;
   Terminating_Mystery_Loop_3:
   while X < Y and I_C < 4 loop
     X := U;
     Y := Z;
     Z := V;
     I_C := I_C + 1;
   end loop Terminating_Mystery_Loop_3;
   ```

   and determine its conditional trace table and part behavior.

3. Consider Mystery_Loop_4 with INTEGER objects declared as

   ```
   type My_Integer is range 0..4;
   First, Second : My_Integer;

   Mystery_Loop_4:
   while First < Second loop
     First := First + 1;
     Second := Second - 1;
   end loop Mystery_Loop_4;
   ```

   and determine its conditional trace table and part behavior. Check termination for all possible values of First, Second in order to determine how long the conditional trace table must be.

4. Consider Mystery_Loop_5 with INTEGER objects declared as

   ```
   type My_Integer is range 0..1_000;
   First, Second : My_Integer;

   Mystery_Loop_5:
   while First < Second loop
     First := First + 1;
     Second := Second - 1;
   end loop Mystery_Loop_5;
   ```

   and determine its conditional trace table and part behavior. The conditional trace table of Mystery_Loop_5 will be quite long, but will be very regular, so identify its regularity and define its elements symbolically in terms of initial values of First, Second.

5. Consider Mystery_Loop_6 with INTEGER objects declared as

```
type My_Integer is range 0..1_000;
First, Second, Third : My_Integer;

Mystery_Loop_6:
while (First + Second) < Third loop
  First := First + Second;
end loop Mystery_Loop_6;
```

and determine its conditional trace table and part behavior. The conditional trace table of Mystery_Loop_6 will be quite long, but will be very regular, so identify its regularity and define its elements symbolically in terms of initial values of First, Second, Third. Will this loop statement terminate for all initial values? Can this loop statement terminate program execution for any initial values?

6. Consider Mystery_Loop_7 with INTEGER objects declared as

```
type My_Integer is range 0..1_000;
First, Second, Third : My_Integer;

Mystery_Loop_7:
while (First + Second) < Third loop
  Second := First * Second;
end loop Mystery_Loop_7;
```

and determine its conditional trace table and part behavior. The conditional trace table of Mystery_Loop_7 will be quite long, but will be very regular, so identify its regularity and define its elements symbolically in terms of initial values of First, Second, Third. Will this loop statement terminate for all initial values? Can this loop statement terminate program execution for any initial values?

7. Consider the Small_Integer data X, Y, Z and the **while** statement Additive_Loop above and determine the range of data X, Y, Z, X < for which the **while** statement fails to terminate correctly.

Chapter 5

Sequential Ada II

Now that you have had the appropriate background, it is time to explore further the programming language Ada. In this chapter, more complex combinations of *data types* will be examined and some new data types will be introduced. Next, more Ada *statements* will be introduced to give you more flexibility and to allow you to express your algorithms in a more natural way. A statement is an action to be performed by the program. It can be a simple statement or it may be arbitrarily complex. Statements can be combined into sequences of statements to perform useful actions.

5.1 Ada Types and Objects

5.1.1 Enumeration Types and Objects

In Chapter 2, the notion of a data type was introduced. Remember that a data type consists of a set of values and a set of operations upon those values. Some of the data types that you have already seen include INTEGER, CHARACTER, and BOOLEAN. There are many more data types; in fact, Ada has provisions for you to create your own data types, yielding an infinite possibility for data types. One of the most useful data types is an *enumeration* data type.

Recall that the CHARACTER data type was described in Chapter 2 as a form of an enumeration type. The full definition of an enumeration type was deferred until other concepts had been discussed. Let us now return to that discussion and see how an enumeration type is defined and how the type CHARACTER is really a specific example of an enumeration type.

Enumeration data types allow what the name implies; namely, declaring an identifier to be the name of a list of other, literally enumerated, identifiers that serve as values. "Literally enumerated" means that each possible value for objects of the type is specified literally in a list. For example, the *enumerated type declaration*

```
   type Color is (RED, ORANGE, BLUE, YELLOW, GREEN, PURPLE);
```

declares the identifier `Color` as a data type with 6 values. `RED` is the first value, `ORANGE` follows, then `BLUE`, and so on. The semicolon (`;`) is part of the declaration. Note that these values are indeed identifiers as defined in Chapter 2. They must start with a letter and be followed by letters, digits, or underlines. Further, underlines can only be used singly to separate letters or digits, and may not start or end an identifier. Note also that the enumeration identifiers are listed in all uppercase letters because that is the style that is being used; Ada views these simply as identifiers and does not distinguish the difference between uppercase and lowercase letters.

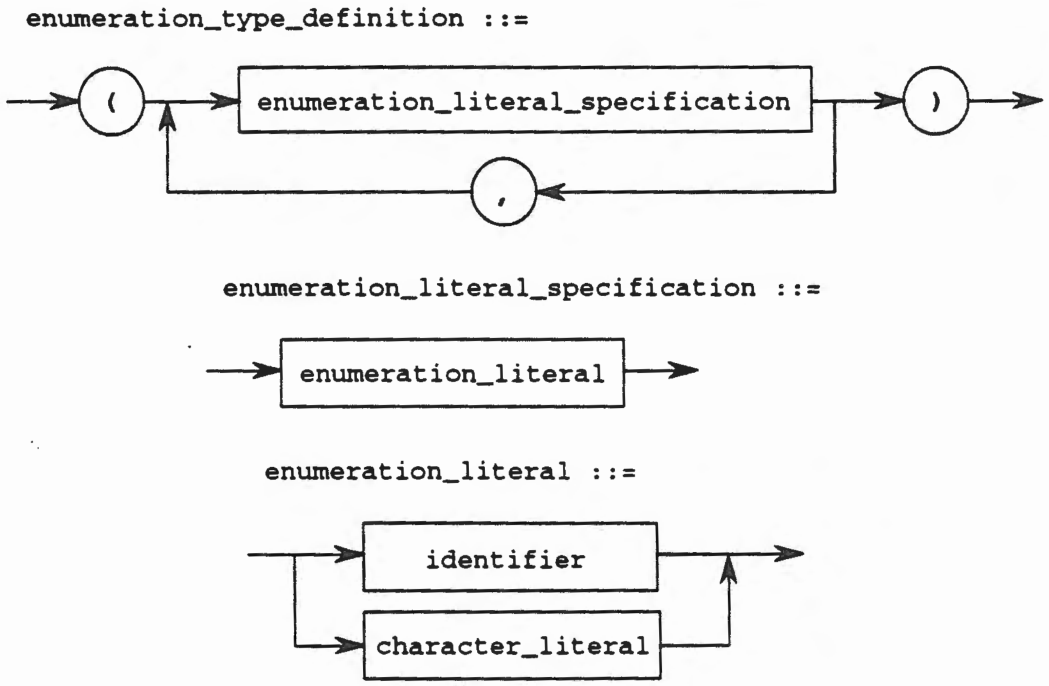
Remember that a type consists of set of values, which in this case are explicitly enumerated in the type definition, *and* a set of operations on those values. Where are operations on this type declared and what are they? The answer is that the operations are *implicitly* declared when you list the set of possible values and the operations themselves are the same for all enumerated types. They consist of such operations as `'SUCC` and `'PRED`. More information on the operations common to all enumerated types, and implicitly declared when the possible values are enumerated (listed), will be provided later in this chapter.

The formal syntax for an enumeration data type in Ada is given in the three syntax productions in Syntax Definition 5.1 with two intermediate words, namely `identifier` and `character_literal`, both defined previously in Section 2.3.2. In Chapter 2, we limited our syntax to that necessary to define the CHARACTER data type. Now we will demonstrate how simple it is to expand this definition to allow other enumeration types besides CHARACTER. Note that in the following Syntax Definition we have added another possibility for an `enumeration_literal`, namely `identifier`. We will now show how this adds flexibility for the user to define new enumerated data types.

```
enumeration_type_definition ::=
  ( enumeration_literal_specification
  {, enumeration_literal_specification} )

enumeration_literal_specification ::=  enumeration_literal

enumeration_literal ::=  identifier | character_literal
```

Enumeration Data Type
Syntax Definition 5.1

This syntax can also be expressed in graphic form as indicated in Syntax Chart 5.1.


enumeration_type_definition ::=
(
enumeration_literal_specification
)
,
enumeration_literal_specification ::=
enumeration_literal
enumeration_literal ::=
identifier
character_literal

Enumeration Data Type
Syntax Chart 5.1

For example:

```
type Important_Dates is (JANUARY_1, APRIL_15, AUGUST_2,
                         DECEMBER_25);
```

is an enumeration type. Note that it is perfectly acceptable to continue the list of possible values for the enumeration type onto the next line, as long as each new line starts with a new identifier, *i.e.*, you may not split an identifier across two lines.

Another example is:

```
type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
              SATURDAY, SUNDAY);
```

For this last case, the ideas of Chapter 2 can be applied as follows to identify two subtypes of the enumerated type `Week`. These are:

```
subtype Work_Week is Week range MONDAY..FRIDAY;

subtype Week_End is Week range SATURDAY..SUNDAY;
```

These two subtype declarations use the reserved word **range** to define a *contiguous* sublist of the values of the type `Week`, which is denoted by the first and last values of the sublist separated by a double period (`..`). Note that the values in the subtype must all be contained in the parent type and that they constitute a contiguous sublist, *i.e.*, all the values are next to each other, without gaps, in the list. In this case, the declarations shown have created two subtypes that have different names and different possible values. However, both subtypes are compatible with the parent type `Week`.

When a subtype is said to be *compatible* with another type or subtype, this means that the operations defined for one apply to the other. Also the types may be freely mixed in expressions, *etc.*, except for possible range constraints. For example, given the object declarations,

```
Pay_Day : Work_Week;
Today   : Week;
```

which are objects declared to have different names for their (sub)types, Ada's strong type checking rules would seem to prohibit mixing these types. However, the following sequence of statements is valid in Ada,

```
Pay_Day := FRIDAY;     -- a valid value for Work_Week
Today   := THURSDAY;   -- a valid value for Week
if Today = Pay_Day then
  TEXT_IO.Put("Today is Payday!");
end if;
```

The comparison operation "=" requires that both types be the same type or compatible types. Thus, since `Pay_Day` is an object of type `Work_Week` and since the type `Work_Week` is a subtype of type `Week`, `Pay_Day` is compatible with the object `Today` which is of type `Week`. Thus, the comparison operation is valid and will not cause an exception to be raised.

Operations

The operations available for all enumerated types include assignment, relational and membership tests, and the attributes given in the next paragraph.

Assignment	:=					
Relational	=	/=	<	<=	>	>=
Membership	in	not in				

The *assignment* operation consists of the ability to assign enumeration literals to objects declared to be of the same enumeration type, as well as the ability to assign an object of an enumerated type to another object of the same enumerated type. If an attempt is made to assign a value to an enumeration object that is not one of the permissible values enumerated in the type definition, then the exception CONSTRAINT_ERROR will be raised. The *relational* operation consists of the ability to determine whether two objects or literals of the same enumeration type are equal (=) or to determine their relationship to one other according to the order in which they are listed in the type definition. For example, in the type `Week` discussed earlier, `MONDAY < THURSDAY` because `MONDAY` appears earlier in the listing of values in the type definition. Similarly, `FRIDAY > TUESDAY` because `FRIDAY` appears after `TUESDAY` in the type definition. Finally, the *membership* operation provides the ability to determine whether or not an enumeration literal or an object is contained in an enumeration type's set of possible values. For example, given the type `Work_Week` as defined above, the membership operator "in" may be used as follows:

```
if TUESDAY in Work_Week then
  TEXT_IO.Put (Item => "This is a valid literal.");
end if;
```

and the complementary operator "not in" may be used as follows:

```
if SATURDAY not in Work_Week then
  TEXT_IO.Put (Item => "Hooray for weekends!");
end if;
```

Attributes

As mentioned previously, an enumerated type also has certain attributes that are implicitly defined and declared when the enumeration type is declared. For example, the enumerated type `T` has attributes that include the following:

`FIRST` the first value of type `T`, denoted `T'FIRST`
`LAST` the last value of type `T`, denoted `T'LAST`
`PRED` if the value of `X` is not `T'FIRST`
then `T'PRED(X)` is the literal that precedes `X`
else CONSTRAINT_ERROR is raised
`SUCC` if the value of `X` is not `T'LAST`
then `T'SUCC(X)` is the literal that succeeds `X`
else CONSTRAINT_ERROR is raised
`POS` the position of the literal or identifier within
the enumeration, starting from zero
`VAL` the value of the enumeration at the given position

For the examples above,

```
Color'FIRST = RED

Important_Dates'LAST = DECEMBER_25

Week'PRED(SUNDAY) = SATURDAY

Week'SUCC(SUNDAY) will raise CONSTRAINT_ERROR.

Week'POS(TUESDAY) = 1 -- MONDAY is at position zero

Important_Dates'VAL(2) = AUGUST_2

Week'VAL(Week'POS(SUNDAY)) = SUNDAY  -- inverse operations
```

Although this discussion has been about enumerated types that can be declared by you, there are certain types that are *predefined* as enumeration types. The type BOOLEAN and the type CHARACTER are nothing more than predefined enumeration types and have no special properties. That is, all of the points just made about enumeration types apply equally to the types BOOLEAN and CHARACTER and all of the points made in our earlier discussion of the types BOOLEAN and CHARACTER apply equally to all other enumeration types.

Enumeration Objects

Enumeration data objects must be declared before their references in the executable parts of programs. For example, the following declarations,

```
Day   : Week;
Dress : Color := RED;
```

declare `Day` as an object of type `Week`, and `Dress` as an object of type `Color` with an initial value of `RED`. The declaration of objects that hold enumeration values is no different than the declaration of objects that hold characters as described in Chapter 2.

Enumeration Expressions

Expressions involving enumeration values adhere to the same type checking rules as discussed previously for CHARACTER values. All of the operations must take values or objects of the same enumerated type. For the relational and membership operations, in every case the results of such operations are boolean values FALSE or TRUE. These operations can be used in more complex boolean expressions.

For example, some enumeration expressions based on the types discussed in this section might be:

```
RED < BLUE                  -- type Color, result is TRUE
SATURDAY in Work_Week       -- FALSE
APRIL_15 >= DECEMBER_25     -- FALSE
MONDAY < BLUE               -- Error - not of the same type
```

5.1.2 Enumeration Input/Output

Ada provides a very elegant capability for direct input/output of enumeration types. This allows a user to read or write enumerated types directly. The manner in which Ada provides this capability for enumeration types is exactly analogous to that which we saw in Chapter 2 for integer types. In future chapters, the complete capability to input/output enumerated types will be fully explained. However, for now a template will be provided to allow you to Get and Put enumeration values for the enumeration types that you declare.

In the package TEXT_IO that has already been partially described for you, there is, in addition to the nested package called INTEGER_IO that we saw earlier, another nested package called ENUMERATION_IO. This package is also not actually a "real" package at all. Instead, it is a template for creating specific instances of packages that will have the ability to read and write the enumerated literals that you declare. Once again, the details of how this is accomplished by the compiler are beyond the scope of our discussions here. For now, we are only concerned that you understand the global concepts of what is happening when an instance of this template is created and that you know how to create an instance for your use.

How do we "fill in" the template called ENUMERATION_IO? The answer is that we must only supply it the name of the enumeration type that we desire to have available for input/output operations, again, just exactly as we did earlier for integer types. For the enumeration type `Color` defined earlier in this section, this is accomplished in the following manner,

```
   package Color_IO is new TEXT_IO.ENUMERATION_IO (Color);
```

This provides us with a new package, called `Color_IO`, that we can use like any other "normal" package, but this package's purpose is to provide us with the ability to perform input/output operations on the type `Color`. Again, for our purposes now, it is not imperative that you fully understand all of the hidden details of what is occurring here, only that you understand how to use this feature for your own Input/Output operations of your own enumeration types.

As another example, to be able to read and write the literals for the type BOOLEAN we would first need to have a line in our program such as,

```
   package Boolean_IO is new TEXT_IO.ENUMERATION_IO (BOOLEAN);
```

which provides us a new package called `Boolean_IO` that will give us the capability of reading and writing BOOLEAN literals directly.

These new packages are used like any other package. It is impossible to detect that these packages, created by this mechanism, are any different than any other package that we might create by simply writing out the specification and body ourselves. This mechanism does have a name. We call it a *generic instantiation* because the package template, in our example ENUMERATION_IO, is called a *generic* package in Ada, and the process of creating an instance of the template is called *instantiation*. Thus, `Boolean_IO` is a generic instantiation of the generic package `TEXT_IO.ENUMERATION_IO`.

We use a generic instantiation in the same manner as any other package. Thus, to read an enumeration literal of the type `Color`, given the generic instantiation `Color_IO`, we would use

```
Color_IO.Get (Item => My_Color);   -- assuming that we had previously
                                   -- declared My_Color : Color;
```

and similarly, we could write this literal as follows,

```
Color_IO.Put (Item => My_Color);
```

It is perhaps instructive to note, however, that enumeration I/O for Ada restricts the output of enumerated types to be either all uppercase letters or all lowercase letters. There is no provision for normal capitalization, *i.e.*, the first letter only is capitalized. Accordingly, and since the default of these two options is uppercase, the style used in this textbook is to always write enumeration literals in all uppercase letters. Thus, on output, the enumeration literals will appear as they are in the programs that produce them.

Recall how the **Width** parameter can be used in an integer Put statement to specify the width of the field in which the number is written. We should naturally wonder if this feature is available for other types, specifically for enumeration types. The answer is yes! The **Width** parameter to an enumeration type Put statement works in the exact same manner as for integer types with one noticeable exception. The values in an enumeration type are output left justified in their field, *i.e.*, the spaces that will be written if the length of the enumeration value to be written out is less than the field width are added at the end of the enumeration value, not placed at the beginning as in the case of integer values. Thus, the following statement,

```
Color_IO.Put (Item => RED, Width => 6);
```

will cause the output printed to be R, E, D, with three spaces following. The total field width is still six spaces, but the values are in the leftmost portion, not the rightmost as they are in integer types.

Normally, you do not need to provide a field width when you output enumeration values. However, if you need to format your output in some special way it is important to know what operations are available to you.

5.1.3 Real Types

Real values conceptually consist of an unbounded, infinitely long set of numeric values with unbounded and infinite precision. In other words, these values consist of all possible values of numeric quantities extending from negative infinity to positive infinity on the number line, with an infinite level of precision. This data type is called the universal_real. Unfortunately, there are no machines that can represent an infinite range of values nor can they represent even a portion of these values with anything approaching infinite precision. Therefore, we are forced by hardware constraints to limit universal_real to a subset of these values that are representable on any given machine. In consequence, each implementation of Ada will provide a machine dependent type called FLOAT that will consist of the representable values of floating point numbers with some machine limited precision. The details of how these values are stored are not important for our purposes, but suffice it to say that these floating point numbers are represented internally differently than the integer values that we have already discussed.

In addition, Ada provides us with the conceptual ability to specify either a relative bound on the error of the actual value and its internal representation, or an absolute bound on this error. What this means and why it is important will be explained in this section.

5.1.3.1 Floating Point Types and Subtypes

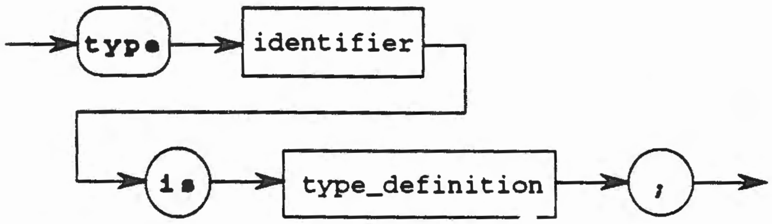
As was previously mentioned, the floating point values in any given implementation of Ada represent the precision and range limitations of the underlying computer. As in the integer types that we have already seen, there is a predefined type called FLOAT that is provided for every Ada implementation. In addition, the user is free to define other floating point types, specifying the relative precision, and optionally the range, to be used. The formal syntax for a floating point type and subtype is given next.

```
full_type_declaration ::= type identifier is type_definition;
type_definition ::= real_type_definition
real_type_definition ::= floating_point_constraint
floating_point_constraint ::= floating_accuracy_definition
                              [range_constraint]
floating_accuracy_definition ::= digits static_simple_expression
subtype_declaration ::= subtype identifier is subtype_indication;
subtype_indication ::= type_mark [constraint]
type_mark ::= type_name | subtype_name
constraint ::= floating_point_constraint
```

Floating Point Type and Subtype Declaration
Syntax Definition 5.2

This syntax can also be shown in graphic form as follows.

full_type_declaration ::=

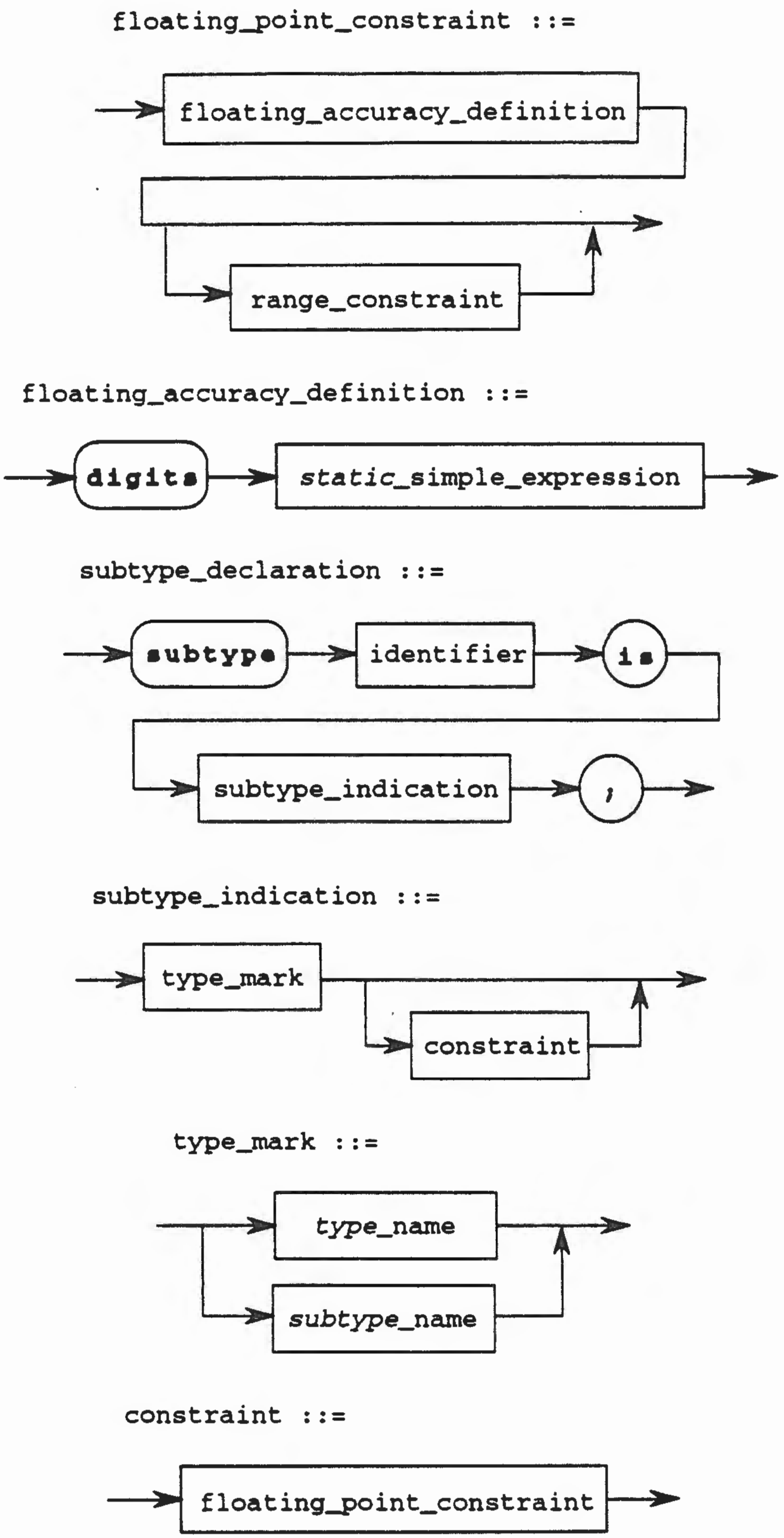


type_definition ::=

real_type_definition

real_type_definition ::=

floating_point_constraint



floating_point_constraint ::=

floating_accuracy_definition ::=

subtype_declaration ::=

subtype_indication ::=

type_mark ::=

constraint ::=

Floating Point Type and Subtype Declaration Syntax Chart 5.2

Sample floating point type and subtype declarations are

```
type Ratio is digits 3 range 2.0 .. 5.3;
type Hand is digits 5;
type Length_Measures is digits 6 range -100.0 .. 100.0;
subtype Small_Hand is Hand digits 3 range 0.0 .. 1.2;
subtype New_Ratio is Ratio;
subtype My_Length_Measures is Length_Measures range -10.0 .. 10.0;
```

In these examples, the type Ratio is defined as a floating point type where objects of this type are represented internally with three digits of precision throughout the range from 2.0 to 5.3. The type Hand is represented with five digits of precision, and takes the default range of values for floating point numbers as provided by the implementor of the Ada compiler. The type Length_Measures defines a floating point type with six digits of precision throughout the range -100.0 to 100.0. Note that a given Ada implementation is allowed to limit the number of digits of precision that it supports, according to the underlying hardware. Thus, on one machine it may be the case that six digits of precision is the maximum that will be allowed, whereas on another machine you might be able to have up to nine digits of precision. The compiler will tell you at compile time if it is unable to support the requested level of precision.

The subtype Small_Hand represents a restricted subset of the values in the type Hand, reducing the required precision to only three digits and introducing a range constraint not present in the type Hand. Note that the precision specified in a subtype declaration must be no greater than the precision required in the type. Thus, you can lower the required precision, but you cannot increase it. The subtype New_Ratio does not restrict the type Ratio in any way, either in range or in precision. In effect, this is simply an alias for the type Ratio and achieves the same effect as a renaming of this type. The subtype My_Length_Measures represents a reduction in the possible range of values from the type Length_Measures without reducing the required precision.

In addition to user defined types, which are based on an analysis of the requirements of the problem to be solved, each implementation of Ada provides a predefined type named FLOAT. This type is used in the same manner as the predefined type INTEGER. The predefined type FLOAT has a degree of precision that is implementor defined and a range that is also implementor defined. The values of these limits can be found in the required Appendix F to the Reference Manual for the Ada Programming Language (LRM) that the vendor must supply with the compiler.

We could declare objects using these types in the same manner as we have seen previously, such as

```
My_Hand : Hand;
The_Distance : My_Length_Measures := 1.7;
Pi : FLOAT := 3.14159;
```

It might be useful to examine what is meant by the term *digits of precision*. What does it mean to specify that there will be five digits of precision? This term is used to refer to the number of significant digits that an object will require. If we have five significant digits, then the leftmost five digits in a canonical representation of the number are the only digits that will be meaningful. This is perhaps easier to see in the illustration given in Figure 5.1.

```
type NUMBERS is digits 3 range 0.0..20_000.0;

0.001, 0.002, 0.003, ... 98.1, 98.2, ... 997.0, 998.0, 999.0,
1000.0, ... 1010.0, ... 10000.0, 10100.0
```

Relative Precision
Figure 5.1

Figure 5.1 also shows what is meant by the term *relative precision.* Note that numbers closer to zero give greater significance to the number of significant digits. Thus, near zero in this example, three digits of precision means that the difference between any two values in this type is only 0.001 or one one-thousandth. As we move further from zero, even with the same number of digits of precision, the significance of those digits lessens. Thus, when we approach 100, we see that with three significant digits we have 98.1, 98.2, *etc.*, where the distance between two successive values in the type is now every tenth instead of every one thousandth. As we pass 100 and approach 1000, we see that 997.0, 998.0, *etc.* are separated by 1.0 which is three orders of magnitude less significant than when we were close to zero and the distance between two successive values was one thousandth! Thus, we say this is *relative precision* because the number of significant digits does not change, but the meaningfulness of those digits increases as we approach zero and decreases as we get further away from zero. Similarly, the number of significant digits is merely the number of digits that have meaning in a number. Thus, the number 10_000_000.0 represented by a type that was declared to be digits 3 would mean that only the leftmost three digits (10000000.0) would have meaning. Adding one to this number would not change it because we would lose the significance of the one, *i.e.*, 10000000.0 + 1.0 = 10000000.0 because 10000001.0 is not representable due to the limitation of only three significant digits.

This phenomena can be a very large problem and may materially affect the results obtained in somewhat innocent looking computations. Thus, you must always be vigilant not to allow this type of a problem to creep into your algorithms. There is a whole field of study called *numerical analysis* which concentrates on solutions to problems of this nature. If you are interested in learning more about roundoff error, relative precision problems, truncation, and internal representation of numeric values you should consider taking a course in numerical analysis.

Attributes

For any given floating point type, say `F`, the attributes available to the user include the following

`FIRST` Yields the first value of type `F`, denoted `F'FIRST`

`LAST` Yields the last value of type `F`, denoted `F'LAST`

`DIGITS` Yields the number of decimal digits in the decimal mantissa of the model numbers for this subtype, denoted `F'DIGITS`

`MANTISSA` Yields the number of binary digits in the binary mantissa of the model numbers of this subtype, denoted `F'MANTISSA`

`EPSILON` Yields the absolute value of the difference between the model number 1.0 and the next model number above for this subtype, denoted `F'EPSILON`

Attributes can be used to query the system at runtime to determine values that may influence the executing algorithm. They are very useful for achieving portability.

As you can see, there is a whole new world of numeric values possible when we include the floating point types. Many computations that would not be possible with only the whole numbers are now available for our use. But there is another kind of real number that we still need to discuss, the fixed point number.

5.1.3.2 Fixed Point Types and Subtypes

Fixed point types are somewhat unique in programming languages. If you have ever used another programming language, then more than likely the real numbers that you have used were floating point numbers. In Ada, we have the choice between the floating point numbers as previously described, and fixed point numbers. You will recall that floating point numbers represented a *relative bound* on the error of a number's representation. Fixed point numbers, in contrast, represent an *absolute bound* on this error.

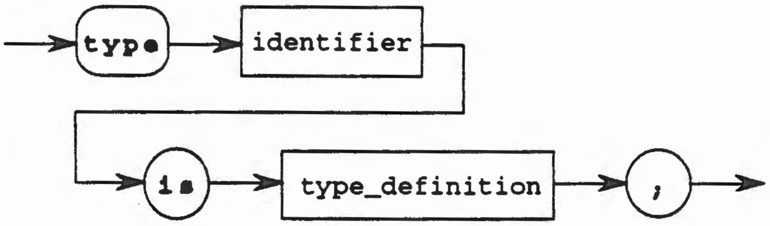
The formal syntax of a fixed point number is given next.

```
full_type_declaration ::= type identifier is type_definition;
type_definition ::= real_type_definition
real_type_definition ::= fixed_point_constraint
fixed_point_constraint ::= fixed_accuracy_definition
                           [range_constraint]
fixed_accuracy_definition ::= delta static_simple_expression
subtype_declaration ::= subtype identifier is subtype_indication;
subtype_indication ::= type_mark [constraint]
type_mark ::= type_name | subtype_name
constraint ::= fixed_point_constraint
```

Fixed Point Type and Subtype Declaration
Syntax Definition 5.3

This syntax can also be shown in graphic form as follows.

full_type_declaration ::=

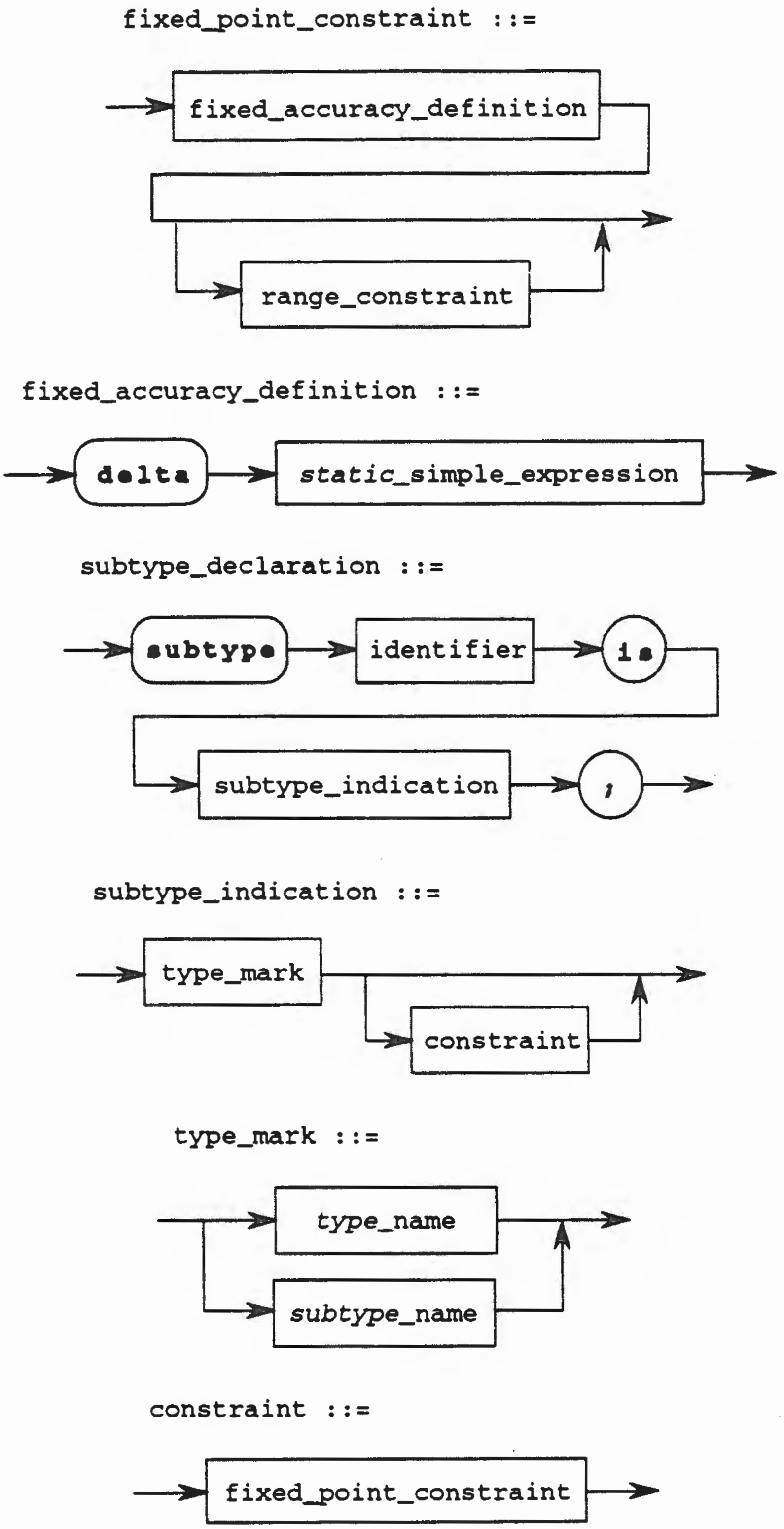


type_definition ::=

real_type_definition

real_type_definition ::=

fixed_point_constraint



fixed_point_constraint ::=

fixed_accuracy_definition ::=

subtype_declaration ::=

subtype_indication ::=

type_mark ::=

constraint ::=

Fixed Point Type and Subtype Declaration
Syntax Chart 5.3

Note that the range constraint is shown as optional. However, when you are declaring a new fixed point **type** the range constraint is *required*. On the other hand, when you declare a fixed point **subtype** the range is optional. Therefore, in the Syntax Definition and Syntax Chart the range must be shown as optional, even though we now know that for a type declaration it is required.

Sample fixed point type declarations are

```
type Dollar_Value is delta 0.01 range 0.0 .. 1.0;
type Very_Precise is delta 0.0006 range -10.185 .. 17.893;
subtype Not_So_Precise is Very_Precise delta 0.001;
subtype Even_Less_Precise is Very_Precise delta 0.01 range -1.0
..1.0;
subtype My_Dollar is Dollar_Value;
```

In these examples, the type `Dollar_Value` is defined as a fixed point type (note the use of the reserved word **delta** to mean fixed point types, while the reserved word **digits** is used for floating point types) where successive values of objects of this type are 0.01 apart throughout the entire range of 0.0 to 1.0. Remember that in floating point types, with relative precision, the closer we were to zero the more precise was our represented value. In this fixed point type, throughout the entire range of values, all values will be 0.01 apart[1]. This is termed an *absolute bound* on the error, contrasted with the relative bound provided by floating point types.

For the type `Very_Precise`, we see that objects of this type will be represented by values separated by 0.0006 throughout the range -10.185 to 17.893. Recall that for fixed point type definitions, the range *must* be provided. The subtype `Not_So_Precise` represents a restriction on the type `Very_Precise` where the delta or difference between successive values has been reduced to 0.001 from 0.0006. As in floating point types, we can reduce the required precision, but we may not increase it. Also, this subtype does not have, nor does it require, a range constraint. Since one is not provided, it will have the same range constraint as the type `Very_Precise`. The subtype `Even_Less_Precise` is also a restriction on the type `Very_Precise`, but in this case we have not only reduced the required precision (delta), we have also reduced the applicable range.

Finally, for the subtype `My_Dollar`, we have merely provided an alias or renaming of the type `Dollar_Value` without restricting either the accuracy requirements or the range. Thus, objects of this type will be identical to those of `Dollar_Value`.

All fixed point types in Ada must be declared by the user. In fact, there is only one predefined fixed point type, `Duration`, which is used to represent time intervals. In particular, there is no predefined type FIXED that can be equated to the predefined type FLOAT.

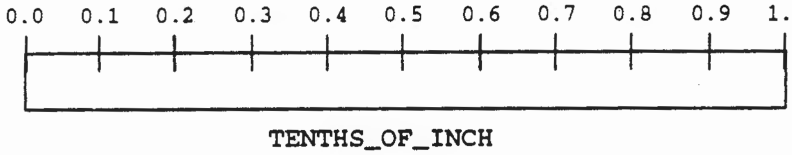
We could declare objects using these types in the same manner as we have seen previously, such as

```
The_Dollar : Dollar_Value;
The_Value : Very_Precise := -9.285;
```

[1] This is not quite true. We have actually specified the largest power of two not greater than the **delta** value. However, this is not a course in numerical analysis, so for our purposes we will infer that the **delta** represents the actual bound on the error.

In the last section, we explained what we meant by the term relative precision by showing a figure that illustrated our points. In this section we will do the same to illustrate the concept of absolute accuracy. Refer to Figure 5.2.

```
type TENTHS_OF_INCH is delta 0.1 range 0.0..1.0;
```



Absolute Error
Figure 5.2

In this figure we illustrate that throughout the entire range of permissible values for this type, the distance between any two representable values is always the same, 0.1. It does not matter whether we are close to zero or far away, the distance between any two values will always be the same. This is what is meant by an ***absolute bound*** on the error. Of course, in real life, we must deal with machines that use binary to represent these values. Therefore, the actual values will not be exactly as illustrated here. However, for our purposes, this explanation will suffice.

Attributes

For any given fixed point type, say `F`, the attributes available to the user include the following

`FIRST`	Yields the first value of type `F`, denoted `F'FIRST`
`LAST`	Yields the last value of type `F`, denoted `F'LAST`
`DELTA`	Yields the value of the delta specified in the fixed accuracy definition for this subtype, denoted `F'DELTA`
`MANTISSA`	Yields the number of binary digits in the binary mantissa of the model numbers of this subtype, denoted `F'MANTISSA`
`SMALL`	Yields the smallest positive (non-zero) model number for this subtype, denoted `F'SMALL`
`LARGE`	Yields the largest positive (non-zero) model number for this subtype, denoted `F'LARGE`

Thus, we see that fixed point numbers are conceptually very intriguing, but due to limitations in their implementations, they are not as useful as we might hope. For this reason, we will deal with floating point values for most of the rest of this course.

5.1.4 Exercises

1. Define an enumeration type named `CARS` that contains the values for any four of your favorite automobiles. What operations are available for your type?

2. Why might you want to have a subtype such as `Work_Week` of a type such as `Week`?

3. Given the type definitions in this section, what will be written to the screen after the execution of the following code segment?

```
if SATURDAY in Week_End then
    TEXT_IO.Put (Item => "Hooray - a day off!");
else
    TEXT_IO.Put (Item => "Too bad you have to work!");
end if;
```

4. Using the type declarations in this section, what would be the result of the following assignment statement? Explain your answer.

```
Dress := Color'VAL (Color'POS( GREEN));
```

5. Write the necessary statement to instantiate an input/output package for the type CARS that you defined in Exercise 1. Will the enumeration values be written in uppercase or lowercase letters?

6. Show what would be output (including all spaces, if any) for each of the following Put statements. Assume the package `Color_IO` has already been properly instantiated for the type `Color`.

```
a) Color_IO.Put(Item => RED, Width => 4);
b) Color_IO.Put(Item => GREEN, Width => 4);
c) Color_IO.Put(Item => ORANGE, Width => 2);
d) Color_IO.Put(Item => PURPLE, Width => 6);
```

7. Is the following user-defined type declaration permissible in Ada? Explain your answer!

```
type Boolean is (TRUE, FALSE);
```

8. What do you think happens if you try to take the predecessor of the first value in an enumeration type? What about the successor to the last value in the type?

9. What is the difference between a floating point number and a fixed point number?

10. Declare a real number with a relative precision of 5 digits over the range 1.7 to 11.3, giving it the name `Conversion_Type`.

11. Declare a number with an absolute error of one ten-thousandth over a range of 2.2 to 5.7 calling it `Calibration_Type`.

5.2 Introduction to One Dimensional Arrays

An array is an object that may contain multiple values simultaneously. It is an object declared from a composite type. We will find that arrays can be quite useful to software engineers to represent objects that exist in the real world such as a deck of cards or a list of grades for any given examination for a class of students. These ideas represent a linear list of component values that are all of the same type. We will then extend these ideas to show how arrays may be constructed to have multiple dimensions.

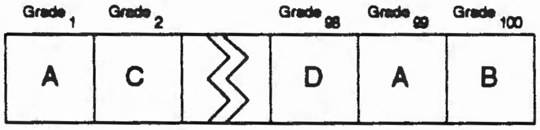
5.2.1 Arrays with Only One Dimension

So far in this textbook, we have used data that contained a single value. Such data are called *scalar* values and thus the types are referred to as scalar data types. Examples include INTEGER, CHARACTER, and BOOLEAN, as well as any enumeration type that you might define. Each object of this type may contain at most a single value of the type at any given time. Thus, given the following declaration,

```
My_Count : INTEGER;
```

the object `My_Count` may contain the value 10, or -24, or 0, but not all of them at the same time. Only a single integer value may ever be contained in the object `My_Count` at any time during the execution of the program.

There are times, however, when it is convenient to speak about values collectively. Data types that can hold multiple values simultaneously are called *composite* types in Ada. For example, suppose that we had to maintain the grades for an entire class of students. Further suppose that this class had 100 members. One way we could accomplish our task would be to declare 100 objects, each of which would then hold a single student's grade. Of course, it would be awkward to access these values because we would need identifiers for these objects such as `Grade1`, `Grade2`, `Grade3`, ..., `Grade99`, and `Grade100`. Needless to say, this would be quite cumbersome and tedious. What we need is a mechanism whereby we could use a single identifier to refer to the grade and a subscript to refer to the student whose grade is sought. This is exactly the way a situation such as this is handled in mathematics. There we would use a subscript like $\texttt{Grade}_1$, $\texttt{Grade}_2$, *etc.* to refer to each individual grade. Note that in mathematics we have used a single identifier; the subscripts then refer to which of several possible values for this identifier we mean. Note that if we were to refer to just `Grade` we would be speaking about all 100 of the grades. This example can be visualized in the following illustration:



Grade Subscripts
Figure 5.3

Similarly, in Ada we can use an *array* to represent our values, with a single identifier denoting the array and a subscript denoting any particular value in the array . The idea is the same as in mathematics; a collection of values of the same type is represented by a single identifier with subscripts This composite of information is then easier for us to deal with and is more natural to discuss. It also eliminates the requirement for creating large numbers of identifiers to represent related objects. One limitation of this representation scheme is that all of the values must be of the same type. Thus, we can have an array of INTEGERs and an array of CHARACTERs, but we cannot have an array of some CHARACTERs and some INTEGERs.

The formal syntax for an array data type in Ada is given by the following syntax productions .

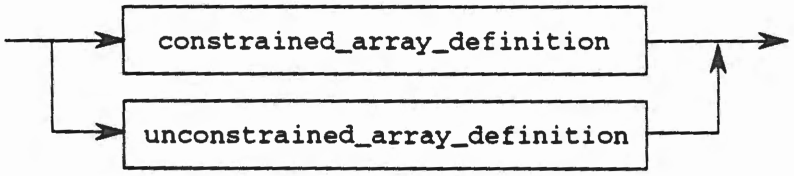
```
array_type_definition ::= constrained_array_definition |
                          unconstrained_array_definition

constrained_array_definition ::=
  array index_constraint of component_subtype_indication
```

```
index_constraint ::= ( discrete_range )

discrete_range ::= discrete_subtype_indication | range

subtype_indication ::= type_mark [constraint]

constraint ::= range_constraint

range_constraint ::= range range

range ::= simple_expression .. simple_expression | range_attribute

attribute ::= prefix ' attribute_designator

prefix ::= name

attribute_designator ::= simple_name [( static_expression )]

component_subtype_indication ::= subtype_indication

unconstrained_array_definition ::=
  array ( index_subtype_definition )
  of component_subtype_indication

index_subtype_definition ::= type_mark range <>

type_mark ::= type_name | subtype_name
```

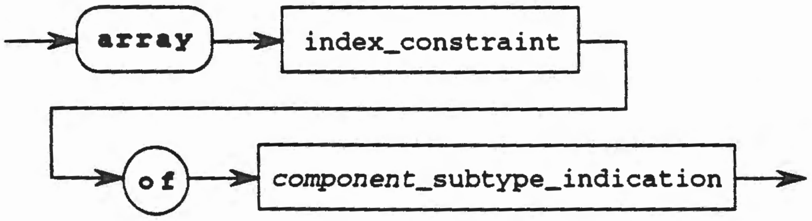
Array Data Type
Syntax Definition 5.4

This syntax can also be expressed in graphic form as indicated in Syntax Chart 5.4

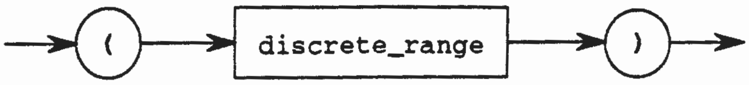
array_type_definition ::=



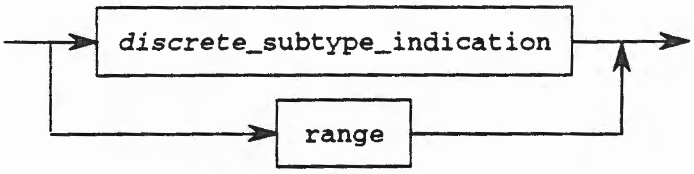
constrained_array_definition ::=



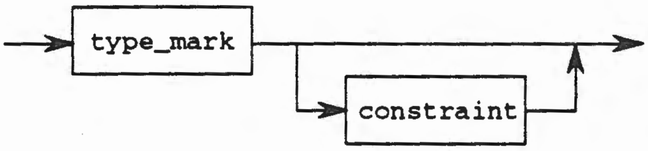
index_constraint ::=



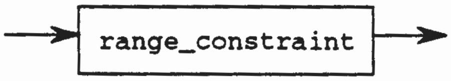
discrete_range ::=



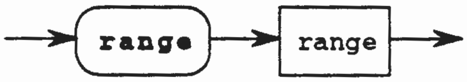
subtype_indication ::=



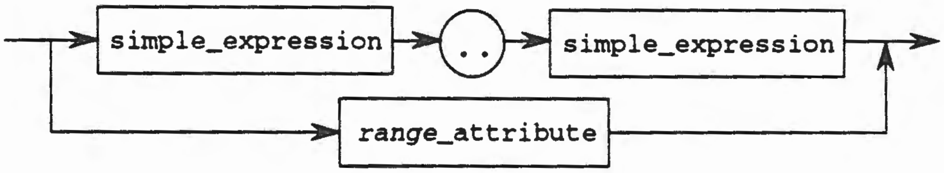
constraint ::=



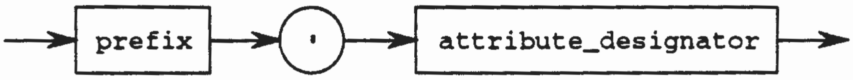
range_constraint ::=



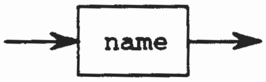
range ::=



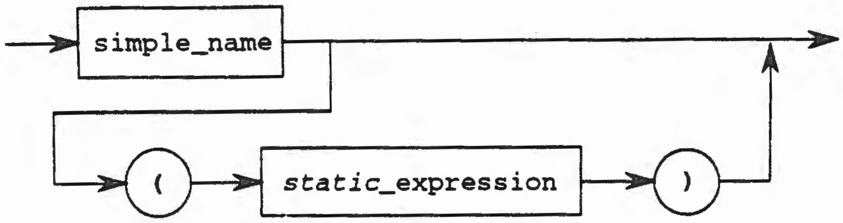
attribute ::=



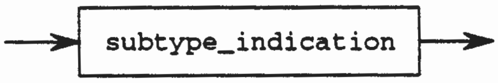
prefix ::=



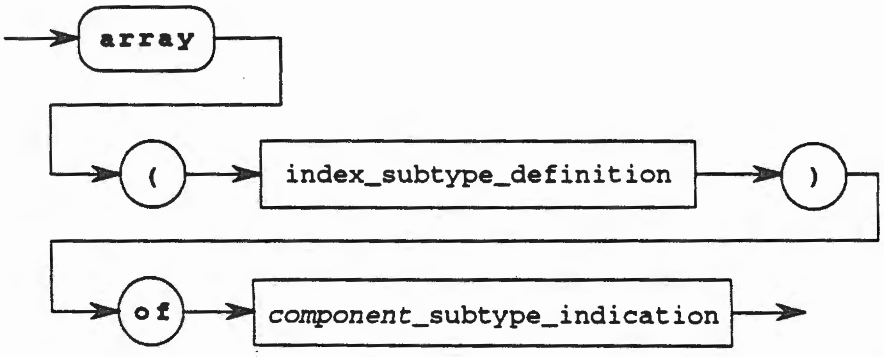
attribute_designator ::=



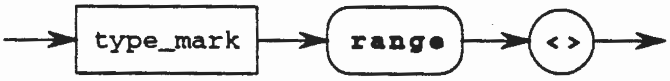
component_subtype_indication ::=



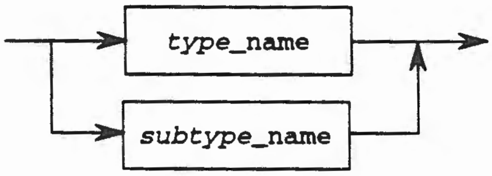
unconstrained_array_definition ::=



index_subtype_definition ::=



type_mark ::=



Array Data Type
Syntax Chart 5.4

For example, in Ada we can define a constrained one dimensional array (later we will discuss a generalization of these concepts to more than one dimension) as a collection of values of the same type represented by a single name[2] using the following declaration,

```
type Grade_Vector is array (1 .. 100) of INTEGER;
```

This declares a new type, an array with the name `Grade_Vector`, to be a contiguous collection of 100 INTEGERs. Then the object declaration,

```
Grade : Grade_Vector;
```

declares an object capable of holding 100 INTEGERs. In order to select any single value in this array we merely subscript the identifier as we would in mathematical notation. Thus,

```
Grade(25) := 88;
```

assigns the value 88 to the 25th position in the collection of INTEGER values called `Grade`.

In all of our previous examples, we have shown the lower bound of the array to be one. This is not a requirement in Ada. The indices used in the array definition can be any discrete value. This means that not only can we have negative indices in the array, we do not even need to have numeric values! Consider the following examples,

```
type Sine_Curve is array (-1 .. 1)of INTEGER;
type Number_Line is array (-100 .. 275) of POSITIVE;
type Work_Hours is array (MONDAY .. SUNDAY) of POSITIVE;
```

all of which define types that are perfectly acceptable in Ada.

5.2.2 Array Operations

The operations available for an array include assignment, indexing, membership, logical, relational, and explicit conversion, as well as the attributes described in the next section.

Aggregate Assignment					:=		
Catenation (One-dimensional only)	&						
Logical (Boolean components only)	and	or	xor	not			
Membership						in	not in
Relational						=	/=
Relational (Discrete components only)	<	<=	>	>=			

Aggregate assignment is the ability to assign all of the values to an array simultaneously. For example, given the following declarations,

```
type Grades is (A, B, C, D, F);
type Class_List is array (1 .. 5) of Grades;
Class_Grades : Class_List;
```

[2] A one dimensional array in Ada is similar to a vector in mathematics.

we could assign the grades to this Class_List by the following single aggregate assignment statement,

```
Class_Grades := (A, C, D, A, B);
```

In this example, the first storage position in the array Class_Grades would get the value A, the second would get the value C, *etc.* Note that we must have *exactly* the same number of grades as we have storage positions in the array. Also note that the entire set of values to be assigned is enclosed in parenthesis and each individual value is separated by a comma.

Sometimes it may be inconvenient to list a value for every storage location in the array, especially if the array is large. Also, it often happens that we want to assign the same value to all of he positions in an array, say to initialize the values. Therefore, there is a mechanism to allow us to do this. We merely use the reserved word **others** followed by an arrow (=>), followed by the value we want for all of the storage positions. For example, suppose that we had the following array declarations,

```
type Books is array (1 .. 100) of NATURAL;
  -- a count of the number of each kind of book
My_Books : Book;
```

then we can set the initial value of this object My_Books to be all zeros by the single assignment statement,

```
My_Books := (others => 0);
```

This is a convenient notation when the array is large and is a handy shortcut for you to remember.

The *Catenation* operation is applicable to all one dimensional arrays (the only kind that you have seen so far), but are used primarily for STRINGs. We will discuss more about this operation in the following section.

The *Logical* operations on one dimensional arrays that have boolean components are exactly the same as on objects that have a single value. These operations allow an entire array to be used in logical tests where conditional expressions are used.

The *Membership* operation allows us to test an array to determine if a particular value exists as one of the components of the array.

The *Relational* operations allow two arrays to be compared. For the relational operators '=' and '/=', the components of the array may be of any type. For comparisons using the operators '<', '<=', '>', or '>=', the type of the components of the array must be boolean. Except for this one restriction, the use of the relational operators for arrays is the same as their use for any scalar type.

5.2.3 Array Attributes

Arrays also have attributes that are implicitly declared when the array is declared. Attributes available to all arrays are:

`FIRST`	the first index position
`LAST`	the last index position
`LENGTH`	the total number of indices
`RANGE`	a shorthand for the range `T'FIRST .. T'LAST` where `T` is any array

For example, given the array declaration and object declaration,

```
type Card_Deck is array (1 .. 52) of Card;
  -- assume that Card is defined for the suits and
  -- ranks of a deck of cards
Cards : Card_Deck;
```

then the following are true,

```
Cards'FIRST = 1
Cards'LAST  = 52
Cards'LENGTH = 52
Cards'RANGE is 1 .. 52
```

There are more attributes defined for certain types of arrays and we will examine them when we discuss multi-dimensional arrays in Section 5.3. For now, these are sufficient to make good use of arrays.

5.2.4 Array Input and Output

Since an array is a collection of many values of the same type represented by a single name, it is usually not possible to put or get an entire array of values in the same manner that a single value is read or written into an object. With the exception of a very special predefined array type that we will study in the next subsection, STRINGs, a different approach is used to perform input and output operations on arrays.

Essentially the approach that we use to get and put values in an array is to examine each separate location in the array using the subscripting operation mentioned above. When we do this, that storage location, even though it is part of an array, is by itself just a single value of the array. If this single value happens to be a scalar object[3], then we can use the normal Put and Get operations defined for that object's type as we have been doing. For example, in the object `Grade` defined above, the array position `Grade(25)` represents a single value in the array `Grade`. In this case, it represents the 25th storage location out of the 100 storage locations defined to be part of the object `Grade`. However, that particular location is just a single location where an object of type INTEGER is stored, since `Grade_Vector` is a contiguous collection of 100

[3] Recall that a scalar object is one that contains a single value, as opposed to a composite object, like the array, that contains potentially multiple values.

INTEGERs, and Grade is an object of that type, and we have subscripted Grade to isolate a single position within it. Thus, Grade(25), or any other subscripted position within Grade, represents an object of type INTEGER no different than any other INTEGER object that we have already studied. Consequently, all of the operations that we were able to perform on these objects we can also perform on a subscripted position of Grade. In particular, we can call Get and Put from the instantiated integer input/output package to perform these operations. This, then, is how we can perform input/output operations on an array. We merely subscript each possible value within the array to isolate a single object of a scalar type, then perform the operation on that single value.

The next question that arises is how to we subscript every location in the array? The answer is that we merely use a looping mechanism. Since we know in advance what the bounds of the array happen to be, we can use a form of deterministic looping, or definite iteration. Such a mechanism is the **for loop**. Thus, to write out the value in the array Grade defined above, assuming that it had already been provided with a complete set of values, we could use the following code segment,

```
Array_Output_Loop:
for I in 1 .. 100 loop -- the known bounds of the loop
  Int_IO.Put( Grade(I) );
end loop Array_Output_Loop;
```

Here we loop through the array indexing each storage location individually, one at a time, from lowest to highest, writing out the value stored there. It should be obvious how to use this mechanism to Get values into the array and so an example of that process will not be given.

From a software engineering perspective, there is a better way to write the last example. Recall that since we knew that there were 100 components in this array, we wrote the loop to iterate from 1 to 100. Suppose that we repeated this several places in our program. Now further suppose that we decided to modify the type so that the range of the array went from 1 .. 200 instead of 1 .. 100. We would now have to go through our entire program and check each loop to make sure that the loop iteration was correct. This is an error-probe operation. A better solution is to use the attributes of the array to dynamically adjust this for us. Consider the following example,

```
New_Array_Output_Loop:
for I in Grade'RANGE loop
  -- using the attributes of the array to set the bounds of the loop
  Int_IO.Put( Grade(I) );
end loop New_Array_Output_Loop;
```

We see in this example that if we change the definition of the array type, *i.e.*, we change the range of the index, this dynamic mechanism will automatically change everywhere it is used. This saves us an enormous amount of work and serves to make our programs more amenable to change. This enhances the maintenance process.

We will see more about the use of arrays later in this course. In fact, we will compare and contrast the two different kinds of arrays, constrained arrays, whose indices are known at compile time (static), and unconstrained arrays, whose indices are dynamic. We have already seen an example of a constrained array so we will now turn our attention to a predefined array that happens to be unconstrained, namely STRING. Later we will return to this topic for a more thorough discussion of these types.

5.2.5 STRING Types and Objects

STRING is a predefined Ada type that is represented as a contiguous collection of characters. In Ada terms, the type STRING consists of a one dimensional unconstrained **array** of CHARACTERs that are indexed by POSITIVE INTEGERs. Having given you the technical jargon, we will now explain the type STRING in more common terms, then return to the Ada definition.

As previously stated, arrays will be defined more generally in Section 5.3. For now we will examine the type STRING as it is declared in Ada, then consider what each portion of the declaration means. Note that although the type STRING is a predefined Ada type, it could have easily been declared by you. The type STRING is an unconstrained array given in two declarations:

```
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

type STRING is array (POSITIVE range <>) of CHARACTER;
```

As defined, an array is a list of identical data objects that can be treated as a single composite object. Thus, the definition of STRING given above states that a STRING is a list of CHARACTERs that can be considered as a single object. The index used to access any given CHARACTER in the STRING must be a POSITIVE number within some unspecified range. When you declare the array object you must provide the actual indices to be used.

Imagine a consecutive list of characters in a row. Collectively, they make up a STRING. However, it is still possible to access individual CHARACTERs that make up the STRING. This is accomplished by naming the STRING and specifying in parenthesis which particular component of the string that you desire to access. This implies that the components of the string must have some positional value so that they can be accessed. This is precisely what is given by the type POSITIVE in the type definition for STRING. For example, some STRING objects are

```
Question : constant STRING := "How Many Characters?";
Name     : constant STRING := "Sherry";
Course   : STRING (1 .. 6);
Address  : STRING (5 .. 27);
```

In the first example, the object `Question` is a `STRING` with a length of 20 (count the characters and spaces) and an initial (and constant) value of `"How Many Characters?"`. The length is implicitly supplied by the length of the actual string used to provide the initial value. In the second example, the length of `Name` is implicitly determined in a similar manner from the length of the initial value, namely `"Sherry,"` so that the number of characters allowed in this string (its length) is 6. In the next example, the object `Course` is declared to be a string that contains 6 characters with positions denoted 1 .. 6. Finally, in the last example, the object `Address` is a string capable of containing 23 characters with the positions of each character numbered in order from 5 to 27. Note that in every case we had to provide POSITIVE numbers to use as the indices, either explicitly as in the last two examples, or implicitly, as in the first two examples.

As was mentioned previously, it is desirable to access individual characters of the STRING on an as needed basis. The mechanism to accomplish this task is to name the string, then provide in parenthesis the positional number of the particular character that is desired from the string. Thus, using the objects declared above,

```
Question(3)  = 'w'  -- third letter in "How Many Characters?"
Question(20) = '?'
Name(3)      = 'e'  -- third letter in "Sherry"
```

and given the following valid assignments to the objects declared above,

```
Course  := "CS1001";
Address := "150 W. University Blvd.";
```

the following are valid accesses to individual characters,

```
Course(2)   = 'S'
Course(6)   = '1'
Address(5)  = '1'  -- the first position was declared to be 5
Address(10) = '.'  -- the sixth character is at position 10
```

Operations

The operations on STRINGS are no different than those on other one dimensional arrays. This is logical since STRINGS are, after all, nothing more than one dimensional arrays of CHARACTERs. There are, however, some special operations available only for objects of type STRING, including one operation that has already been discussed. Recall that the & operator is used for catenation. This operator can be used to join two STRINGs or to join a CHARACTER to the beginning of a STRING, or to join a CHARACTER to the end of a STRING. Thus, the following declarations:

```
Full_Name  : STRING (1 .. 10);
First_Name : STRING (1 .. 5) := "Mike ";
Last_Name  : STRING (1 .. 5) := "Smith";
A_Letter   : CHARACTER := 'A';
```

allow the following operations to be performed:

```
Full_Name := First_Name & Last_Name;
TEXT_IO.Put (Item => A_Letter & First_Name);  -- puts "AMike"
TEXT_IO.Put (Item => Last_Name & A_Letter);   -- puts "SmithA"
```

Note that in the first case, the length of the STRING `First_Name` is 5 characters and the length of the second STRING `Last_Name` is 5 characters, so the STRING object to which the catenation of these two STRINGs is assigned, namely `Full_Name`, must be declared to be 10 characters long in order to contain the catenation of the two STRINGs. If the STRING to which the catenated objects are to be assigned is not long enough to contain all of the characters from both objects, then an error results and the exception CONSTRAINT_ERROR is raised. Similarly, if the STRING to which the catenated objects are to be assigned is bigger than then both objects combined, then CONSTRAINT_ERROR will also be raised. In short, the target STRING (the one on the left of the assignment operator) must be *exactly* the same length as the sum of the lengths of the two objects on the right hand side of the assignment operator.

Input and Output for STRINGs

`TEXT_IO.Get` and `TEXT_IO.Put` procedures will get and put components of STRINGs as sequences of characters. A call on Get will attempt to read as many characters as declared for the STRING.

Consider the following program

```
with TEXT_IO;
procedure String_Program is
  Name : STRING (1 .. 9);
begin
  TEXT_IO.Get (Item => Name);
  if Name(1) < Name(2) then
    TEXT_IO.Put (Item => "Name starts upward");
  else
    TEXT_IO.Put (Item => "Name starts downward");
  end if;
  TEXT_IO.Put (Item => " with full name " & Name);
exception
  when others =>
    TEXT_IO.Put (Item => "An error occurred in this program.");
end String_Program;
```

If a value of Name is not found, this program will terminate after the exception handler writes out its message. The value for Name, if one exists, must be exactly nine characters long because the object Name is specified to be nine characters long. Any longer and only the first nine characters will be read; any shorter and the terminal will seem to hang while the program waits for the rest of the input characters.

Note also that the program compares the first two CHARACTERs of the STRING Name by using the notation Name(1) to mean the first CHARACTER in the STRING Name and Name(2) to mean the second CHARACTER in the STRING. The rest of the program is a simple explanation to the user based upon the relationships between these first two characters.

5.2.6 Exercises

1. Declare an array type called Work_Hours that is indexed by the enumeration values representing the days of the week having components that represent the number of hours worked by an individual during the week. Ensure that you properly define appropriate types to use in your array type declaration.

2. Suppose that you had an array object named Time_Card of type Work_Hours (see Exercise 1). What are two different ways to initialize all components of this array to the value zero?

3. Write a segment of Ada code to output all of the values in the array object in Exercise 2. What would you have to assume had been done before you could Put these values, besides providing values to the array components?

4. Write a segment of Ada code to output all of the values in the array object in Exercise 2. For this exercise, you must use the most appropriate attribute.

5. Declare an array type called Deck_Of_Cards that consists of 52 storage locations for objects of type Card. Use the appropriate type to represent the type Card. (Hint: remember that a card consists of a value and a suit.)

6. Declare an array called Regional_Burger_Sales. This array will be used to store information about the number of hamburgers consumed daily at each location within the region. There are 27 locations within the region, and the record sales for any given day is 25,000 hamburgers.

5.3 Multi-Dimensional Arrays

In the previous section we discussed a means by which we could represent multiple values in a single identifier, namely an array. This concept is fine as far as it goes, but it can easily be generalized into a more useful form. In this section we will study multi-dimensioned arrays to see how the concepts inherent in these data structures allow us to map real-world problems into internal abstractions that can be used in the solution of these problems.

5.3.1 Arrays with More than One Dimension

One dimensional arrays are quite useful and allow us to create abstractions that model real-world entities. This, in turn, allows us to solve many problems in a more straightforward way, which allows us to maintain these software systems in a less cumbersome manner. However, the abstraction does breakdown when we desire to represent entities that are not vectors, but are matrices. One simple solution to this problem is to have an array that has elements that are also arrays. For example, to represent a chessboard we might have:

```
type Chess_Piece is (PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING, EMPTY);
subtype Chess_Board_Size is POSITIVE range 1 .. 8;
type Chess_Board_Row is array (Chess_Board_Size) of Chess_Piece;
type Chess_Boards is array (Chess_Board_Size) of Chess_Board_Row;
Chess_Board : Chess_Boards;
```

This simple example represents the possible chess pieces as an enumerated type. The value EMPTY is included so as to allow for a given square to not have any chess pieces upon it. The subtype is created to limit the number of the rows and columns to that which make up the chessboard. Then each possible row is represented as an array that consists of eight storage locations, each of which can hold a value of type Chess_Piece. Finally, the chessboard type is represented as an array of eight of the chess board rows.

As you can see, while it possible to represent the chessboard in this manner it is confusing and unnatural. Further, references to any given board location are syntactically strange. For example, to indicate that we want to place a KNIGHT on row three, column five of the chessboard we would need to write

```
Chess_Board(3)(5) := KNIGHT;
```

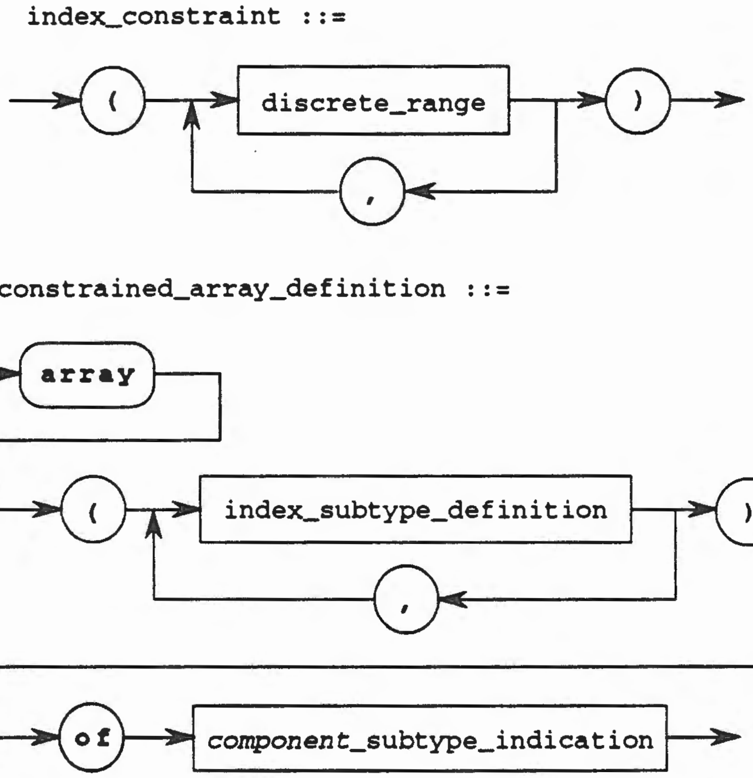
which looks very ugly. To see that this is syntactically correct, we see that the identifier Chess_Board represents an array that we index with the value three. This gives us the third component of the array Chess_Board which we know is itself an array. Thus, we have to further index into this component array at the fifth position to obtain the component located there which we know to be a storage location capable of holding an object of type Chess_Piece. Therefore, we see that this syntax is indeed correct, even though it is not intuitive or "pretty."

To solve this problem we extend the notion of an array to more than one dimension. The syntax for this extension is shown next in Syntax Definition 5.5 which is formed by simply extending the syntax definition given previously as Syntax Definition 5.4.

```
index_constraint ::= ( discrete_range {, discrete_range} )

unconstrained_array_definition ::=
  array ( index_subtype_definition {, index_subtype_definition} )
  of component_subtype_indication
```

Multi-Dimensional Array Data Type
Syntax Definition 5.5

This can also be shown graphically as Syntax Chart 5.5 as follows:



Multi-Dimensional Array Data Type
Syntax Chart 5.5

In order to show you how this simple extension extends our power of representation, consider the same chessboard definition but with the use of a multi-dimensioned array. We will use the first dimension to represent each row of the chessboard and the second dimension to represent each column. Thus, we obtain the following type definitions:

```
type Chess_Piece is (PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING, EMPTY);
subtype Chess_Board_Size is POSITIVE range 1 .. 8;
type Chess_Boards is array (Chess_Board_Size, Chess_Board_Size)
  of Chess_Piece;
Chess_Board : Chess_Boards;
```

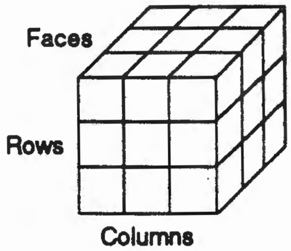
This allows us to make the same assignment as before in the following manner:

```
Chess_Board (3,5) := KNIGHT;
```

which looks somewhat less ugly than the last version. This syntax is more intuitive because it suggests that we are indexing into the array at row three, column five, to obtain the component that is located there. This is a storage location for an object of type `Chess_Piece` as before, but the notion of how to access this object is "cleaner" and more intuitive, *i.e.*, it involves the selection of a row and a column.

As you study the syntax definition you will note that there is no limit placed by the language on the number of dimensions that an array may contain. Thus, you may place a dimension for each portion of the problem that you logically want to model. Thus, to represent a Rubik's Cube as shown in Figure 5.4, you might have the following declarations:

```
subtype Rows is POSITIVE range 1 .. 3;
subtype Columns is POSITIVE range 1 .. 3;
type Faces is (FACE_1, FACE_2, FACE_3, FACE_4, FACE_5, FACE_6);
type Face_Colors is (RED, YELLOW, BLACK, ORANGE, BLUE, GREEN);
type Rubik_Cubes is array (Faces, Rows, Columns) of Face_Colors;
Rubik_Cube : Rubik_Cubes;
```



Rubik Cube
Figure 5.4

In this series of declarations, there are three dimensions, represented by the three ranges inside of the parentheses in the type declaration. The first dimension is the face of the cube that we are currently interested in, the next is the row on that face and the last is the column in the row on that face. If you think about it (and it may not hurt to pull out your old Rubik's cube to verify this), each of the colored squares that make up the Rubik's cube is completely and uniquely defined by specifying which face, row, and column you desire to talk about. In this manner, colors may be assigned to these locations and the cube may be completely described.

5.3.2 Array Operations and Attributes

The operations on a multi-dimensioned array are the same as those available for a single dimension array as described in Section 5.2.2, except where it was specifically noted that the operation applied to one-dimensional arrays only. There are some accommodations that must be made for the additional dimensions. For example, aggregate assignment in multi-dimensioned arrays is more difficult because we must specify each of the dimensions separately. Thus, to initialize the `Chess_Board` declared above to be all `EMPTY`, we must use the following notation:

```
Chess_Board := (others => (others => EMPTY));
```

which states that for all indices in the first dimension (Rows) and for all indices in the second dimension (Columns) the value to be assigned is EMPTY. Alternatively, we could have stated:

```
Chess_Board := (1 .. 8 => (1 .. 8 => EMPTY));
```

which would do the exact same thing. Thus, when we deal with multi-dimensioned arrays we must remember to consider and name all of the dimensions when we do an aggregate assignment. As a final example, the Rubik's cube could be initialized to all RED with the following aggregate assignment:

```
Rubik_Cube := (FACE_1 .. FACE_8 => (1 .. 3 => (1 .. 3 => RED)));
```

which says that for each of the six faces, for each row (all three of them) and for each column (again all three of them) assign the value RED to that location. This has the effect of initializing the cube to all locations being RED. Alternatively, we could have used:

```
Rubik_Cube := (others => (others => (others => RED)));
```

which would have accomplished the same thing.

The attributes are also the same as those described in section 5.2.3 with the addition of some specifically useful for multi-dimensioned arrays. These include

FIRST(N)	the first index position in the Nth dimension
LAST(N)	the last index position in the Nth dimension
LENGTH(N)	the total number of indices in dimension N
RANGE(N)	same as before but for dimension N

Thus, we can see again that the extension of the array concept to multiple dimensions is very straightforward and logical and introduces no new concepts.

5.3.3 Multi-Dimensional Array Input and Output

There is one item with regard to multi-dimensioned arrays that makes their use more difficult than one-dimensioned arrays and that is input and output operations. Since we have to consider multiple dimensions it is not as easy to get or put values with arrays that contain more than one dimension. Therefore, we need a mechanism to allow us to examine each element in each dimension. In short, we will need a loop, one for each dimension.

For example, to get a value for the Chess_Board declared above we would need a code segment that looks like the following:

```
Get_The_Row:
for This_Row in Chess_Board_Size loop
  Get_The_Column:
  for This_Column in Chess_Board_Size loop
  Chess_IO.Get(Chess_Board(This_Row, This_Column));
  -- Assume Chess_IO is an instantiation to allow
  -- Chess_Piece IO
  end loop Get_The_Column;
end loop Get_The_Row;
```

This code segment iterates over each row and again over each column within each row to get a value for the chess piece that corresponds to that location. Note the use of two loops because we have a two dimensional array.

To further illustrate this point, consider reading values into the Rubik's cube. We would need three loops because we have three dimensions. Thus, we would have a code segment such as:

```
Get_The_Face:
for This_Face in Faces loop
  Get_The_Row:
  for This_Row in Rows loop
    Get_The_Column:
    for This_Column in Columns loop
      Face_Color_IO.Get(
        Rubik_Cube(This_Face, This_Row, This_Column));
    -- Assume Face_Color_IO is an instantiation to allow
    -- IO for Face_Colors as defined above
    end loop Get_The_Column;
  end loop Get_The_Row;
end loop Get_The_Face;
```

Thus, we see that except for some special considerations necessitated by the additional dimensions, the concept of a multi-dimensioned array is very similar to the concept of array with a single dimension.

5.3.4 Exercises

1. Given the following object declaration

```
Name : STRING (6 .. 15) := "Clark Kent";
```

what is the value of each of the following:

a. Name(8)
b. Name(3)
c. Name(15)
d. Name(10)

2. What are the bounds (upper and lower limits) of the following object?

```
Notice : constant STRING := "You like Software Engineering!";
```

3. What is wrong with the following code segment and how could it be fixed?

```
Month : STRING (1 .. 8) := "December";
Day   : STRING (1 .. 3) := "22,";
Year  : STRING (1 .. 4) := "1991";
Date  : STRING (1 .. 10) := Month & Day & Year;
```

4. Determine the behavior of the following program including the possibility of termination from lack of input data.

```
with TEXT_IO;
procedure Week_End_Program is
  Day : STRING (1 .. 3) := "FRI";
begin
  TEXT_IO.Get (Item => Day);
  if Day = "SUN" then
    TEXT_IO.Get (Item => Day);
  else
    TEXT_IO.Put (Item => "Day is " & Day);
  end if;
exception
  when others => TEXT_IO.Put (Item => "An error has occurred.");
end Week_End_Program;
```

5. Determine the behavior of the following program including the possibility of abnormal termination from lack of data.

```
with TEXT_IO;
procedure Trend_Program is
  Trend : STRING (1 .. 9);
begin
  TEXT_IO.Get (Item => Trend);
  if Trend(1) < Trend(9) then
    TEXT_IO.Put (Item => "Trend is upward");
  else
    TEXT_IO.Put (Item => "Trend is downward");
  end if;
  TEXT_IO.Put (Item => " with full history " & Trend);
exception
  when others => TEXT_IO.Put (Item => "An error has occurred.");
end Trend_Program;
```

4. Write the appropriate statements to declare a checker board to be used in a game of checkers. Use a two-dimensional representation of the game board and an enumerated type to represent the checkers.

5. Is it possible to have an array with 100 dimensions? Justify your answer.

6. Show the code segment that would be used to read in values of the initial positions of the checkers in the checkerboard from Exercise 4.

7. Show how to initialize the checkerboard from Exercise 4 using an aggregate assignment. Will you need to extend your declarations from Exercise 4?

8. Suppose that we have a class with 25 students each with a unique student number in the range 0 .. 1_000. Each student has three test scores in the range 0 .. 100. Define an array to hold these students names and scores.

9. Suppose that we have five different classes exactly like the one described in Exercise 8 (except with different student names!). How could you extend the definition of your array to handle all of these classes?

10. Show the code segments that you would use to get values for the array declared in Exercise 8. Be sure to include prompts so that the user knows what data to enter at each Get statement.

11. Show the code segments that you would use to get values for the array declared in Exercise 9. Be sure to include prompts so that the user knows what data to enter at each Get statement.

5.4 Introduction to Records

In the previous section we introduced a composite data type called an array. Array objects are used to store lists of component values that are all of the same type. For example, a list of quiz grades for 100 students could easily be handled by an array object. However, arrays are not appropriate in cases where the data components to be stored are of different types. To handle these cases, we now introduce the concept of a record.

5.4.1 Records

A record is a list of component values that may be of different types. Records are used to store different, but related values as a single unit. For example, a student record might consist of an id number, a major code, and a year level. As another example, consider the date today. This composite value usually consists of a month name, a day number, and a year number. Thus it seems natural to define a single object, say `Date`, with three related components for the month, day, and year.

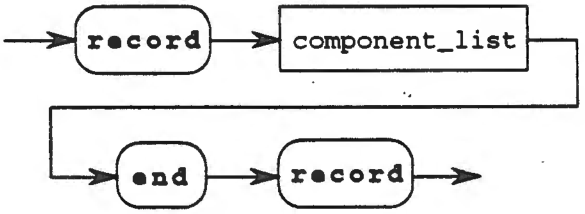
The formal syntax for a record data type in Ada is given by the following syntax productions.

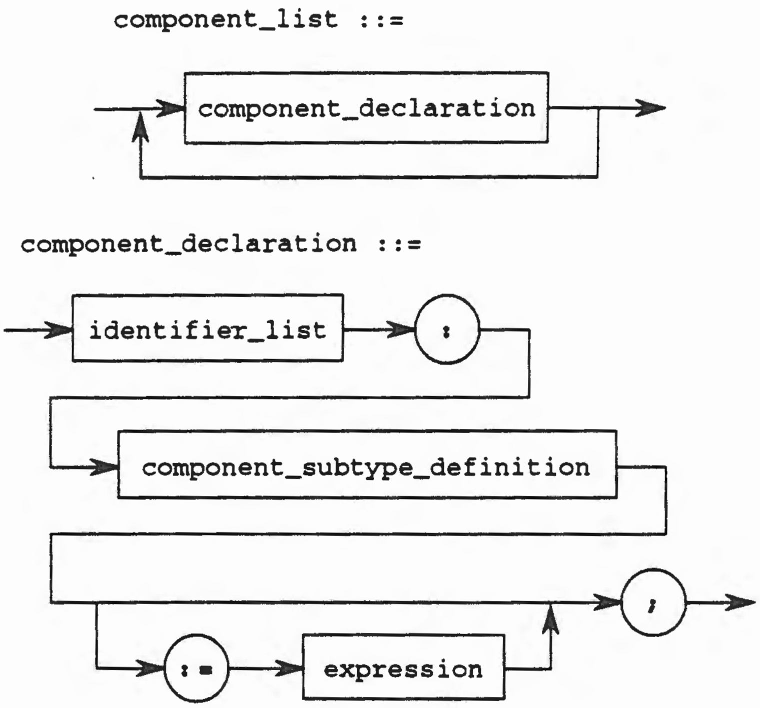
```
record_type_definition ::=
  record
    component_list
  end record

component_list ::= component_declaration {component_declaration}

component_declaration ::=
  identifier_list : component_subtype_indication [:= expression];
```

Record Data Type
Syntax Definition 5.6

The syntax can also be expressed in graphic form as indicated in Syntax Chart 5.6

record_type_definition ::=





Record Data Type
Syntax Chart 5.6

Suppose we wish to define a record type with a suitable structure for the student record mentioned earlier. We will begin by analyzing the component values. For the student id number we will use values between `0` and `9999`. For the major code we will use values of `CS`, `CP`, `EE`, or `MA`. Finally, for year level we will use values of `1`, `2`, `3`, or `4`. Thus the full record type definition looks like the following:

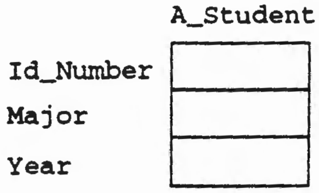
```
type Student_Id_Number is range 0 .. 9999;
type Major_Code is (CS, CP, EE, MA);
type Year_Level is range 1 .. 4;
type Student_Record is
record
  Id_Number : Student_Id_Number;
  Major : Major_Code;
  Year : Year_Level;
end record;
```

Note that `Student_Record` now describes a type with three components, `Id_Number`, `Major`, and `Year` of differing types. Note also that the type of each component must be indicated. Any data type may be used as the component of a record.

Now that we have described the type `Student_Record`, we are free to create objects of that type. For example,

```
A_Student : Student_Record:
```

declares an object named `A_Student` with enough contiguous storage for the three component values desired. Pictorially we have the following object:

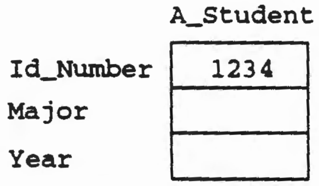


A_Student Object
Figure 5.5

Each component location within the record structure is called a field. We may access the various fields of a record directly through the use of a "dot notation". For example,

```
A_Student.Id_Number := 1234;
```

places the value `1234` into the `Id_Number` field of the record `A_Student`. That is, our picture now looks like this:

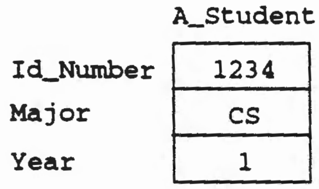


Initializing Id_Number
Figure 5.6

To continue with our initialization suppose we perform the following assignments:

```
A_Student.Major := CS;
A_Student.Year := 1;
```

Now our picture is:



Initialized A_Student Object
Figure 5.7

5.4.2 Record Operations

The operations available for records include assignment, membership, component indication, relational, and explicit conversion.

Aggregate Assignment	:=	
Membership	in	not in
Relational	=	/=

Aggregate assignment for records is similar to aggregate assignment for arrays. It provides the ability to assign values to all fields of a record at the same time. For example, if we define a record structure and create objects suitable for various dates:

```
type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN, JUL,
  AUG, SEP, OCT, NOV, DEC);
type Day_Number is range 1 .. 31;
type Year_Number is range 1900 .. 2100;
type Date_Record is
record
  Month : Month_Name;
  Day : Day_Number;
  Year : Year_Number;
end record;

Today : Date_Record;
Birthday, Holiday : Date_Record;
```

we could assign values to the objects by the following aggregate assignment statements.

```
Today := (MAR, 30, 1992);
Birthday := (Month => MAY, Day => 15, Year => 1900);
Holiday := (DEC, 25, 1992);
```

The statement

```
Today := (MAR, 30, 1992);
```

is equivalent to the three statements:

```
Today.Month := MAR;
Today.Day := 30;
Today.Year := 1992;
```

The statement

```
Birthday := (Month => MAY, Day => 15, Year => 1990);
```

uses named association of field and value. That is, `Month => MAY` means for the `Month` field use (or assign) the value `MAY`. Named association permits field values to be given in orders that may differ from the order specified in the type definition. Thus an equivalent assignment is:

```
Birthday := (Day => 15, Year => 1990, Month => MAY);
```

Aggregate assignment may also be used to initialize a record object as it is created. For example,

```
Today : Date_Record := (MAR, 30, 1992);
```

creates the object Today and initializes the three fields with the values indicated.

The *membership* operation allows us to test a record to determine if a particular value exists as one of the components of the record.

The *relational* operations of "=" and "/=" allow two record objects to be compared. For example,

```
if Today = Birthday then
  TEXT_IO.Put (Item => "Happy Birthday!");
else
  TEXT_IO.Put (Item => "Just another day.");
end if;
```

will compare the corresponding fields of the two record objects Today and Birthday. If the corresponding fields contain identical values, then the message "Happy Birthday!" will be printed. Otherwise the message "Just another day." will appear.

5.4.3 Default Initial Values

In addition to aggregate assignment and single component assignment, record objects may be initialized through the use of default initial values given in the declaration of the record type. These values are specified using the ":=" operator in the type definition. Some or all of the components in a record may be given a default value. For example,

```
type Gender is (M, F);
type Age_Range is range 0 .. 150;
type Person is
record
  Name : STRING (1 .. 5);
  Sex : Gender := M;
  Age : Age_Range := 20;
end record;

First_Person : Person;
Second_Person : Person := ("Linda", F, 30);
```

the record type Person specifies default initial values of M and 20 for the fields Sex and Age respectively. The declaration of First_Person makes use of these default values while leaving the Name field uninitialized. On the other hand, the declaration of Second_Person overrides the default values by placing the string "Linda" in the Name field, F in the Sex field, and 30 in the Age field.

5.4.4 Record Input and Output

Since a record is a collection of values of differing types, it is not possible to put or get an entire record in the same manner that a single value is read or written into an object. Instead each component must be treated as an object of a particular type and the corresponding methods of input and output employed. See the following example.

```
with TEXT_IO;
procedure Test_Days is
  type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN, JUL,
    AUG, SEP, OCT, NOV, DEC);
  type Day_Number is range 1 .. 31;
  type Year_Number is range 1900 .. 2100;
  type Date_Record is
  record
    Month : Month_Name := JAN;
    Day : Day_Number := 1;
    Year : Year_Number := 1993;
  end record;

  package Month_IO is new TEXT_IO.ENUMERATION_IO (Month_Name);
  package Day_IO is new TEXT_IO.INTEGER_IO (Day_Number);
  package Year_IO is new TEXT_IO.INTEGER_IO (Year_Number);

  Today : Date_Record := (MAR, 30, 1992);
  Birthday, Holiday : Date_Record;

begin
  TEXT_IO.Put (Item => "In what month were you born?");
  Month_IO.Get (Item => Birthday.Month);
  TEXT_IO.New_Line;
  TEXT_IO.Put (Item => "On what day?");
  Day_IO.Get (Item => Birthday.Day);
  TEXT_IO.New_Line;
  TEXT_IO.Put (Item => "In what year?");
  Year_IO.Get (Item => Birthday.Year);
  TEXT_IO.New_Line;
  Holiday := (JUL, 4, 1992);
  if Today = Birthday then
    TEXT_IO.Put (Item => "Happy Birthday!");
  elsif Today = Holiday then
    TEXT_IO.Put (Item => "Hurray - a holiday!");
  else
    TEXT_IO.Put (Item => "It's just ");
    Month_IO.Put (Item => Today.Month);
    Day_IO.Put (Item => Today.Day);
    Year_IO.Put (Item => Today.Year);
  end if;
  TEXT_IO.New_Line;
end Test_Days;
```

5.4.5 Exercises

1. Describe the difference between a record and an array.

2. Define a record type named `Person_Record` with fields for a person's name, age, height, and weight.

3. Suppose you had a record object named `A_Person` of type `Person_Record` (see Exercise 2). How can you initialize this object?

4. Write a segment of Ada code to output the values in the object `A_Person` (see Exercise 3).

5. Consider the record objects Today, Birthday, and Holiday. Describe how the same information could be stored using parallel arrays.

5.5 Ada Statements

In Chapter 2 many of the statements of Ada were introduced. In this section, extensions to those statements will be described. In addition, new statements will be introduced.

5.5.1 If Statements

Recall that in Chapter 2 a statement was introduced to allow the software engineer to choose different possible action based upon the value of some conditional expression. Such a statement provided alternation control and was called an **if** statement. In the form that was introduced at that time, an **if** statement looked like this:

```
if This_Number > Last_Number then
  This_Number := This_Number + 1;
else
  This_Number := Last_Number;
end if;
```

and was interpreted to mean that if This_Number was greater than Last_Number then This_Number was to be incremented by one. Otherwise, This_Number was set to the value of Last_Number.

When the choices that are to be made are binary, *i.e.*, when two mutually exclusive options are possible based upon the evaluation of the conditional expression, then this form of **if** statement is perfectly adequate. Unfortunately, not all choices that must be made in a program are binary. Consider a problem where it is desired to make a count of all of the characters in a file that are in the range 'A' .. 'L', as well as those that are in the ranges 'M' .. 'S' and 'T' .. 'Z'. Since there are three possibilities an **if** statement of the form studied so far is not useful to solve this problem. Of course, it is possible to *nest* the **if** statements so that one solution using only what has been presented so far might be:

```
if Letter in 'A' .. 'L' then
  First_Range := First_Range + 1;
else
  if Letter in 'M' .. 'S' then
    Second_Range := Second_Range + 1;
  else
    -- the only possibility left is the range 'T' .. 'Z'
    -- assuming that Letter contains an uppercase letter
    Third_Range := Third_Range + 1;
  end if;
end if;
```

In this solution, the value of `Letter` is checked to determine if it is in the range 'A' .. 'L' and if it is then the `First_Range` counter is incremented by one. If the first condition is FALSE however, then the **else** part of the outermost if statement is executed. But in this case, the statement in the **else** part of the outer if statement is another if statement. This inner if statement is said to be *nested* within the outermost if statement. The action of this inner if statement is to check to see if the value of `Letter` is in the range 'M' .. 'S', given that we already know that it cannot be in the range 'A' .. 'L' (otherwise the **then** part of the outer if statement would have been executed). If the value is within the range 'M' .. 'S', then the **then** part of the inner if statement is taken and the value of the counter `Second_Range` is incremented by one. Finally, if the value of `Letter` is not in the range 'M' .. 'S', then it must be in the range 'T' .. 'Z' (since all of the other possibilities for the value of `Letter` have been eliminated, assuming that we know that the value in `Letter` is an uppercase letter) so the **else** part of the inner if statement is taken and the value of the counter `Third_Range` is incremented by one.

As the analysis of the action of this nested if statement shows, it is possible to solve a problem with an arbitrary number of possibilities by merely nesting the if statements and checking different conditions until one of the conditions is true. Unfortunately, this nesting leads to unnecessarily complex code segments that are hard to understand and even harder to maintain as the size of the code gets large and the number of conditions gets large. Therefore, Ada allows a generalized form of the if statement.

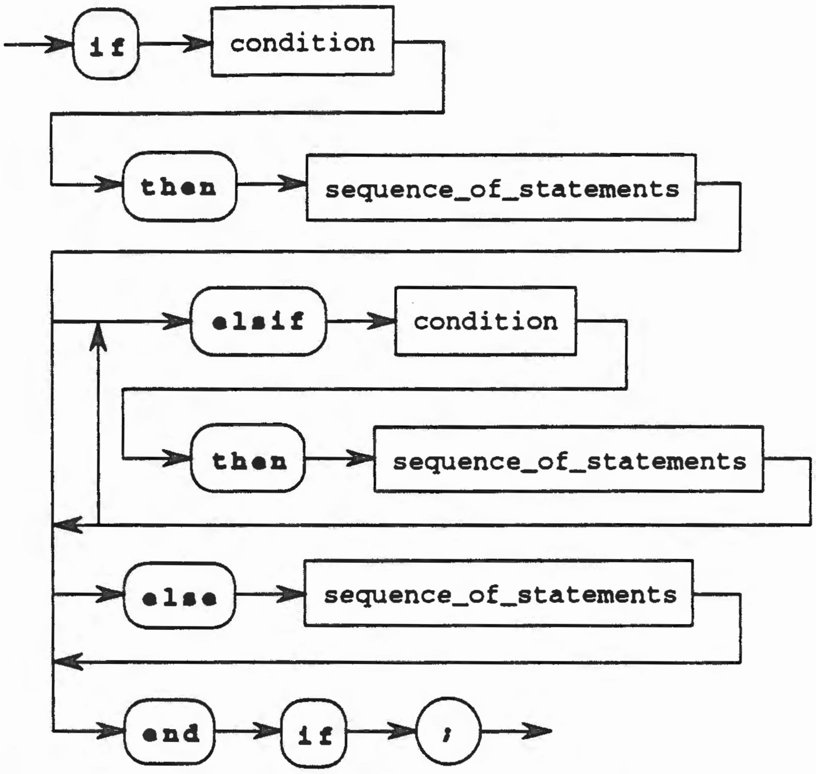
The syntax of an if statement that has been shown so far only allowed binary choices to be made in conditional statements. The full syntax definition of an if statement is shown next in Syntax Definition 5.7.

```
if_statement ::= if condition then
                   sequence_of_statements
                 {elsif condition then
                   sequence_of_statements}
                 [else
                   sequence_of_statements]
                  end if;

condition ::= boolean_expression
```

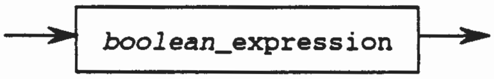
If Statement
Syntax Definition 5.7

The syntax for a complete if statement may also be shown graphically as illustrated in Syntax Chart 5.7 below.

if_statement ::=



condition ::=



If Statement
Syntax Chart 5.7

Analysis of the syntax definition notes that there may be as many **elsif** parts of the **if** statement as desired, but only a single **else** part. If an **else** part is used, it must be last, after any **elsif** parts that are used.

Consider the rewritten **if** statement from above,

```
if Letter in 'A' .. 'L' then
  First_Range := First_Range + 1;
elsif Letter in 'M' .. 'S' then
  Second_Range := Second_Range + 1;
else
  -- the only possibility left is the range 'T' .. 'Z'
  -- assuming that Letter contains an uppercase letter
  Third_Range := Third_Range + 1;
end if;
```

Note that it has been rewritten as a single **if** statement instead of the nested **if** statements that were previously required. The logic is still the same in that the first action is to compare `Letter` to the first range `'A' .. 'L'`, and if it is in that range then `First_Range` is incremented. If not, the **elsif** part is checked to determine if `Letter` is in the range `'M' .. 'S'`. If so, then `Second_Range` is incremented. If not, then the **else** part is executed and `Third_Range` is incremented. Note that since there is only one **if** statement, there is only one **end if**. Note also that the indentation is such that all of the **elsif** parts and the **else** part, if any, are aligned under the reserved word **if**. This keeps the statement on the page when multiple conditions are required to be checked; without this, each condition would be a nested **if** statement and the indentation could eventually move the statement off the page to the right!

Consider another example of this extended form of the **if** statement given below,

```
if Cumulative_Average >= 90 then
  Grade := 'A';
elsif Cumulative_Average >= 80 then
  Grade := 'B';
elsif Cumulative_Average >= 70 then
  Grade := 'C';
elsif Cumulative_Average >= 60 then
  Grade := 'D';
else
  Grade := 'F';
end if;
```

This statement determines grades for some fictional course. Note that several **elsif** parts are used, each checking for a different condition. The conditions are evaluated in order from top to bottom until one is found to be true. The first condition that evaluates to TRUE is the part whose sequence of statements get executed. Flow of control then passes to the statement after the **if** statement, *i.e.*, the evaluation of additional conditions or the execution of additional statements in this **if** statement is not possible. It is also important to note that the conditions that are evaluated do not need to be mutually exclusive. They may overlap in any manner, but the first one evaluated to TRUE will prevent any other condition from being checked.

The importance of the last point is obvious when the **if** statement above is rewritten with the conditions checked in the reverse order. Does this change the logic of the statement?

```
if Cumulative_Average >= 60 then
  Grade := 'D';
elsif Cumulative_Average >= 70 then
  Grade := 'C';
elsif Cumulative_Average >= 80 then
  Grade := 'B';
elsif Cumulative_Average >= 90 then
  Grade := 'A';
else
  Grade := 'F';
end if;
```

Yes, it does! Consider that if a student has a `Cumulative_Average` of 92, then in the first version of the if statement the first condition checked would be true and the student would receive the `Grade` of 'A'. In this new version of the if statement, the first condition is also TRUE, but the student receives a grade of 'D'! This is caused by the fact that a `Cumulative_Average` of 92 is not only greater than 90, it is also greater than 60. In other words, the conditions are not mutually exclusive, so the order in which they are evaluated is very important. A software engineer should keep this in mind when writing programs.

5.5.2 Case Statements

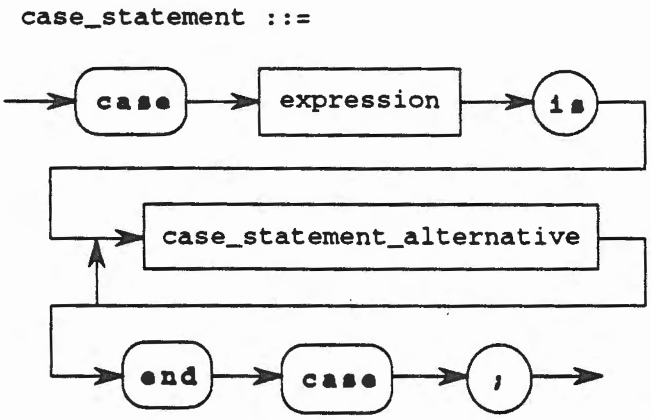
Now that we understand about one kind of statement that allows the software engineer to choose different possible actions based upon the value of some conditional expression, lets examine another possibility. We will look at another statement that provides alternation control, like an if statement, but in an even more general form. This statement is called a **case** statement.

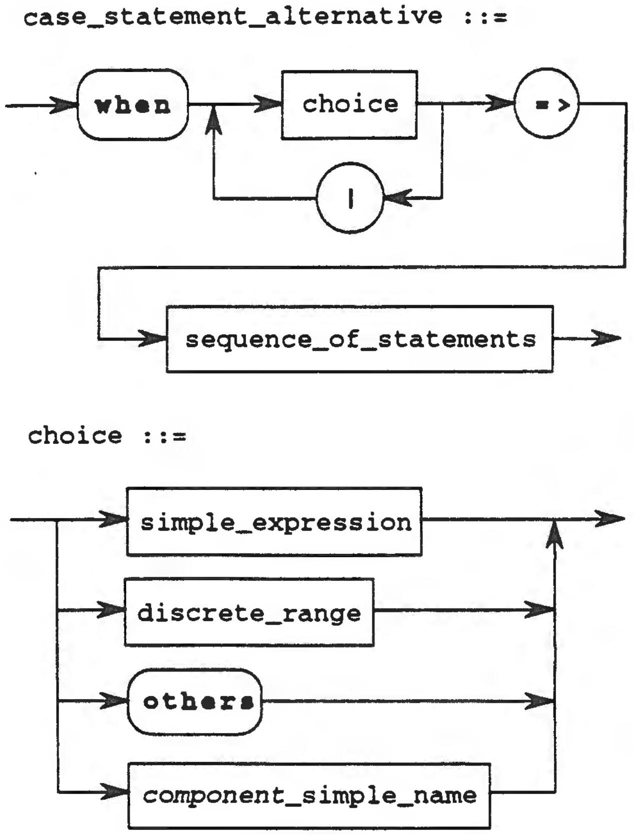
The **case** statement is another statement that can be used by a software engineer when choosing between several possible values. The case selector expression is evaluated and exactly one of several possible sequences of statements is selected for execution based upon the value of the selector expression, a more general term for an expression that serves the same purpose as the conditional expression did for the if statement. The syntax for the **case** statement is given in Syntax Definition 5.8 below.

```
case_statement ::= case expression is
                    case_statement_alternative
                    {case_statement_alternative}
                   end case;

case_statement_alternative ::=
  when choice {| choice} => sequence_of_statements

choice ::= simple_expression | discrete_range | others |
           component_simple_name
```

Case Statement
Syntax Definition 5.8

In a graphic form this syntax can be expressed as in Syntax Chart 5.8 below,





case_statement_alternative ::=

choice ::=

Case Statement
Syntax Chart 5.8

The **case** statement is useful when there are several possible actions to be taken depending upon the value of a selector expression. For example, consider the following **case** statement,

```
case Cumulative_Average is
  when 90 .. 100 => Grade := 'A';
  when 80 .. 89  => Grade := 'B';
  when 70 .. 79  => Grade := 'C';
  when 60 .. 69  => Grade := 'D';
  when others    => Grade := 'F';
end case;
```

that provides the same function as the nested **if** statement discussed in the previous subsection. The action that occurs when this statement is reached is the evaluation of the expression after the reserved word **case**. In the example, this is simply the value currently contained in the object `Cumulative_Average`. The flow of control then passes to exactly one of the **when** parts, depending upon the value of `Cumulative_Average`. If it is in the range 90 .. 100, inclusive, then the statement after the arrow (**=>**) is executed and flow of control then continues with the statement following the **case** statement, skipping all of the other statements in this **case** statement. Similarly, the evaluation of `Cumulative_Average` causes the flow of control to branch to the sequence of statements following the arrow in whichever **when** part is selected. If the evaluation of the expression yields a value not specifically mentioned, say 55 in this example, then flow of control branches to the sequence of statements after the arrow in the **when others** part.

There are certain rules that must be followed when using a **case** statement. The first is that the alternatives (the ranges or values after the **when**) must be mutually exclusive and exhaustive. Mutually exclusive means that there can be no overlap in any of the **when** parts, or more properly the **case** alternatives. This is necessary so that the evaluation of the selector expression determines exactly one **case** alternative to which control will branch. Exhaustive means that every possible value for the selector expression must be contained in the **case** alternatives. Thus, if the selector is an INTEGER object, then every possible value for the type INTEGER must be covered in the **case** alternatives. This is necessary so that the selector expression never is evaluated to a value that is not provided for in the **case** alternatives; if so, then what might happen would be unpredictable. Therefore, the rules of Ada require that every possible choice must be considered for each selector expression.

In some selector expressions, providing a **case** alternative for each possible value is not a difficult chore. However, for some selector expressions there many be a very large set of possible choices. In these situations, it is usually the situation that only a small subset of the possible choices are of particular interest; the rest are either not important or have the same sequence of statements that should be executed. In such circumstances, the choices that are of concern can be specifically mentioned, then the **when others** choice can be used.

When the **when others** choice is used, it must be the last choice in the **case** alternatives. Only one **when others** choice is allowed per **case** statement. The **when others** choice is chosen when the selector expression evaluates to anything that is not specifically mentioned in the **case** alternative list. For example, consider the following **case** statement,

```
case The_Speed is   -- The_Speed is an object of type INTEGER
  when  0 .. 5    => TEXT_IO.Put (Item => "You are walking.");
  when  6 .. 65   => TEXT_IO.Put (Item => "You are driving.");
  when 66 .. 80   => TEXT_IO.Put (Item => "Speeding!");
  when 81 .. 150  => TEXT_IO.Put (Item => "You are flying.");
  when others     => TEXT_IO.Put (Item => "Too fast for me.");
end case;
```

All of the possible values for `The_Speed` up to 150 are included specifically and so evaluation of the value of `The_Speed` to a value between zero and 150, inclusive, will cause the flow of control to branch to one of the sequences of statements in the corresponding **case** alternative. If `The_Speed` is above 150, then flow of control will branch to the **when others** part because no **case** alternative has been explicitly provided for a value in these ranges. However, the software engineer has perhaps forgotten that since `The_Speed` is of type INTEGER, there are an equal number of negative values possible. These also would branch control to the **when others** part since they are not specifically provided with a **case alternative.**

As another example, consider the following **case** statement about university administrators that makes use of the following enumerated type definition and object declaration,

Declarations:

```
type Work_Times_Type is (EARLY_AM, MID_AM, NOON,
  EARLY_AFTERNOON, MID_AFTERNOON, LATE_AFTERNOON);

Work_Day : Work_Times_Type;
```

Statement:

```
case Work_Day is
  when EARLY_AM | EARLY_AFTERNOON => TEXT_IO.Put("Drink Coffee");
  when MID_AM                    => TEXT_IO.Put("Prepare For Lunch");
  when NOON                      => TEXT_IO.Put("Eat Lunch");
  when MID_AFTERNOON             => TEXT_IO.Put("Stay Awake");
  when LATE_AFTERNOON            => TEXT_IO.Put("Prepare To Go Home");
end case;
```

This is an example of the same sequence of statements serving for two different possible choices of the selector expression using the | symbol to mean EARLY_AM or EARLY_AFTERNOON. Also, no **when others** part is needed since every possible value for the selector expression has been considered and a **case** alternative has been provided for each. It would not have been an error to have put a **when others case** alternative in this statement; it would simply always be ignored. Therefore, as a stylist point, consider putting a **when others case** alternative in all of your **case** statements as a good defensive programming practice.

The **case** statement is very useful when there are many possible actions depending upon the value of some choice mechanism in an algorithm. The most important thing to remember is that the choices in a **case** statement must be mutually exclusive and exhaustive.

5.5.3 For Loop Statements Revisited

In previous discussions of **for loops**, we have always used a numeric value to represent the upper and lower bounds of the loop. This is not a requirement in Ada. Any discrete type can be used as the bounds to a **for loop**. In this section we will demonstrate some of the uses of this capability.

The range specified may be any discrete range of any discrete type. For example, consider the **for loop** below that makes use of the declarations given next,

```
type Colors is (RED, YELLOW, ORANGE);

Loop_By_Colors:
for This_Color in Colors loop
  TEXT_IO.Put (Item => "Another pretty color.");
end loop Loop_By_Colors;
```

In this example, the identifier This_Color is initially given the value RED, the first value in the type Colors. After one iteration of the loop, the successor to RED is assigned to This_Color, which in this example is YELLOW. Since this is not the last value in the type another iteration of the loop is executed. Now This_Color is given the value of the successor to YELLOW, namely ORANGE and another iteration of the loop is executed. Finally, there being no successor to the value ORANGE in the type Colors, the loop is terminated normally. Thus, the loop identifier does not need to be an integer, but any discrete type is allowed.

Now that we have explained enumerated types, their use in for loops as indices should be apparent. This is a very useful feature of Ada that you should use in your solutions to programming problems.

5.5.4 Exercises

1. Rewrite the following code segment so that it has a single **if** statement but performs the same logical operation:

```
if Letter in 'T' .. 'Z' then
  First_Range := First_Range + 1;
else
  if Letter in 'A' .. 'L' then
    Second_Range := Second_Range + 1;
  else
    -- the only possibility left is the range 'M' .. 'S'
    -- assuming that Letter contains an uppercase letter
    Third_Range := Third_Range + 1;
  end if;
end if;
```

2. Rewrite the following code segment using a more appropriate mechanism as the loop controller:

```
Color_Counter:
for This_Color in Color'FIRST .. Color'LAST loop
  TEXT_IO.Put ("There is another color.");
end loop Color_Counter;
```

3. Rewrite the code segment in Exercise 1 using a **case** statement.

4. Write a **for loop** that iterates over the following enumeration values in the opposite order in which they are declared.

```
type Animals is (DOG, CAT, PIG, HORSE, COW, FISH, BIRD);
```

5. Write a code segment that gets an INTEGER value from the user using a prompt, then uses a **case** statement to write a message as to whether the value entered is negative, positive, or zero.

6. Rewrite the code segment in Exercise 5 to use an **if** statement. Which of the two alternatives is more appropriate in this case? Justify your answer.

5.6 Block Statements And Exceptions

Ada introduces some new notions in programming that make it especially suited for software engineering. One of these concepts is localization, which states that things that are logically related should also be physically related. In other words, things that belong together should be physically located together for ease of maintenance and understandability. Ada supports this concept with **block** statements. In addition, a program in execution should not be allowed to fail simply because something exceptional has occurred. In other words, if the program attempts to divide by zero in a numerical calculation, the program should not be allowed to "crash." This allows the programmer to decide what should happen when an exceptional situation occurs. This capability in Ada is called exception handling. In this section we will discuss both of these features to see how they might be useful to us. Before we can understand these new features, there is one more thing we need to understand, namely the meaning of scope and visibility.

5.6.1 Scope and Visibility

There are two related terms that are often used in the discussion of programs that need to be explained before we go any further. These terms are *scope* and *visibility.* Scope refers to the region of a program where an identifier is accessible, *i.e.*, the region where the meaning of this identifier is known. It is also the region of potential visibility.

Visibility is the region of a program where an identifier is directly accessible. It is the same as the scope of the variable except for places where more local declarations may hide the outer location. Thus, scope is potential visibility and visibility is the region where an identifier may be directly accessed.

Given the following program,

```
with TEXT_IO;
procedure Scope_Demo
is                                Scope of My_Value as an INTEGER
  My_Value : INTEGER;        <----------------------------------------
begin                                                               |
  My_Value := 10;                                                   |
  declare                                                           |
    My_Value : CHARACTER;    <----                                  |
  begin                          | Scope and Visibility of My_Value |
    My_Value := 'A';             | as a CHARACTER                   |
  end;                       <----                                  |
end Scope_Demo;              <----------------------------------------
```

the scope of My_Value as an INTEGER is from the declaration of the identifier until the end of the program. The visibility of this same identifier is NOT the same region; it excludes the portion from the declaration of My_Value as a CHARACTER inside the declare block until the end of the declare block. The scope and visibility of My_Value as a CHARACTER extends from the declaration of this identifier in the declare block until the end of the declare block; in this case both the scope and the visibility of this identifier are the same.

In general, it is not a good engineering practice to redefine an identifier within its own scope. Thus, usually the scope and visibility of an identifier are the same. However, there are cases where the reuse of an identifier makes good engineering sense and in these instances scope and visibility considerations are important.

There are times when it is convenient to get some information from the user and then use that information in the declaration of new types and objects. The problem that arises is that you must be in the executable portion of the program to interact with the user, while you must be in the declarative part of the program to make declarations. Furthermore, the declarative part must come first. Thus, there seems to be no way to interact with the user and use this information in declaring objects.

5.6.2 Introduction to Block Statements

To avoid this problem, Ada has a feature called a **block** statement, sometimes also called a declare block for reasons that will be obvious later. Essentially, this statement opens up a new level of scope, allowing the declaration of new types and objects in the midst of the executable portion of the program. The declarations made in this block are created in the declare block and disappear when the declare block is exited.

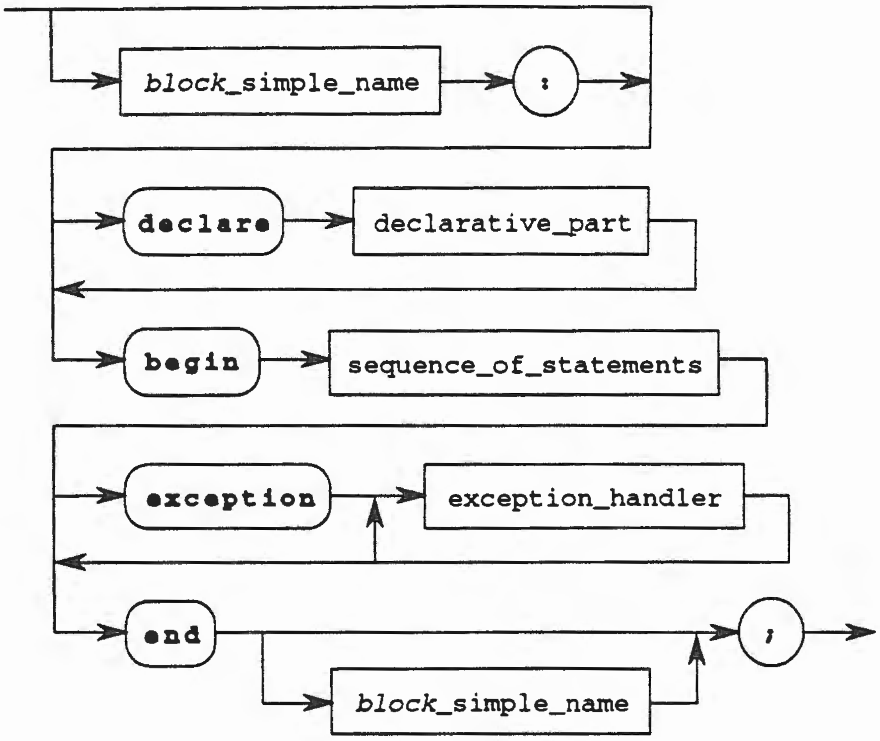
The formal syntax of a block statement in Ada is given in the 9 syntax productions in Syntax Definition 5.9.

```
block_statement ::= [block_simple_name :]
  [declare declarative_part]
  begin sequence_of_statements
  [exception exception_handler {exception_handler}]
  end [block_simple_name] ;

declarative_part ::=
  {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::= basic_declaration | use_clause

use_clause ::= use package_name {, package_name} ;

basic_declaration ::= object_declaration | number_declaration |
                      type_declaration | subtype_declaration

number_declaration ::= identifier_list : constant :=
                       static_expression ;

later_declarative_item ::= body | subprogram_declaration |
                           package_declaration | use_clause

body ::= proper_body

proper_body ::= package_body | subprogram_body

exception_handler ::= when exception_choice {| exception_choice} =>
                      sequence_of_statements

exception_choice ::= exception_name | others
```

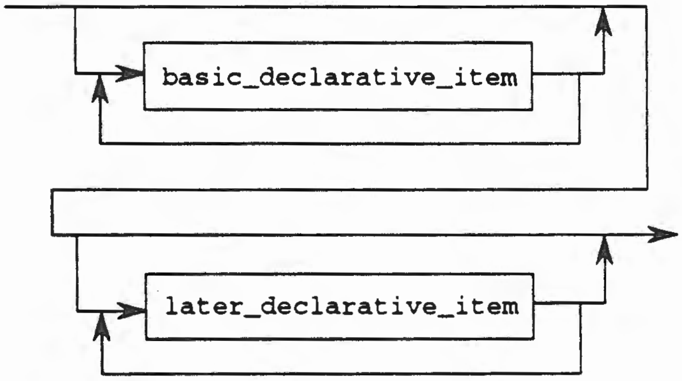
Block Statement
Syntax Definition 5.9

This syntax can also be expressed in graphical form as indicated in Syntax Chart 5.9.

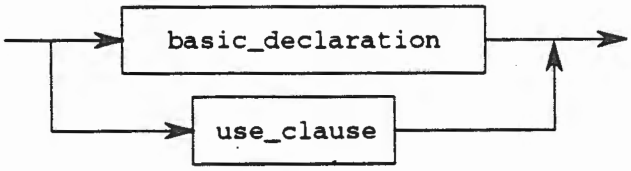
block_statement ::=



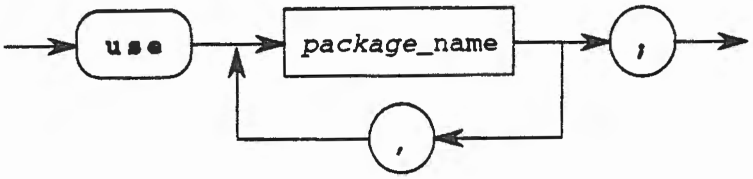
declarative_part ::=



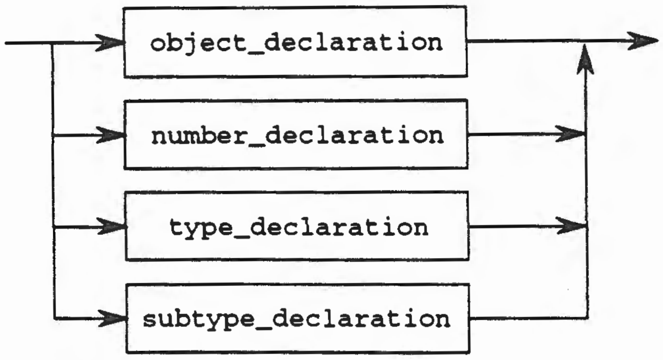
basic_declarative_item ::=



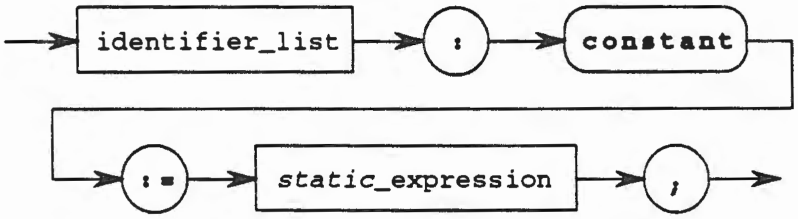
use_clause ::=



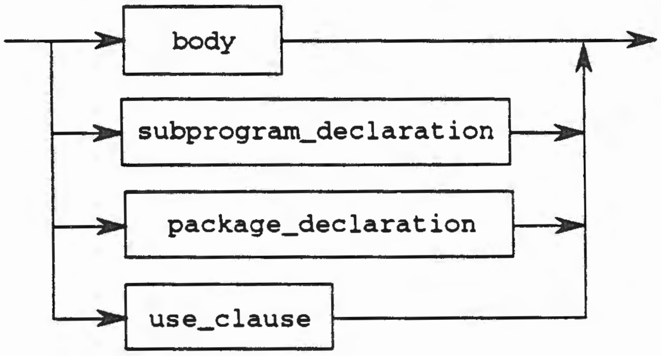
basic_declaration ::=



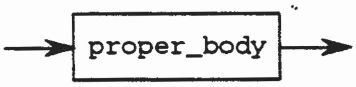
number_declaration ::=

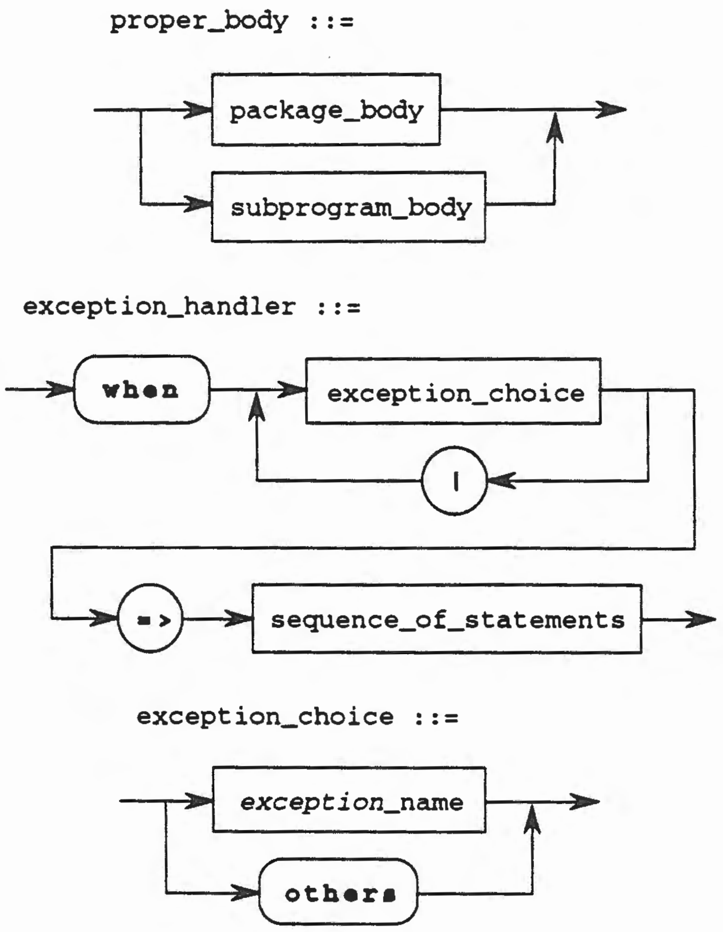


later_declarative_item ::=



body ::=





proper_body ::=

exception_handler ::=

exception_choice ::=

Block Statement
Syntax Chart 5.9

An example may serve to demonstrate these ideas. Consider the following program,

```
with TEXT_IO;
procedure Declare_Demo
is
  Length : POSITIVE;
  package Int_IO is new TEXT_IO.INTEGER_IO (POSITIVE);
begin
  Text_IO.Put(Item => "Length of String? ");
  Int_IO.Get (Item => Length);
  declare
    Demo : STRING (1 .. Length);
  begin
    Text_IO.Get (Item => Demo);
  end;  -- declare block
end Declare_Demo;
```

In this example, the number of characters in the index of the array was unknown. The program asked the user for the value of the last index and then entered a declare block. The declare block opened up a new level of scope and thus allowed for the declaration of a new array type using as an upper bound the value read in from the user. This would not have been possible without the declare block.

Later we will return to this idea and see that the declare block can be used for other more useful purposes whenever a new level of scoping is needed. This will be particularly useful in exception handling. In certain circumstances, even the word declare can be left out, yielding just the **begin** and **end** pair as the entire declare block.

5.6.3 Introduction to Exceptions

It would be an ideal world if you could anticipate all of the uses that would be made of your program. Similarly, it would be nice if you could anticipate all of the possible input that your program might have to accept. Unfortunately, this is rarely the case. Consequently, your program may fail for a variety of reasons, not all of which can be anticipated by you. In many languages, such a situation causes the computer to *"crash."* This is a term used by computer professionals to indicate that the program has terminated abnormally. Usually the operating system can be counted upon to trap these errors and simply stop executing your program, indicating the error if it can. Other times, your entire system may *"lock-up"* and need to be manually reset. Until now, this was just an accepted part of programming.

However, in Ada we have a built-in mechanism to detect and potentially correct errors while the program is still executing. Depending on what the software engineer wants to do, it is possible that such faults as would normally cause the program to *"crash"* can be detected, handled, and the program can continue in any manner that is desired. This facility in Ada is called an *exception handler* and the exceptional situation that caused a fault is called an *exception*.

5.6.3.1 The Use of Exceptions

Exceptions are both predefined and user-defined. Early in this text we will use the predefined exceptions as a convenience. Later, we will use both the predefined exceptions and the user-defined exceptions that we will define that are unique and specific to our problem domain.

A complete definition and explanation of exceptions will be deferred until later. For now, we will introduce a simple example of an exception handler. This mechanism should be a part of all of your programs from this point forward. Later we will see how to tailor this generalized exception handler to fit the specific problem that we are solving with our programming system.

```
with TEXT_IO;
procedure Demonstrate_Exception
is
  Value : POSITIVE;
  package Int_IO is new TEXT_IO.INTEGER_IO (POSITIVE);
begin
  TEXT_IO.Put (Item => "Enter a value => ");
  Int_IO.Get (Item => Value);
exception
  when others => TEXT_IO.Put(Item => "Oops - an error!");
end Demonstrate_Exception;
```

In this example, the part between the reserved words **begin** and **exception** are the normal program's statements that will be executed as we have seen before. The part between the reserved words **exception** and **end** is called the *exception handler part*. In this example, there is a single exception handler, designated by the **when others**. **When others** is a short cut means for saying that all exceptions are to handled by this exception handler.

When something goes wrong and an exception occurs, we say that it has been *raised*. Execution of the normal sequence of statements ceases and control passes to the exception handler, if there is one, otherwise, the exception is propagated through the calling chain until either an exception handler is found or the exception reaches the outer level. If that occurs, then the program will *"crash"* as in any other programming language.. If there is an exception handler part and the proper exception is handled in that exception handler, or if there is a **when others**, the exception is said to be *lowered* and the statements following the arrow (=>) are executed. There are no limitations on what statements may be in an exception handler. When these statements have been executed, control returns to the calling procedure (if there is one) normally, *i.e.*, as if there had been no exception. The calling program is not able to determine that an exception has ever been raised.

There are five predefined exceptions that you should know about. These include CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, and TASKING_ERROR. The last exception is not likely to be seen by you until later in this course, so we will ignore that one until later. The rest of the exceptions are predefined and raised automatically by the runtime system. The explanation of each of these exceptions and the circumstances that result in their being raised are as follows:

CONSTRAINT_ERROR This is one of the most frequently seen exceptions. It is raised whenever a range constraint or an index constraint is violated. There are some other special cases when this exception is raised, but we will not study them until later.

NUMERIC_ERROR This exception is relatively rare. It is raised whenever the execution of a predefined numeric operation cannot deliver a correct result. In the future this exception will probably be subsumed by the CONSTRAINT_ERROR exception.

PROGRAM_ERROR This exception can be raised automatically in a variety of ways. If you attempt to use a program unit that has not yet been elaborated, this exception will be raised. Note that we will describe elaboration in more detail later. There are several other ways for this exception to be raised, but a full discussion of them will be deferred until we have seen more of the features of the language.

STORAGE_ERROR This exception is raised whenever there is insufficient storage (usually memory) to perform the operation requested. We will see more about this exception later when we study access types.

5.6.3.2 Predefined Input/Output Exceptions

In order for you to plan for the potential exceptions that may be raised in your programs, it is essential that you understand the predefined exceptions defined in the previous section. In addition, it is useful to understand the predefined exceptions that are raised during input/output operations. These include STATUS_ERROR, MODE_ERROR, NAME_ERROR, USE_ERROR, DEVICE_ERROR, END_ERROR, DATA_ERROR, and LAYOUT_ERROR. These exceptions are defined for all of the predefined input/output packages, such as the one that we have been using, TEXT_IO. The definitions and causes of the input/output exceptions, paraphrasing heavily from the *Reference Manual for the Programming Language Ada* [LRM], are as follows:

STATUS_ERROR
This exception is raised when you attempt to operate on a file that is not open or when you attempt to open a file that is already open. We will explain more about files later in this book and the meaning of this exception will become more clear at that time.

MODE_ERROR
This exception is raised when you attempt to read from, or test for the end of, a file that has a current mode on OUT_FILE. It can also be raised by an attempt to write to a file that has a current mode of IN_FILE. It can also be raised by specifying a file with an improper mode to any of the procedures in TEXT_IO that depend upon a mode for a given file, such as calling the procedure SET_OUTPUT with a parameter of a file type whose mode is IN_FILE.

NAME_ERROR
This exception is raised by a call of the procedure CREATE or OPEN if the string given for the parameter NAME does not allow the identification of an external file. Thus, if the OPEN procedure is called with a string given for the NAME parameter that does not match any of the external files.

USE_ERROR
This exception is raised if the user requests an operation that cannot be performed on the external file due to its physical or logical characteristics.

DEVICE_ERROR
This exception is raised if the requested operation cannot be performed because of a malfunction of the underlying system.

END_ERROR
This exception is raised when the user attempts to read beyond the end of a file.

DATA_ERROR
This exception may be raised when a Get is unable to interpret the input sequence as belonging to the type of the object for which a value is being sought. For example, if the procedure Get is attempting to read an INTEGER value and it finds the character 'Z' in the input sequence.

LAYOUT_ERROR
This exception is raised when the user attempts to set a column beyond the maximum allowed for that line, or when a line number is attempted to be set beyond the maximum number of lines in a page. It also is raised if the user attempts to put a string with more characters than the maximum line length.

5.6.4 Exercises

1. Define and differentiate between the terms *scope* and *visibility.*

2. What is the purpose of the block statement in the following code segment?

```
with TEXT_IO;
procedure Mystery_Segment is
  Mystery_Value : POSITIVE;
  package Int_IO is new TEXT_IO.INTEGER_IO (POSITIVE);
begin
  Int_IO.Get (Item => Mystery_Value);
  declare
    Demo : STRING (1 .. Mystery_Value);
  begin
    TEXT_IO.Put (Demo);
  end;  -- declare block
end Declare_Demo;
```

3. What is the purpose of the exception mechanism in Ada?

4. Given the following code segment:

```
begin
  -- some statements
exception
  when CONSTRAINT_ERROR =>
    TEXT_IO.Put(Item => "Constraint Problem.");
  when others => TEXT_IO.Put("Something else is wrong.");
end;
```

   What will happen if a PROGRAM_ERROR is raised? Explain your answer.

5. Explain the sequence of steps that occurs when an exception is raised. Include details up until the time that the exception is lowered. You may assume that an appropriate exception handler is in place.

6. Explain the sequence of steps that occurs when an exception is raised. Include details up until the time that the exception is lowered. You may *not* assume that an appropriate exception handler is in place.

7. Is it possible to get a LAYOUT_ERROR when doing input operations only? Explain.

8. Is it a good idea to have a system in a language that allows the user to control what happens when errors occur? Defend your answer.

Chapter 6

Program Verification

Chapter 5 has introduced some additional Ada constructs, extensions to types and objects, and statements. We next look at these constructs for the behavior they add to Ada programs. In each case, they permit new means of expression, more efficient ways of computation. Then, beyond these extensions in Ada, we broaden our horizon to defining what programs should do—called **specifications**, and evaluating whether the programs meet such specifications by both analytic and experimental evidence.

Given a program **specification** (a relation of input and output files, which is possibly a function) and a **program** (whose behavior is ordinarily a function but possibly a relation), the program is said to be **correct** if it terminates from every input file in the domain of the specification and produces an output file which is paired with that input file in the specification.

Programs are correct when they meet their specifications. Another question is whether the specifications are right. Deciding what software should be is a deep and broad question which will be addressed later. Defining good specifications needs effective analyses based on understandings of user needs as well as computer capabilities.

In summary, three major tasks in producing software are **specification, development, verification.**

The first task is to **specify** the software that will best fulfil the mission that has been assigned to the software. At this stage the software is represented by a behavior, namely a mathematical function or relation, that defines what the software is to do in order to fulfil its assigned mission. The behavior is documented in the software specification.

The second task is to **develop** the program that will satisfy the entire domain of the specification. The software engineer must find a **behavior rule** that correctly implements the entire domain of the behavior defined by the specification that will execute properly in the target operating environment. The rule is documented in the program.

The third task is to **verify** that the program as developed does indeed satisfy the specification correctly. Interactively with the design for the program, namely the rule for the behavior required, the program should be verified as meeting the specification step by step as it is being developed. The verifications should be reviewed by peer engineers.

This Chapter 6 deals with two of these three tasks, namely with **specification** and **verification**. The remainder of the book will deal with the **development**, especially the **design**, of software. In this Chapter, the first section deals with how to analyze the new Ada constructs introduced in Chapter 5. The second section introduces specification as a formal activity. The third section introduces verification, using the ideas of program behavior of the preceding chapters and its comparison with a required specification. The fourth and fifth sections deal with verification using trace tables.

6.1 New Ada Rules for Behavior

6.1.1 Ada Rules for Declarations

6.1.1.1 Enumeration Types and Objects

Enumeration types and objects have meanings that are very straight forward, but as discussed in Chapter 5, distinct types and subtypes can be compatible and used together. The declaration of any type, subtype, or object expands the the state of a program on the spot. For example, a program declaration starting with type WEEK, as defined in Chapter 5

```
with TEXT_IO;
procedure Check_Weekday
is
  type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY);
...
```

will augment the program state from

```
⟨Input, Output⟩
```

to

```
⟨Input, Output, Week⟩
```

or, more formally, no matter where it appears,

```
[type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
  SATURDAY, SUNDAY);]
  = {⟨⟨...⟩, ⟨... , Week⟩⟩ | type Week is days of week beginning with
     MONDAY, then TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
     SUNDAY}
```

The extension of the declaration to an object `Today` of type Week will further augment the program state as shown next.

```
with TEXT_IO;
procedure Check_Weekday
is
  type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY);
  Today: Week;
...
```

will further augment the program state from

```
⟨Input, Output, Week⟩
```

to

```
⟨Input, Output, Week, Today⟩
```

or, more formally,

```
[Today: Week;]
  = {⟨⟨...⟩, ⟨..., Today⟩⟩ | Today is any day of Week}
```

In this case, since Today is not initialized, there are exactly seven distinct states possible.

6.1.1.2 Array Types and Objects

Array types and objects permit storing collections of single data types in a single vector, or Ada array. The objects can be CHARACTERS, BOOLEANS, INTEGERS. When declared as either types or objects, they expand the data known to a program. For example, building on the partial program above, we add an array type and an array object of that type next.

```
with TEXT_IO;
procedure Check_Weekday
is
  type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY);
  Today: Week;
  type Grade_Vector is array (1 .. 100) of INTEGER;
  Grade: Grade_Vector := (others => 100);
...
```

will further augment the program state from

```
⟨Input, Output, Week, Today⟩
```

to

```
⟨Input, Output, Week, Today, Grade_Vector, Grade⟩
```

or, more formally,

```
[type Grade_Vector is array (1 .. 100) of INTEGER;
Grade: Grade_Vector := (others => 100);]
  = {⟨⟨...⟩, ⟨..., Grade_Vector, Grade⟩⟩ | Grade_Vector is type array
     of dimension 1 to 100 with INTEGER components, Grade is of type
     Grade_Vector with each component initially 100}
```

In this case, Grade was initialized for each component to its maximum possible value.

6.1.1.3 STRING Types and Objects

STRINGS are special kinds of arrays which are unconstrained arrays of CHARACTERS, indexed by INTEGERS. As above, we expand the declarations to include a STRING object.

```
with TEXT_IO;
procedure Check_Weekday
is
  type Week is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY);
  Today: Week;
```

```
    type Grade_Vector is array (1 .. 100) of INTEGER;
    Grade: Grade_Vector;
    Full_Name: STRING (1 .. 10);
  ...
```

will further augment the program state from

```
  ⟨Input, Output, Week, Today, Grade_Vector, Grade⟩
```

to

```
  ⟨Input, Output, Week, Today, Grade_Vector, Grade, Full_Name⟩
```

or, more formally

```
  [Full_Name : STRING (1 .. 10);]
  = {⟨⟨...⟩, ⟨..., Full_Name⟩⟩ | Full_Name is 10 CHARACTER STRING}
```

In this case, `Full_Name` could have value "`Mike`" & "`Smith`".

6.1.2 Rules for Ada Statements

6.1.2.1 If Statements

The full definition for if statements which permits **elsif** parts defines new meanings. The **Syntax Definition 5.8 of the form**

```
  if condition
  then
    sequence of statements
  {elsif condition
  then
    sequence of statements}
  [else
    sequence of statements]
  end if;
```

allows any number of **elsif** parts, and the **if** condition and **elsif** conditions are checked in order during execution. The first such condition that is TRUE, if any, turns execution control over to the sequence of statements for it, otherwise the **else** part, if it exists, is executed, and finally, if no condition is TRUE and no **else** part exists, the **if** statement behaves as a **null** statement. One item to be careful about is that the **elsif** conditions must be checked in order. It is not enough simply for an **elsif** condition to be TRUE unless all previous conditions, including the **if** condition are FALSE.

For example, consider the **if** statement shown in Chapter 5, assuming `Cumulative_Average` is INTEGER in region 0 .. 100,

```
  if Cumulative_Average >= 90
  then
    Grade := 'A';
  elsif Cumulative_Average >= 80
  then
    Grade := 'B';
```

```
elsif Cumulative_Average >= 70
then
  Grade := 'C';
elsif Cumulative_Average >= 60
then
  Grade := 'D'
else
  Grade := 'F';
end if;
```

Its part behavior is the disjoint behavior union of the **if then** part for the **if** condition TRUE, each successive **elsif then** part for all prior conditions FALSE and this **elsif** condition TRUE, and the **else** part if all prior conditions FALSE. This can be expressed as follows.

```
if 100 >= Cumulative_Average >= 90 -> Grade <- 'A';
if  90 >  Cumulative_Average >= 80 -> Grade <- 'B';
if  80 >  Cumulative_Average >= 70 -> Grade <- 'C';
if  70 >  Cumulative_Average >= 60 -> Grade <- 'D';
if  60 >  Cumulative_Average >= 0  -> Grade <- 'F';
```

This part behavior might be better expressed as follows.

```
if 90 <= Cumulative_Average <= 100 -> Grade <- 'A';
if 80 <= Cumulative_Average <=  89 -> Grade <- 'B';
if 70 <= Cumulative_Average <=  79 -> Grade <- 'C';
if 60 <= Cumulative_Average <=  69 -> Grade <- 'D';
if  0 <= Cumulative_Average <=  59 -> Grade <- 'F';
```

The part behavior of this if statement has the following five subparts

```
if 90 <= Cumulative_Average <= 100 -> [Grade <- 'A';],

if 80 <= Cumulative_Average <=  89 -> [Grade <- 'B';],

if 70 <= Cumulative_Average <=  79 -> [Grade <- 'C';],

if 60 <= Cumulative_Average <=  69 -> [Grade <- 'D';],

if  0 <= Cumulative_Average <=  59 -> [Grade <- 'F';]
```

In summary, this can be put together as follows.

```
[if Cumulative_Average >= 90
then
  Grade := 'A';
elsif Cumulative_Average >= 80
then
  Grade := 'B';
elsif Cumulative_Average >= 70
then
  Grade := 'C';
elsif Cumulative_Average >= 60
then
  Grade := 'D'
else
  Grade := 'F';
end if;]
```

```
= {⟨⟨Input, Output, Cumulative_Average, Grade⟩,
  ⟨Input⟩, Output, Cumulative_Average, Grade$_1$⟩⟩ |
  if Cumulative_Average in 90 .. 100 -> Grade$_1$ <- 'A' and
  if Cumulative_Average in 80 .. 89  -> Grade$_1$ <- 'B' and
  if Cumulative_Average in 70 .. 79  -> Grade$_1$ <- 'C' and
  if Cumulative_Average in 60 .. 69  -> Grade$_1$ <- 'D' and
  if Cumulative_Average in  0 .. 59  -> Grade$_1$ <- 'F'}
```

6.1.2.2 Case Statements

Case statements are another form of alternation control that give another alternative to **if** statements. An alternative to the **if** statement just above converting `Cumulative_Average` into `Grade` was given in Chapter 5 as follows.

```
case Cumulative_Average
is
  when 90 .. 100 => Grade := 'A';
  when 80 .. 89  => Grade := 'B';
  when 70 .. 79  => Grade := 'C';
  when 60 .. 69  => Grade := 'D';
  when others    => Grade := 'F';
end case;
```

The part behavior of this **case** statement has the following five subparts as identified in the preceding **if** statement

```
if 90 <= Cumulative_Average <= 100 -> [Grade <- 'A';],

if 80 <= Cumulative_Average <=  89 -> [Grade <- 'B';],

if 70 <= Cumulative_Average <=  79 -> [Grade <- 'C';],

if 60 <= Cumulative_Average <=  69 -> [Grade <- 'D';],

if  0 <= Cumulative_Average <=  59 -> [Grade <- 'F';]
```

In this case the **when** parts must be distinct, with no common conditions between them (unlike the **elsif** parts which can have overlapping conditions). The **case** part function is quite direct, except for dealing with **when others** which needs to be described explicitly. For example, the part function for the **if** statement above is a satisfactory form for this **case** statement, namely

```
[case Cumulative_Average
is
  when 90 .. 100 => Grade := 'A';
  when 80 .. 89  => Grade := 'B';
  when 70 .. 79  => Grade := 'C';
  when 60 .. 69  => Grade := 'D';
  when others    => Grade := 'F';
end case;]
```

```
= {⟨⟨Input, Output, Cumulative_Average, Grade⟩,
  ⟨Input⟩, Output, Cumulative_Average, Grade$_1$⟩⟩ |
  when Cumulative_Average is in the range 90 .. 100 -> Grade$_1$ <- 'A',
  when Cumulative_Average is in the range 80 .. 89  -> Grade$_1$ <- 'B',
  when Cumulative_Average is in the range 70 .. 79  -> Grade$_1$ <- 'C',
  when Cumulative_Average is in the range 60 .. 69  -> Grade$_1$ <- 'D',
  when Cumulative_Average is in the range  0 .. 59  -> Grade$_1$ <- 'F'}
```

which is precisely that of the meaning for the **if** statement above. Note that **when others** is explicitly described in this form.

6.1.2.3 Block Statements

Block statements introduced in Chapter 5 permit the grouping of both declarations and statements, just as can be done in procedures and functions. Since the meanings of both declarations and statements have already been treated, **block** statements have no new meanings, just putting them together. But we have already seen declarations and statements put together in procedures and functions, so **block** statements simply follow the way of procedures except for their naming and calling conventions.

For example, the **block** illustrated in Chapter 5 was

```
with TEXT_IO;
procedure Declare_Demo
is
  Length : POSITIVE;
  package INT_IO is new TEXT_IO.INTEGER_IO (POSITIVE);
begin
  Text_IO.Put (Item => "Length of String? ");
  INT_IO.Get (Item => Length);
  Check_Data:
  declare
    Demo: STRING (1 .. Length);
  begin
    TEXT_IO.Put ("My name is " & Demo);
  end Check_Data; -- declare block
end Declare_Demo;
```

The **block**, itself is the third and final statement in the procedure **begin** block, following the one line statements of `Put` and `Get`. This **block** statement will make use of declared data `Demo`, a STRING, but on completion `Demo` will disappear (just as a **for** index will disappear). That is, the part function for **block** `Check_Data` will possibly transform `Input`, `Output`, `Length`, using `Demo` in the process, but not returning `Demo`. The result is

```
[Check_Data:
declare
  Demo: STRING (1 .. Length);
begin
  -- do something
end Check_Data; -- declare block]
= {⟨⟨Input, Output, Length⟩, ⟨Input$_1$, Output$_1$, Length$_1$⟩⟩ |
  Output$_1$ = Output & "My name is  " & Demo}
```

6.1.2.4 For Statements

The use of general discrete types as ranges in **for** statements provides more descriptive power with little extra logical effort. For example, the **for** statement illustrated in Chapter 5 is

```
type Colors is (RED, YELLOW, ORANGE);
...
Loop_By_Colors:
for This_Color in Colors
loop
  TEXT_IO.Put (Item => " Another Pretty Color.");
end loop Loop_By_Colors;
```

which executes sequentially for `This_Color` taking all values of `Colors`, but `This_Color` disappears on the exit from the **for loop**. In this case, the part function is as follows.

```
[Loop_By_Colors:
for This_Color in Colors
loop
  TEXT_IO.Put (Item => " Another Pretty Color.");
end loop Loop_By_Colors;]
= [TEXT_IO.Put (Item => " Another Pretty Color.");
  TEXT_IO.Put (Item => " Another Pretty Color.");
  TEXT_IO.Put (Item => " Another Pretty Color.");]
= {⟨⟨Input, Output, ...⟩, ⟨Input, Output_1, ...⟩⟩ | Output_1 = Output &
  " Another Pretty Color." & " Another Pretty Color." &
  " Another Pretty Color."}
```

In this case the output did not depend on the value of `This_Color` in detail, but it did depend on the cardinality of `Colors` for the count of simple messages. Another version of the **for** statement with different responses for each iteration is

```
[Loop_By_Colors:
for This_Color in Colors
loop
  TEXT_IO.Put (Item => This_Color);
  TEXT_IO.Put (Item => " is another Pretty Color.");
end loop Loop_By_Colors;]
= [TEXT_IO.Put (Item => " RED is another Pretty Color.");
  TEXT_IO.Put (Item => " YELLOW is another Pretty Color.");
  TEXT_IO.Put (Item => " ORANGE IS another Pretty Color.");]
= {⟨⟨Input, Output, ...⟩, ⟨Input, Output_1, ...⟩⟩ | Output_1 = Output &
  " RED is another Pretty Color." & " YELLOW is another Pretty
  Color." &
  " ORANGE is another Pretty Color."}
```

6.1.3 Ada Exceptions

Ada **exceptions** move execution out of the current sequence of statements for a procedure into another sequence of statements, followed by termination unless more exceptions arise to move execution to even other sequences of statements. Within each such sequence, the meaning of the statements are identical. The format for exceptions is as follows

```
with <packages>
procedure <name of procedure>
is
  <declarations>
begin
  <sequence_of_statements>
exception
  <sequence_of_statements>
end <name of procedure>;
```

For example, `Mystery_Program_5b` illustrated in Chapter 4 is of the form

```
with TEXT_IO;
procedure Mystery_Program_5b
is
  Choice: CHARACTER := ' ';
  Tries: INTEGER range 0 .. 5 := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z'
    then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
exception
  when others =>
    TEXT_IO.Put ("Execution Halted, too many Tries.");
end Mystery_Program_5b;
```

In this case, even though the **while** condition permits `Tries` to have values up to 10, `Tries` is declared only in the range 0 .. 5, so if it is attempted to move `Tries` beyond 5, exception takes over with a single statement before the program terminates.

The exception introduces a new method for transferring control beyond the ordinary sequences, alternatives, and looping. It is a transfer based on the data in the program to a new part of the program. There is no return to the statements exited. In effect, a new program is entered, but everything else is completely natural. In `Mystery_Program_5b`, the single statement

```
Tries := Tries + 1;
```

has two possible exits, not just the ordinary one. The first exit simply remains in the program, in this case returning to the **while** condition as long as `0 <= Tries <= 5`, and the second exit is to the exception through **when** others when `Tries < 0` or `Tries > 5`, which in effect enters a new program, quite short in this case, but not necessarily so. As a result of an exception, the program behavior is expanded by augmenting the main program behavior with additional behavior defined by the exception section. For example, in this example, if the exception section is removed, the program fails unless an upper case letter is entered into `Input` in the first five tries. That is, the program does not terminate in a predictable way. But with the exception section, the program terminates whether or not an upper case letter is entered, with the message in the exception section. In illustration, the behavior of this program can be stated rather simply as follows.

```
[with TEXT_IO;
procedure Mystery_Program_5b
is
  Choice: CHARACTER := ' ';
  Tries: INTEGER range 0 .. 5 := 0;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
    Tries <= 10
  loop
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z'
    then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
exception
  when others =>
    TEXT_IO.Put ("Execution Halted, too many Tries.");
end Mystery_Program_5b;]
  = {⟨⟨Input, Output⟩, ⟨Input_1, Output_1⟩⟩ |
     Input_1 is of length 1 to 5 and last component is upper case
     letter and
     Output_1 = Output & "Welcome Aboard " & ℏ(reverse (Input_1)) or
     Input_1 is of length 5 and no component is upper case letter and
     Output_1 = Output & "Execution Halted, too many Tries."}
```

Note that, calling `Mystery_Program_5b` with exception removed the program `Mystery_Program_5r`, that its behavior is only the first part of the behavior above.

```
[Mystery_Program_5r]
  = {⟨⟨Input, Output⟩, ⟨Input_1, Output_1⟩⟩ |
     Input_1 is of length 1 to 5 and last component is upper case
     letter and
     Output_1 = Output & "Welcome Aboard " & ℏ(reverse (Input_1))}
```

Its domain is limited to cases in which an upper case letter is received in the first five tries.

6.1.4 Exercises

1. Can enumeration data types be declared as a mixture of character_literals and identifiers? What about the single letter A, is that a character_literal, an identifier, or both?

2. Declare an array object called Pass of 50 BOOLEAN values and initialize them all to FALSE. How would you handle a need to have odd values FALSE and even values TRUE?

3. Declare a STRING object called Text of POSITIVE number Length which is to contain the text of a book. What special characters might be used as symbols to organize the content of Text into lines, blank lines, centered text within lines, pages?

4. Define an **if** statement to deal with checking a Roman Digit, called Next, for correct position in a Roman Numeral, given that the last Roman Digit, Last, is known. Can **elsif** parts help?

5. Define a **case** statement to deal with checking a Roman Digit, called Next, for correct position in a Roman Numeral, given that the last Roman Digit, Last, is known.

6. **Block** statements allow local data declarations, unlike **for loop** statements. But **for loop** statements allow an implicit declaration of a loop variable. Show how **block** statements can simulate **for loop** statements using an initialized **while loop**. For example, consider

```
Output_List:
for Alpha in 1 .. 25
loop
  TEXT_IO.Put (Item => Text(Alpha);
  TEXT_IO.New_Line;
end loop Output_List;
```

   and define an equivalent **block** statement.

6.2 Program Specifications

6.2.1 Program Specifications as Mathematical Relations

A **program specification** defines what a program is required to do in all possible circumstances, namely its expected behavior. Suppose such a program is headed by a

```
with TEXT_IO;
```

statement, or a similar statement. Then the program has standard Input and Output files which are empty at the start, before declarations and execution, say $\texttt{Input}_0$ and $\texttt{Output}_0$. Next, during execution data may be entered as Input, say $\texttt{Input}_1$. The result of execution can bring data to Output, say $\texttt{Output}_1$. As a result of program execution, the empty files $\texttt{Input}_0$ and $\texttt{Output}_0$ are transformed into $\texttt{Input}_1$ and $\texttt{Output}_1$, so that during the entire execution of the program, generalized assignments are made in the forms

$\texttt{Input}_0$ <- $\texttt{Input}_1$

$\texttt{Output}_0$ <- $\texttt{Output}_1$

Continuing, a **program part specification** defines what a program part is required to do in all possible circumstances, namely its expected behavior. Then the program part of a program headed by a

```
with TEXT_IO;
```

statement possibly has some data stored in $\texttt{Input}_0$ and $\texttt{Output}_0$ at the start of the declarations and execution of the program part. And during execution additional data may be entered into $\texttt{Input}_1$. The result of execution can bring additional data into $\texttt{Output}_1$. As a result of program part execution, $\texttt{Input}_0$ and $\texttt{Output}_0$ are expanded by $\texttt{Input}_1$ and $\texttt{Output}_1$, again in the form of generalized assignments

$\texttt{Input}_0$ <- $\texttt{Input}_0$ & $\texttt{Input}_1$
$\texttt{Output}_0$ <- $\texttt{Output}_0$ & $\texttt{Output}_1$

If what is required is not unique for every possible initial state and input, the specification is a mathematical relation, a set of ordered pairs such that each initial state and input defines a final output. If what is required is unique for every possible initial state and input, the specification is a mathematical function, which is also a relation.

For example, let S be a program specification, say of the form

$$\texttt{S} = \{\langle\langle \texttt{Input}_0, \texttt{Output}_0\rangle, \langle \texttt{Input}_1, \texttt{Output}_1\rangle\rangle \mid \ldots\}$$

where the ... signifies whatever conditions that relate Ada Text_IO files $\texttt{Input}_0$, $\texttt{Output}_0$, $\texttt{Input}_1$ to $\texttt{Output}_1$ in this specification. Specification S can be quite arbitrary in its domain, possibly not defined for all values of $\texttt{Input}_0$, $\texttt{Output}_0$, $\texttt{Input}_1$. In illustration, a specification involving arithmetic division will not be defined for an argument in which a divisor may become zero.

As noted, in the case of a full Ada program, the standard Input and Output files, $\texttt{Input}_0$ and $\texttt{Output}_0$, will always be empty strings. So it is possible in full Ada programs for the specification S to be written as

$$\texttt{S} = \{\langle\langle\langle\rangle, \langle\rangle\rangle, \langle \texttt{Input}_1, \texttt{Output}_1\rangle\rangle \mid \ldots\}$$

We will ordinarily use the first form, with $\texttt{Input}_0$, $\texttt{Output}_0$ visible in form, even though known to be empty for a full Ada program. But as also noted above, in Ada program parts the initial files $\texttt{Input}_0$ and $\texttt{Output}_0$ need not be empty. And more broadly, even full programs in other programming languages may be defined for $\texttt{Input}_0$ and $\texttt{Output}_0$ not empty. In addition, other files than Input and Output in Ada will be introduced later, and such files need not be empty initially.

In illustration, a specification to list the characters of $\texttt{Input}_1$ in $\texttt{Output}_1$, with no requirement on their order, will be a relation, say List, with the form

$$\begin{aligned}\texttt{List} = \{&\langle\langle \texttt{Input}_0, \texttt{Output}_0\rangle, \langle \texttt{Input}_1, \texttt{Output}_1\rangle\rangle \mid \\ &\texttt{Output}_1 \text{ is any permutation of } \texttt{Input}_1\}\end{aligned}$$

For example, to list the characters of $\texttt{Input}_1$ with string <c, a, b>, any of six strings,

```
<a, b, c>, <b, c, a>, <c, a, b>, <b, a, c>, <a, c, b>, <c, b, a>
```

would be correct for Output$_1$. With identical characters, say <c, b, c>, only three distinct strings,

<b, c, c>, <c, b, c>, <c, c, b>

would be correct for Output$_1$.

But a specification to list the characters of Input$_1$ in sorted order, say in increasing values, will be a function, say Sort, with the form

Sort = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ |
Output$_1$ is the sorted list of Input$_1$}

To list the members of <c, a, b> in sorted order, only the string <a, b, c> would be correct for Output$_1$. If identical characters appear, they will be duplicated. But it may be desired to only list sorted distinct characters depending on the uses of the specification, say in function Dsort with the form

Dsort = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ |
Output$_1$ is the sorted list of distinct components of Input$_1$}

With identical characters, say <c, b, c>, only the string <b, c> would be correct for Output$_1$.

Still another specification might be to list the components of Input$_1$ string, say <c, a, b>, in reverse order which will also be a function, with only the string <b, a, c> correct for Output$_1$, given as

Reverse = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ |
Output$_1$ is the reverse list of Input$_1$}

Program specifications can be large and complex, and often must be stated in large part in natural language. But a specification will still be a relation or function, whether expressed in mathematical notation or natural language. In either case good logical discipline should be used. There are many ways to describe sets, relations, or functions, depending on the context of the description. Specifications need to be understood by potential users of programs as well as by the people who will create the programs.

6.2.2 A Specification for Program_1

For example, a specification called Specify_1 implies the required behavior of Program_1 of Chapter 3, as follows, where Input$_0$, Output$_0$ are empty character strings to begin with.

Specify_1 = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ |
Input$_1$ contains a single character called Char and
Output$_1$ contains a request for Char followed by either
the message "Welcome Aboard " followed by Char if Char
received is a capital letter or the message
"Not a capital letter!"}

The dynamics of generating Input$_1$ and Output$_1$ interactively needs to be spelled out when not obvious, as it may be here. In this case, the time sequence of interaction between Output and Input will be in the order

```
Output1.1 : request for Char
Input1.1  : input of Char
Output1.2 : message either "Welcome Aboard " & Char or
                           "Not a capital letter!"
```

In this case the "request for Char" is undefined in detail, just required in some form. In Program_1, this request for Char has the detailed form of "Enter a capital letter => " which appears to be a reasonable way to make the request. The name "Char" in the specification is not strictly necessary, but can be expressed in terms of Input$_1$, namely as the element *ft*(reverse(Input$_1$)), or since Input$_1$ is of length 1, even *ft*(Input$_1$). Yet such a shorter and generic name "Char" may be useful in communicating with users. In any case, there is a difference between the one character string making up Input$_1$ and the character in the string.

The final form of both Input$_1$ and Output$_1$ can be very different in different situations with more complex interaction in data generation, although quite simple here. In particular, in this program there is only one sequence of interaction between Output and Input, but in general many different sequences may be possible that require much more thought in their description.

A more detailed form of Specify_1 that defines "request for Char" more specifically, say Specify_1d, is

```
Specify_1d = {⟨⟨Input0, Output0⟩, ⟨Input1, Output1⟩⟩ |
               Input1 contains a single character called Char and
               Output1 contains a message "Enter a capital letter => "
               followed either by the message "Welcome Aboard "
               followed by Char if Char received is a capital letter
               or by the message "Not a capital letter!"}
```

This is exactly the specification for Program_1, with no freedom for changing messages.

A more general form of Specify_1 that gives freedom for defining the message of failure, say Specify_1g, is

```
Specify_1g = {⟨⟨Input0, Output0⟩, ⟨Input1, Output1⟩⟩ |
               Input1 contains a single character called Char and
               Output1 contains a request for Char followed either by
               the message "Welcome Aboard " followed by Char if Char
               received is a capital letter or by a message of
               failure}
```

In these three examples, Specify_1d gives a complete specification with no choice in its response, Specify_1 gives some choice in making the request for Char, and Specify_1g gives choice in both the request for Char and a failed input message. An even more general specification can be given in making the message for a successful input arbitrary in actual text returned. Which level of detail is best depends on the situation, in particular on the user situation. Even the most general form still requires a program design that is correct up to the actual messages served up. Those messages can well be parameters to be filled in depending on the expected usage.

These specifications are intended to be understandable by users, who may be software engineers. They need to be precise, but also need to be readable as well. As in the case of `Program_1`, there are many choices in the actual form to be used. For example, another alternative is of the form given next.

```
Specify_1 = A ∪ B

where A = {(⟨Input₀, Output₀⟩, ⟨Input₁, Output₁⟩) |
           Input₁ contains a single character Char which is a capital
           letter and Output₁ contains a request for Char followed by
           the message "Welcome Aboard " followed by Char}

  and B = {(⟨Input₀, Output₀⟩, ⟨Input₁, Output₁⟩) |
           Input₁ contains a single character Char which is not a
           capital letter and Output₁ contains a request for Char
           followed by the message "Not a capital letter!"}
```

and "`request for Char`" found in both A and B are identical in meaning. Although the specification seems different in form, it defines the same specification as shown above. In real practice it may be an important point to verify they are the same, even though written differently.

An even simpler form from the standpoint of the user might be

```
Specify_1 = Request input character, if input is capital letter
            return "Welcome Aboard " followed by the input, otherwise
            return"Not a capital letter!"
```

It is easy for the software engineer to convert this into a more formal form as given above if the user is more comfortable and agrees with this less formal form.

In any case, the users need to participate and understand specifications in order to establish correct technical objectives for software engineering.

6.2.3 A Specification in Roman Numeral Arithmetic

Checking roman numerals is more complex and interesting than checking roman digits. A roman numeral must not only be composed of roman digits, the roman digits must be in correct order and not too many. As already noted, early roman numerals can be defined in context-free syntax, as strings of letters with rules about their possible order and values in a single syntax production.

Carrying out arithmetic among roman numerals is more complex and interesting than just checking their legality. Suppose a program is required for handling roman numeral arithmetic. The problem statement is given in general text form below, followed by a specification. In what follows assume only the knowledge of the ancient romans about numbers and arithmetic. In particular they did not know place notation for numbers or methods such as long division.

6.2.3.1 Roman Numeral Syntax and Semantics

A roman numeral is a nonempty sequence, call it RN, of roman digits which each appear, if at all, in the order and maximum repetitions given next, and whose value is the sum of the roman digit values appearing:

`M` : one thousand (any number)
`D` : five hundred (at most one)
`C` : one hundred (at most four)
`L` : fifty (at most one)
`X` : ten (at most four)
`V` : five (at most one)
`I` : one (at most four)

For example, MDCLXVI, CCCLXXXVIII, X, II are roman numerals but CDM, IX, R are not roman numerals. IX might be considered a roman numeral in another definition, but not here.

A program is required to accept problems in roman numeral arithmetic of the following types:

`V:` Validate that a sequence of characters is a roman numeral.

`C:` Compare two roman numerals in magnitude.

`A:` Add one roman numeral to another.

`S:` Subtract one roman numeral from another.

`M:` Multiply one roman numeral by another.

`D:` Divide one roman numeral by another.

It is recognized that any of the problems may be illegal and the alleged roman numerals must be checked. Since there are no negative roman numerals a larger roman numeral cannot be subtracted from a smaller one. Also, since there is no zero in roman numerals, equal roman numerals cannot be subtracted nor a smaller roman numeral divided by a larger one.

6.2.3.2 Roman Numeral Arithmetic Commands and Responses

For each type of problem, a specific syntax for a command is given followed by separator : and the possible responses. The form (A | B) means that either A or B must be chosen. For example, the phrase (is | is not) means that either "is" or "is not" must be chosen.

V RN : RN (is | is not) a valid roman numeral
Examples

```
V MD
  MD is a valid roman numeral
V DM
  DM is not a valid roman numeral
```

C RN1, RN2 : Larger of RN1, RN2 is (RN3 | undefined)
Examples

```
C MDXXXXIIII, MDL
  Larger of MDXXXXIIII, MDL is MDL
C MD, DM
  Larger of MD, DM is undefined
```

A RN1, RN2 : RN1 plus RN2 is (RN3 | undefined)
Examples

```
A MD, MDV
  MD plus MDV is MMV
A XXVII, IIVXX
  XXVII plus IIVXX is undefined
```

S RN1, RN2 : RN1 less RN2 is (RN3 | undefined)
Examples

```
S L, XXVII
  L less XXVII is XXIII
S XXVII, L
  XXVII less L is undefined
```

M RN1, RN2 : RN1 times RN2 is (RN3 | undefined)
Examples

```
M L, XXVII
  L times XXVII is MCCCL
M L, XXL
  L times XXL is undefined
```

D RN1, RN2 : RN1 divided by RN2 is (RN3 with (remainder RN4 | no remainder) | undefined)
Examples

```
D CCC, XXX
  CCC divided by XXX is X with no remainder
D CCC, XXXX
  CCC divided by XXXX is VII with remainder XX
D XXX, CCC
  XXX divided by CCC is undefined
```

None of the above Commands : Illegal command
Examples

```
+ XXV, L
  Illegal command
/ XXV by L
  Illegal command
```

In this case the specification for roman numeral arithmetic is a function. Every response is uniquely determined.

6.2.3.3 Roman Numeral Arithmetic Specification

The roman numeral arithmetic specification will be defined in terms of the syntax in `Input` and subsequent responses in `Output`. `Input` can hold several problems, separated by commas, and `Output` will hold answers for each problem separated by line ends. The syntax is given next, with literals shown in boldface

```
input_file     ::= problem {problem}

output_file    ::= response {line_end response}
```

```
problem          ::= V roman_numeral ,

                   | C roman_numeral , roman_numeral ,

                   | A roman_numeral , roman_numeral ,

                   | S roman_numeral , roman_numeral ,

                   | M roman_numeral , roman_numeral ,

                   | D roman_numeral , roman_numeral ,

roman_numeral ::= {character}

response          ::= roman_numeral (is | is not) a valid roman
                        numeral.

                    | Larger of roman_numeral, roman_numeral is
                        (valid_roman_numeral . | undefined.)

                    | roman_numeral plus roman_numeral is
                        (valid_roman_numeral . | undefined.)

                    | roman_numeral less roman_numeral is
                        (valid_roman_numeral . | undefined.)

                    | roman_numeral times roman_numeral is
                        (valid_roman_numeral . | undefined.)

                    | roman_numeral divided by roman_numeral is
                        (valid_roman_numeral with remainder
                        valid_roman_numeral .

                     | valid_roman_numeral with no remainder. |
                        undefined.)

                     | Illegal command.

valid_roman_numeral ::= {M}[L]{C:4}[D]{X:4}[V]{I:4}
```

The term character is described no further, so `roman_numeral` may be a valid roman numeral or not. With each operation, only valid `roman_numeral` operands can create a `valid_roman_numeral`, and the response is **undefined** otherwise. In addition to correct syntax, the arithmetic operations must provide correct answers when they are legal. For example, as noted above, both Subtract and Divide require certain comparisons to hold between valid roman numeral operands. The subtractor must be strictly less than the subtrahend, and the divisor must be less than or equal to the dividend.

6.2.4 Exercises

1. Create specifications for ARRAYS of INTEGERS that correspond to `List`, `Sort`, `Dsort`, `Reverse` for CHARACTER STRINGS, called `List_Array`, `Sort_Array`, `Dsort_Array`, `Reverse_Array`. Are there degrees of freedom for ARRAYS of INTEGERS not present in CHARACTER STRINGS?

2. Create a specification, called `Repeats`, to list all characters repeated (adjacent characters are identical) entered in `Input`$_1$ until digit '`0`' is entered.

3. Create a specification, called `Squares`, to sum the squares of digits in entered in `Input`$_1$ until '`0`' is entered.

4. Create a specification, called `Trend`, that `Trend_Program`, Exercise 8, Section 5.1 , would satisfy.

5. Create a specification, called `More`, that `Program_6`, Exercise 7, Section 3.2, would satisfy.

6. Create a specification, called `Sides`, that program `Triangles`, Exercise 3, Section 4.1, would satisfy.

7. Create a specification, called `No_Mystery`, that program `Mystery_Again`, Exercise 6, Section 5.2, would satisfy.

8. Expand the specification for Roman Numerals to modern forms that permit preceding certain digits with lower level digits, such as IV as a possible alternative for IIII. First, identify all correct additions to the forms, second identify the numerical values of each, third define an algorithm for identifying all such modern forms.

6.3 Program Verification

6.3.1 Program Correctness

Given a specification relation (which is possibly a function) and a behavior of a program (which is possibly a function or a relation), the program is said to be **correct** if it terminates from every argument in the domain of the specification and produces a value which is paired with that argument in the specification. As before, let `S` be a specification, say of the form

$$S = \{\langle\langle \mathtt{Input}_0,\ \mathtt{Output}_0\rangle,\ \langle \mathtt{Input}_1,\ \mathtt{Output}_1\rangle\rangle \mid \ldots\}$$

and also let `P` be a program, so `[P]` is the program behavior of `P` of the form

$$[P] = \{\langle\langle \mathtt{Input}_0,\ \mathtt{Output}_0\rangle,\ \langle \mathtt{Input}_2,\ \mathtt{Output}_2\rangle\rangle \mid \ldots\}$$

In either case, either every value of `Input`$_1$ or `Input`$_2$ can be associated with a unique value of `Output`$_1$ or `Output`$_2$ (`S` or `[P]` is a function) or some values of `Input`$_1$ or `Input`$_2$ may be associated with a set of two or more `Output`$_1$ or `Output`$_2$ values (`S` or `[P]` is a relation).

As noted, specification `S` can be quite arbitrary, possibly not defined for all arguments of `Input`$_1$. In this case the domain of `S` will be of the form

$$\mathtt{domain}\ (S) = \{\mathtt{Input}_1 \mid \mathtt{Output}_1\ \mathtt{is\ defined}\}$$

For an Ada program `P`, program behavior `[P]` can accept some values of `Input`$_2$ and create a value of `Output`$_2$. But, for example, values for `Output`$_2$ will not be defined for potential arguments of `Input`$_2$ which lead to infinite looping in executing program `P` and no termination. Another example is attempts to divide by zero with no exception handing. So the domain of specification `S` will be as defined, and the domain of program behavior `[P]` will be defined by program `P` termination, even termination by exception. Again, the domain of `[P]` will be of the form

$$\mathtt{domain}\ ([P]) = \{\mathtt{Input}_2 \mid \mathtt{Output}_2\ \mathtt{is\ defined}\}$$

For P to be correct, it must first terminate from every member of the domain of S. That is, the domain of [P] must include the domain of S, namely

$$\texttt{domain (S)} \subset \texttt{domain([P])},$$

and, second, for any argument in domain of S, every value from program behavior [P] must agree with some value paired with that argument in the relation S, namely

$$\begin{array}{l}\texttt{if Input}_1 \in \texttt{domain (S)}\\ \texttt{then } \langle \texttt{Input}_1\texttt{, [P](Input}_1\texttt{)}\rangle \in \texttt{S}\end{array}$$

That is, if [P] is a function, there will only be one value paired with each argument to be found in the relation of S. But if [P] is a relation, then every value paired with each argument must be found in the relation of S.

Once understood, there is a simpler way to combine these two conditions into a single condition. The second condition identifies pairs

$$\langle \texttt{Input}_1\texttt{, [P](Input}_1\texttt{)}\rangle$$

that must be members of S, and these pairs are also clearly members of [P]. That is, these pairs are members of

$$\texttt{[P]} \cap \texttt{S,}$$

while the first condition requires that every member in the domain of S be a first component $\texttt{Input}_1$ in a pair

$$\langle \texttt{Input}_1\texttt{, [P](Input}_1\texttt{)}\rangle$$

of both [P] and S. Putting this together, the **condition of correctness** for program P with specification S is that

$$\texttt{domain ([P]} \cap \texttt{S) = domain (S)}$$

At first glance, this seems to be a very simple condition. But it just the condition that is needed.

When specification S is a function, it is clear that [P] must be a function as well. Otherwise different values of [P] could not equal a single value from S. In this case, this condition of correctness can be stated more simply as

$$\texttt{S} \subset \texttt{[P]}$$

because in this case

$$\texttt{[P]} \cap \texttt{S = S}$$

and

$$\texttt{domain([P]} \cap \texttt{S) = domain (S)}$$

directly.

6.3.2 Program Correctness Example

As an example of program correctness, suppose a specification named `Welcome` is of the following form.

```
Welcome = {Request up to ten input characters, by messages
           "Try input. " until a capital letter is returned, if ever.
           If capital letter is returned, respond "Welcome Aboard "
           followed by the capital letter; if no capital letter is
           returned respond with rejection message.}
```

Specification `Welcome` can be reformulated more formally as follows, assuming $\texttt{Input}_0$, $\texttt{Output}_0$ are empty strings.

`Welcome` = {⟨⟨$\texttt{Input}_0$, $\texttt{Output}_0$⟩, ⟨$\texttt{Input}_1$, $\texttt{Output}_2$⟩⟩ |
`length` ($\texttt{Input}_1$) `<= 10 and if UCL is first and only upper`
`case letter in` $\texttt{Input}_1$ `then` $\texttt{Output}_2$ `= c * "Try input. "`
`(1 <= c <= 10) & "Welcome Aboard " & UCL else`
`length` ($\texttt{Input}_1$) `= 10 and no upper case letter in` $\texttt{Input}_1$
`and` $\texttt{Output}_2$ `= 10 * "Try input. " & rejection message}`

where `c * "Try input. "` (1 <= c <= 10) means a number `c` of `"Try input. "` messages where `c` is an integer between 1 and 10.

From Chapter 4, `Mystery_Program_5` is a candidate to meet the specification `Welcome`. In the interest of better communications with the user, we augment `Mystery_Program_5` with an additional `Put` statement in the **loop** to make the search for a character more specific, calling it `Search_Program`.

```
with TEXT_IO;
procedure Search_Program
is
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 1;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if 'a' <= Choice and Choice <= 'z'
    then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
  end if;
end Search_Program;
```

with program behavior is

```
[Search_Program] =
  {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_2$⟩⟩}
```

where

```
if ℎ(reverse (Input$_1$)) in 'A' .. 'Z')
length (Input$_1$) <= 10,
no preceding component of Input$_1$ in 'A' .. 'Z'
  Output$_2$ = c * ("Try input. ") (1 <= c <= 10) & "Welcome Aboard " &
    ℎ(reverse (Input$_1$))

if ℎ(reverse (Input$_1$)) in 'a' .. 'z'
length (Input$_1$) = 10
no component of Input$_1$ in 'A' .. 'Z',
  Output$_2$ = 10 * ("Try input. ") & "Lower Case Data Input " &
    ℎ(reverse (Input$_1$))

if ℎ(reverse (Input$_1$)) not in 'a' .. 'z'
length (Input$_1$) = 10
no component of Input$_1$ in 'A' .. 'Z',
  Output$_2$ = 10 * ("Try input. ") & "Try Again"
```

More formally, the domains of [Search_Program] and Welcome are identical, namely all sets Input$_1$, and every member of [Search_Program], namely

```
⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_2$⟩⟩
```

is also a member of Welcome, first if an upper case letter is in Input$_1$, and second if not, in which Welcome combines the other two cases in [Search_Program]. Thus, in particular,

```
domain (Welcome * [Search_Program]) = domain (Welcome),
```

as was needed to be shown.

6.3.3 A Simpler Program

It is clear that Search_Program could be simplified and still satisfy Welcome, calling it Simple_Program, as

```
with TEXT_IO;
procedure Simple_Program
is
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 1;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  -- else
    -- if 'a' <= Choice and Choice <= 'z'
    -- then
      -- TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  -- end if;
  end if;
end Simple_Program;
```

or, removing the comments showing the statements removed

```
with TEXT_IO;
procedure Simple_Program
is
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 1;
begin
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  end if;
end Simple_Program;
```

In this case, the program behavior of `Simple_Program` is

$$[\texttt{Simple\_Program}] = \{\langle\langle \texttt{Input}_0, \texttt{Output}_0\rangle, \langle \texttt{Input}_1, \texttt{Output}_3\rangle\rangle\}$$

where

if $\hbar$(reverse (Input$_1$)) in 'A' .. 'Z')
length (Input$_1$) <= 10,
no preceding component of Input$_1$ in 'A' .. 'Z'
  Output$_3$ = c * ("Try input. ") & "Welcome Aboard " &
    $\hbar$(reverse (Input$_1$))

length (Input$_1$) = 10
no component of Input$_1$ in 'A' .. 'Z',
  Output$_3$ = 10 * ("Try input. ") & "Try Again"

More formally, the domains of [Simple_Program] and Welcome are identical, namely all sets $\texttt{Input}_3$ and every member of [Simple_Program]

$$\langle\langle \texttt{Input}_0, \ \texttt{Output}_0\rangle, \ \langle \texttt{Input}_1, \ \texttt{Output}_3\rangle\rangle\}$$

is also a member of Welcome and vice versa. Thus, in particular.

$$\texttt{domain (Welcome} \cap \texttt{[Simple\_Program])} = \texttt{domain (Welcome)}$$

as was needed to be shown.

6.3.4 A Top Level Program for Roman Numeral Arithmetic

Given the specification for Roman Numeral Arithmetic, a top level program can be designed to recognize problems and call the proper procedures to solve them. In this case, these procedures will be provided in a package called Roman_Numeral_Operators that must be included in the opening with clause of the program. The specifications for these next level procedures become an integral part of the top level program solution.

```
with TEXT_IO;
with Roman_Numeral_Operators;
procedure Roman_Numeral_Arithmetic
is
  Terminate : BOOLEAN := FALSE;
  Problem : CHARACTER;
begin
  Roman_Numeral_Problems:
  while Terminate = FALSE
  loop
    TEXT_IO.Put (Item => "Next Problem? "
    TEXT_IO.Get (Item => Problem);
    case Problem
    is
      when 'V' =>
        Validate;
      when 'C' =>
        Compare;
      when 'A' =>
        Add;
      when 'S' =>
        Subtract;
      when 'M' =>
        Multiply;
      when 'D' =>
        Divide;
      when 'T' =>
        Terminate := TRUE;
      when others =>
        Error;
    end case;
  end loop Roman_Numeral_Problems;
end Roman_Numeral_Arithmetic;
```

The package Roman_Numeral_Operators in the with clause contains the procedures called for in the case statement. The specifications for the procedures in Roman_Numeral_Operators are given next.

Validate: Validate that the next sequence of characters up to a comma in Input, called roman_numeral, is a valid roman numeral or not. Put to Output the proper message with the syntax

```
roman_numeral (is | is not) a valid roman numeral.
```

Compare: Compare the next two sequences of characters separated by a comma and ended by a comma, called roman_numeral_1, roman_numeral_2, as roman numerals in magnitude. Put to Output the proper message with the syntax

```
Larger of roman_numeral_1, roman_numeral_2 is
   (roman_numeral_3 . | undefined.)
```

Add: Add the next two sequences of characters separated by a comma and ended by a comma, called roman_numeral_1, roman_numeral_2, as roman numerals. Put to Output the proper message with the syntax

```
roman_numeral_1 plus roman_numeral_2 is
   (roman_numeral_3 . | undefined.)
```

Subtract: Subtract the second from the first of the next two sequences of characters separated by a comma and ended by a comma, called roman_numeral_1, roman_numeral_2, as roman numerals. Put to Output the proper message with the syntax

```
roman_numeral_1 less roman_numeral_2 is
   (roman_numeral_3 . | undefined.)
```

Multiply: Multiply the next two sequences of characters separated by a comma and ended by a comma, called roman_numeral_1, roman_numeral_2, as roman numerals. Put to Output the proper message with the syntax

```
roman_numeral_1 times roman_numeral_2 is
   (roman_numeral . | undefined.)
```

Divide: Divide the first by the second of the next two sequences of characters separated by a comma and ended by a comma, called roman_numeral_1, roman_numeral_2, as roman numerals. Put to Output the proper message with the syntax

```
roman_numeral_1 divided by roman_numeral_2 is
   (roman_numeral_3 with remainder roman_numeral_4 .
   | roman_numeral with no remainder.
   | undefined.)
```

Terminate: Terminate computation

```
Terminate := TRUE;
```

Error: Put to Output the message

```
Illegal command.
```

The procedures specified must be designed and coded in Ada. Some of these procedures will be useful to others. For example, procedure Validate will be useful for all the other procedures, to check if their input character strings are indeed roman numerals. Every other procedure can report a problem as undefined if the strings are not roman numerals. The procedure Compare will be useful for both Subtract and Divide procedures. Once over the hurdle of legal roman numerals, Subtract and Divide both must determine if the problem is legal by comparing the sizes of the two roman numerals. Procedure Compare might also be useful for the Add and Multiply procedures for optimization purposes. There may be an advantage to knowing which of the two roman numerals are smallest.

The design of these procedures will be taken up later, beginning with Chapter 7. The data structure of STRING will be very useful in designs for these procedures. It will also be useful to think through how these procedures can pass data in STRING form rather than file forms as defined by the top level program.

6.3.5 Exercises

1. Is it possible for a program P that does not initialize declared objects before their use be correct with respect to a specification S? What if [P] is a relation, not a function?

2. Given a specification S which is a function and a program P such that [P] is a relation, not a function, can program P be correct with respect to S?

3. In writing down Search_Program for execution, a small mistake was made, interchanging the **while loop** and the if statement Call the resulting program Search_Program_2 and determine its behavior.

4. If Simple_Program is found deficient because of lack of final communication if no upper case letter is found, how could it be fixed with a single message that ignores lower case letters?

5. Consider the specification

```
Check = {Check up to five input characters, by request messages
         until a digit is returned, if ever. If a digit is
         returned, respond with positive message, otherwise
         respond with negative message.}
```

   Convert this specification to a more formal form, then modify Simple_Program with program name Input_Digit.

6.4 Program Part Verification

6.4.1 Program Verification in Hierarchical Structures

As already noted, Ada programs have hierarchical control structures which can be used in organizing their verifications. Each sequence, if, or **loop** statement in the hierarchy can be verified, based on its specification and the specifications of its component parts. Each of those parts can, in turn, be verified against its specification using the specifications of its components in turn. These verifications can be carried out at various levels of formality. The more formal the verification, the less fallibility can be expected, but the more time it takes. The question is not whether to verify programs or not, but what level of formality to use in verification. There is not a simple or single answer to this question. Some programs are more critical than others and different levels of effort will be put into them. Programs to be used only once or twice for noncritical purposes may not be worth the effort of those used many times with human life dependent on correct programs. The same person or team can encounter both situations in their work simultaneously.

At the top level of the hierarchy the specifications will be in the context of the application area. But as these specifications are decomposed by the structure of the program, the specifications of lower level parts become more program oriented, and specifically, become more like conditional or concurrent assignment specifications. Such **conditional or concurrent forms** may be relations by providing freedom in assignments in explicit ways. For example, the concurrent form

```
x1, y2, z3 <- arbitrary, z3, y2;
```

is a relation that places no requirement on `x1` while interchanging `y2` and `z3`. In illustration, either of the following sequences will accomplish this interchange.

```
x1 := y2;
y2 := z3;
z3 := x1;   ** x1 ends with value of initial y2

x1 := z3;
z3 := y2;
y2 := x1;   ** x1 ends with value of initial z3
```

But, for example, another sequence will do, such as

```
x1 := y2;
y2 := z3;
z3 := x1;
x1 := 0;
```

which initializes `x1`, possibly for another use.

Even at the top level, the conditional and concurrent forms are always possible, and provide an effective discipline for specifications in application contexts. As such, they are useful in creating structured specifications in forms that are understandable by both users and designers.

6.4.2 Hierarchical Structure of Search_Program

Recall that the predecessor of `Search_Program` was analyzed in its hierarchical structure in Chapter 4, where part behaviors were discovered and combined into larger part behaviors from the bottom up until the program behavior was discovered.

6.4.2.1 Low Level If Statement

`Search_Program` has a low level if statement

```
if1 =
  if 'a' <= Choice and Choice <= 'z'
  then
    TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  end if;
```

with part behavior

$$[\texttt{if1}] = \{\langle\langle \texttt{Input}_0,\ \texttt{Output}_0,\ \texttt{Choice},\ \ldots\rangle,\ \langle \texttt{Input}_1,\ \texttt{Output}_1,\ \texttt{Choice},\ \ldots\rangle\rangle\}$$

where

```
if 'a' <= Choice and Choice <= 'z',
  Output1 = "Lower Case Data Input " & Choice

if not ('a' <= Choice and Choice <= 'z'),
  Output1 = "Try Again"
```

This very assertion that if1 has part behavior [if1] is a matter for proof. In this case it is quite straightforward, but mistakes are possible and independent inspections can reduce mistakes significantly. It is clear by inspection that the only object being changed is Output$_1$, and that "Lower Case Data Input " & Choice or "Try Again" will be added to Output$_1$ depending on the value of Choice. In particular, Choice will be used but not altered altered in this if statement.

Note one important thing about Output. It is whatever is in Output at the moment, not the Output mentioned at the start of the whole program. As noted above, unlike full programs whose initial values for Input$_0$ and Output$_0$ are always empty strings, program parts, such as this if statement may start with values for Input$_0$ and Output$_0$ which are not empty strings. In particular, in this full program, the while statement preceding the nested if statement will have generated data that will be in Input$_0$ and Output$_0$ now. That previous data is not part of Output$_1$.

An inspection of if1 shows that the additions are made correctly.

6.4.2.2 Next Level If Statement

Next, if1 is part of the larger if statement if2, where

```
if2 =
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    if1
  end if;
```

whose part behavior is

```
[if2] = {⟨⟨Input0, Output0, Choice, ...⟩,
         ⟨Input1, Output2, Choice, ...⟩⟩}
```

where bringing the results of if1 above forward and calling the new Output file Output$_2$ for convenience at the moment

```
if 'A' <= Choice and Choice <= 'Z'
    Output2 = "Welcome Aboard " & Choice

if not ('A' <= Choice and Choice <= 'Z') and
  'a' <= Choice and Choice <= 'z'
    Output2 = "Lower Case Data Input " & Choice
```

```
if not ('A' <= Choice and Choice <= 'Z') and
  not ( 'a' <= Choice and Choice <= 'z')
    Output2 = "Try Again"
```

Again, the assertion that if2 has part behavior [if2] needs proof. In this case again, the only object being changed is Output$_2$, either by the then part with the TEXT_IO.Put statement or the else part if1, already examined and verified. As before, an inspection indicates the correct additions are made to Output for the conditions.

6.4.2.3 Initialized While Loop Statement

Continuing, the initialized while loop statement just before the outer if statement if2 is

```
wh1 =
  Choice := ' '; -- from declaration
  Tries := 1;    -- from declaration
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice >= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
```

with part behavior as follows

```
[wh1] = {⟨⟨Input0, Output0, Choice, Tries⟩,
          ⟨Input1, Output1, Choice1, Tries1⟩⟩}
```

where

Input$_1$ will be a string of 1 to 10 characters, only the last, if any, an upper case letter.

Output$_1$ will be a string of messages "Try input. " of same number as Input$_1$ components.

Choice$_1$ will be last value of the Input$_1$ string, *h*(reverse(Input$_1$)).

Tries$_1$ will be an integer between 2 and 11, one more than the length of Input$_1$.

This behavior can be determined directly from wh1 by direct analysis. First, Choice and Tries have been initialized when declared, and the while statement is the first statement executed. Next, the first entry into the while loop will be successful in passing the while test because of how the variables Choice and Tries have been initialized. In this first loop a message "Try input. " is sent to Output$_1$, then a value for Choice$_1$ is returned to Input$_1$, and Tries$_1$ is incremented to 2. If Choice$_1$ is now an upper case letter, the loop is terminated next with just those values for the data. If Choice$_1$ is not an upper case letter, the loop is continued with new values added to the Input$_1$ and Output$_1$ strings, and new values created for Choice and Tries. The loop continues until either Choice contains an upper case letter or Tries exceeds 10. Since Tries is incremented each time through the loop, termination is guaranteed, but the appearance of an upper case letter can terminate the loop before that.

This analysis for the initialized **while** statement is simplified considerably because of where it appears as the opening statement in the program. If not initialized as found here, the **while** statement must be prepared for any possible values for `Choice` and `Tries`, and the analysis is much more complex, as will be seen later.

6.4.2.4 Entire Executable Part

Next, the entire executable part of the program is the sequence `seq` where

```
seq =
  wh1
  if2
```

and

```
[seq] = ⟨⟨Input$_0$, Output$_0$, Choice, Tries⟩,
         ⟨Input$_1$, Output$_2$, Choice$_1$, Tries$_1$⟩⟩
```

which is determined by the two steps of the sequence. As above, we suppose values Input$_1$, Output$_1$, Choice$_1$, Tries$_1$ result from the first step `wh1`. The first step repeated from just above determines values for the data as follows.

Input$_1$ will be a string of 1 to 10 characters, only the last, if any, an upper case letter.

Output$_1$ will be a string of messages "Try input. " of same number as Input$_1$ components.

Choice$_1$ will be last value of the Input$_1$ string, h(reverse (Input$_1$)).

Tries$_1$ will be an integer between 2 and 11, one more than the length of Input$_1$.

The second step uses these values to create a final Output$_2$ from the intermediate values of Input$_1$, Output$_1$, Choice$_1$, as follows.

```
if 'A' <= Choice$_1$ and Choice$_1$ <= 'Z'
    Output$_2$ = Output$_1$ & "Welcome Aboard " & Choice$_1$

if not ('A' <= Choice$_1$ and Choice$_1$ <= 'Z') and
  'a' <= Choice$_1$ and Choice$_1$ <= 'z'
    Output$_2$ = Output$_1$ & "Lower Case Data Input " & Choice$_1$

if not ('A' <= Choice$_1$ and Choice$_1$ <= 'Z') and
  not ('a' <= Choice$_1$ and Choice$_1$ <= 'z')
    Output$_2$ = Output$_1$ & "Try Again"
```

where Output$_1$ is given as above, namely

Output$_1$ will be a string of messages "Try input. " of same number as Input$_1$ components.

Substituting the results of `wh1` into `if2`, we need to formulate the conditions of `wh1` above into the three conditions of `if2` which account for the history of conditions in `wh1` that are unknown in `if2`. For example, in `if2` any upper case letter in Input$_1$ will be printed, whether last or not, no limit exists for the length of Input$_1$, and so on. Going back to `wh1` and putting its results in terms needed for the sequence with if2, we get the following expansion in the if2 step just above in terms of Input$_1$ at the termination of seq.

```
if length (Input_1) <= 10 and
  not ('A' <= x <= 'Z' and x in (t(reverse (Input_1))) and
  'A' <= h(reverse (Input_1)) <= 'Z'
    Choice_1 = h(reverse (Input_1)),
    Tries_1 = length (Input_1) + 1,
    Output_2 = length (Input_1) * "Try input. " & "Welcome Aboard " &
      h(reverse (Input_1))

if length (Input_1) = 10 and
  not ('A' <= x <= 'Z' and x in Input_1) and
  'a' <= h(reverse (Input_1)) <= 'z'
    Choice_1 = h(reverse (Input_1)),
    Tries_1 = 11,
    Output_2 = 10 * "Try input. " & "Lower Case Data Input " &
      h(reverse (Input_1))

if length (Input_1) = 10 and
  not ('A' <= x <= 'Z' and x in Input_1) and
  not ('a' <= h(reverse (Input_1)) <= 'z')
    Choice_1 = h(reverse (Input_1)),
    Tries_1 = 11,
    Output_2 = 10 * "Try input. " & "Try Again"
```

6.4.2.5 Entire Program

And finally, the entire program is

```
with TEXT_IO;
procedure Search_Program
is
  Choice : CHARACTER := ' ';
  Tries  : INTEGER := 1;
begin
  seq
end Search_Program;
```

with the program behavior

```
[Search_Program] = {⟨⟨Input_0, Output_0⟩, ⟨Input_1, Output_2⟩⟩}
```

where

```
if length (Input_1) <= 10 and
  not ('A' <= x <= 'Z' and x in (t(reverse (Input_1))) and
  'A' <= h(reverse (Input_1)) <= 'Z'
    Output_2 = length (Input_1) * "Try input. " & "Welcome Aboard " &
      h(reverse (Input_1))
```

```
if length (Input$_1$) = 10 and
  not ('A' <= x <= 'Z' and x in Input$_1$) and
  'a' <= $\hbar$(reverse (Input$_1$)) <= 'z'
    Output$_2$ = 10 * "Try input. " & "Lower Case Data Input " &
      $\hbar$(reverse (Input$_1$))

if length (Input$_1$) = 10 and
  not ('A' <= x <= 'Z' and x in Input$_1$) and
  not ('a' <= $\hbar$(reverse (Input$_1$)) <= 'z')
    Output$_2$ = 10 * "Try input. " & "Try Again"
```

To show this, first, the statement with TEXT_IO; makes the standard IO package available for this program.

Second, the procedure of the program is named Search_Program.

Third, Ada reserved word is is followed by Choice: CHARACTER := ' '; Tries: INTEGER := 1; that declares the object named Choice to be of type CHARACTER with initial value ' ' and object named Tries of type INTEGER with initial value 1.

Fourth, begin will set the executable part named seq into action, with the result given above, which uses the availability of TEXT_IO and objects named Choice and Tries just declared and initialized.

Fifth, after the line end Search_Program;, all declared data is eliminated. That is, the program behavior of the entire program, including the with clause and declaration, both used implicitly in the derivation of the executable part behavior, is as shown above.

This program has been verified part by part in its hierarchical structure. At each step a working hypothesis equates the part with its asserted part behavior, and the proof involves an analysis that depends on the kind of part it is, whether a sequence, an if statement, a while loop statement, a declaration, or the entire program.

6.4.2.6 Summary of Search_Program

In summary, Search_Program can be analyzed step by step to determine its behavior with respect to a specification. The program provides more response than the specification requires, but is correct. It is given all filled in next, with comments that identify the names of program parts used above.

```
with TEXT_IO;
procedure Search_Program
is
-- begin seq
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 1;
begin
  -- begin whl
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  -- end whl
  -- begin if2
  if 'A' <= Choice and Choice <= 'Z'
  then
    TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
  else
    -- begin if1
    if 'a' <= Choice and Choice <= 'z'
    then
      TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
    else
      TEXT_IO.Put (Item => "Try Again");
    end if;
    -- end if1
  end if;
  -- end if2
-- end seq
end Search_Program;
```

As noted above, program parts called if1, if2, whl, seq, Search_Program are identified and related in building a hierarchical analysis of program behavior.

6.4.3 Exercises

1. Given a specification Trend for Exercise 4, Section 6.2, namely

   Trend = {⟨⟨$Input_0$, $Output_0$⟩, ⟨$Input_1$, $Output_1$⟩⟩ | beginning with
   $Output_1$ empty and getting the first nine characters from
   $Input_1$, compare the first and ninth characters and if the
   ninth is greater than the first put to $Output_1$ the
   message "Trend is upward with full history " followed by
   the nine characters, otherwise put to $Output_1$ this
   message with "upward" replaced by "downward"}

   Create a hierarchical structure of the parts of Trend_Program, Exercise 8, Section 5.1, for a verification that Trend_Program would satisfy Trend.

2. Carry out the verifications of the program parts of the previous Exercise 1.

3. What if the simple exchange discussed above which initializes x1, possibly for another use, is not found together, but spread out among other text as follows?

   ```
   x1 := y2;
   if z3 > 0
   then
     y2 := x1;
   end if;
   y2 := z3;
   ```

```
if y2 < 0
then
  z3 := y2
end if;
z3 := x1;
if y2 < z3
then
  x1 := z3;
end if;
x1 := 0;
```

First, does the interchange still work? Second, can that be proven?

4. Given a specification More for Exercise 5, Section 6.2, namely

    More = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ | if first character
    of Input$_1$ is upper case letter and second character is
    lower case letter, Output$_1$ = "Valid Input Data"}

    create a hierarchical structure of the parts of Program_6, Exercise 7, Section 3.2, for a verification that Program_6 would satisfy More.

5. Given a specification Sides for Exercise 6, Section 6.2, namely

    Sides = {⟨⟨Input$_0$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ | for each triple
    of integers in Input$_1$, put "Sides " followed by the three
    integers, followed by the proper message about these
    integers from " do not make up a triangle.",
    " make up a right triangle.",
    " make up an acute triangle.",
    " make up an obtuse triangle." to Output$_1$}

    create a hierarchical structure of the parts of Triangles, Exercise 3, Section 4.1, for a verification that program Triangles would satisfy Sides.

6. Given a specification No_Mystery for Exercise 7, Section 6.2, namely

    No_Mystery = {⟨⟨Input$_1$, Output$_0$⟩, ⟨Input$_1$, Output$_1$⟩⟩ |
    Output$_1$ will contain the number of characters in
    Input$_1$}

    create a hierarchical structure of the parts of program Mystery_Again, Exercise 6, Section 5.2, for a verification that program Mystery_Again would satisfy No_Mystery.

6.5 Program Part Verification with Behavior Tables

6.5.1 Verifying Opening Sequence Parts

Sequence parts are simple in form, although their component parts may be complex and extensive to describe. In any case, the sequence part and its components can be described as conditional or concurrent forms from input states to output states. The forms may involve many objects, but they are no more than generalized conditional or concurrent assignments. The level of formality required depends on the stakes in designing the program. The number of conditions in the sequence depends on the conditional forms found in its components, and is the product of the number of conditional forms in each component.

For example, the entire executable part of Search_Program analyzed above is defined between the two comments -- begin seq and -- end seq which can be abstracted as

```
-- begin seq
  Choice := ' ';
  Tries := 1;
  wh1
  if2
-- end seq
```

This verification of seq can be carried out more systematically in trace tables, as shown next. The notation for entries in the trace tables will be simplified to a macro level, referencing new values for variables, not simply new lines of the tables.

From the analysis already carried out, it is clear that three different conditions on Input$_1$ will lead to three different execution patterns and can be handled in three separate trace tables. In this case we will deal with statements in general position. As shown above, the three different conditions and consequences are as follows.

```
if length (Input1) <= 10 and
  not ('A' <= x <= 'Z' and x in (ŧ(reverse (Input1))) and
  'A' <= ħ(reverse (Input1)) <= 'Z'
    Choice1 = ħ(reverse (Input1)),
    Tries1 = length (Input1) + 1,
    Output2 = length (Input1) * "Try input. " &
      "Welcome Aboard " & ħ(reverse (Input1))

if length (Input1) = 10 and
  not ('A' <= x <= 'Z' and x in Input1) and
  'a' <= ħ(reverse (Input1)) <= 'z'
    Choice1 = ħ(reverse (Input1)),
    Tries1 = 11,
    Output2 = 10 * "Try input. " & "Lower Case Data Input " &
      ħ(reverse (Input1))

if length (Input1) = 10 and
  not ('A' <= x <= 'Z' and x in Input1) and
  not ('a' <= ħ(reverse (Input1)) <= 'z')
    Choice1 = ħ(reverse (Input1)),
    Tries1 = 11,
    Output2 = 10 * "Try input. " & "Try Again"
```

The trace tables that deal with these three cases are as follows.

Trace Table 6.1

```
if length (Input1) <= 10 and
  not ('A' <= x <= 'Z' and x in (t(reverse (Input1))) and
  'A' <= h(reverse (Input1)) <= 'Z'
    Choice1 = h(reverse (Input1)),
    Tries1 = length (Input1) + 1,
    Output2 = length (Input1) * "Try input. " &
      "Welcome Aboard " & h(reverse (Input1))
```

statements	Condition	Input	Output	Choice	Tries
Choice := ' ';				' '	
Tries := 1;					1
whl	'A'<=h...	I_1	n*"Try..."	h(r(I_1...))	length
	and Tries<=10				(I_1)+1
if2	'A'<=h...		n*"Try..."&		
			"Welcome..."		

where

```
'A'<=h... = 'A' <= h(reverse (Input1)) <= 'Z'
I1 = Input1 = n characters, only last is upper case letter
'Try..." = "Try input. "
"Welcome..." = "Welcome Aboard " & h(reverse (Input1))
h(r(I1...)) = h(reverse (Input1))
```

Trace Table 6.2

```
if length (Input1) = 10 and
  not ('A' <= x <= 'Z' and x in Input1) and
  'a' <= h(reverse (Input1)) <= 'z'
    Choice1 = h(reverse (Input1)),
    Tries1 = 11,
    Output2 = 10 * "Try input. " & "Lower Case Data Input " &
      h(reverse (Input1))
```

statements	Condition	Input	Output	Choice	Tries
Choice := ' ';				' '	
Tries := 1;					1
whl	'A'<=h...	I_1	10*"Try..."	h(r(I_1))	11
	and Tries<=10				
if2	not('A'<=h...)				
	'a'<=h...		10*"Try..."&		
			"Lower..."		

where, in addition to those above,

```
'a'<=... = 'a' <= h(reverse (Input1)) <= 'z'
"Lower..." = "Lower Case Data Input " & h(reverse (Input1))
```

Trace Table 6.3

```
if length (Input1) = 10 and
  not ('A' <= x <= 'Z' and x in Input1) and
  not ('a' <= ℏ(reverse (Input1)) <= 'z')
    Choice1 = ℏ(reverse (Input1)),
    Tries1 = 11,
    Output2 = 10 * "Try input. " & "Try Again"
```

statements	Condition	Input	Output	Choice	Tries
Choice := ' ';				' '	
Tries := 1;					1
wh1	'A'<=ℏ... and Tries<=10	I_1	10*"Try..."	ℏ(r(I_1))	11

if2	not('A'<=ℏ...)
	not('a'<=ℏ...)
			10*"Try..."&
			"Try Again"		

These three trace tables confirm the results obtained more informally above. The three conditions on Input$_1$ and the outcomes of each trace table provide the conditional concurrent assignment for `[seq]` given above, in alternate form, namely

```
[seq] =
  (length (Input1) = n <= 10 and
   'A' <= ℏ(reverse (Input1)) <= 'Z' and
   no preceding character in Input1 in 'A' .. 'Z' ->
    Input1, Output2, Choice, Tries
      <- n * Char,
         n * "Try input. " & "Welcome Aboard " &
           ℏ(reverse (Input1)),
         ℏ(reverse (Input1)),
         length (Input1) + 1;

  length (Input1) = 10 and
  'a' <= ℏ(reverse (Input1)) <= 'z'
  no preceding character in Input1 in 'A' .. 'Z' ->
    Input1, Output2, Choice, Tries
      <- 10 * Char,
         10 * "Try input. " & "Lower Case Data Input " &
           ℏ(reverse (Input1)),
         ℏ(reverse (Input1)),
         11;

  length (Input1) = 10 and
  not ('a' <= ℏ(reverse (Input1)) <= 'z')
  no character in Input1 in 'A' .. 'Z' ->
    Input1, Output2, Choice, Tries
      <- 10 * Char,
         10 * "Try input. " & "Try Again",
         ℏ(reverse (Input1)),
         11;)
```

6.5.2 Verifying General Sequence Parts

Consider now a subsequence of seq, namely the sequence given by

```
seq1 = wh1
       if2
```

that might appear any place in a program, not just start it out. In particular, Choice and Tries are not initialized to special values as above. Then wh1 may never execute its loop if Choice is an upper case letter or Tries is greater than 10. And wh1 may execute its loop a large number of times if Tries is a negative number of large absolute value and no upper case letters are found. When wh1 terminates, if2 will execute normally. The main idea in looking at seq1 is to reexamine the looping condition, in terms of initial values of Choice and Tries. If Choice is an upper case letter initially, the loop will not be executed at all, and nothing really prevents this case in the conditions as expressed. However, the loop conditions on Tries, expressed in terms of Input$_1$ before, are better expressed directly in Tries. These conditions need to allow for the possibility that Tries initially exceeds 10, and since the loop will not execute, will not be changed. The second recognizes that Tries may have an arbitrary value less than 10, rather that initial value 1 as before. The trace tables for this case are organized in two groups of three each, by whether the loop is executed or not, and become as follows. If the loop is executed, the model already used above is very useful. The first three trace tables deal with the loop not executing at all, namely

```
('A' <= Choice and Choice <= 'Z') or
  not (Tries <= 10)
```

In this case, Choice and Tries are simply inherited from the foregoing declarations and statements leading to this point. Choice will then be examined by the if2 statement following wh1.

Trace Table 6.4

```
if 'A' <= Choice <= 'Z'
    Choice$_1$ = Choice,
    Tries$_1$ = Tries,
    Input$_1$ = ⟨⟩,
    Output$_2$ = "Welcome Aboard " & Choice
```

statements	Condition	Input	Output	Choice	Tries
wh1	'A'<=...				
if2	'A'<=...		"Welcome..."		

where

```
'A'<=... = 'A' <= Choice <= 'Z'
```

Trace Table 6.5

```
if Tries > 10 and
  'a' <= Choice <= 'z'
    Choice$_1$ = Choice,
    Tries$_1$ = Tries,
    Input$_1$ = ⟨⟩,
    Output$_2$ = "Lower Case Data Input " & Choice
```

statements	Condition	Input	Output	Choice	Tries
wh1	Tries>10				
if2	'a'<=...		"Lower..."		

where

```
'a'<= = 'a' <= Choice <= 'z'
```

Trace Table 6.6

```
if Tries > 10 and
  not ('A' <= Choice <= 'Z') and
  not ('a' <= Choice <= 'z')
    Choice$_1$ = Choice,
    Tries$_1$ = Tries,
    Input$_1$ = ⟨⟩,
    Output$_2$ = "Try Again"
```

statements	Condition	Input	Output	Choice	Tries
wh1	Tries>10				

if2	not('A'<=...)
	not('a'<=...)
			"Try Again"		

With the three cases of immediate termination of the loop completed, we now look at the cases when the loop is entered.

Trace Table 6.7

```
if not (Tries > 10) and
  not ('A' <= Choice <= 'Z') and
  Tries <= 10 and ** note Input$_1$ will not be empty
  not ('A' <= x <= 'Z' and x in (𝑡(reverse (Input$_1$))) and
  'A' <= ℎ(reverse (Input$_1$)) <= 'Z'
    Choice$_1$ = ℎ(reverse (Input$_1$)),
    Tries$_1$ = length (Input$_1$) + Tries,
    Output$_2$ = length (Input$_1$) * "Try input. " &
      "Welcome Aboard " & ℎ(reverse (Input$_1$))
```

statements	Condition	Input	Output	Choice	Tries
wh1	'A'<=ℎ... and Tries<=10	I_1	n*"Try..."	ℎ(r(I_1...))	length (I_1-I)+T
if2	'A'<=ℎ...		n*"Try..."& "Welcome..."		

where

```
'A'<=ℎ... = 'A' <= ℎ(reverse (Input$_1$)) <= 'Z'
I$_1$ = Input$_1$ = n characters, only last is upper case letter
"Try..." = "Try input. "
"Welcome..." = "Welcome Aboard " & ℎ(reverse (Input$_1$))
ℎ(r(I$_1$...)) = ℎ(reverse (Input$_1$))
length(I$_1$-I)+T = length (Input$_1$) + Tries
```

Trace Table 6.8

```
if not (Tries > 10) and
  not ('A' <= Choice <= 'Z') and
  Tries <= 10 and  ** note Input$_1$ will not be empty
  not ('A' <= x <= 'Z' and x in Input$_1$) and
  'a' <= ħ(reverse (Input$_1$)) <= 'z'
    Choice$_1$ = ħ(reverse (Input$_1$)),
    Tries$_1$ = max (11, Tries),
    Output$_2$ = n * "Try input. " & "Lower Case Data Input " &
      ħ(reverse (Input$_1$))
```

statements	Condition	Input	Output	Choice	Tries
whl	'A'<=ħ...	I$_1$	n*"Try..."	ħ(r(I$_1$))	max(11,
	and Tries<=10				Tries)
if2	not('A'<=ħ...)				
	'a'<=ħ		n*"Try..."&		
			"Lower..."		

where, in addition to those above,

```
'a'<=... = 'a' <= ħ(reverse (Input$_1$)) <= 'z'
"Lower..." = "Lower Case Data Input " & ħ(reverse (Input$_1$))
```

Trace Table 6.9

```
if not (Tries > 10) and
  not ('A' <= Choice <= 'Z') and
  Tries <= 10 and  ** note Input$_1$ will not be empty
  not ('A' <= x <= 'Z' and x in Input$_1$) and
  not ('a' <= ħ(reverse (Input$_1$)) <= 'z')
    Choice$_1$ = ħ(reverse (Input$_1$)),
    Tries$_1$ = max (11, Tries),
    Output$_2$ = n * "Try input. " & "Try Again"
```

statements	Condition	Input	Output	Choice	Tries
whl	'A'<=ħ...	I$_1$	n*"Try..."	ħ(r(I$_1$))	max(11,
	and Tries<=10				Tries)
if2	not('A'<=ħ...)				
	not('a'<=ħ...)		n*"Try..."&		
			"Try Again"		

The method of trace tables is readily generalized to conditional and concurrent forms in relations rather than functions. In place of unique behavior from row to row, the behavior may become a range of possibilities. For example, consider the sequence of specification parts in the trace table shown next.

Trace Table 6.10

statement	x	y	z
x1, y2, z3 <- y2 or z3, z3, y2;	$y2_0$ or $z3_0$	$z3_0$	$y2_0$
x1, y2 <- y2 or z3, x1;	$z3_0$ or $y2_0$	$y2_0$ or $z3_0$	
y2, z3 <- x1, y2;		$z3_0$ or $y2_0$	$y2_0$ or $z3_0$

which leads to the final concurrent form

```
x1, y2, z3 <- z3 or y2, z3 or y2, y2 or z3;
```

which defines a relation, not a function.

6.5.3 Verifying If Statements

While **if** statements are directly described in conditional forms that connect their **then** parts and **else** parts, their specifications may not make use of the same conditionals as in the **if** statements. In a simple case, the specification assignment

```
x1 := max(y2, z3);
```

has no visible condition whereas a design such as

```
if y2 > z3
then
  x1 := y2;
else
  x1 := z3;
end if;
```

defines a conditional assignment

```
(y2 > z3 -> x1 <- y2; y2 <= z3 -> x1 <- z3;)
```

which must still be shown to be equivalent to the specification.

For example, the if2 statement of Search_Program analyzed above is

```
-- begin if2
if 'A' <= Choice and Choice <= 'Z'
then
  TEXT_IO.Put (Item => "Welcome Aboard " & Choice);
else
  -- begin if1
  if 'a' <= Choice and Choice <= 'z'
  then
    TEXT_IO.Put (Item => "Lower Case Data Input " & Choice);
  else
    TEXT_IO.Put (Item => "Try Again");
  end if;
  -- end if1
end if;
-- end if2
```

and was verified informally above. This verification of the sequence can be carried out more systematically in conditional trace tables, as shown next.

Trace Table 6.11

```
if 'A' <= Choice <= 'Z'
    Output$_2$ = "Welcome Aboard " & Choice
```

statements	Condition	Input	Output	Choice	Tries
if2	'A'<=...		"Welcome..."		

where

```
'A'<=... = 'A' <= Choice <= 'Z'
"Welcome..." = "Welcome Aboard " & Choice
```

Trace Table 6.12

```
if 'a' <= Choice <= 'z'
    Output$_2$ = "Lower Case Data Input " & Choice
```

statements	Condition	Input	Output	Choice	Tries
if2	not('A'<=...)				
if1	'a'<=...		"Lower..."		

where, in addition to those above,

```
'a'<=... = 'a' <= Choice <= 'z'
"Lower..." = "Lower Case Data Input " & Choice
```

Trace Table 6.13

```
if not ('A' <= ħ(reverse (Input$_1$)) <= 'Z') and
   not ('a' <= ħ(reverse (Input$_1$)) <= 'z')
    Output$_2$ = "Try Again"
```

statements	Condition	Input	Output	Choice	Tries
if2	not('A'<=...)				
if1	not('a'<=...)		"Try Again"		

These three trace tables confirm the results obtained more informally above. The three conditions on `Choice` and the outcomes of each trace table provide a conditional concurrent assignment for `[if2]` given above, in alternate form, namely

```
[if2] =
  ('A' <= Choice <= 'Z' ->
      Output <- "Welcome Aboard " & Choice;
   'a' <= Choice <= 'z' ->
      Output <- "Lower Case Data Input " & Choice;
   not ('A' <= Choice <= 'Z') and
     not ('a' <= Choice <= 'z') ->
      Output <- "Try Again";)
```

6.5.4 Verifying While Loops

As already noted in determining program and program statement behavior, **loop** statements are more complex than sequence or **if** statements, and require more analysis for verification. If by some direct way the **loop** statement behavior can be determined directly, then that behavior can be compared with the specification relation for correctness.

For example, the initialized **while loop** statement whl in Search_Program is

```
-- begin seq
  Choice: CHARACTER := ' ';
  Tries: INTEGER := 1;
begin
  -- begin whl
  Search_For_Upper_Case_Letter:
  while not ('A' <= Choice and Choice <= 'Z') and
        Tries <= 10
  loop
    TEXT_IO.Put (Item => "Try input. ");
    TEXT_IO.Get (Item => Choice);
    Tries := Tries + 1;
  end loop Search_For_Upper_Case_Letter;
  -- end whl
```

and it was found that part behavior was as follows

$$\texttt{[whl]} = \{\langle\langle \texttt{Input}_0,\ \texttt{Output}_0,\ \texttt{Choice},\ \texttt{Tries}\rangle,\ \langle \texttt{Input}_1,\ \texttt{Output}_1,\ \texttt{Choice}_1,\ \texttt{Tries}_1\rangle\rangle\}$$

where

Input$_1$ will be a string of 1 to 10 characters, only the last, if any, an upper case letter.

Output$_1$ will be a string of messages "Try input. " of same number as Input$_1$ components.

Choice$_1$ will be last value of the Input$_1$ string, *h* (reverse (Input$_1$)).

Tries$_1$ will be an integer between 2 and 11, one more than the length of Input$_1$.

As noted above, this behavior can be determined directly from whl by direct analysis. Choice and Tries have been initialized when declared, and the while statement is the first statement executed. The first entry into the **while loop** will be successful in passing the while test. A message "Try input. " is sent to Output$_1$, then a value for Choice$_1$ is returned to Input$_1$, and Tries$_1$ is incremented to 2. If Choice$_1$ is now an upper case letter, the loop is terminated next with just those values for the data. If Choice$_1$ is not an upper case letter, the loop is continued with new values added to the Output$_x$ and Input$_x$ strings, and new values created for Choice$_1$ and Tries$_1$. The loop continues until either Choice$_1$ contains an upper case letter or Tries$_1$ exceeds 10. Since Tries is incremented each time through the loop, termination is guaranteed, but the appearance of an upper case letter in Choice can terminate the loop before that. Within the loop we will identify current values for Input and Output with separate numbers called x, and the next value supplied Input$_x$ called nextx.

Trace Table 6.14

```
if 0 <= x <= 10
  and length (Input$_x$) <= 10
  and not 'A' <= ℎ(reverse (Input$_x$)) <= 'Z'
    Inputx+1 = Input$_x$ & next$_{x+1}$
    Choice$_1$ = ℎ(reverse (Input$_x$)),
    Tries$_1$ = length (Input$_x$) + 1,
    Outputx+1 = Output$_x$ & "Try input. "
```

statements	Condition	Input	Output	Choice	Tries
Choice := ' ';				' '	
Tries := 1;					1
whl	not 'A'<=ℎ...	I$_1$=<)	"Try..."	ℎ(r(I$_1$...))	2
	and Tries<=10	&next$_1$			
whl	not 'A'<=ℎ...	I$_2$=I$_1$	2*"Try..."	ℎ(r(I$_2$...))	3
	and Tries<=10	&next$_2$			
	...				
whl	not 'A'<=ℎ...	I$_n$=I$_{n-1}$	n*"Try..."	ℎ(r(I$_n$...))	length
	and Tries=10	&next$_n$			(In)+1

where

```
'A'<=ℎ... = 'A' <= ℎ(reverse (Input$_1$)) <= 'Z'
In = Inputn = n characters, only last is upper case letter
'Try..." = "Try input. "
ℎ(r(I$_x$...)) = ℎ(reverse (Input$_x$))
```

In this case the execution can terminate at each whl if the condition is not satisfied. It is certain to terminate on the final whl with either an upper case letter or a count of 10 Tries.

This analysis for the initialized while statement is simplified considerably because of where it appears as the opening statement in the program. If not initialized as found here, the while statement must be prepared for any possible values for Choice and Tries, and the analysis is much more complex, as seen next in two trace tables, one for immediate termination

Trace Table 6.15

```
if length (Input) > 10
  or 'A' <= Choice <= 'Z'
    Input$_1$ = Input$_1$,
    Choice$_1$ = Choice$_1$,
    Tries$_1$ = Tries,
    Output$_1$ = Output$_1$
```

statements	Condition	Input	Output	Choice	Tries
whl	'A'<=ℎ...				
	or Tries<=10				

or, the other for later termination

Trace Table 6.16

```
if  x <= 10
  and length (Input$_1$) <= 10
  and not 'A' <= ℏ(reverse (Input$_1$)) <= 'Z'
    Input$_2$ = Input$_1$ & next$_1$
    Choice$_2$ = ℏ(reverse (Input$_1$)),
    Tries$_2$ = Tries$_1$ + 1,
    Output$_2$ = Output$_1$ & "Try input. "
```

statements	Condition	Input	Output	Choice	Tries
wh1	not 'A'<=ℏ...	I_1=I0	"Try..."	ℏ(r(I_1...))	2
	and Tries<=10	&next$_1$			
wh1	not 'A'<=ℏ...	I_2=I_1	2*"Try..."	ℏ(r(I_2...))	3
	and Tries<=10	&next$_2$			
	...				
wh1	not 'A'<=ℏ...	In=I_{n-1}	n*"Try..."	ℏ(r(In...))	length
	and Tries=10	&next$_n$			(In)+1

6.5.5 Exercises

1. Create trace tables for `Input_Digit` defined above in Exercise 5, Section 6.3.5.

2. Create subprogram behavior and trace tables for the statement part of `Input_Digit`, which might appear any place in a program, not just at the beginning and initialized as above. It is

```
Search_For_Digit:
while not ('0' <= Choice and Choice <= '9') and
      Tries <= 5
loop
  TEXT_IO.Put (Item => "Next digit? ");
  TEXT_IO.Get (Item => Choice);
  Tries := Tries + 1;
end loop Search_For_Digit;
if '0' <= Choice and Choice <= '9'
then
  TEXT_IO.Put (Item => "Digit is " & Choice);
else
  TEXT_IO.Put (Item => "Why no digit?");
end if;
```

3. If the while loop and if statement of `Input_Digit` are reversed as follows

```
if '0' <= Choice and Choice <= '9'
then
  TEXT_IO.Put (Item => "Digit is " & Choice);
else
  TEXT_IO.Put (Item => "Why no digit? ");
end if;
Search_For_Digit:
while not ('0' <= Choice and Choice <= '9') and
      Tries <= 5
loop
  TEXT_IO.Put (Item => "Next digit? ");
  TEXT_IO.Get (Item => Choice);
  Tries := Tries + 1;
end loop Search_For_Digit;
```

what does the sequence do as behavior? Could it ever be useful?

Chapter 7

Software Design and Certification

As already noted, Ada programs are made up of declarations and statements. Both declarations and statements require the best of thinking. Inappropriate declarations may be good enough to allow a solution to be defined in the statements. But the statement part of the solution may be unnecessarily complex, require more time and/or space for computation than is really required. Even with well conceived declarations, the statements may be poorly designed for computation in time and/or space requirements, or so poorly explained and documented that the entire solution is overly complex and difficult to understand.

In this Chapter 7, we begin the process of program design, dealing with declarations and statements of a single subprogram or package. In Chapters 8 and 9, we will expand the process of program design to how subprograms and packages can be put together into program solutions for complex problems. The fact is, of course, that subprograms and packages are put together by nothing more than declarations and statements, either poorly or well for meeting functional and performance requirements.

The second main topic of this Chapter 7 is program testing and certification. If program tests are invented by programmers, the tests are ad hoc executions from which no scientific conclusions can be made. The program tests may be well intended, but still allow no general conclusions. And self invented tests may skip unlikely events that can still happen and need to be tested. Instead, if program tests are generated by statistical methods that are based on the expected usage of the program to meet all of specifications, sound scientific conclusions of the correctness program behavior can be **certified**.

Programs are correct when they meet their specifications, shown by both verification and certification. Another question is whether the specifications are right. Deciding what programs should be is a deep and broad question which needs be addressed always. Defining good specifications needs effective analyses based on understandings of user needs as well as computer capabilities.

A critical task is to certify that the program as developed does indeed satisfy the specification correctly. It must be shown that representative inputs are executed with correct outputs in all tests. If failures arise, the program must be returned to the developers for fixing and retested by the certifiers until no more failures are found or the program is rejected if failures continue to appear.

7.1 Designing Programs to Meet Specifications

Using effective program design principles is at the center of software engineering. We will address four topics in these principles. First, we consider the good use of specification principles to provide the direction required for the program design. Second, we consider the good use of trace tables to confirm the correctness of the design in meeting its specifications as it unfolds. Third, we recognize the need for finding good algorithms for software which not only meet functional and performance requirements, but also verification and documentation needs as well. Fourth, we recognize the need for organizing software designs into manageable parts for intellectual and engineering control during its development and subsequently during its use.

7.1.1 Good Use of Specification Principles

As pointed out in Chapter 3, we can write software with good intentions and then wonder if they are right, but it is better engineering practice to design software from the start to meet well defined specifications. This means spending enough time up front to get and understand the specifications that need to be satisfied, then subjecting the design, as it unfolds, to sufficient analysis to ensure that it will indeed satisfy the specifications.

Whatever the representation, any specification is a mathematical relation, possibly a function, between inputs and outputs. A critical first step in design is to ensure that the specification is truly a relation, and that its representation is defined is as effectively as possible. When users first define specifications they may not be complete, describe what is required in common situations, but neglect uncommon situations. Specifications should describe satisfactory behavior in all possible situations. There may be cases in which a specification is completed during the design process, but that situation should be understood and the incomplete specification parts should be recognized and no design pursued for those parts until the specification parts are completed.

In defining a specification it is important to distinguish between the specification and its good use. The specification itself needs to describe exactly what the programmed computer is to output for each input, no more, no less. For example, with an interactive system, each input during an interactive session will describe the next response as an output, which will be a very small step in the entire session. But the good use of the interactive system may involve critical decisions on what inputs to enter in each step, and in what order. That is another topic which should be treated separately from the specification itself. It is an important topic but not part of the specification itself. The specification only describes the outputs for whatever inputs are provided.

Once software is specified, the simplest solution is to discover existing software that already meets the specification, not only meets the functional requirements, but also meets the performance requirements. It goes without saying that the software should be correct, without unexpected failures in use. Software may already exist to meet a functional specification, but the performance is inadequate, either in time or space requirements. In such a case there may be lessons of interest in such software, but the software itself is insufficient. It may be perfectly good software for some other situation, but too slow or too big for the current need. There will be many implementation circumstances for software that goes beyond its specification. What kind of computer is being used? How much storage is available? How much time is available? What programming language is available? The software needs for a small space computer may be very different than for a large general purpose computer.

If no solution to a specification exists in existing programs, there may be many parts to a solution in existing programs. Again, the program parts to be used must be satisfactory in both function and performance. When program parts can be used they may be imported just as they are to carry out certain subfunctions required, or they may be used as models from which modifications are needed to meet functional and performance requirements. Existing program parts may exist in a different programming language than is available. Of course, it goes without saying that the software must be correct.

Another creative solution may be to recognize that software already exists that does not exactly meet the specification, but can be used to solve the underlying problem addressed by the specification. Specifications should be created and defined to meet problems that can be solved with computer programs. It may be wise to alter specifications during the design process, either to make use of existing software or to make software more feasible to create. For example, specifications may define responses that are more accurate than are really required. Less accurate responses may permit entirely different declarations and computations and much simpler software. In illustration, numerical solutions to two decimal places may be sufficient, but a program exists to find solutions to ten decimal places that takes the ratio five squared times as long, and is too expensive.

Computers do elementary operations remarkably fast, compared to humans. For example in simple arithmetic or character operations, computers typically operate at microsecond levels, humans at second levels. But more intelligent operations that require computer software may take more and more time for computers—more time than might be expected at first thought. For example, finding pictures of critical objects in a two dimensional framework will be very difficult for computers, but very easy for humans. As surprising as it might seem, humans will do many intelligent operations faster than computers. Humans can approximate data processing steps, such as picture recognition, possibly with some level of errors, whereas computers can only do exactly as told, perhaps with large time or space requirements. So, as programs are designed to carry out more intelligent operations, it becomes critical to analyze the performance possible by computers, in both space and time. In various applications, either space or time may become critical. For example, in a spacecraft the amount of computer storage may be limited, so space is especially critical. In an automobile, the processor power may be limited, so time may be critical.

While computers do elementary operations remarkably fast, it is easy for programs to ask for more processing than is practical. For example, let's print out all the natural numbers described in a 32 bit word. Such a program would not be hard to write. In fact it shouldn't take more than a dozen lines or so of Ada. Declare a variable, say `NATURAL_NUMBER`, initialize it to zero, execute a loop to print and increment `NATURAL_NUMBER` in its binary form until it reaches 2**32 - 1. How long would an execution of this short and simple program take? Just long enough to print the natural numbers from 0 to 2**32 - 1, a little more than a billion numbers. In illustration, suppose it takes a microsecond to print one number, then it will take more than a million seconds, which is

```
    1,000,000/(60*60*24) days = 11.57... days.
```

That's not very practical. A printer ten times as fast will still take over a day. And who will ever look at all those numbers, anyway? That is, a short and simple program can take extremely long for execution and produce more data than is useful for people, anyway.

So not all programs, even simple ones, are practical. In addition, many problems that have programs that are practical also have other programs that are not practical. Programs that solve the same problem may take very different amounts of space or time in execution. For example, given a file of a thousand data items to sort into ascending order, a bubble sort will require a thousand squared comparisons to carry out, while a merge file will require a thousand times the logarithm of a thousand comparisons, a total of some ten thousand comparisons. If comparisons take a microsecond, the bubblesort will take a million seconds, which is 11.57... days, whereas the merge sort will take only some ten seconds. So the bubblesort might be judged impractical and the mergesort practical for the same problem.

7.1.2 Good Use of Trace Tables

Trace tables are straightforward to set up, except for iteration, and even then are not so difficult with some analysis of the program parts. It is primarily taking the time and effort to lay out the computation in general form. The alternative of guessing at the computation may seem to save some time and effort at the outset, but risks much more time and effort to get the program parts debugged later on.

Sequence parts, including declarations as well as statements, go directly into trace tables, with a column for each variable, a row for each declaration or statement. Such variables only come into being by declarations, and disappear when the range of the declaration is exceeded. So the trace table columns have limitations in the range of rows in which each variable is defined. For many analyses within executable statements, all variables exist in all rows, but that is not necessary. For an input file, two columns for subfiles, of items 1) already read and 2) yet to be read, need to be defined.

The values of the variables in the trace table are defined by function or relation composition defined by the declarations and statements going down the trace table. The trace table may be of large size for large parts, so it may need considerable paper. Even better, anything possible on paper can be organized in word processing form in computers for serious work that is to be documented and stored.

Note that **for loop** statements are actually sequence parts with a convenient notation. Such **for loop** statements may in fact define very long sequences, though very regular, and additional specific analyses of the semantics of the **for loops** may be in order. While the iteration by iteration analysis may be in order for short **for loops**, an additional generic analysis may be in order for longer **for loops**. The rationale behind the **for loop** should provide such analysis or the **for loop** may be a piece of guesswork and likely wrong.

Alternative parts, such as **if** statements and **case** statements, require a separate part and its trace table for each alternative, with the condition (**if, elsif, else,** or **case**) literally a prefix for the part in each separate table. That is, the **then** part of **if** statement `<if p then g else h end if;>` is not simply `<g>`, but is `<p -> g>` which is undefined if p is not TRUE. The trace table has an additional column with the conditions that must hold as reached in execution. Each separate part and trace table defines a behavior (function or relation) with a specific domain, and the behavior of the entire alternative part is the set union of these separate parts.

A **while** or **exit loop** statement defines a trace table, but a possible exit for each possible computation sequence of its internal statements. For example, with the **while loop**, there will be sequences of 0, 1, 2, ... repetitions of the internal statements, and a possible exit preceding each sequence. The condition column will require the **while** (or **exit**) condition to be TRUE (or FALSE) to continue, FALSE (TRUE) to terminate. Each separate set of internal statements of the trace table defines a behavior (function or relation) with a specific domain, and the behavior of the entire **loop** statement is the set union of the behaviors of these separate parts.

Such a set of trace tables for a **loop** statement can be embedded in a single generalized trace table with the possibility of an **exit** at each **exit** condition (**while** or **exit** condition). All trace tables for sequence or alternation parts discussed above execute through every row, top to bottom. This **loop** trace table permits exits at intermediate rows, depending on the value of the **exit** condition with the current variable values. The initial data defines which (if any) exit will terminate the execution of the loop statement. In this case the initial data partitions the domain of the part behavior by the exits taken (which condition after how many iterations). Each exit from this single trace table is reached by a sequence of the internal statements and the value of each condition in the **loop** statement to reach this exit. Again as in the case of the **for loop** statements, general **loop** statements may require generic analyses to deal with longer **loops**.

Internal declaration statements expand trace tables in the variable columns direction. Each new call of a subprogram with declarations will create a new set of variables (parameters), hiding the previous variables. That adds this set of new realizations of the variables in new columns across the trace table, just as for any declaration of variables. As each such call is completed, its set of added variables disappears below that point in the trace table, just as when declarations terminate. As for iteration, internal declarations may call for generic analysis to carry out the necessary reasoning for the operations.

Although trace tables may seem directed to software engineers, and are, other people can use them as well. Although program users may not know or care about the internal form of programs, they may well be interested in how best to use programs, particularly interactive programs, so the external use of interactive programs can be described in trace tables, as well. People using interactive programs define a people computer system that can be described and studied in trace tables. People exercise systems in sequential, alternative, iteration forms and instructions for people look like programs under different names.

7.1.3 Finding Good Algorithms for Software

As noted for sort programs, there will typically be many ways to carry out sort software design. And some or many of these ways may be better than others, some impractical. In another illustration, before computers, tables of trigonometric values were used extensively by humans to support geometric analysis and manual computation. While these tables can be stored in computers and entries looked up as needed, that method of use is seldom the best way in the computer. Instead, methods for computing individual entries can be programmed, and such entries calculated as needed rather than looked up. The methods for calculating trigonometric values from scratch go fast in computers, slow for humans, while table look ups are much faster for humans, but take more space for computers. The same principles apply to other tables for hand computation, such as logarithms, powers and roots, interest rates, *etc.* These values will also be calculated as needed rather than stored up ahead of time.

As mentioned before, arithmetic in place notation and algorithms for long division were not known to ancient Romans. We hardly think of these as algorithms, but they are and are so fundamental in today's business, scientific and engineering world. But computers now bring a new dimension to arithmetic because of their finiteness. Strictly speaking, computers can not do general arithmetic, but only arithmetic in limited domains. For example, two INTEGERs selected at random from the finite set of a given computer, can be added or subtracted only half the time, the other half leading to an overflow. Two INTEGERs, say `X1` and `X2` selected randomly from the range 1 .. 1_000_000 will have a sum `X1 + X2` that exceeds upper bound 1_000_000 half the time. That may be a surprise at first glance, but true.

The same two INTEGERs can only be multiplied correctly much less often, based on the actual bounds. For example, suppose `X1` and `X2` are selected randomly from the range 1 .. 1_000_000. If `X1` = 1, `X1 * X2` = `X2`, alright always, but `X1` = 1 only one time in a million. If `X1` = 2, `X1 * X2` is alright only if `X2` <= 500_000, half the time for `X2`. If `X1` = 10, `X1 * X2` is alright only if `X2` <= 100_000, a tenth of the time for `X2`. These values for `X1` are possible in a random distribution but unlikely—one in a million for each case. More likely, if `X1` <= 100, `X2` must be 10_000 or below, so `X1` <= 100 only one time in 10_000 and `X2` <= 10_000 only one time in 100. Even more likely, if `X1` <= 1_000, `X2` must be 1_000 or below, so `X1` <= 1_000 only one time in 1_000 and `X2` <= 1_000 only one time in 1_000.

So even arithmetic which seems so simple and straight forward with pencil and paper, with unlimited memory, is not so simple and straight forward in computers with finite memory. As a result, declarations of INTEGER data with arithmetic operations, especially multiplication, requires much more range than might appear offhand.

For example, if X1 and X2 are both always in the range 1 .. 1_000 but are declared in the range 1 .. 1_000_000, additions and multiplications will always go through. That condition itself can be explicitly stated and checked during arithmetic. That will take longer, but it may be worth it if the alternative is worse. For example, the simple multiplication

```
X3 := X1 * X2;
```

might be rewritten as

```
if (X1 in 1 .. 1_000) and (X2 in 1 .. 1_000)
then
  X3 := X1 * X2;
else
  -- recovery goes here
end if;
```

7.1.4 Organizing Software Designs Into Manageable Parts

In dealing with software designs, one must proceed with both top down and bottom up considerations simultaneously. Only top down, with no bottom up, considerations to start with can lead to dead ends at the bottom, unintentionally pushing unsolvable problems deeper and deeper until the design effort founders for lack of solutions. Only bottom up, with no top down, considerations can lead to no solution at all. Even though the bottom parts all work just fine, they do not work together for one reason or another.

It may be hard to believe that large groups of good programmers, say fifty or a hundred, have worked several years and never got a software project completed. The first IBM PL/I optimizing compiler was worked on in the mid sixties by over fifty good programmers for over two years, and was finally abandoned without producing any compiler at all. Lots of good code was written, but it did not work together as a compiler. There are also many examples of pure top down efforts that failed because unsolvable problems were pushed deeper and deeper without recognizing or solving them. So both top down and bottom up efforts are needed to manage and complete software projects.

On the other hand, major software projects under good intellectual control can come together very well. For example, the software for the NASA Space Shuttle was created in the '80s by IBM with many deliveries, and never missed a schedule or budget. Space shuttle software has been recognized by the U. S. government as some of the very best created. On the first attempted shuttle flight, a failure was found in synchronizing five computers. But since then, with intelligent use by the astronauts, no failures occurred in flight. It is possible to build large complex software under intellectual and effective control. But it takes good engineering discipline and technical management to do so.

One of the easiest ways to lose control in a software project is to begin design without a specification. It may seem simple enough to just start writing the code when it seems clear what the software should do, especially in the details. "Why bother with specifications? Isn't that just a waste of time? Who needs them anyway?" Computers and software are so young as human activities, it has taken some time to understand the need for mathematical rigor and engineering discipline in software development. As noted before, when civil engineering was this age, the right triangle was not known. When accounting was this age, the double entry bookkeeping idea was not known. Specifications are indeed needed in real software projects. The specifications may not be completely defined. But they should be sufficient to define what software will be created first to the specifications known. In a real sense, as software needs change, their specifications will change as well. So changing specifications may be required, followed by changing software to meet the new specifications.

Another way to lose control in a software project is to design and write code for execution without serious verification. Again, it may seem simple to just write the code and discover by testing what little mistakes may be left behind. But that has been another false premise of a new human activity. The problem is not in finding the little mistakes—it is in the new and deeper mistakes entered while debugging the little mistakes found. Even with the best of intentions, fifteen per cent or more of these debugging fixes produce deeper failures. At first hearing, fifteen per cent seems very high, but that is really the case. The perspective in debugging is narrower than in original design, and the local problem may well be fixed, but a new problem on a broader basis may be introduced unintentionally.

For example, the software supporting IBM's main line of computers has been built and maintained with extensive debugging and fixing using traditional methods. It has been studied extensively for its failures and fixes. They have been fixed many times with an estimated fifteen per cent of the fixes adding new failures. At this point it is believed that practically all the failures are due to previous fixes and not the original code. This study has changed the way management permits fixes to be made. A failure reported only once will not be fixed for fear of creating another more potent set of failures. The failure rates found run from under two years to three thousand years. No one imagined the failure rates would vary so much. Fixing a three thousand year rate failure may create a new failure with rate under two years, a very bad case to avoid. So failures must appear frequently enough to justify the danger of a failure with higher rate from a fix.

This fundamental difficulty in heuristic programming and debugging in creating failure free software is only recently recognized. The new concept of functional verification is not an alternative to debugging. It is a new human capability only beginning to be understood. People do make mistakes in verification. But the failures of well verified programs are of a very different kind than failures of heuristically invented programs. When well verified programs are tested, a few failures of very simple kinds may be found and fixed right away, and no more failures introduced. For example, the 1980 U. S. census used a program created by Paul Friday of about 25,000 lines of Pascal. He used functional verification. After coding the program, and testing and fixing failures due to mistakes in functional verification, the program ran the entire census operation with no failures ever found during its use. Mr. Friday received the highest award of the Commerce Department for the programming achievement of zero defect software.

Functional verification allows software engineers to review each other's programs in systematic ways, without insulting their personal pride. A long term precedent is in arithmetic, using place notation and long division. Long division permits perfect arithmetic, but humans can still make mistakes in local steps, for example multiplying 7 times 3 and getting 28. That mistake may be found by the divider by multiplying out the answers to check against the original problem. But even then it might not be found. For important divisions, colleagues can check long divisions step by step and find such mistakes as well. There is no harm to human pride, only thanks for catching such mistakes. In the same way, functional verifications can be checked by fellow software engineers. For example, the IBM software for advanced typewriters with some 65,000 lines of code for three microcomputers created by a dozen programmers was functional verified before going into test. It has been used by millions of people with no failures ever found. So zero defect software is really possible if it is important for its use.

7.1.5 Exercises

1. Given the need for an interactive program for spell checking English text as part of a word processing system, what might its specification be? Distinguish between data stored and obtained from the system and data received from and returned to the user.

2. Given a two dimensional pattern of blanks and dots, 1_000 by 1_000, and the need to recognize all lines, horizontal, vertical, diagonal of five dots or more, assume dots appear independently of each other one time in ten at each spot and estimate how many lines might appear. What effort would be required to find these lines by automatic computation? If the pattern was increased in size to 1_000_000 by 1_000_000, how would the required effort be increased?

3. Can trace tables be used in checking specifications for their correctness in meeting external requirements?

4. Can trace tables be used in user documentation of specifications of interactive systems? Can sequence, alternation, and iteration be expanded to the usage of the system?

5. It was noted above that trigonometric tables were not needed in computers, but that individual entries could be calculated as needed. What is different about spelling tables? Can correct spelling of words be calculated as needed or are words necessarily stored in tables for reference?

6. The multiplication alternative described above

   ```
   if (X1 in 1 .. 1_000) and (X2 in 1 .. 1_000)
   then
     X3 := X1 * X2;
   else
     -- recovery goes here
   end if;
   ```

   can be improved on, by somehow requiring only that the product X1 * X2 be no more than 1_000_000, not that each X1 and X2 be no more than 1_000. How might that be done?

7. Given a program or program part that "does the right thing," but has no explicit specification, how might a specification be defined?

8. Arithmetic with place notation, long addition, long subtraction, long multiplication, long division, square root, *etc.* was important a hundred years ago before computers or preceding hand calculators. How might such operations be defined to maximize human capabilities for both efficiency and correctness? How might possible human mistakes enter into such definitions?

7.2 Designing Data Types, Subtypes, and Objects

Creating good data types, subtypes and objects is central to good software design. As we have already seen, Ada has a rich set of data types, subtypes and objects for solving software problems. We have also seen that objects can be initialized at declaration. Data can be designed in scalar and array types. Scalar data can be of enumerated and INTEGER types, enumerated types including CHARACTER and BOOLEAN as special cases. Array types allow lists of identical objects, including the STRING type with CHARACTER data, and constrained arrays and unconstrained arrays with enumerated data and INTEGERs. The ranges of constrained arrays are defined at compile time, ranges of unconstrained arrays are defined at execution time.

Good data types, subtypes, and objects come from the problem being solved, as defined by the specification. They certainly anticipate the uses of types, subtypes, and objects in statements. But they must be well defined first in order to create the statements. Often, the solution to a problem falls naturally from the choice of data representation. One part of data design is the types selected themselves. Scalar types may be CHARACTER or BOOLEAN, which are small types that may well fit an occasion. But enumerated types (of identifiers and/or character literals) and INTEGER types have more power in storing data, and INTEGER types have more operations available for their objects. Similarly, array types have different powers. STRING types are one dimensional unconstrained arrays that may only have CHARACTER elements indexed by POSITIVE INTEGERs. Constrained arrays may have any elements (all the same type) in one or more dimensions, each indexed by any scalar type. General unconstrained arrays may also have any elements in one or more dimensions indexed by any scalar types.

As already noted, Ada is a strong programming language in data definition with two levels of description. The first level is in data types and subtypes, to describe classes of data that may be used in subprogram, package, and block designs. The next level is in the data objects themselves to be used from these data types and subtypes. It takes as much thinking to design the data as to use it in good program design. Good data can make programs much easier to read and verify, and thereby much easier to design the programs. Data types and subtypes provide the first level of design in meeting data requirements and opportunities. What kind of data and how it is to be used needs good analysis and anticipation of the specifications and statements that will be needed. The data types and subtypes anticipate the data objects needed at the second level. The data objects also depend on strategies for meeting specifications with statements.

For example, a subtype of CHARACTER, say `CAPITAL_LETTER`, and an object `This_Letter` might be declared, then `This_Letter` initialized at the start of execution as

```
-- ...
subtype CAPITAL_LETTER is CHARACTER range 'A' .. 'Z';
This_Letter : CAPITAL_LETTER;
-- ...
begin
  This_Letter := 'A';
  -- ...
```

but it is better to initialize `This_Letter` at declaration, such as

```
-- ...
subtype CAPITAL_LETTER is CHARACTER range 'A' .. 'Z';
This_Letter : CAPITAL_LETTER := 'A';
-- ...
```

There are some advantages to providing the initial value at the point of the declaration as shown in this second example, but there are also disadvantages. Perhaps the most glaring is that an exception raised as a result of this initialization cannot be handled by this block and would instead be raised at the point of the call to this block. The details of the debate are beyond the scope of this textbook, however, suffice tit to say that for now, it is recommended that you always initialize objects when you declare them.

If the problem being solved makes use of both capital letters and small letters, it may be useful to start with SMALL_LETTER as another subtype as well, such as

```
-- ...
subtype CAPITAL_LETTER is CHARACTER range 'A' .. 'Z';
subtype SMALL_LETTER   is CHARACTER range 'a' .. 'z';
This_Letter : CAPITAL_LETTER := 'A';
That_Letter : SMALL_LETTER   := 'a';
-- ...
```

noting that CAPITAL_LETTER and SMALL_LETTER are distinct subtypes with no common characters and are not adjacent in CHARACTER.

Still further, a declaration hierarchy may be useful, defining a subtype LETTER within which to fit CAPITAL_LETTER and SMALL_LETTER, as follows,

```
-- ...
subtype LETTER is CHARACTER;
subtype CAPITAL_LETTER is LETTER range 'A' .. 'Z';
subtype SMALL_LETTER   is LETTER range 'a' .. 'z';
This_Letter : CAPITAL_LETTER := 'A';
That_Letter : SMALL_LETTER   := 'a';
-- ...
```

so that both 'a' and 'A' are of base type LETTER, for example. Note that comparisons are possible between This_Letter and That_Letter, but assignments are not. For example, an if statement beginning with

```
if This_Letter < That_Letter
then
  ...
else
  ...
end if;
```

is legitimate (always with the same result), but an assignment

```
This_Letter := That_Letter;
```

will always fail.

This is true since these are both subtypes of the base type CHARACTER, making them compatible types. When types are compatible, it means that operations are permitted on them such as comparison, the test for equality, *etc.* However, since the ranges of these subtypes are disjoint, *i.e.,* they do not overlap, objects of these two types may never be assigned to each other. There would not be a problem in the type matching, since they are subtypes of the same base type, but since the ranges are different CONSTRAINT_ERROR would be raised at every attempted assignment.

The Ada data types and subtypes that we have seen so far can be divided into three general classes. The first class is scalar data types and subtypes, divided further into character data and INTEGER data. Character data are rich in formats, with CHARACTER and BOOLEAN special classes. INTEGER data are rich in operations of arithmetic. The second class is the general one dimensional arrays, including the special STRING type and two kinds of one dimensional arrays, one with dimensions defined at compile time, the other defined at execution time. The third class is the multidimensional arrays of the two kinds. These three classes are treated next. The creation of data objects in these classes permit effective engineering design of data for programs.

7.2.1 Designing Scalar Data Types, Subtypes and Objects

Unlike earlier programming languages with more limited definitions of data, Ada provides a wide variety of data definitions, which can be designed to meet problems being addressed. As noted, in scalar data types and subtypes, the major distinction is between character data and INTEGER data. INTEGER data permit the operations of arithmetic when that is necessary. But character data can be tied directly to the meaning of nonarithmetic needs and made very readable. As already noted, INTEGER data is only capable of arithmetic in a limited range. If arithmetic operations are attempted that cause overflow, the operations are invalid, so it is up to the software engineer to be sure that such overflows do not take place.

All scalar data types and subtypes define data with an order. Such order is to be expected for INTEGER data, but it holds as well for all character data as well. CHARACTER and BOOLEAN data has order by definition, as described in Table 2.5 in Chapter 2. They are not necessarily what you might think at first glance, but part of the Ada definition using ASCII. For example, it may seem clear that 'A' < 'B', but is 'A' < 'a' or 'A' > 'a', or even 'A' = 'a'? Is '0' < 'A' or '0' > 'A'? Is '&' < 'A' or '&' > 'A', etc.? These relations are defined by ASCII, so the ASCII definition must be used. For character data defined by type or subtype declarations, the order is exactly as declared. For example,

```
type Decimal_Digit is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
type Octal_Digit_1 is ('0', '1', '2', '3', '4', '5', '6', '7');
type Vowel         is ('A', 'E', 'I', 'O', 'U', 'Y');
type Octal_Chars   is ('7', '6', '5', '4', '3', '2', '1', '0');
```

In this case the contents of `Octal_Digit` are character literals that could be redeclared as a subtype as follows:

```
subtype Octal_Digit_2 is CHARACTER range '0' .. '7';
```

But neither of the other two types can be redeclared as subtypes of CHARACTER. Vowel has values in order from CHARACTER, but they are not in sequence in CHARACTER. `Octal_Chars` has values from CHARACTER, but they are in inverse order.

Character data can also be identifiers, defined in type and subtype declarations. For example

```
type Decimal_Term is (Zero, One, Two, Three, Four, Five, Six, Seven,
                      Eight, Nine);
subtype Octal_Term is Decimal_Term range Zero .. Seven;
```

are distinct types with the same logical content as `Decimal_Digit` and `Octal_Digit` above. But if a need arises to recognize their possible treatment as one place INTEGERs, that would be more difficult (but not impossible). Even though both convey numeric information, they are character literals/identifiers and could not be used in numeric computation without manipulation.

Notice again that data types and subtypes do not hold string literals. For example,

```
"0", "1", "2", "3", "4", "5", "6", "7",
```

are not character literals, but are string literals. They do not make up data types or subtypes as CHARACTERs and identifiers do.

The choice of character and INTEGER data for program design starts with the problem being solved. As noted, BOOLEAN types and subtypes are natural for dealing with logic in a problem. The BOOLEAN values FALSE and TRUE are identifiers with obvious meaning. To what extent a computation can depend on BOOLEAN values, they are worth identifying early. Likewise, CHARACTER types and subtypes provide another level of data definition that are worth identifying. But as problems become more complex and need more widespread understanding in both creating programs and using them, richer character type data can become very useful. Data types and subtypes can be invented with identifiers appropriate to the problem in Ada that permit real thought in making programs understandable in terms of the external needs.

When data needs arithmetic applied to it, it is clear that it should be INTEGER based. There are many uses of arithmetic, and some or all arithmetic operations become needed. The main choice in arithmetic is the range of data to be declared. For some uses, negative INTEGERs are not needed. For example, the number of coins of a specific kind in a candy machine will never be negative. For other uses some arithmetic operations are not needed. For example, calendar dates may be subtracted to determine time spans, but will never be added or multiplied. But a time span may be added to or subtracted from a calendar date to get a new calendar date. So the use of INTEGER types or subtypes may need all or only part of the arithmetic operations. Again, the good definition and declaration of the data will stem from the external problem being solved.

In illustration, calendar dates and time spans in day units BC and AD might be declared in terms of INTEGER types, then objects as follows.

```
-- ...
type CALENDAR_DATE is range -5_000_000 ... 2_000_000;
type TIME_SPAN     is range -7_000_000 ... 7_000_000;
This_Date : CALENDAR_DATE := 0;
This_Span : TIME_SPAN     := 0;
-- ...
```

In this case, the range of TIME_SPAN covers the possible spans of CALENDAR_DATE but the ranges are unique and distinct. These types could also be declared as subtypes to allow for compatibility between objects of these types, as follows.

```
-- ...
subtype CALENDAR_DATE_1 is INTEGER range -5_000_000 ... 2_000_000;
subtype TIME_SPAN_1     is INTEGER range -7_000_000 ... 7_000_000;
This_Date : CALENDAR_DATE_1 := 0;
This_Span : TIME_SPAN_1     := 0;
-- ...
```

With these declarations, the different meanings of CALENDAR_DATE_1 and TIME_SPAN_1 are clear even though they are each subtypes of INTEGER. Because of the ranges involved in these declarations, CALENDAR_DATE_1 could be declared as a subtype of TIME_SPAN_1, itself a subtype of INTEGER, but because of their different meanings that would seem wrong.

These initial values may not seem very realistic, and a real problem may suggest other initial values, but it does not hurt anything for objects to be declared with specific initial values. In particular, this reduces one kind of needless variability in execution and testing with possible different behavior with identical input data. If there is an unintentional failure from lack of initialization or incorrect initialization, it will be easier to find.

As already noted CALENDAR_DATE_1 and TIME_SPAN_1, while both INTEGER subtypes, will be used in specific ways. For example, one might expect an assignment such as

```
   This_Span := This_Date_1 - This_Date_2;
```

or

```
   This_Date_1 := This_Date_2 + This_Span;
```

or

```
   This_Date_1 := This_Date_2 - This_Span;
```

to make sense, but assignments such as

```
   This_Span := This_Date_1 + This_Date_2;
```

or

```
   This_Date_1 := This_Span - This_Date_2;
```

would not make sense from the intended meaning of the data, even though they are logically possible.

7.2.2 Designing One Dimensional Array Data Types, Subtypes and Objects

One dimensional array data types and subtypes offer an additional way to structure and access similar data within a single dimension. All the items of the array have the same name as an identifier, with an additional scalar object being used to identify which one of the array elements is being referenced. Some operations can be defined for the entire array, for example, the array can be initialized at declaration so that all elements begin with the same value. As noted in Chapter 5, the special case of type STRING is a one dimensional unconstrained array of CHARACTERs, defined by two declarations

```
   subtype POSITIVE in INTEGER range 1 .. INTEGER'LAST;

   type STRING is array (POSITIVE range <>) of CHARACTER;
```

From this basis, subtypes of STRING with finite ranges may be very useful. For example

```
   subtype Line      is STRING (1 .. 80);
   subtype Pass_Word is STRING (1 .. 8);
```

are subtypes of STRING with finite arguments. With these declarations of types and subtypes, specific STRING objects can be declared, such as

```
   Line_Contents      : Line;
   Last_Line_Contents : Line;
   My_Pass_Word       : Pass_Word;
```

More general one dimensional arrays permit broader forms of both the elements than CHARACTERs and the indices than substrings of POSITIVE. In particular, the indices may be defined as any enumerated or INTEGER forms. As already noted, enumerated forms have their data ordered, and may be useful and descriptive in describing how arrays are accessed. For example, the declarations

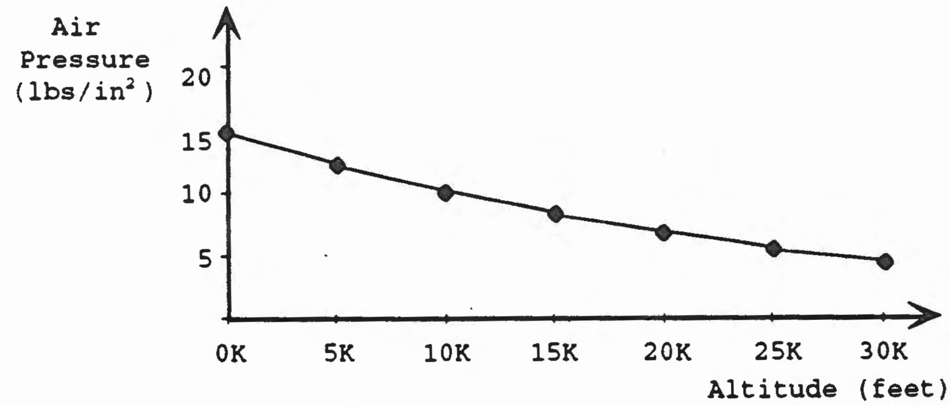
```
type Color_Type is (Red, Yellow, Blue, Green, Purple);
Color : Color_Type;
type Color_Frequency is array (Color_Type) of INTEGER;
type Color_Value     is array (Color_Type) of CHARACTER;
type Color_Presence  is array (Color_Type) of BOOLEAN;
```

deal with three distinct properties, one an INTEGER, one a CHARACTER, one a BOOLEAN in three arrays indexed by Color_Type. For example a sequence of assignments to these arrays might be as follows.

```
Color_Presence (Yellow) := FALSE;
Color_Frequency (Green) := 1_500;
Color_Presence (Blue)   := TRUE;
Color_Value (Purple)    := 'C';
Color_Presence (Yellow) := TRUE;
```

In some cases, arrays are obvious ways of describing data, for example when geometric pictures show data along a line and an array can hold the values along the line. Such insights from geometry may provide a unique relation between the array and the data. For example, the air pressure measured at various altitudes will define an array directly. But in other cases the relation between array and data may be more arbitrary. In the Color_Type example above, the colors themselves were listed in a certain sequence not so explainable as altitudes. In this case the colors were just listed without further explanation. There may be a reason, but it will usually be debatable and discussable in arriving at the sequence of array data. In the case of altitudes for air pressure data, the relation between successive altitudes is very clear and very useful beyond their appearance in the array. In fact, an array for air pressure could be listed in other sequences than increasing altitudes, but would simply make understanding and use more difficult. So the natural ordering of array data is very useful when it exists. But even when no natural ordering exists, array data can be useful as well, when the ordering used is documented.

One useful capability with arrays is to show data in two dimensional form, indices in one dimension, array values in the other. In the case of air pressure, the two dimensional form will be very regular, with air pressure reducing as altitude increases. In this case the two dimensional form shows a fundamental physical law, with air pressure reducing by half about every 18,000 feet of altitude.



Air Pressure versus Altitude
Figure 7.1

More precisely, air pressure versus altitude is given in the following table.

Altitude (Feet above Sea Level)	Air Pressure (Pounds/Square Inch)
0,000	14.7...
5,000	12.2...
10,000	10.1...
15,000	8.3...
20,000	6.8...
25,000	5.4...
30,000	4.4...

Table 7.1
Variation of Air Pressure with Altitude

With Color_Frequency as given above, the two dimensional form is not quite so regular as given, but Color_Frequency does relate to elements of Color_Type in a specific way. With more study, the type Color_Type might be redefined to make Color_Frequency more regular. In this case, a new and carefully defined Color_Type could be related to Color_Frequency just as regularly as air pressure relates to altitude. But in any case, a two dimensional form can be displayed to relate the indices with the array values which may be very useful in understanding the data. Curiously, when no physical law is known for the array data, its display may be very much more useful than a display for a known law.

7.2.3 Designing Multidimensional Array Data Types, Subtypes and Objects

Multidimensional array data types and subtypes offer even more ways to structure and access similar data over several dimensions. All the items of the array have the same name as an identifier, with several scalar objects being used to identify which one of the array elements is being referenced. Some operations can be defined for the entire array, for example, the array can be initialized at declaration so that all elements begin with the same value.

The various dimensions of an array can be entirely different to suit what is being defined. In three dimensional space, all dimensions may be the same type of INTEGER. For example,

```
   Distance : array (INTEGER, INTEGER, INTEGER) of INTEGER;
```

which uses an anonymous array type. A better way to introduce an explicit type name is as follows.

```
   type Distance_Type is array (INTEGER, INTEGER, INTEGER) of INTEGER;
   Distance : Distance_Type;
```

But in four dimensional space and time, the time dimension may be a different form of INTEGER than the space dimensions.

In an economic problem, various kinds of dimensions may be appropriate. For example, if Expense_Type refers to the expense of a picture frame, which depends on Color_Type, Shape_Type, and Size_Type, its declaration might look as follows.

```
   type Color_Type       is (Red, Yellow, Blue, Green, Purple);
   type Shape_Type       is (Square, Rectangular, Circular, Oval);
   type Size_Type        is range 10 .. 50;
   type Cost_Type        is range 100 .. 1_000;
   type Expense_Type     is array (Color_Type, Shape_Type, Size_Type) of
                           Cost_Type;
```

Then, the use of these types in declaring objects for computation could be something like

```
   Color          : Color_Type;
   Frame_Shape    : Shape_Type;
   Frame_Size     : Size_Type;
   Frame_Cost     : Cost_Type;
   Frame_Expense  : Expense_Type;
```

Multidimensional arrays are very powerful objects for storage and computation of closely related data. It is useful to identify how objects may depend on several indices in planning for multidimensional array design. As already noted, but worth reinforcing, multidimensional arrays are important capabilities for practical and large scale software design. Designing them well begins during and before the declaration of types, subtypes and objects, which precedes the design of executable statements.

Just as for one dimensional arrays, multidimensional arrays can be used when obvious in geometric situations but also in cases when data depends on indices in no geometric ways, but does depend on them. Defining array data needs study in the application, and thinking how data might be put together for the software development. Two dimensional arrays can be diagrammed in three dimensions, one for each indice, one for the array value, for better visualization. These diagrams can be imagined in more dimensions, depending on the types and meanings of the indices and array values.

7.2.4 Exercises

1. If objects are required to be initialized at declaration, but seem to have no reason to be, can you provide any arguments that they should be initialized for higher reasons of intellectual control in documentation?

2. Why are data types and subtypes an important part of software design in Ada, and what opportunities are there to simplify subsequent object declarations and statement definitions?

3. For scalar data types and subtypes which require arithmetic comparison but no other arithmetic operations, should they be declared INTEGER or not? Are there other considerations to be considered?

4. Given a three dimensional logic based on "false, maybe, true" how might its data be defined and what would its operations be?

5. Can a STRING be used to handle a set of digits accessed by bank accounts that might be negative?

6. In the example dealing with arrays Color_Frequency, Color_Frequency, Color_Presence, their declarations are implicit. How might explicit declarations be given?

7. How might a two dimensional space be declared as a subtype and two object declared in it both initialized to the origin?

8. How might a cost array be declared that deals with three forms of interactive activities in producing fruit juices, namely picking, canning, and delivering?

7.3 Hierarchical Design with Subprograms

Correct sequence, branching, looping steps are central to subprogram design. Every subprogram execution part is a sequence of one or more statements. Each statement may itself be a sequence, branching, or looping statement. And each of those may be expanded again into one of the three kinds of steps. Sometimes a given statement may be designed as any one of a sequence, a branching, or a looping statement, but usually, one of the three will be most natural for what is needed. However, within each kind, there will be further choices. For example, a sequence may be defined as a `sequence_of_statements` or as a `for_statement`. A branching statement may be an **if** statement or a **case** statement. Even then, there will be further choices of the internals, such as whether **elsif** parts are appropriate in an **if** statement, or how parts will be defined for a **case** statement.

In each case, a specification is needed to state the objectives of the statement being designed. The specification must not only describe what is needed under normal circumstances, but also what is needed in abnormal circumstances. Once the design is defined, a verification that it meets the specification needs to be carried out. For complex statements trace tables may be called for in their verification. The performance as well as the functional requirements for the specification must be satisfied. The performance requirements may be implicit, but may be explicit if the subprogram must execute in real time or fit and execute in constrained space.

As noted before, both top down and bottom up thinking must go into design. But the design must be carried out top down. There may well be studies on what lower level designs will look like, as part of bottom up thinking. Even so, the final assembly of design must go top down, step by step. For example, consider an initialized **while loop** under development with specification called `initialized_loop_spec`—say of the form

```
initial_step;
loop_spec:
while
  loop_test
loop
  loop_body;
end loop loop_spec;
```

The most interesting and creative part may well be `loop_body`, and this may motivate its creation and verification first. This is the wrong thing to do. As noted, there may be bottom up analysis preceding the plan for the `loop_body`, to ensure the design is possible and efficient enough. But once the technical ideas are worked out, the design should proceed top down.

Let's go back and examine the design process required for this section of program. The specification `initialized_loop_spec` will be satisfied by a sequence of `initial_step` followed by the entire **loop** called `loop_spec`. Following that, `loop_spec` will be satisfied by a **while loop** with test called `loop_test` and body called `loop_body`. Then, finally, `loop_body` will be satisfied with its actual Ada statements. So as described here there are five separate designs and verifications needed, namely for

```
initialized_loop_spec
initial_step
loop_spec
loop_test
loop_body
```

and in terms of logic all verifications are needed in any order. But in terms of software engineering they are better carried out in the order shown. The specification `initialized_loop_spec` is known first about what is needed, and is all that is known. At that point, specifications for `initial_step` and `loop_spec` can be designed jointly to satisfy `initialized_loop_step`. And finally, specifications for `loop_test` and `loop_body` can be designed jointly to satisfy `loop_spec`.

7.3.1 Sequence Statement Design

As noted above, sequence statement designs are not only among simple assignment statements or procedure statements, but may contain possibly complex branching or looping statements, as well. It is the relation of sequential behavior among the statements that is referenced, not their simplicity. The example of an initialized **while loop just** above is typical. Another example is initialized branching steps, either **if** statements or **case** statements. Other examples include branching statements before or after looping statements, and sequences of more than two statements, for example before and after **loop** statements. Frequently, a sequence of several assignment statements are required to complete a logical process. Practically any form of sequence statements are useful in one situation or another.

The trace tables for sequence statements on the surface may look to be single forms. But the number of trace tables and their logical forms may be quite complex. The number of trace tables required and where they apply depend on how many trace tables each member of the sequence define. The total number of trace tables for the sequence will be the product of the number of each member. In some cases some such trace tables are vacuous. For example with component trace tables dependent on values of the same variables, a variable can not be both positive and negative in different trace tables at the same time unless it has been altered. So the worst case in possible number of trace tables that apply is known, but the actual number may be somewhat smaller.

The design of a sequence of statements begins with its specification, as noted above. From the standpoint of logic, the specifications of each of the statements of the sequence can be invented before or after the specification for the entire sequence. But from the human engineering standpoint, it is better to understand the overall specification first and not lose sight of it in designing the sequence of statements. Then, the sequence of specifications and their statements should be created and verified a step at a time, from **front** or **back**, but not from inside out. Inventing an inside specification and statement first gives nothing to hang onto in its verification. But inventing an outside specification and statement, say **front**, breaks the sequence into two parts, the **front** and the remainder to be designed, call it the **rest**. Now, when a specification for the **front** is designed, it must first fit the start of the specification for the sequence, and the **rest** must take the execution to the end of the specification for the sequence. That is, a specification for the **rest** is implied and needed to argue for the correctness of the **front**. If the **rest** is still a sequence of more than one statement, this process is repeated. If **rest** is a single statement, it is a final step to verify. Once understood, these new definitions and arguments may go very fast in simple cases, but can be surely carried out in complex cases.

For example, consider a specification to exchange CHARACTER variables `Won`, `Lost`, using CHARACTER variable `Referee`, and leaving `Referee` equal to the minimum of the final values of `Won`, `Lost`. This design could begin as above to exchange `Won`, `Lost`, followed by setting `Referee` accordingly, as follows.

```
Referee := Won;
Won     := Lost;
Lost    := Referee;
if Won < Lost
then
  Referee := Won;
end if;
```

This is a sequence of four statements, the first three assignments as seen before (with different variables), the last an **if** statement. Check that `Referee` received the right value.

A major design decision among statements will be the decision to break design among subprograms or packages. We will get into Ada subprograms and packages in Chapter 8. For the moment we realize that subprogram calls may be the best way to organize design. We also realize that many designs can make use of part or all of existing programs or program parts.

7.3.2 Branching Statement Design

Branching statement design, either **if** statements or **case** statements, are required to deal with different logic situations that may arise. In an **if** statement the test condition identifies the central breakout of the situation into **if ... then, elsif ... then, else** sequences of statements. In a **case** statement the **case** test expression identifies **when** which **case** alternative sequence of statements is to be selected. The two branching statements are equivalent in capability, the selection of sequences of statements at the next level based on conditions in **if** statements and on expressions in **case** statements. But they provide different descriptions of branching statement designs and one may be simpler than the other in specific situations. For example, **if** statements may be simpler when only a few—say two or three—alternatives are possible. And **case** statements may be simpler when many alternatives—say ten or a hundred—are possible. But as noted, both have the same logical power of statement description.

Branching statements identify multiple trace tables to account for the various sequences of statements that are possible. As a result, the executions occur over the several trace tables, each with a subdomain of the branching statement. The branching statement itself may have a limited domain due to previous branching or looping in the program hierarchy. As before, the specification of the branching statement must be understood before defining the specifications for the sequences of statements it contains. While there is no logical reason for this priority in specifications, there is human engineering reason in keeping the intellectual and inventive process under good control. Just as in long division, the process can be verified a step at a time, and each step corrected if necessary before going on. That needs using the specifications from the top down, and creating lower level specifications to match the one above.

For example, as above consider a specification to exchange CHARACTER variables `Won`, `Lost`, using CHARACTER variable `Referee`, and leaving `Referee` equal to the minimum of the final values of `Won`, `Lost`. It was solved above as a sequence of statements. But it can also be solved as an **if** statement, as follows.

```
if Won < Lost
then
  Referee := Lost;
  Lost    := Won;
  Won     := Referee;
```

```
else
  Referee := Won;
  Won     := Lost;
  Lost    := Referee;
end if;
```

Check that Referee is left with the right value.

7.3.3 Looping Statement Design

Looping statement design adds another level of complexity to design, but also another level of computing power. One concern of looping is whether the statement terminates. With good design that concern can be limited. But with poor design looping termination can be real problem. So first order is to ensure termination as required. Another concern of looping is the actual behavior with a variable number of iterations, different numbers for different initial data. As already seen, looping behavior can be studied and understood, but takes well defined analyses that account for all possibilities during execution. Good design must also make the determination of looping behavior as reasonable as possible. It is not sufficient to create loops too hard to verify with high hopes they will operate correctly anyway. So the design of looping statements involves not only creating correct statements, but also creating statements that can be shown to be correct.

As for other statements, designing correct looping statements begins with specifications. They must be complete over all the domain required, but needn't deal with initial data outside the domain. For example, many looping statements are initialized before their execution. They needn't handle conditions not possible in the initialization. In illustration, consider the initialized **while** statement Square_Root_Sum with INTEGER variables Total and Number, shown next.

```
Total, Number : INTEGER; -- declaration
...
Total  := 0;
Number := 1;
Square_Root_Sum:
while Number <= 10
loop
  Total  := Total + Sqrt (Number);
  Number := Number + 1;
end loop Square_Root_Sum;
```

where Sqrt is an Ada function that returns the INTEGER closest to the square root of the argument (*e. g.*, Sqrt (3) = 2). Note that this entire part is a sequence with a **while** statement at the end. But this **while** statement will not have to deal with Sqrt of a negative INTEGER, as otherwise might be required.

But having worked out a **while loop** for this problem, it is easy to see that a **for loop** will also work, and is preferred for simplicity, shown next.

```
Total : INTEGER; -- declaration
...
Total  := 0;
Square_Root_Sum_1:
for Number in 1 .. 10
loop
  Total  := Total + Sqrt (Number);
end loop Square_Root_Sum_1;
```

In this case, Square_Root_Sum_1 is two statements shorter than Square_Root_Sum and simpler to understand. The **loop** body is exercised ten times so a **for loop** is satisfactory and preferred. Variable Number does not exist outside the **for loop**. If Number were needed for some other reason, it would have to be declared.

However, if the foregoing **loop** were used in a more flexible way, so that the number of square roots determined was a variable number, the **for loop** would not work. For example, if the range for Number was defined on each use by a lower bound, Low, and an upper bound, High, a **while loop** would be preferred, as follows.

```
Total, Low, High, Number : INTEGER; -- declaration
...
Total  := 0;
Number := Low;
Square_Root_Sum_2:
while Number <= High
loop
  Total  := Total + Sqrt (Number);
  Number := Number + 1;
end loop Square_Root_Sum_2;
```

However, it is now important to implement some discipline on variables Low and High coming into the **loop**, so that square roots of negative numbers is not required. One way to implement such discipline would be to embed to **loop** into an if statement, such as follows.

```
Total, Low, High, Number : INTEGER; -- declaration
...
Total  := 0;
if Low >= 0 and High >= 0
then
  Number := Low;
  Square_Root_Sum_3:
  while Number <= High
  loop
    Total  := Total + Sqrt (Number);
    Number := Number + 1;
  end loop Square_Root_Sum_3;
else
  null; -- think up what to do now
end if;
```

Another way to implement such discipline would be to declare Low, High, Number as nonnegative INTEGERs as follows.

```
Total : INTEGER; -- declaration
Low, High, Number : INTEGER range 0 .. max; -- declaration

...
Total  := 0;
Number := Low;
Square_Root_Sum_4:
while Number <= High
loop
  Total  := Total + Sqrt (Number);
  Number := Number + 1;
end loop Square_Root_Sum_4;
```

In this case, if previous text attempted to give `Low` and/or `High` negative values, the execution would be halted right there before getting to this loop. In summary, there are many ways to recognize wrong data during execution. Which is best depends on circumstances.

7.3.4 Exercises

1. Let a specification be required to bring a set of student grades up to date with a list of scores achieved on the last test. The student grades are maintained in alphabetical student name order of those who are enrolled in the class, but the list of test scores include only those who took the test. Students who did not take the test will be given a zero score. The student grades must be initialized to zero at the beginning of the course. But even before that, the student names in the course must be identified, say as the members of a character data type, in alphabetical order. For example, consider

   ```
   type Student_Names is (Allen_John, Criswell_Sue, Hancock_Sally,
     Kron_Tom, Miller_Amy, Polk_Sam, Rabin_Bob, Williams_Henry);
   ```

   and suggest how to complete the specification.

2. A design for a specification called bracketed_if_spec seems on early assessment to be of the form

   ```
   initial_step;
   if_test
   then
     then_body;
   else
     else_body;
   end if;
   final_step;
   ```

   Identify a set of subspecifications to complete it and a top down sequence of designs and verification required.

3. Give a sequence design to rotate a sequence of values declared

   ```
   X1, X2, X3, X4, XT : INTEGER;
   ```

   to meet specification

   ```
   X1, X2, X3, X4 <- X2, X3, X4, X1;
   ```

   and give its verification.

4. Give a for statement design to rotate a sequence of values declared

   ```
   X1, X2, X3, X4 : INTEGER;
   ```

   to meet specification

   ```
   X1, X2, X3, X4 <- X2, X3, X4, X1;
   ```

   and give its verification.

5. Give an **if** statement design to reverse a sequence of values declared

   ```
   X1, X2, X3 : CHARACTER;
   ```

   unless X1 is max of set {X1, X2, X3} and give its verification.

6. Give a **while** statement design to rotate a sequence of values declared

   ```
   X1, X2, X3, X4 : INTEGER;
   ```

   so that the rotated sequence begins with the largest value and give its verification.

7.4 Program Usage Specifications

7.4.1 Program Usage

Programs are typically developed for use by others than the developers. Users are frequently not programmers themselves, and seldom see the actual source code or design documentation of the programs. Instead, they will be familiar with **users guides** that describe the externals on what to provide as input and what to expect as output. Such users guides need to be readable and understandable by the users, while being precise and accurate about the programs behavior, namely the program behaviors.

Programs may well be used in quite different ways by different users. In widely used programs, it may be reasonable to have several users guides about the same programs, to match several underlying knowledge bases. For example, a set of programs to maintain inventory control with item additions and deletions, daily reports, and special reports for abnormal events, can be used in many situations, say in managing a wholesale warehouse, or in maintaining a financial clearing house, or in tracking a complex engineering operation. Since each group of users is familiar with different methods of operation and communication, it may make sense to provide each with a distinct users guide that describes the same set of programs. In fact, without such a distinct users guide, such a group may refuse to use the set of programs in its area.

Prior to such program usage, in specification and development, how a program or set of programs will be used will be critical in arriving at the specification and carrying out the development. If no users will ever be found, the specification and development simply goes to waste. So anticipating program usage is a critical part of deciding on the specification and development. It is not only important that specifications define useful programs for a set of users, but that development creates programs that operate correctly and efficiently enough. As will be seen in sorting, the same specification can be implemented in ways that computer time required differs by orders of magnitude. And it is critical that programs operate as expected without failures that impact its users.

As part of the specification and development, how users will make use of the programs is needed. That is, how often are what programs, commands, and data will be called up, and in what kind of order will such calls be made. For example, in an inventory control system, how often are new items added or deleted from the system, as well as how much addition and deletion goes in within each item from day to day or month to month. Such information will go into design and choices of basic strategy in dealing with performance in computer execution and data storage. And such information is also critical in testing the programs to better ensure they work properly and timely as needed by the users.

Thus, in addition to the behavioral and performance specifications already discussed, **program usage specification** is critical in confirming that good design and development has met the behavioral and performance specifications. In program usage specification for testing, the precise statistical expectation of use is developed. Such precise statistics will provide a basis for completely objective **statistical testing** and the **certification** of the programs before actual use.

The program usage specifications divide programs into two major classes in their usage, first, programs whose each use is independent of any data of previous uses, and second, programs whose uses depend on data of previous uses. An example of the first class is a sort program, which is provided new data with each use, unrelated to any previous data. The inventory control system mentioned above is an example of the second class, which maintains data between uses for reference to data of previous uses. These two classes of programs will be treated separately, the first in the next section, the second later when external Ada storage files have been introduced. In each case, enough probability theory is introduced before dealing directly with each class of programs.

7.4.2 Probability Distributions

A **probability distribution** over a set of **possible events** is a function whose domain is the set of possible events, and whose range is a set of real numbers (**probabilities**) such that each is nonnegative (>=0) and their sum is 1. That seems simple enough, and it is just that simple. For a program of the first class, a probability distribution over the domain of its program behavior is its program usage specification. Such a probability distribution may be quite complex to discover and describe, but the theory is quite simple. For a program of the second class, a Markov process which identifies interactive stepping points with a domain of program behavior from each stepping point is its program usage specification. In this case, each step moves from one point to another interactively. Thus from each stepping point a separate probability distribution holds for the next input data.

7.4.2.1 Uniform Probability Distributions

The simplest class of probability distributions is **uniform distributions.** For example, a fair flipped coin has uniform distribution of both head and tail given probability 1/2. A fair thrown die of six sides, 1 ... 6, has uniform distribution of each side given probability 1/6. A special case of the uniform distribution is for a domain of a single event with probability 1.

A closely related set of probability distributions to the uniform distribution comes by observing repeated events from the same distribution and accumulating the results. For example, flipping a coin twice (or flipping two coins), four events are possible, namely

head, head
head, tail
tail, head
tail, tail

which is, itself, another uniform distribution, each event (of two previous events) with probability 1/4. However, if these four events are mapped into the number of heads, tails in the two flips, these events

two heads
one head, one tail
two tails

are not events of a uniform distribution, because two heads occur with probability 1/4, one head, one tail with probability 1/2 (can occur in either order), two tails with probability 1/4. Continuing, with three coin flips, if the sequence of events is counted, there are eight events each with probability 1/8, namely

head, head, head
head, head, tail
head, tail, head
head, tail, tail
tail, head, head
tail, head, tail
tail, tail, head
tail, tail, tail

but if these events are mapped into the number of heads, tails in the three flips, these events

three heads
two heads, one tail
one head, two tails
three tails

are not events of a uniform distribution, because three heads occur with probability 1/8, two heads, one tail with probability 3/8, one head, two tails with probability 3/8, three tails with probability 1/8.

In general, the probabilities of repeating the events of a two possible event uniform distribution n times are

n heads	`1/2**n`
n-1 heads, 1 tail	`n/2**n`
n-2 heads, 2 tails	`n*(n-1)/2**n`
n-3 heads, 3 tails	`n*(n-1)*(n-2)/2**n`
...	

in which the numerators are the coefficients found in the repeated powers of binomial terms.

This process of repeating and counting simple events as part of larger events generalizes directly to uniform distributions of more than two possible events, such as throwing two dice and adding the two numbers as a new kind of event. In this case, the probabilities are easily seen in a table with rows from one die, columns from the other, and their totals in the table

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Die Possibilities
Table 7.2

In this case, the domain of the new distribution is 2 .. 12, with probabilities easily determined by counting the number of entries (up the diagonals of the table) to find

2	1/36
3	2/36
4	3/36
5	4/36
6	5/36
7	6/36
8	5/36
9	4/36
10	3/36
11	2/36
12	1/36

More generally, repeating any distribution, uniform or not, and counting events, can be treated by combining the events with coefficients of powers of the probabilities.

In another direction, consider the probability of reaching a particular new event from repeated simple events. For example, consider a uniform distribution of two events with probabilities x and (1 - x), where 0 <= x <= 1. Then the probability of reaching the first event in trials taken over and over is as follows

1	x
2	x*(1 - x)
3	x*(1 - x)**2
4	x*(1 - x)**3
...	

The number of these new events is infinite, and if x > 0 the sum of their probabilities is

```
sum      x*(1 - x)**(i - 1) = x/x = 1
i=1...
```

7.4.2.2 Nonuniform Probability Distributions

Many probability distributions are nonuniform, arising from existing activities. We discuss two examples of nonuniform probabilities that may not seem to be probabilities, but are. They are from word processing and Ada programs. It is not obvious what these probabilities are at first glance, nor why they might be useful. In each case, the probabilities are complex, but samples are observable all the time, if only they are recognized as such. With some thought, underlying definitions can be developed, but why bother? Such definitions permit statistical testing and correctness demonstrations of software, in contrast with ad hoc testing that permits hopeful conclusions that may or may not be valid.

For example, in word processing the distribution of characters in memoranda, reports, books, are far from uniform. It is easy to notice the characters such as 'e', 't', will occur in ordinary text much more often than 'z', 'q'. But the space character ' ' (the blank), will happen even more often, and characters period and comma '.' and ',' will happen quite often, too. In ordinary English text, with words averaging five characters, each word not ending a sentence will be followed by a blank, so about one character in six will be a blank. Words are organized into sentences, each ended in periods or other suitable characters such as '!' or '?'. If sentences average ten words in length, about one character in sixty will be an end of sentence symbol, mostly periods.

But there is another aspect of the distribution of characters in English text, namely, how characters appear in the context of other nearby characters. Words are made up of letters that fit together in making up possible words. Actually a very small fraction of letter strings do make up words. Most strings of five letters, such a "evrst" or "maaeq" are not words. And as noted above, in strings that are words, different letters appear different fractions of the time. So vowels such as 'e', 'a', 'o' appear relatively frequently, as do constants 't', 's', 'r', while other constants such as 'z', 'q', 'j' appear much less frequently. In fact, in English text, it is the words that appear, which happen to be made up of letters. However, there are so many words that for some purposes it may be more useful to examine the letters, assuming they assemble into words for the moment.

At the next level in English text, words do not fit together into sentences arbitrarily any more than characters assemble into words. Just as a small fraction of letter strings make up words, a small fraction of word strings (separated by blanks) make up valid sentences. It is an even greater step to assemble words into sentences. We do have dictionaries that identify words, dictionaries of various sizes and completeness for different uses. But we do not have dictionaries that identify all possible sentences or anything close to that. Even so, there is a general theory on how sentences are formed from words, and sentence structures. With such a theory, specific sentences can be analyzed for the words they contain and how the words fit together into sentences. Word processing systems today do just these kinds of operations. They require software much as you have already seen in dealing with text in word and sentence forms.

In illustration, the 95 characters of Ada introduced in Chapter 2 include 26 upper and 26 lower case letters, the ten digits, the space character, 19 "special characters" (including ".", "," ...) and 13 "other special characters" (including "!", "?", ...). Typical text will not only contain characters, words, and sentences, but also paragraphs, tables, figures, *etc*. Even lines holding words and sentences are frequently padded with blanks because the text does not exactly fit. For example if text is right justified, extra blanks will be distributed between words. So these blanks are characters we hardly need to notice, but which the computer must. Between paragraphs may be lines of blanks as well. And tables, say of numbers, are filled with blanks where the numbers aren't. So the estimate above of one blank for five nonblanks is typically too small—more like one blank for two to three nonblanks would be more realistic.

The treatment of text as probability based is not always recognized. Why isn't each new problem just another specific example of text, with no underlying background? That is certainly a point of view people can and do have. But creating specific text is indeed a probability act in a broader sense. It is that the methods used in creating specific text are general, and good methods recognize and profit from the probability class of text that must be handled. For example, current text processors uniformly recognize the need to deal with blanks in effective ways, because blanks are so widespread. As a result, any paragraph can be ended any place in a line and the line will be filled out with blanks automatically. Likewise, a line of blanks requires only a single step for its first character, not the entire line.

As already noted, the specific generation of probabilities of characters in text processing is not easy, but not intellectually difficult either. The analysis of typical past text examples can create a probability model, and samples of such examples used in testing directly.

Another example of nonuniform statistical distribution is Ada programs and program parts themselves. As noted above for English text, Ada text will be composed of the equivalent of "words" at the lowest level. Such "words" may be Ada lexical units of six basic types, namely

identifiers
numeric literals
character literals
string literals
delimiters
comments

Then, these basic types are used to create declarations and statements at the next levels, and at high levels, procedures, functions, and packages will be defined in terms of the declarations and statements. These Ada programs and program parts are designed to address data processing problems. As discussed before, sound Ada text can be organized in various ways into lines. Good practices will make the Ada material as easy to read as possible. Just as good English text, good Ada programs meet their objectives, namely their specifications. But such Ada programs are also a very small fraction of the text possible. That is, given a random piece of text, it will be Ada text with very small probability. First, the syntax of Ada text is very special, for example as an assignment statement

```
Alpha := Beta;
```

or an if statement

```
if Cost > Profit
then
  Post_Loss;
else
  Post_Profit;
end if;
```

which are legal Ada statements. But in the larger sense, while legal statements they may or may not be legal in a larger program part or program. The variables in these statements must be defined for use previously. For example, if Alpha has not been declared the assignment statement is not legal.

7.4.3 Statistical Program Usage

In creating behavioral and performance specifications for a program or set of programs, how users will make use of the programs must be anticipated. The purpose of the program usage specifications is to formulate this possibly qualitative anticipated usage into a precise quantitative probability distribution over the input domains of the behavioral specifications. This may require a deeper analysis of the way the programs may be used, possibly including interrogations of potential users to better understand their problems and how the programs will help them address their problems.

As for any statistical statement about a class of people, it is important to remember that individual use may differ from the collective statistical use. For example, there are statistics about the death rates of people by ages. But very few people die at their average age of death, most hoping to live longer and about half doing so. In the same way, an individual user may use the programs for a particular set of problems that others do not have, and use different data. Of course, correct programs will handle all data provided them properly, and statistical testing will confirm that the programs are correct over all data they are required to handle.

In some circumstances, uniform distributions may be good bases for developing reasonable usage specifications. But as noted above, finding the right bases may take some thought and analysis. In illustration, consider a sort program, and what usage statistics might be appropriate. For convenience, let the sort program expect in the input file an INTEGER Array with length in range 1..1_000 and components in range 0..1_000_000. That is, the input file contains first an INTEGER length, followed by the INTEGER length of INTEGER components. One basis of statistics might be to consider length to be uniformly distributed over 1..1_000 and each component uniformly distributed over 0..1_000_000.

However, in discussions with potential users of the sort program, they may bring up different ideas on the statistics they expect in use. For example, on the matter of the length of the Array, they may expect the distribution more heavily weighed to the longer side, for example the probability of length is proportional to length itself, so a length of 1_000 is 10 times as often as a length of 100, 100 times as often as a length of 10, and 1_000 times as often as a length of 1.

On the matter of the Array to be sorted, they may expect a different kind of distribution of the components than independently uniform. For example, they may expect the Arrays to be nearly sorted already, which may suggest some sort algorithms might be more efficient than others. But what does "nearly sorted" mean? One measure of "unsortedness" is to count the number of adjacent component inversions in the array, that is to examine each pair of adjacent components, counting how many pairs are already in sorted order, how many not in sorted order. If the length is n, there will be n - 1 comparisons divided between sorted pairs, say s of them, unsorted pairs, say u of them, so s + u = n - 1. In a sorted Array, s = n - 1, u = 0, and in a totally inverted Array, s = 0, u = n - 1. A possible distribution could weight Arrays by the values s + 1 (values s would mean no totally inverted Arrays), from n down to 1. It would take some more design of such a distribution to complete it. For example, unsorted pairs should be uniformly distributed in the Array, as well.

On the other hand, users may have quite different suggestions about distributions. For example some characteristics of usage can be derived by understanding the specific application. An illustration would be the use of roman numerals. In section 3.1.2 and 7.2.3 the concept of recognizing and performing arithmetic on roman numerals was presented. If, a particular user intended to have these numerals represent years up to the present date, inferences can be made about the expected distribution of digits. In this distribution at most one M, L, D, and V will occur, but up to four C, X, and I may occur. If the application another user intended was to represent the date a motion picture was produced, additional information about the distribution would be known because it could only be in the range of 1920 to the present. This distribution would always contain one M, one L and four C. By knowing what the expected usage will be, an appropriate distribution can be created.

Knowing what input data will likely be encountered is only part of specifying the program behavior. The correct estimate of the distribution of input data can also provide insight into the relative frequency with which the various components of the program will be executed. In the roman numeral example the procedure *valid_roman_numeral* should be used to test each operand that is input before any of the arithmetic operators are used. Thus the frequency of using this procedure would be considerably higher than any of the individual arithmetic procedures. The concept of verifying that a given input is within the desired range is important in most applications and thus this portion of the program will likely be exercised frequently. In Ada, part of this validation can be accomplished through the use of range constraints that are placed on objects at the time of declaration.

Users can provide insight into other factors affecting the frequency of execution of program components by indicating how they intend to use a program. If a specific user intended to use the roman numeral program in an application to deal with the dates movies were produced, procedures named **Larger** and **Smaller** would likely be used to determine if Back To The Future II was, in fact, produced before Back To The Future III and by how many years. On the other hand, it is not likely that the roman numeral arithmetic operators of **Times** or **Divided_By** would used in such an application.

The frequency with which a particular Ada procedure or function is executed is not intended to be the only measure of the importance of that component to the overall program. It does, however, reflect the weight it carries in the probability distribution of the execution of all procedures or functions of a given Ada program. In the case of the movie application of roman numerals, the validation procedure will be used at least three times as often as the arithmetic procedures, once to validate each operand and once to validate the result. The more that is known about this distribution the better a set of tests can be constructed to reflect the true usage profile of the components of a program. The actual distribution component usage could be empirically determined by monitoring the program execution over multiple sets of input data. In most cases this would be impractical if not impossible. Instead it may be possible to establish an estimated distribution of component execution based on insight provided by the user.

7.4.4 Markov Models in Usage Distributions

A Markov model is an extension of a probability distribution with no correlation between successive events to a set of distributions that apply in various subsets of the domain of the distribution. In each subset a specific probability distribution holds, and a specific event will take the process to another subset (possibly the same) with a given probability. For example, let a game be defined with a coin and a die. If the coin turns head, coin is the next object to use, but if tails, the next object is the die. If the die turns up 1, the coin is the next object to use, otherwise, the die is the next object. In illustration, the sequence

HHT3521T2

is possible, with probability $((1/2)**3)*((1/6)**4)*(1/2)*(1/6)$, but

HH234TH

is not possible. There are two subsets in this case, {H, T} and {1, 2, 3, 4, 5, 6} of a total set {H, T, 1, 2, 3, 4, 5, 6}. For each item of the set, a probability distribution applies, namely as shown in the table given next.

	Next Item							
Item	H	T	1	2	3	4	5	6
H	1/2	1/2	0	0	0	0	0	0
T	0	0	1/6	1/6	1/6	1/6	1/6	1/6
1	1/2	1/2	0	0	0	0	0	0
2	0	0	1/6	1/6	1/6	1/6	1/6	1/6
3	0	0	1/6	1/6	1/6	1/6	1/6	1/6
4	0	0	1/6	1/6	1/6	1/6	1/6	1/6
5	0	0	1/6	1/6	1/6	1/6	1/6	1/6
6	0	0	1/6	1/6	1/6	1/6	1/6	1/6

Probability Distribution
Table 7.3

A Markov model enables us to specify a usage profile that accounts for the serial correlation of uses of software. For example, batch programs, such as a square root program or even a compiler, perform independently of previous uses on each input. But an interactive program, such as a flight control program or a word processor, will be used over and over within a flight or document preparation with high correlation between the successive inputs. The Markov model is a general model which can be applied to many interactive software usage situations. The use of the software may go from state to state, based on previous inputs, and in each state the probabilities of the next input will be different. But the next specific input that appears will take the process to another state (possibly the same), and so on until the interactive program terminates.

7.4.5 Exercises

1. A common deck of playing cards has four suits, of Spades, Hearts, Diamonds, and Clubs, each suit with number cards 2 through 10 and face cards of Jack, Queen, King and Ace.

   a. What is the probability of drawing a face card from a randomly shuffled deck?

   b. What is the probability of drawing two face cards in two consecutive draws, assuming the first card is replaced in a random fashion?

   c. What is the probability of drawing two face cards in two consecutive draws, assuming the first card is not replaced?

   d. What is the probability of drawing three cards of the same suite on three consecutive draws, assuming each card drawn is replaced in a random fashion?

   e. What is the probability of drawing three cards of the same suite on three consecutive draws, assuming each card drawn is not replaced?

2. What are the possibilities and probabilities for a set of randomly thrown dies in the following cases.

   a. For the sum of three dies?

   b. For the maximum of two dies?

   c. For the difference of two dies?

   d. For the product of two dies?

   e. For the maximum of three dies?

3. Make an estimate of the distribution of characters to be found in this Chapter—what fraction will be blanks? what letters? what digits? what other characters?

4. Make an estimate of the distribution of the six Ada lexical units in Ada programs and program parts in this chapter.

5. Make an estimate of the distribution of roman numerals required for defining cafeteria meal bills that run from I to M uniformly.

6. Assume that the `Trend_Program` **(Exercise 8, Section 5.1)** and as verified in Exercise 3, Section 6.2.6 was to be used in analyzing a historical record of temperatures recorded during a given year. What could be said about the expected distribution of the input data? What could be said if the input was the variation from the temperature last year and not the actual temperature this year?

7. The Ada procedure `Traffic_Signals` described in **Section 5.2.1** was modified in **Exercise 3 of Section 5.2.4** to provide a parameter `DURATION` that specified how long a light would stay red and green. Assume that such a traffic regulator worked best if the duration was lengthened during peak loads. How could a traffic engineer, as the user, supply useful information regarding the input data that could be used to establish the parameter `DURATION`?

8. From your understanding of the various operations that are included in the package TEXT_IO what information could you provide that would help to estimate usage of `Get`, `Put`, `Open_Files`, and `Close_Files`?

7.5 Software Certification

There are two ways to address the question of program correctness, either by **program verification to prove correctness**, or by **program certification** through testing and observation of output to **demonstrate correctness**. But both ways are fallible. While proving correctness is mathematically sound, by comparing the specification relation with the program behavior, humans are fallible in carrying out the mathematics, especially as specifications and programs get large and complex. While demonstrating correctness is possible for all tests executed, the number of possible inputs for all but the simplest programs is astronomical and beyond practical possibility of complete testing. Even for tests carried out, human fallibility also applies to determining if outputs are correct. Failures may not always be observed when they occur.

While this fact of human fallibility and pragmatics may seem at first to make getting programs correct impossible, that is not the case. However, it is well to recognize reality to start with and to then discover that getting programs correct is indeed possible in practical terms. The practical basis for getting programs correct is to use both program verification and program testing together, in synergism. They support each other in remarkable ways.

Program testing can be viewed as a form of checking verification. If the program is executed with inputs from the domain of the specification, the outputs can be checked as correct for those inputs. And no matter how carefully programs are proven correct they should be tested as well. Correctness proofs, by individuals or by teams, are subject to human fallibility. In illustration, in ordinary arithmetic, long division with place notation can be done very accurately, but human mistakes are still possible. So checking division with reverse multiplication testing is useful in important arithmetic. In a similar way, a testing is a valuable check for correctness proving.

But program testing can be a science in its own rights, too. When the programmer makes up tests, errors may well be found. The main results of just making up tests, no matter how intelligently, are anecdotal. Different tests may have turned up different errors. And if no errors show up, no one really knows whether there are no errors, or whether the testing was inadequate. Instead, testing should be on a scientific, statistical basis. For programs well used by many people, the very use, the choice of inputs, is a statistical use. Some inputs are more likely than others. And inputs may be related sequentially, with the statistics of use dependent on recent or past history of previous use.

When a program is to be seriously used it is not enough to simply know its specification for testing it. One should know the expected statistics of use. One should also know, of course, the severity of different kinds of uses. Some failures may be annoying, some life threatening, so those factors should go into the statistics gathering, as well. That is the statistics of testing should be stratified, depending on importance of use. A stratus may contain a small subset of tests of great importance, to be sure those cases are sufficiently addressed. Some cases may be so important as to make up a stratus by themselves, with one single test in such a stratus. Strictly speaking, there may seem to be no statistics in a stratus with a single test, but it is indeed a legitimate status in a total stratified statistics definition.

7.5.1 Correct Software

Software needs to be correct in two ways.

First the specifications need to be right. Good specifications define software that will fulfil its assigned mission, to the full satisfaction of everyone that has a stake in the software, if the software is implemented in accordance with the specifications. This is the deep problem in creating software, determining what the software should do. It requires knowledge of the subject matter as well as techniques for writing specifications.

Second the software must meet the specifications. This means determining if the software as developed is a correct rule for the behavior required by the specifications. If the software defines a correct rule then by definition a user of the software will never experience a failure while using the software since the software operates in accordance with its assigned behavior. Conversely if the software is not a correct rule for the entire domain of the assigned software then the software will from time to time produce an incorrect result according to the specification. In this case a **failure** has occurred. The greater the divergence between the rule and the behavior the more likely it is that failures will occur.

In some cases the software will produce results that are not to the liking of some user but they are in accordance with the specification. In this case correct software exists and a faulty specification may be the difficulty. But it may be that the specification is indeed right and this particular user is the problem, either in understanding how to use the software or even whether this software should be used for the problem the user has. The user may hope that software will solve some problem that it doesn't, but other users understand that the software does not solve this problem, but another problem.

Software correctness can be observed by examining the results produced by software and finding consistency between the results expected by the specification and the results produced by the program. Any inconsistency is called a failure. When this observation takes place in real world operations and a failure is observed it can sometimes have disastrous consequences. That is why enlightened software developers want to develop software that has zero probability of ever performing an operation that results in a failure during use, especially a disastrous failure. The ideal is to develop correct software, with strong evidence that the software is indeed correct before it is deployed.

Correct software is software that will never produce a failure, that is for every combination of stimuli (proper or improper) the program produces the combination of responses required by the specification. Such a program is a correct rule for the behavior defined by the specification.

7.5.2 Correctness Evidence

There are two aspects to asserting the correctness of software. One is **verification**, as discussed previously, and the other is **certification**.

Verification evidence consists of arguments that have been made as shown above (and hopefully recorded) that support the conviction that the rule (program) is correct for the behavior (specification). If complete (exhaustive) verification arguments have been made that show the rule is correct for the behavior then there is a reasonable probability that the rule is correct for the behavior. But since humans make verification arguments and humans are fallible there is also a chance that the rule is incorrect for the behavior. The better the process that guided the development of the rule from the specification behavior and the preparation of the verification arguments the higher the probability that the rule is correct for the behavior. It has been shown that correct software can be developed using only informal verification arguments when the engineers know what verification is and carry it out informally. Even then, engineers may need to elaborate arguments when inspecting engineers do not follow the informal arguments or have specific doubts about the correctness of the program.

An analogy for the software design and verification process is long division. A long division problem is decomposed into a logical sequence of simpler problems, each finding the single next digit for the quotient. Each simple problem is solved in turn. But before proceeding to the following simple problem the person performing the long division performs a verification to increase the probability that the next digit solution to the simple problem is indeed correct. First, this next digit is multiplied by the divisor and the result compared with the local dividend. If the product is larger than the local dividend, the next digit tried is too large. If the local dividend minus the product is larger than the divisor, the next digit is too small. If either error is found the simple next digit problem is solved again. This process continues until the verification indicates that the solutions to all the simple problems are correct. In this way it has been found that third and fourth graders have a very high probability of correctly solving long division problems that Archimedes would have found quite difficult. But there is still a small chance of a numerical error, even though the process is known, including checking for mistakes.

Software design problems and verifications are more difficult than long division problems and verifications but in principle the logic is the same and in the hands of dedicated teams good software design processes exhibit a high probability of producing correct software.

Therefore, the first evidence of correct software is the process by which the software was produced. If the process is reproducible and has a track record of producing correct software then there is some expectation that the process is likely to once again produce correct software. On the other hand if the process is not repeatable (*i.e.*, it is heuristic, trial and error) and it has a track record of producing software in which failures are observed then it is likely that the process will once again produce incorrect software.

A second evidence of software correctness is the verification arguments. If the arguments consist of anecdotal arguments that indicate recent testing has yielded few or even no failures then the verification arguments are weak. If on the other hand the verification arguments may consist of a documented trail of logical arguments that have been presented to a jury of critical peers. If everyone agrees the verifications show the rule as developed fully defines the full domain of the specification then there is a reasonable probability that the software is correct.

Certification evidence beyond verification consists of observations made during software usage of proper or improper inputs. It is best if the software developers perform no testing of the software. In this case then the first testing begins with independent usage testing to begin certifying the claim of software correctness. The best evidence of software correctness is to never encounter incorrect operation of the software. Extensive usage with no instance of incorrect operation is not a fool proof proof of program correctness. But it is strong evidence that the software has a high probability of performing correctly in the future.

A third form of correctness evidence is the observed failure and correction history of the software. A software system with a good failure history (that is few failures have been observed and the observed failures are evidence of mathematical fallibility rather than programming mistakes) is preferred to one with a poor performance failure history (that is many failures have been observed and many of the observed failures are evidence of programming mistakes that have required deep and extensive corrections in the software).

As discussed above, the best way to make certification measurements is with a valid testing process where the tests are statistical representatives of actual usage. It is necessary that these tests are selected at random so that statistical inference theory can be utilized to develop valid projections of future failure performance. All tests should be representative of actual usage. Therefore, it is important that the tests performed prior to actual use are representative of actual use.

It turns out that it is possible to use a Markov model to represent all interactive realizations of software use. As a result the model can be used to construct tests in accordance with the expected usage profile of the software. In running software tests it is necessary to select the actual tests at random so the tests provide the basis for the development of statistically valid projects. If tests are selected in any other manner the only projections possible are the development of anecdotal arguments which are very unsatisfying and provide only marginal value.

Therefore, the last evidence of software correctness except for real use is the execution of randomly selected tests to usage statistics that perform correctly.

For a high probability of failure free software performance all four forms of correctness evidence are needed. Using correctness evidence to project the probability of future failure free software performance is the subject of the next section.

7.5.3 Certifying The Correctness of Software

As noted, software is either correct or incorrect. If incorrect it may operate correctly most of the time, but there will be conditions in which it operates incorrectly. If software proves to be incorrect with an observed failure, it may be corrected. When corrected, the software may become correct, or may remain incorrect. It may remain incorrect because other incorrections still remain undetected and show up later, or because the correction just made is itself incorrect.

However, correctness is entirely observed through the execution of the software, and is never knowable absolutely. Even so, software can indeed prove to be correct with extensive correct behavior.

Certifying the correctness of software means executing the software to statistically generated usage inputs with no failures occurring, and recording the failure free execution. If a failure occurs, the software is not correct, but can be corrected. As noted above, subsequent execution may or may not show additional failures. At first glance, this may seem a weak method of certifying the correctness of software. In particular, a piece of software with many unobserved failures can be declared correct until the next failure shows up. But human intelligence has no trouble identifying such a situation. The truth to recognize is that there is no stronger statement possible about correctness. If software is indeed correct (unknowable by humans) it will be failure free in usage (which is observable by humans) and can be certified to the level observed.

Newly developed software may likely have early failures from errors of mathematical fallibility, which once corrected lead to no more failures. For example, a 5,000 line section (5 KLOC) of an IBM Cobol Structuring Facility (SF) was certified in a sequence of 4 test segments of 30 tests each, 120 tests in total. In the first segment, failures were found in the first test, and when fixed, the next 29 tests ran failure free. In the second segment, failures were found and fixed in the first two tests, then 28 tests ran failure free. In the third and fourth segments, the experience was identical to the first, only the first test revealing failures. No further failures were ever found in this section of the prototype. This is a very different experience than expected with trial and error programming and debugging. In each set of 30 tests, failures will be found in several runs more than one or two, and spread out over all the tests. Then failures will typically occur now and then after the first 120 tests.

Software with unit debugging often has deeper failures harder to find and fix. Failures do not show up immediately, as do errors of mathematical fallibility. Large software systems of today are typically of this category. Hundreds or thousands of failures are found, and failures continue to occur, no matter how many have been removed. As software matures, few such failures are due to the original software. They are due to the fixes made, of which some one in five have produced an additional failure. But software failures of verified programs are fixed more readily under team scrutiny with very few, if any, subsequent failures.

An important piece of history for any piece of software is all the failures previously discovered and corrected, with the time to failure in each case. If the software has a small number of tests with failures, say 5 or so as in the IBM Cobol S/F prototype above, they will show up quickly, and when corrected one by one, no more failures ever show.

In summary, the correctness of a piece of software is certified by failure free execution to statistically generated usage inputs, including real use. Many software parts working together can have different levels of confidence, based on the times executed without failures. But a software procedure or function with no errors will require time to establish that possibility, and never will be established failure free except by continued use without failures.

7.5.4 Statistical Test Generation

Given the program usage specification, namely a probability distribution over the possible inputs for the program, usage testing generates tests randomly according to the probability distribution. The correct outcomes of these tests must be determined, for comparison with the executed outcomes. The correct outcomes are determined by the functional specifications and may need to be calculated by other programs. For example, if a program is being developed to replace another program, say to execute more efficiently, or on different computers, the correct outcomes may be available already. Of course, such outcomes should be checked to make sure they are indeed correct. When a new program disagrees with an old program, either program may be at fault.

As already noted, some failures may be more drastic that other failures. One failure may produce a misspelled word in a comment back to the user, and another failure may bring a whole system down, possibly losing data as well. In order to recognize the varying importance of correct behavior, testing can be defined in **stratified statistics**, to better ensure that critical facilities of the software are sufficiently tested. So a stratus of testing defines a subset of possible inputs for this testing. Such a stratus can identify a subset of important inputs which the certifiers want to give especial notice in test strategy. In this sense, stratified statistics must be designed by the certifiers to better ensure completeness in the testing.

One stratus of testing may be the entire input set of the system with probabilities associated with each specific input. But some inputs may be more critical for system behavior than others, so other strata collecting such inputs into subsets will be very important. In the extreme, some strata may contain only a single input, to be sure it is tested. Large systems may call for hundreds or thousands of strata to cover all possible circumstances in adequate ways. There are two choices required in defining stratified testing, namely what the strata are, and what level of testing is required for each stratus. For strata with only a single input, the level is easy, namely one test. For strata with small numbers of inputs a small number of tests may be called for, but the importance of the system behavior on these inputs may call for more tests. The distribution of tests among the strata is a critical design issue, to address both the statistical and the importance issues of the system inputs.

In illustration of a software system dealing with nuclear power safety, it is noted that simple real time test generation hardly does justice to the safety problem. Most of the time, nothing dangerous is happening. When dangerous things are happening they have been preceded by certain kinds of observable events. So a strategy of test design is to spend 50% of the time in testing in normal time, 50% of the time in testing at 4 orders of magnitude faster looking for dangerous observable events.

In large systems, many critical situations can arise that need to be recognized as critical when they arise and tested thoroughly. Thus, many strata can be identified to deal with these situations, some with single inputs, some with small sets of critical inputs, some with larger and larger sets of inputs. With these strata definitions and a level of testing for each stratus, there is a basis for statistical testing for the entire system.

When Ada procedures and functions have been developed for broad use in other programs, it may be desirable to test them explicitly for their correctness. Just as for programs, their usage specifications by other programs is needed to define a probability distribution of the arguments they may be called with. In addition, a **shell testing program** will be needed to get test data from the input file, call the procedure or function with the arguments provided, and put the results to the output file.

In view of the foregoing, it is clear that selected testing, that is, testing with inputs directly selected by a programmer for a program, will provide no statistical evidence of the reliability of the program. A programmer may feel better psychologically by creating some tests to see that a program works on them. This will be especially so if intuitive guesses have played a role in the design and coding of the program. With effective functional verification, there is much less motivation to try out this or that input on the program.

In addition to testing the functional behavior of a program, it is also possible to create tests that are sure to execute various parts of a program. In particular, it is possible to devise a set of tests that execute all parts of a program, or make use of all data declared in a program. Such testing to cover all of certain aspects of a program or its execution is called **coverage testing.** At first glance, coverage testing would seem a comprehensive method of testing, to insure that no failures remain in execution. Unfortunately, coverage testing does not insure the discovery of all such points of failure. And, while counter intuitive, coverage testing is not so efficient in finding the important failures of a program. This will be shown in a later section below.

7.5.5 Measuring Testing Results

When statistical testing has been carried out, the results allow a statement of the correctness of a program, procedure, or function. The testing has been carried out in a sequence and possibly a failure in the output has been identified. In this case, the program is clearly incorrect. The failure should be analyzed and the program corrected for subsequent testing. The symbol TTF will be used to denote a "Time To Failure" that has been identified, measured from either the start of testing or the last previous correction. The "Time" in TTF may be defined in various ways, depending on the program or subprogram. For programs that execute quickly with very similar time of execution for all data, "Time" may mean number of times executed to the failure. For programs that execute over considerable time periods, possibly quite different execution times with different data, "Time" can mean accumulated elapsed time of executions to the failure.

Also, it is possible than no failures have been identified, either from the start of testing or since fixing the last failure found. The symbol TWF will be used to denote "Time Without Failure" that has been identified. As for TTF, TWF can be measured in appropriate ways, depending on the nature of the execution.

TTF's represent the finding of a relatively rare event, a failure, on some execution, and namely the first such rare event as discussed above. When new programs are first tested, trivial failures may be found for a time from human and mathematical mistakes. As these trivial failures are found and fixed, failures may occur less frequently or entirely disappear. That was the experience in the COBOL/SF prototype mentioned above, namely failures on each first or second run of program, and no failures ever after.

As already noted, statistical certification of software involves, first, the specification of usage statistics in addition to behavior and performance specifications. Such usage statistics provide a basis for assessing the correctness of the software being tested under expected use.

As each specified increment is completed by the developers, it is delivered to the certifiers, combined with preceding increments, for testing based on usage statistics. As noted, the structured specification must define a sequence of nested increments which are to be executed exclusively by user commands as they accumulate into the entire system required. Each subsequence represents a subsystem complete in itself, even though not all the user function may be provided in it. For each subsystem, a certified correctness is determined from the usage testing and failures discovered, if any.

It is characteristic that each increment goes through a maturation during the testing, becoming more reliable from corrections required for failures found, serving thereby as a stable base as later increments are delivered and added to the developing system. For example, the HH60 flight control program (for an Air Force helicopter) had three very efficient increments of about 12 KLOC of software each. Increment 1 required 27 corrections for failures discovered in its first appearance in increment 1 testing, but then only 1 correction during increment 1/2 testing, and 2 corrections during increment 1/2/3 testing. Increment 2 required 20 corrections during its first appearance in increment 1/2 testing, and 5 corrections during increment 1/2/3 testing. Increment 3 required 21 corrections on its first appearance in increment 1/2/3 testing. In this case 76 corrections were required in a system of some 36 KLOC, some 2.1 corrections per KLOC for verified and inspected software, with no previous execution.

The COBOL/SF, version 2, consisted of 80 KLOC, 28 KLOC reused from previous products, 52 KLOC new or changed, designed and tested in a pipeline of five increments, the largest over 19 KLOC. A total of 179 corrections were required during certification, some 2.1 corrections per KLOC for all 80 KLOC of software, under 3.5 corrections per KLOC for software with no previous execution. The productivity of the development was 740 LOC per person month, including all specification, design, implementation, and management, in meeting a very short deadline.

7.5.6 Certification Tasks

In parallel with the development team, the certification team prepares to certify the software up to and including the increment being developed by the development team. The certification team uses the usage profile and the portion of the specification that is applicable to the increments to be verified to prepare test cases including proper outputs to tests.

When the development team has completed an increment, the certification team creates one or more successive **versions** of the accumulated system up through this increment. For each version the certification team compiles the increment, combines it with previous increments, and certifies the accumulated system through this version. If a failure is encountered in the certification of a version, it is returned to the development team for analysis and engineering changes to whatever increments are causing the failure. While a failure is likely to be caused by the latest increment added, previous increments may be at fault and changed as well, as noted in HH60 experience. Each redelivery of changed increments defines a new version. If no failures are encountered in the certification of a version, no additional versions are required.

Within each version of the accumulating system, tests are constructed at random for the statistical strata in accordance with the specified usage statistics profile and then exercised. Test results are compared to a standard and either a failure occurs or the result was correct.

7.5.7 Testing Procedures and Functions

When Ada procedures and functions have been developed for broad use in other programs, it may be desirable to test them separately and explicitly for their reliability. Just as for programs, their usage specifications by other programs is needed to define a probability distribution of the arguments they may be called with. In addition, a **shell testing program** will be needed to get test data from the input file, call the program or function with the arguments provided, and put the results to the output file for evaluation.

As Ada programs become large systems it is desirable to test subunits separately and then together. This involves using shell testing programs to function as those subunits that have not yet been developed or completed. As each of the subunits is developed it is tested based on the appropriate probability distribution of input data. Then the subunits can be tested together again using data derived from the probability distribution that reflects the usage frequency of each subunit. In the case of the system for roman numeral arithmetic, each arithmetic operation could be created and tested using data that reflected appropriate probability distribution. All the operations could then be tested together with the expected usage of verification operation at least three time that of any other operation. Although this is a simple example, it illustrates the concept of having the expected usage determine the testing data and sequence. The larger the system the more complex the development and testing sequence is and the more control is needed to manage the entire development process.

7.5.8 Exercises

1. What would be an appropriate probability distribution of test data for the roman numeral arithmetic system for a specific distribution? Why?

2. What would be an appropriate probability distribution of test data for the temperature trend program described in Exercise 3 in Section 5.3.5?

3. Why are the results generated by test program be equally suspect if discrepancies occur with the original system? What would need to be done to resolve this conflict?

4. Given the information provided in answering Exercise 5 of Section 7.3.5, what would be an appropriate probability distribution of input over a daily or weekly period?

5. Given the estimate of the usage profile of the various operations included in the *TEXT_IO* package, made in Exercise 6 of Section 7.3.5, what would be an appropriate probability distribution of input?

6. What are sort specifications for arrays of INTEGERs based on their declarations in which duplicate objects are retained in the final array. Given statistics of specific arrays to be sorted and the relative importance of all arrays to be sorted correctly, define stratified sets of tests that would make good use of sort times in testing.

7. What are distinct sort specifications for arrays of INTEGERs based on their declarations in which duplicate objects are replaced by a single object in the final array. Given statistics of specific arrays to be distinctly sorted and the relative importance of all arrays to be sorted correctly, define stratified sets of tests that would make good use of distinct sort times in testing.

8. Given a specification to be implemented in a sequence of four sectors of increasing subspecification until the entire specification is reached, with statistics for the entire specification, determine what the statistics should be for each of the four accumulated subspecifications.

9. Given a statistical test process for a program with the following possible results of times to failure, followed by fixing the failure and resuming testing, where * means no failure in the last time,

   a. 1, 1, 4, 250*
   b. 250, 1, 1, 4*

   discuss what to conclude in the two cases.

10. Given a statistical test process for a program implemented in a sequence of four sectors of increasing subspecification in the following form

```
v1:  1, 1, 4, 250*
v2:  1, 3, 7(v1), 250*
v3:  1, 4(v2), 6(v1), 250*
v4:  1, 2, 3(v2), 5(v3), 250*
```

in which failures due to previous sectors are identified, determine what the conclusions are.

Chapter 8

Sequential Ada III

In this chapter we will introduce sequences of statements that collectively serve as an abstraction of a single operation; such sequences of statements will be handled as if they were a single indivisible operation and will be given a name. Ada calls these abstractions *subprograms*. Ada recognizes two distinct classes of subprograms, procedures and functions. A procedure is a statement in the language and can be used wherever a statement is valid. A function is an expression and can be used wherever an expression is valid. In addition, we will examine a very powerful new concept in Ada that allows us to compose software systems from collections of logically related entities. We call these collections of logically related entities *packages*. They are architectural tools that allow us to build systems. We will study them in this chapter and learn how to use them to create software solutions that are more efficient and maintainable. More powerful features of Ada will be deferred until Chapter 11.

8.1 Ada Packages

When we are designing large systems, it is desirable to be able to handle logically related items as a "single unit," or module. In Ada, this is accomplished by use of another programming unit called a *package*. The package is a programming unit specifically designed to collect together all logically related items into a single physical location. While there are no restrictions placed by the language on what may be in a package, it is the responsibility of the software engineer to provide everything needed for the use of the abstract entity encapsulated in the package.

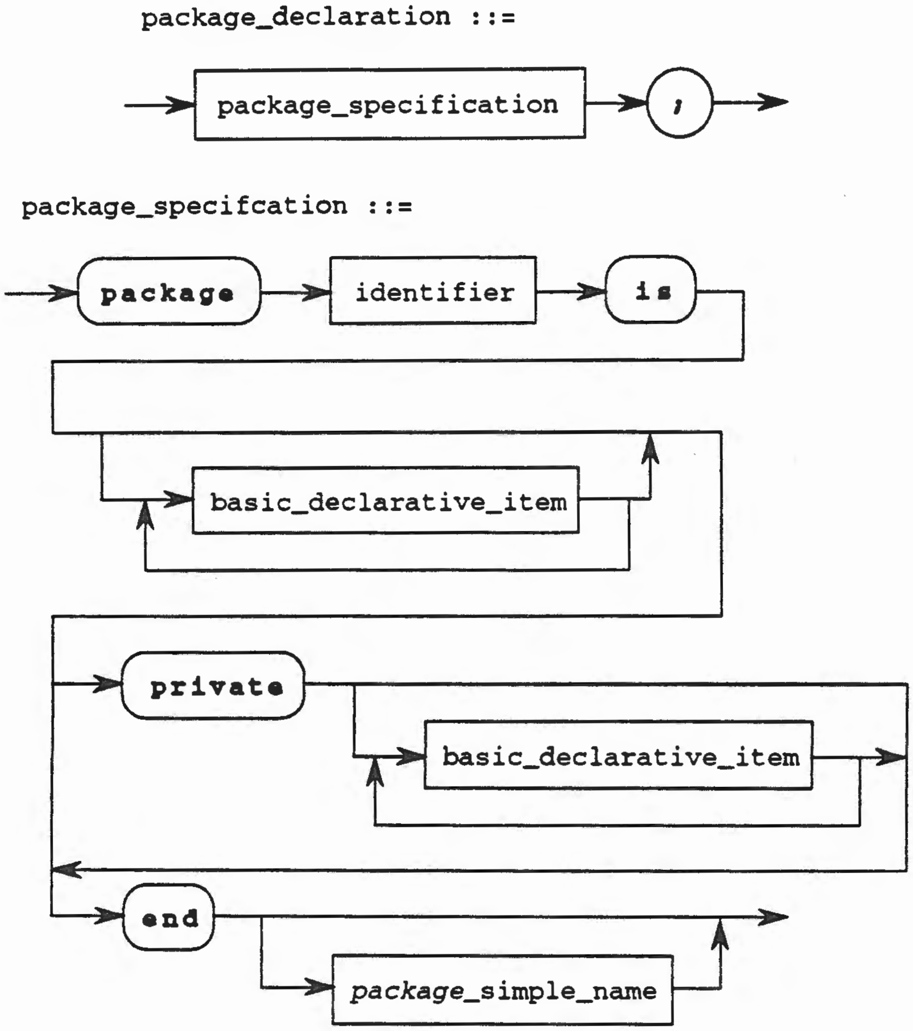
8.1.1 Introduction to Packages

We have talked about packages before and you have used them without fully understanding their purpose. The most important package that you have already seen and used is TEXT_IO. This package is nothing more than a collection of logically related entities, types, objects, procedures, and functions. You should refer to the Reference Manual for the Ada Programming Language [LRM 14.3.10], starting on page 14-26, for a complete listing of this package. For now, suffice it to say that it is a collection of the entities logically related to input and output operations. We will look further at this package and at others in this section.

The syntax for a package looks very much like that for a subprogram which we will see shortly. Packages come in two distinct parts—the specification and the body. The following syntax charts describe the syntax of Ada package specifications.

```
package_declaration ::= package_specification;

package_specification ::= package identifier is
   {basic_declarative_item} [private {basic_declarative_item}]
   end [package_simple_name]
```

Ada Package Specification
Syntax Definition 8.1

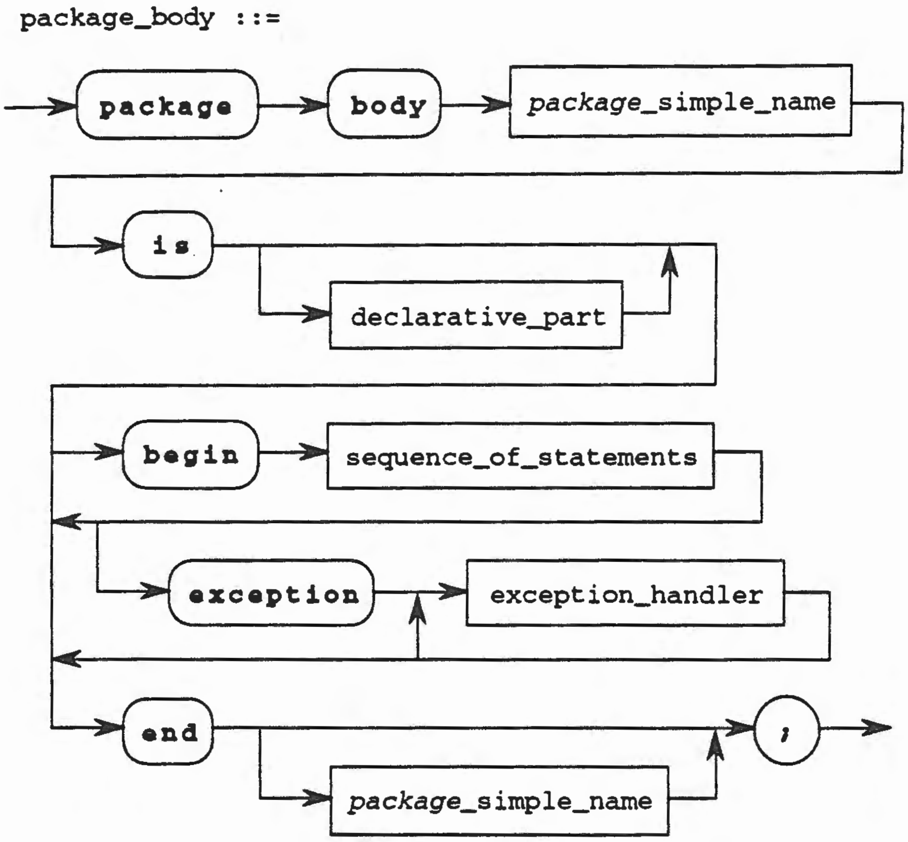


Ada Package Specification
Syntax Chart 8.1

The following syntax charts describe a package body, an entity that we will explain fully in this section.

```
package_body ::= package body package_simple_name is
   [declarative_part] [begin sequence_of_statements
   [exception exception_handler {exception_handler}]]
   end [package_simple_name];
```

Ada Package Body
Syntax Definition 8.2



Ada Package Body
Syntax Chart 8.2

8.1.2 Package Specifications and Bodies

As any other programming unit, packages are composed of two components, a *specification* and a *body*. The specification of the packages serves as an interface for using units. Everything that is to be made available for external users of this package is listed in the package specification. It summarizes the things that this package will export. It is, in that sense, a contract with the user, promising the operations that will be provided by the package and listing any types or objects that are available to use the operations.

The other part of the package is the body. It is in the body that the actual implementations of the subprograms are written. The body is where the *work* of the package is done that was promised in the specification. Thus, in the body of a package we can expect to see the bodies of all of the subprograms that had specifications in the package specification. Additionally, the body can be used to define other subprograms that are used in the body of the package, but that are not exported for the use of the using programs.

Packages should be thought of as architectural tools. They allow the program designer to structure the system. All of the items that are logically related can then be physically associated so as to contain them all in a single logical unit. This facilitates the design of large software systems and encourages the reusability. Reusability is accomplished because the packages are logically complete and can be thought of as building blocks that compose large systems. As such, they are somewhat like the integrated circuits "chips" that hardware designers use. A complete package can be reused in a system design in the same manner that an IC chip is reused in a hardware design.

Another property of packages, shared by all Ada program units, is that the specification and body are separately compilable. This means that we can compile the specification without having written the body. This allows the system designer to describe all of the interfaces in the system under development before a single line of executable code has been written. The compiler will check the interfaces and guarantee that everything is correct and that no interface problems will occur. All of this is done without a single line of executable code! Note that the guarantee is simply that the number, order, and type of the parameters is correct and that all cross-module references are accurate. It does not guarantee that the functionality to be provided will be the one that the user expects.

Perhaps an example will serve to illustrate these points. Consider the following package:

```
package Geometric_Figures
is
  procedure Circle (Radius : Positive);
  procedure Square (Side : Positive);
  procedure Rectangle (Side_1, Side_2 : Positive);
  procedure Parallelogram (Side_1, Side_2 : Positive;
    Angle : Degrees);
end Geometric_Figures;
```

In this example, we have a package that contains procedures for drawing several different kinds of geometric shapes. All of these entities are logically related and thus we have physically related them by packaging them up into a single program unit with the name Geometric_Figures. Our "contract" with the user promises that we will draw these four figures, as long as they provide us with the necessary parameters.

Now this package specification could be compiled and placed in the library. A using program could then make use of this package as follows:

```
with Geometric_Figures;
procedure Draw_Pictures
is
  Geometric_Figures.Circle(5);
  Geometric_Figures.Square(10);
end Draw_Pictures;
```

In this example, the using program (Draw_Pictures) uses the facilities of the Geometric_Figures package to draw a circle and a square. Note that this is accomplished by merely requesting the the compiler import this package from the library (that is the purpose of the clause with Geometric_Figures). Now we merely specify the name of the procedure we want to execute and provide it with appropriate parameters. When this is compiled, the compiler will verify that the subprograms that we have used do exist in the visible portion of the package Geometric_Figures and that the number, order, and type of the parameters is consistent. In this case, the code segment will meet these requirements, and thus successfully compile. Note that the compilation is successful even though we have yet to write the body! This is because we are only concerned now that the interfaces are correct and these are specified in the specification. We do not need the bodies until we want to link the code together.

Thus, we now would write the body of the package Geometric_Figures. It might look like this:

```
package body Geometric_Figures
is
  procedure Circle (Radius : Positive)
  is
    -- actual code to draw the circle would need to be placed here
    null;
  end Circle;

  procedure Square (Side : Positive)
  is
    -- actual code to draw the square would need to be placed here
    null;
  end Square;

  procedure Rectangle (Side_1, Side_2 : Positive)
  is
    -- actual code to draw the rectangle would need to be here
    null;
  end Rectangle;

  procedure Parallelogram (Side_1, Side_2 : Positive,
    Angle : Positive)
  is
    -- actual code to draw the parallelogram would need to be here
    null;
  end Parallelogram;

end Geometric_Figures;
```

In this example, the subprogram bodies have not been completed because their contents are immaterial to this discussion. However, you can see where the body of each subprogram in this package would be filled out. Note that the package body can be identified by the word body being included in the first line. The name of the package specification and the package body must be the same.

The specification and body do not need to be compiled together as indicated above. In addition, these two portions of the package may be in different external files. When they are compiled, however, they will be placed in the same library.

In some instances it makes sense to combine in one place all of the logically related constants that a project may use. In this case, there are no subprograms that need to be implemented. Consequently, the specification is all that is needed. The body is not needed. For example,

```
package Project_Constants
is
  e  : constant := 2.71828;
  Pi : constant := 3.14159;
  Golden_Ratio : constant := 1.61803;
  Radius_of_Earth : constant := 6378.388;  -- km
end Project_Constants;
```

This example demonstrates the case where no package body is needed. Everything exported by this package is completely defined in the package specification and so no package body is needed.

Another example can be used to demonstrate the case where subprograms are available in the package body that are not exported by the package specification. For example,

```
package Robot_Control
is
  type Speed is range 0..100;
  type Distance is range 0..500;
  type Degrees is range 0..359;
  procedure Go_Forward (How_Fast : in Speed;
                        How_Far : in Distance);
  procedure Reverse_Direction (How_Fast : in Speed;
                               How_Far : in Distance);
  procedure Turn (How_Much : in Degrees);
end Robot_Control;
```

This package specification exports the functions necessary to control a small robot. In addition to some special types defined to represent the limits of the robot's activities, such as Speed, Distance, and Degrees, there are subprograms that can manipulate the robot. These provide the operations of Go_Forward, Reverse_Direction, and Turn. This package represents the interface by which the users of this robot can cause the robot to perform the actions necessary to solve their problems. We could compile this into our library and make use of it in programs designed to manipulate the robot. For example, the following program uses the Robot_Control package to make the robot move in a square:

```
with Robot_Control; -- Provides access to Robot_Control
procedure Square is
begin
  Robot_Control.Go_Forward (How_Fast => 100,
                            How_Far => 20);
  Robot_Control.Turn (How_Much => 90);
  Robot_Control.Go_Forward (How_Fast => 100,
                            How_Far => 20);
  Robot_Control.Turn (How_Much => 90);
  Robot_Control.Go_Forward (How_Fast => 100,
                            How_Far => 20);
  Robot_Control.Turn (How_Much => 90);
  Robot_Control.Go_Forward (How_Fast => 100,
                            How_Far => 20);
  Robot_Control.Turn (How_Much => 90);
end Square;
```

We could compile this program and place it in the library even though the body of the Robot_Control package had not been written. We could not link these programs, however, until we supply the body that will describe how we will accomplish the actions of Go_Forward, Reverse_Direction, and Turn. Thus, the following body could be the one created to provide these functions:

```
package body Robot_Control is
  --local declarations

procedure Reset_System is
begin
  --implementation
end Reset_System;

procedure Go_Forward…is…
procedure Reverse_Direction…is…
procedure Turn…is…
end Robot_Control;
```

In this example package body, the details of how each of the subprograms is implemented are not important and so these are shown with ellipses (...). The point to be made is that they would need to be implemented in the package body. Note also that in the package body there is a subprogram, `Reset_System`, that is implemented, yet it does not appear in the package specification. This is an example of a subprogram that is needed by the other subprograms in the body in order to make their implementation easier or more efficient, yet it is not exported for use by the general user. As you can guess, the author of this package has decided that the user does not need the capability of resetting the system and thus has chosen not to export it. However, some of the subprograms may need this capability, especially in an error situation (an exception has been raised) and so the functionality must be provided.

This example serves to illustrate an asymmetry in the relationship between specifications and bodies of packages. Any and all subprograms listed in the specification of a package *MUST* be implemented in the body of the package. However, there may be subprograms implemented in the body of the package that are not listed in the specification of the package.

8.1.3 Initialization of Packages

It is sometimes necessary to have objects that have known initial values when a system is started. In older languages this is accomplished by having the first several statements in the program assign initial values to the objects in the system. However, this practice tends to have a detrimental effect on maintenance because it clutters up the main portion of the algorithm; it hides the algorithm being implemented in several extraneous statements serving merely to initialize the objects that will be manipulated in the algorithm and make up the solution.

We already know one way in Ada to avoid this problem and that is to provide initial values for all objects at the time that they are declared. Thus, we might have a statement such as

```
Items_In_Stack : NATURAL := 0;
```

where we have declared the object `Items_In_Stack` to be an object with the properties of a NATURAL number and have initialized it to the value zero. This makes sense because it says that when we start the system we have no values in the stack initially.

While this approach works for static initial values, it does not work for objects whose initial values are not known at compile time or are not constants. A possible work-around for this situation is to get the value that is needed and then enter a declare block, using the value obtained to provide an initial value. Thus, we might have the following code segment,

```
   TEXT_IO.Put (Item => "Enter the maximum number of characters => ");
   TEXT_IO.Get (Item => Max_Characters);
   declare
     Name : STRING (1 .. Max_Characters);
   begin
     -- use the string Name
```

This example shows that we can prompt the user for the value that will be needed to constrain the length of the sting. We are able to do this by entering a declare block after we had obtained this information and then using the value that we had already obtained from the user. While this approach avoids the issue of having to have all known values at compile time, it requires that we again wait until execution to get these values and thus suffers from the same problems as previously described for older languages where the initial values are obtained during the execution of the algorithm. This complicates maintenance.

Thus, we have a dilemma. How can we provide initial values for objects declared in a package , but not use time during execution? The answer is that we do it during *elaboration*. Elaboration is that process that must occur just prior to execution in which the declarations achieve their effect. That is to say, when we declare a variable such as the previously described `Items_In_Stack`, the runtime system must allocate a space in memory for this variable and must provide its initial value. This does not actually occur at execution time since we are free to assume that it has already occurred at that point. It is actually done whenever the block containing that declaration is entered, just prior to the actual execution of that block. All of the machine level actions that are required to allocate the memory and return the address, as well as to set up initial values prior to execution, is called elaboration. Traditional languages do not mention this process, but rather lump it into the overall process called execution. Ada makes a distinction because there are things that occur at elaboration that can't occur at execution and *vice versa*. All actions in Ada are defined as to their effect at elaboration time and at execution time. We will return to this subject in detail later. For now, let's see how we can use the elaboration process to help us initial objects in a package.

Recall that a package is a program unit with both a specification and a body. The body is where any types and objects needed by the body and not exported to the user are declared. In addition, all subprograms specified in the package specification just be implemented in the body, as well as any subprograms not exported to the user but provided for the use of the subprograms implemented in the body. In addition to all of these things, there is another part of the package body that we have not yet discussed. It is the initialization portion. This portion of the package body is physically the last thing in the package body. It starts with the word begin and ends with the end of the package. This region is called the initialization portion of the package body. Let's see an example before we describe it further. This,

```
   package body Robot_Control is
     --local declarations

     procedure Reset_System is
     begin
       --implementation
     end Reset_System;

     procedure Go_Forward…is…
     procedure Reverse_Direction…is…
     procedure Turn…is…
   begin
     Reset_System;
   end Robot_Control;
```

In this package body we have included as the last thing in the body a `begin` and a sequence of statements, in this case only a single statement that happens to be a call to the subprogram `Reset_System`. This will cause this package body to execute the procedure `Reset_System` whenever the package is elaborated, which is whenever this package is mentioned in a `with` clause in some using program unit. Thus, every time a user wants to use this package, the system will be reset so that its initial state has some known value.

There are no limits on what kinds of statements may be placed in the initialization portion of the package body. We have already seen that this portion of the package body is completely optional, in which case the **begin** is not even provided. This portion of the package body may contain input/output statements, it may have loops, subprogram calls; in short, anything that is legal anywhere else. However, its intended use is to provide initial values for objects needed in the package without requiring the user to have to call an initialize routine prior to using the package. In this course, that is how you should use the initialization portion of the package body.

8.1.4 State Variables in Packages

Objects that are declared in a package body have another property that makes them unique among objects declared in other program units. This property is that these objects retain their values between uses. For this reason, we refer to these objects as *state variables* because they retain their state, or value, between uses.

An example may serve to clear up any confusion that exists on this point. Consider the following subprogram,

```
with TEXT_IO;
procedure Demonstrate
is
  This_Value : NATURAL;
begin
  TEXT_IO.Get (Item => This_Value) ;
end Demonstrate;
```

This simple procedure gets a new value for `This_Value` every time that the procedure is invoked. Every time that the procedure is called, the object `This_Value` is elaborated and then during execution, provided with a value by the `Get` procedure.

Consider now this nested set of procedures, making use of a non-local reference,

```
with TEXT_IO;
procedure More_Demo
is
  This_Value : NATURAL;
  package Int_IO is new TEXT_IO.INTEGER_IO(NATURAL);

  procedure Nested
  is
    New_Value : NATURAL;
  begin
    Int_IO.Get (Item => New_Value);
    This_Value := New_Value;
  end Nested;
```

```
begin
  This_Value := 10;
  Silly_Loop:
  for Count in 1 .. 10
  loop
    Int_IO.Put(Item => This_Value);
    Nested;
  end loop Silly_Loop;
end More_Demo;
```

In this example, the nested procedure called `Nested` references a non-local object, namely `This_Value`. `This_Value` is non-local because it is not declared within the procedure `Nested`. It is visible because of the scope and visibility rules that were discussed earlier. This use of the object `This_Value` is considered bad programming style because of the potential for uncontrolled change of this object. It is often called a *global variable* or object because it is visible globally and has the potential to be changed by any executable statement that has visibility to it, *i.e.*, all executable statements.

Now let's consider a similar situation within a package body. Suppose that we have a stack package that defines and manipulates a bounded stack, *i.e.*, a stack that has a limited number of positions in which we may push values. A skeleton for this package body may be as follows,

```
package body Stack
is
  Top_of_Stack : NATURAL := 0;

  procedure Push (The_Item : NATURAL;
                  The_Stack : in out Stack)
  is
    Top_of_Stack := Top_of_Stack + 1;
    The_Stack(Top_of_Stack) := The_Item;
  end Push;

  procedure Pop (The_Item : out NATURAL;
                 The_Stack : in out Stack)
  is
    The_Item := The_Stack(Top_of_Stack) ;
    Top_of_Stack := Top_of_Stack - 1;
  end Pop;

-- Other stack manipulation subprograms implemented here

end Stack;
```

Note that in this example, `Top_of_Stack` is an object that is non-local to both of the procedures illustrated. However, it is not considered to be a global variable. This is the case because this object is not available to the user of the package. This insures that only the implementor of the package body has access to this object and the degree of control is greater than it would be if this were a global variable. We also can see from this example that it sis desirable for the object to retain its value between invocations of these procedures. If it did not, then the user would need to maintain this information in the using procedure and pass it as an additional parameter. This would make the abstraction that we are attempting to create less useful, especially in a reusability context. Consequently, it is desirable for this non-local (at least to the package body) object to retain its value. In a sense, it has retained *state information*. For this reason, objects used in this manner in a package body are referred to as *state variables*. We will see more state variables later in this course.

8.1.5 Exercises

<< To be added.>>

8.2 Ada Procedures

In Ada, there are two program units that serve as abstractions of operations that are performed on data. Since many of the ideas and concepts in these two program units are so similar, it is convenient to have a single name by which to reference both. This term is subprogram and it is used when we want to talk collectively about procedures and functions. In some sense, procedures and functions are the building blocks of Ada. At the lowest level, they are the active program units used in our algorithms to solve users problems. They are often grouped together by logical function into larger units that will be discussed later. For now, let's examine these very powerful program units by studying each of them in turn.

8.2.1 Procedure Specifications and Bodies

A **procedure** is an abstraction of an operation. In Ada, a procedure is one of two kinds of structures that make up a program unit called a **subprogram**. Procedures perform an action on behalf of a calling unit by executing the sequence of statements that make up their body. A procedure is a statement and can be used anywhere that a statement can be used. Procedures may also have parameters that allow them to be individualized to perform differently under different circumstances.

A procedure is a sequence of statements that perform an action. In Ada, a procedure may be the **main program** or it may be a subordinate unit called on behalf of another program unit. In fact, the same procedure may be used in both ways because Ada does not require anything to distinguish a main program, other than the fact that if it is a procedure than it must be a parameterless procedure.

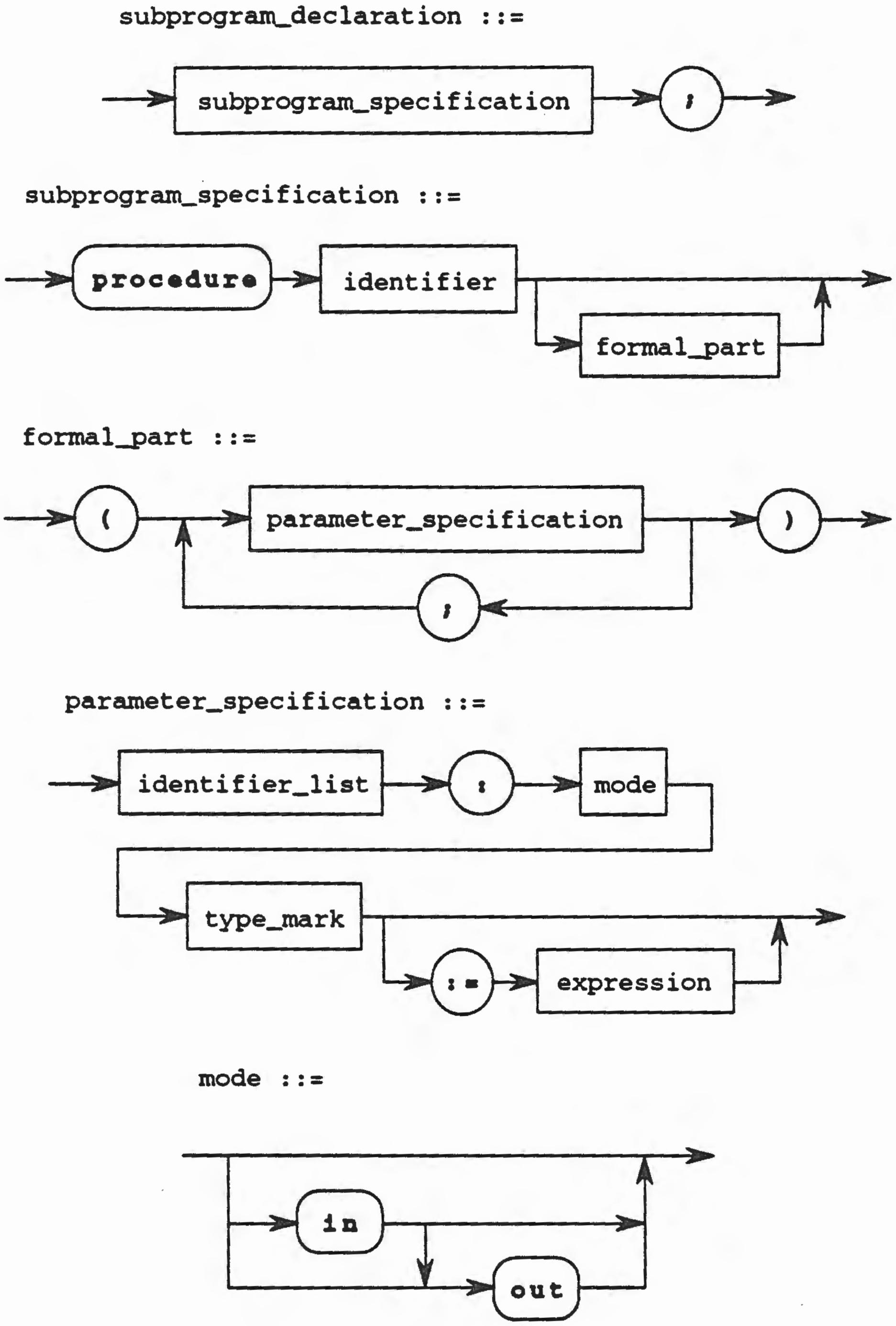
The syntax for a procedure, and for all program units, is actually composed of two parts. The first part is called the procedure **specification** and contains all of the information needed by other program units in order to make use of the procedure. The specification tells *what* the procedure does; it does not say *how* it does it. The syntax for a procedure specification is given in Syntax Definition 8.3.

8.2.1.1 Procedure Specifications

```
subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
   procedure identifier [formal_part]

formal_part ::=
   (parameter_specification {; parameter_specification})

parameter_specification ::=
   identifier_list : mode type_mark [:= expression]

mode ::= [in] | in out | out
```

Procedure Specification
Syntax Definition 8.3

This syntax may also be expressed in graphical form as given in Syntax Chart 8.3 below.



Procedure Specification
Syntax Chart 8.3

For example, the following are three different procedure specifications. The first,

```
procedure Draw_Circle;
```

is an example of a procedure without any parameters. This procedure will perform the same action each time that it is invoked (called) and cannot be tailored or modified by the user. Parameters, which will be discussed more fully in the next section, allow the user to make certain changes or modifications to the actions to be performed by the procedure, allowing more flexibility and adaptability.

The second example,

```
procedure Put (Item : in CHARACTER);
```

is a specification for the procedure Put in the package TEXT_IO. It contains all of the information needed by other program units to use this procedure. It specifies the name (Put) and the parameters to be used, namely one formal parameter that has the name Item and the type CHARACTER. The mode of this formal parameter is specified to be in. A full discussion of the meaning of the modes for parameters will be deferred until the next section of this chapter.

As another example,

```
procedure Sum (First_Number, Second_Number : in INTEGER;
               Result : out INTEGER);
```

is a procedure specification for a procedure named Sum. It requires three parameters, all of which are of type INTEGER. The mode of the first two parameters is in, while the mode of the third parameter is out. The formal names of the parameters are also provided by the procedure specification, in this case, First_Number, Second_Number, and Result.

8.2.1.2 Procedure Bodies

Recall that the procedure specification described *what* the procedure does. By itself, a specification can perform no actions because the software engineer has not provided the information about how it does what it does. Therefore, for every procedure specification there must be a second part called the procedure body that describes *how* the procedure does what it does. The syntax for a procedure body is shown in Syntax Definition 8.4. The optional designator after the **end** must be the same as the identifier given after the reserved word **procedure** in the procedure specification.

```
subprogram_body ::= subprogram_specification
                    is
                       [declarative_part]
                    begin
                       sequence_of_statements
                    end [designator];

declarative_part ::=
   {basic_declarative_item} {later_declarative_item}

basic_declarative_part ::= basic_declaration
```

basic_declaration ::= object_declaration | type_declaration |
   number_declaration | subtype_declaration | subprogram_declaration

later_declarative_item ::= body | subprogram_declaration

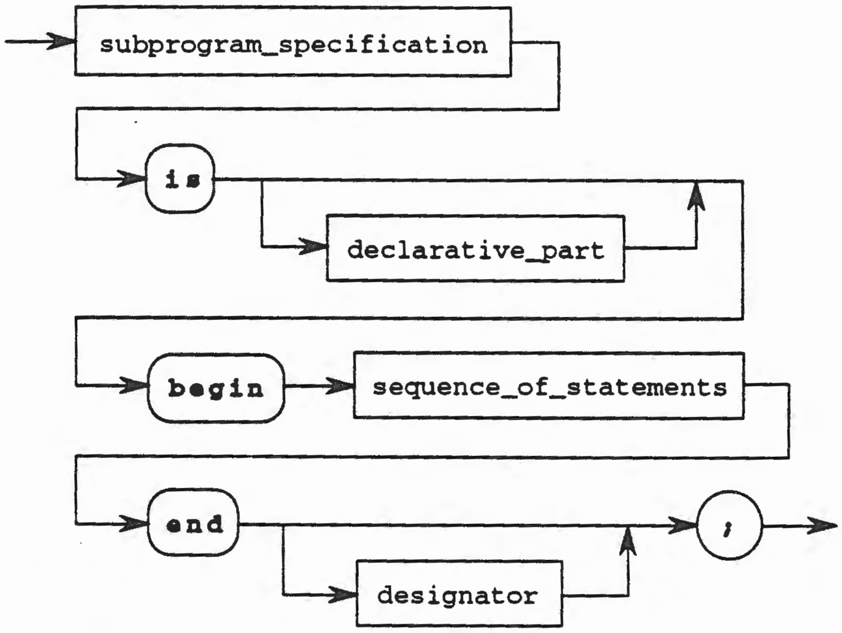
body ::= proper_body | body_stub

body_stub ::= subprogram_specification **is separate;**

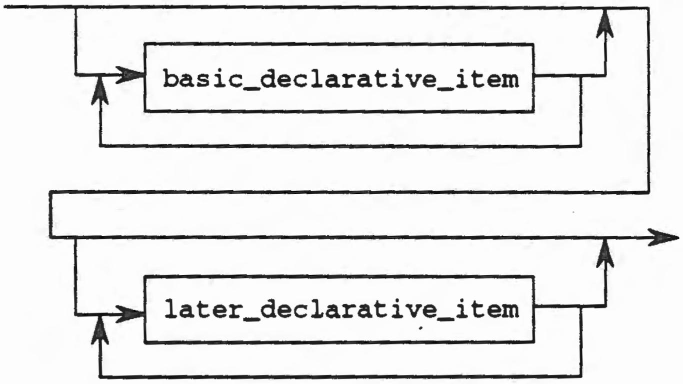
Procedure Body
Syntax Definition 8.4

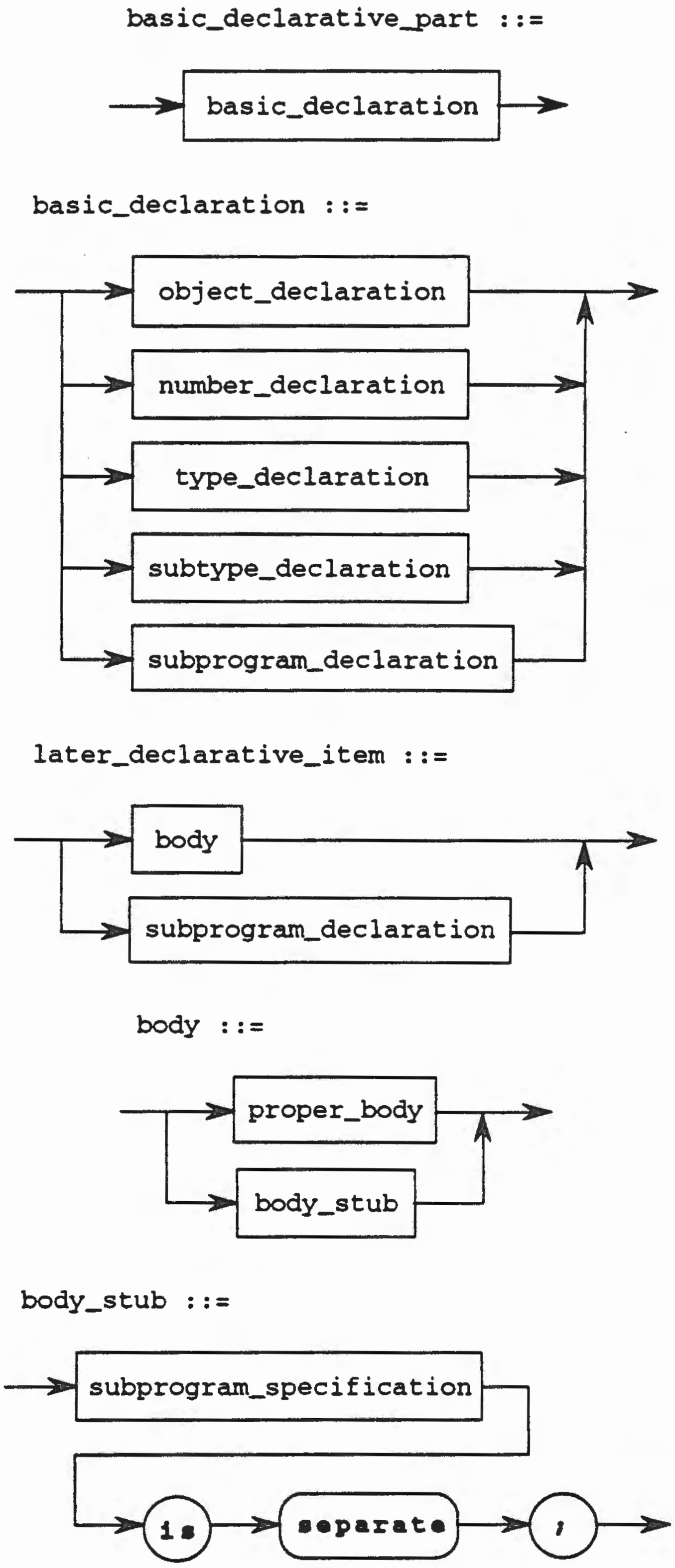
This syntax may also be expressed in graphical form as given in Syntax Chart 8.4 below.

subprogram_body ::=



declarative_part ::=





Procedure Body
Syntax Chart 8.4

For example, the procedure body below

```
procedure Sum (First_Number, Second_Number : in INTEGER;
               Result : out INTEGER)
is
begin
  Result := First_Number + Second_Number;
end Sum;
```

provides a body for the specification of the procedure `Sum` whose specification was provided earlier. It should be clear that the action performed by this procedure is to add the first two parameters and place the result in the third parameter. Thus, in this case, the name of the procedure, `Sum`, is an accurate indication of what the procedure does. In general, however, the name of a procedure is merely an identifier and may not provide much insight into the action performed by the procedure. Therefore, the function of each procedure will need to be determined using the methods already seen in Chapters 3 and 4. It is good practice when creating procedures to name them with a name that is indicative of the action performed by the procedure, but in general, a user cannot depend upon this since there is no way for the machine to enforce the proper naming of a procedure.

Another example of a procedure body is,

```
procedure Traffic_Signals
is
  type Color_Type is (RED, YELLOW, GREEN);
  Signal : Color_Type := GREEN;
  Signal_Light_Activated : BOOLEAN := TRUE;
begin
  Traffic_Control_Processing:
  while Signal_Light_Activated
  loop
    Green_Light_Delay:
    for I in 1 .. 60
    loop
      null;
    end loop Green_Light_Delay;  -- serves to cause a delay
    Signal := YELLOW;
    Yellow_Light_Delay:
    for I in 1 .. 5
    loop
      null;
    end loop Yellow_Light_Delay;  -- serves to cause a shorter delay
    Signal := RED;
    Red_Light_Delay:
    for I in 1 .. 60
    loop
      null;
    end loop Red_Light_Delay;  -- serves to cause a delay
    Signal := GREEN;
  end loop Traffic_Control_Processing;
end Traffic_Signals;
```

which controls the traffic signal at the intersection of two major roads. Note that this procedure performs an action, namely controlling the traffic signal. Note also that the while loop in this procedure is an **infinite loop**, *i.e.*, it runs forever. In some applications having an infinite main processing loop makes sense for the application. This is an example for which that applies because traffic signals must run continuously until they break down or need to be adjusted, in which they case they can be restarted when fixed. Another interesting point is the use of a **for loop** with an embedded null statement. You may wonder why we would loop several times performing a statement that explicitly does nothing. The answer is that we want to cause the system to pause or delay, with the light of the traffic signal a particular color. This is a simple way to make a system delay, although in actuality, the number of times to loop to achieve a useful delay would be much greater than the values illustrated. This need for a system to delay is so frequent that the designers of Ada provided a separate statement to do it. Although we did not use it here, we could have used the single statement,

```
delay 60.0;  -- seconds
```

to achieve our purpose. This statement will be discussed in more detail later. For now, we can say that the delay statement causes the program to be delayed for at least the specified number of seconds.

The purpose of a procedure as mentioned previously is to abstract an operation. This may involve many statements, although in general it is inadvisable to have procedures that are overly long. A procedure should perform a single logical operation. If many logical operations are to be performed then many procedures should be written, *i.e.,* do not have a procedure that performs multiple operations.

It should also be mentioned that the procedure specification is an optional part of the procedure. Since the body of the procedure repeats the specification, no information is lost by allowing the specification to be optional. The reason that specifications exist will become more clear later when mutually referential procedures are discussed. Also, in a package specification only type and object declarations and other program unit specifications may be used. Therefore, in declaring a package that contains procedures, the procedure specifications will be required. But in other circumstances the specification for a procedure is optional.

8.2.2 Procedure Calls

The discussion above addresses how to read (and write) procedures and concerns itself with what their syntax and semantics ought to be. It does not mention how one uses procedures. This is accomplished by a **procedure call**. A procedure call is the means by which the action of the procedure is invoked. It is accomplished by merely naming the procedure within the executable portion of another procedure and supplying the appropriate parameters, if any. For example, to invoke the procedure `Draw_Circle`, a parameterless procedure whose specification appears earlier in this section, a using procedure might look like

```
with TEXT_IO, Draw_Figures;
procedure Demo_Program
is
  Lines : NATURAL;
begin
  TEXT_IO.Get (Item => Lines);
  Center_Circle:
  while Lines > 0
  loop
    TEXT_IO.New_Line;
    Lines := Lines - 1;
  end loop Center_Circle;
  Draw_Figures.Draw_Circle;   -- This is a call to the procedure
                              -- Draw_Circle
end Demo_Program;
```

The action of the procedure Demo_Program is to get the number of Lines to skip in the output file, then enter a loop to skip the appropriate number of lines (which could have been done with a single statement, namely, TEXT_IO.New_Line(Lines)), then call the procedure Draw_Circle. Actually, there are three procedures being used by Demo_Program. The first is the procedure Get from the package TEXT_IO, the next is the procedure New_Line from the same package, and finally, the procedure Draw_Circle is called. Thus, using a procedure is a very simple matter of specifying its name.

The actual sequence of events that occur when a procedure is called will not be detailed here. Suffice it to say that the effect is as if the lines of code of the called procedure were textually substituted at the point of the call.

Consider now the procedure Sum declared above. A call to this procedure might be as follows,

```
with TEXT_IO, Math_Routines;
procedure New_Demo_Program
is
  Number_1, Number_2 : INTEGER;
  Answer             : INTEGER;
  package Int_IO is new TEXT_IO.INTEGER_IO (INTEGER);
begin
  Number_1 := 5;
  Number_2 := 7;
  Math_Routines.Sum (First_Number  => Number_1,
                     Second_Number => Number_2,
                     Result        => Answer);
  Int_IO.Put (Item => Answer);
end New_Demo_Program;
```

This program assigns values to two INTEGERs and then calls the procedure Sum. The call to Sum must include values for the parameters that are needed by Sum, namely two INTEGERs with values, and another INTEGER object to hold the computed result. In the call to Sum inside of the procedure New_Demo_Program, the **formal** parameters that were declared in the specification (and body) of the procedure Sum are listed followed by the **actual** parameter provided by the calling procedure New_Demo_Program. The association between these two parameters (formal and actual) is explicitly indicated by the arrow (=>) symbol. Details of parameter passing will be explained in the next section.

8.2.3 Exercises

<< To be added.>>

8.3 Ada Functions

As was previously presented, Ada subprograms are of two distinct types, namely procedures and function. Procedures are statements that perform a single action and were previously described. A function is also an abstraction of an operation. In Ada, it is a program unit that returns a value. It is therefore always a part of an expression and may be used wherever expressions are allowed. It is not a statement and may not appear in places where a statement is needed. In this section, Ada functions will be introduced and comparisons will be made to procedures. Examples of functions will be given to assist in presenting many of the concepts that are introduced.

An Ada function is a program unit that returns a value. The syntax of a function, like that of a procedure is composed of two parts. The function **specification** contains all of the information that a calling program unit needs to make use of the function. It tells *what* the function does, but not *how* it does it.

8.3.1 Function Specifications

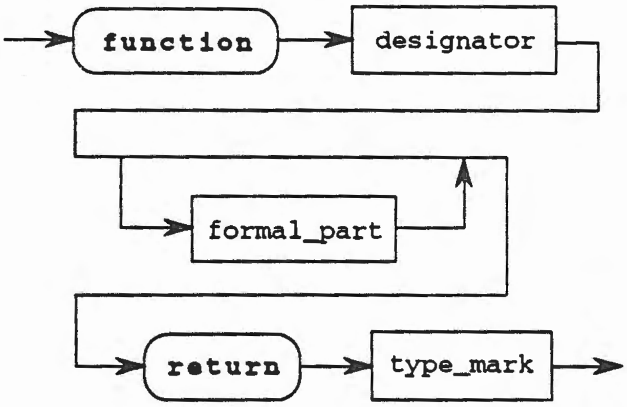
The syntax for a function specification is given in Syntax Definition 8.5.

```
subprogram_specification ::=
   function designator [formal_part] return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

formal_part ::=
   (parameter_specification {; parameter_specification})

parameter_specification ::=
   identifier_list : mode type_mark [:= expression]

mode ::= [in]
```

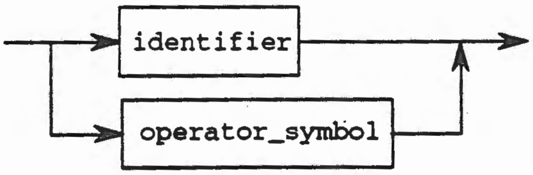
Function Specification
Syntax Definition 8.5

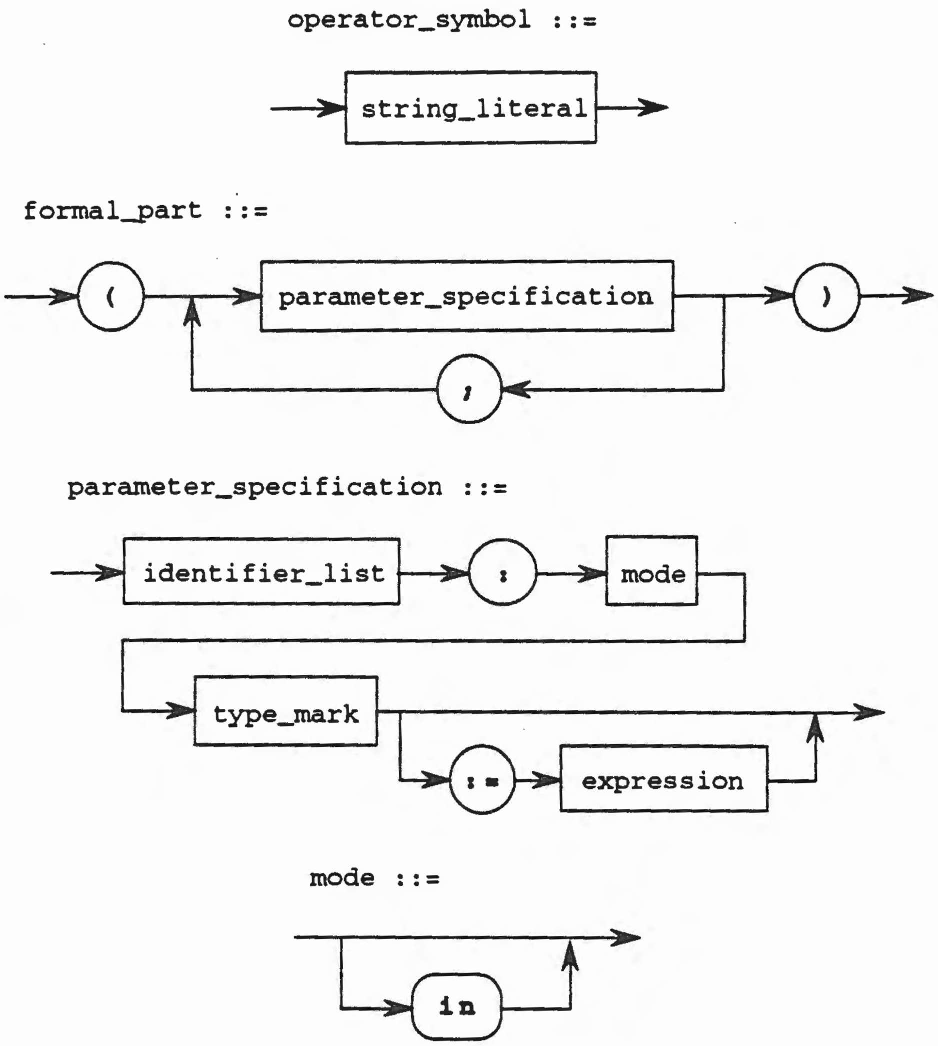
This syntax can also be expressed in graphic form as indicated in Syntax Chart 8.5.

subprogram_specification ::=



designator ::=





Function Specification
Syntax Chart 8.5

For example, the following are three different function specifications. The first,

```
function Clock return Time;
```

is a parameterless function that is actually declared inside of a predefined package named `CALENDAR`. This specification indicates that a call to the function `Clock` will return a value of type `Time`.

The second example,

```
function Increment (Old_Value : INTEGER) return INTEGER;
```

is a specification for a function named `Increment` that takes a single parameter of type INTEGER and returns a value of type INTEGER. Note that there is no guarantee from the name of the function that it will perform any particular operation. The name of the function is, according to Syntax Definition 8.5, merely a designator, which is either an identifier or an operator symbol. Thus, the name of the function does not necessarily by itself provide any indication of what the function actually does. On the other hand, it is a good engineering practice to always name subprograms with an identifier that accurately reflects the operation that they perform.

As another example,

```
function Dot_Product (Left, Right : in VECTOR) return INTEGER;
```

is a function specification for the function `Dot_Product`. It has two parameters, both with mode **in**, of type VECTOR, with formal names `Left` and `Right`, and it returns a value of type INTEGER.

It can be seen that function specifications convey all of the information that a calling program unit needs to use the function, *i.e.*, it completely specifies the interface needed to use the function. However, function specifications do not convey any information as to *how* they do what they do; that is the purpose of the function body.

8.3.2 Function Bodies

As was the case with procedures, a function specification requires that a function body be provided to perform the action "promised" in the function specification. Thus, the second part of every function is the function body. The syntax for a function body is identical to the syntax for procedure body, shown in Syntax Definition 8.4. The optional designator after the **end** must be the same as the designator given after the reserved word **function** in the function specification.

For example, the function body below

```
function Increment (Old_Value : INTEGER) return INTEGER
is
  return Old_Value + 1;
end Increment;
```

provides the body for the function `Increment` specified above. The action performed by this function is add one to the parameter supplied and return the resultant sum.

Another example of a function body is

```
with TEXT_IO;
function Dot_Product (Left, Right : Vector) return INTEGER
is
  Sum      : INTEGER := 0;
  Final_Value : INTEGER;
begin
  if Left'FIRST = Right'FIRST and Left'LAST = Right'LAST
  then
    Dot_Product_Computation:
    for I in Left'RANGE
    loop
      Sum := Sum + Left(I) * Right(I);
    end loop Dot_Product_Computation;
    Final_Value := Sum;
    else
      TEXT_IO.Put (Item => "Error - cannot compute Dot " &
                            "Product.");
      Final_Value := 0;
    end if;
    return Final_Value;
  end Dot_Product;
```

This function computes the dot product of two vectors, where the type `Vector` is declared in some enclosing scope. It checks to see that the vectors are of the same length and have the same upper and lower bounds, then computes the dot product if the test is true and writes an error message if it is false. Note that if the test is false and the dot product cannot be computed the function still provides a value to the calling program unit. There are many other ways to handle a situation such as this, but for now this solution should suffice.

Functions should perform a single logical operation. The size of a function then is related to the logical operation that is to be performed. In general, however, functions should be relatively small and should only perform the logical operation that is required. Any extraneous operations are likely to be side-effects of the function and should be avoided.

As was the case for procedures, a function specification is optional. All of the requisite interface information can be obtained from the function body since the first portion of the function body repeats the function specification. There are certain circumstances that will be discussed later that require a function specification, but in general they are optional.

8.3.3 Function Calls

All of the discussion of functions so far has been concerned with their definition. Much has been made of the similarity of function to procedures. Since these are two forms of subprograms, each has its intended purpose and is suited to different uses. Procedure calls, the means to invoke a procedure, were discussed in the previous section. A function call is, similarly, the means to invoke a function. It is accomplished by naming the function in any place where an expression is permissible in the syntax, including as part of a larger expression. The function's parameters, if any, are indicated in parenthesis in a manner consistent with that for parameter passing in procedures. For example, to use the parameterless function `Clock` whose specification appears earlier in this section, a using program unit would look like,

```
with TEXT_IO; with CALENDAR;
procedure Demo_Clock
is
  The_Time : CALENDAR.TIME;   -- the type TIME is declared in
                              -- the package CALENDAR
  The_Day  : CALENDAR.DAY_NUMBER;  -- similarly for DAY_NUMBER
begin
  The_Time := CALENDAR.Clock;
  The_Day  := CALENDAR.Day (Date => The_Time);
  TEXT_IO.Put (Item => "The day of the month is ");
  TEXT_IO.Put (Item => The_Day);
end Demo_Clock;
```

where the package `CALENDAR` referenced in the context clause (`with CALENDAR;`) is a predefined package in Ada that provides date/time information and operations. You should look at Chapter 9 of the Reference Manual for the Ada Language [LRM 9.6, paragraph 7, page 9–11], to see the complete specification of this package. The procedure `Demo_Clock` makes use of two functions inside of the package `CALENDAR`, namely `Clock` and `Day`. `Clock` is a function that returns an object of type TIME and Day is a function that takes a parameters of type TIME and returns an object of subtype DAY_NUMBER, which is defined in package CALENDAR to be a subtype of INTEGER. Thus, this procedure calls Clock to get an object of type TIME, called The_Time in this procedure, then passes that object as a parameter to the function Day to get an object of type DAY_NUMBER. Since this is just an INTEGER subtype, the TEXT_IO package can write out the result, which will be the current day of the month in the range 1 .. 31.

Note that the function calls are not statements on a line by themselves, but they are part of an expression (in this case they are the complete expression) that must be evaluated to determine the value to assign to the objects on the left side of the assignment statement. Compare their use with that of the procedure TEXT_IO.Put that is called in this same procedure.

The actual sequence of events that occur when a function is called is beyond the scope of this discussion, but it is sufficient to say that the effect is the same as if the lines of code that make up the function body were textually substituted at the place of the call.

Consider now the function Increment that was declared above. This function could be used as,

```
with TEXT_IO;
procedure Demo_Increment
is
  A_Value : INTEGER;

  function Increment (Old_Value : INTEGER) return INTEGER
  is
    return Old_Value + 1;
  end Increment;

begin
  TEXT_IO.Get (Item => A_Value);
  TEXT_IO.Put (Item => "The original value is ");
  TEXT_IO.Put (Item => A_Value);
  A_Value := Increment (Old_Value => A_Value);
  TEXT_IO.Put (Item => "The new value is ");
  TEXT_IO.Put (Item => A_Value);
end Demo_Increment;
```

which embeds the function in a procedure. The procedure gets a value and writes it out with an appropriate message. Then it computes a new value for A_Value by calling the function Increment and passing the function as a parameter the old value of A_Value. The function increments A_Value by one and returns the result which can then be assigned as the new value of A_Value. Finally, the new value is written out along with an appropriate message.

Another use of Increment might be,

```
with TEXT_IO;
procedure Demo_Increment_2
is
  A_Value : INTEGER := 10;

  function Increment (Old_Value : INTEGER) return INTEGER
  is
    return Old_Value + 1;
  end Increment;
```

```
begin
  TEXT_IO.Put (Item => "The original value is ");
  TEXT_IO.Put (Item => A_Value);
  A_Value := 7 * Increment (Old_Value => A_Value);
  TEXT_IO.Put (Item => "The new value is ");
  TEXT_IO.Put (Item => A_Value);
end Demo_Increment_2;
```

which is similar to the previous use. The difference is found in the initialization of the object `A_Value`, eliminating the call to the procedure `TEXT_IO.Get`. The value 10 is written out along with an appropriate message. Next, the function `Increment` is called and passed as a parameter the current value of `A_Value`. The function `Increment` increments this object and returns the result which is 11. Finally, this result is multiplied by the other operand, 7, and the final result of 77 is assigned to the object `A_Value`. This value is written out along with an appropriate message. Note that in expressions, functions have a higher precedence than any of the other operators.

Finally, the function `Dot_Product` declared above could be used in the following manner,

```
with TEXT_IO;
procedure Vector_Operations
is
  type VECTOR is array (INTEGER range <>) of INTEGER;
  Distance, Time : VECTOR (1 .. 50);
  Total : INTEGER;

  function Dot_Product (Left, Right : Vector) return INTEGER
  is
    Sum     : INTEGER := 0;
    Final_Value : INTEGER;
  begin
    if Left'FIRST = Right'FIRST and Left'LAST = Right'LAST
    then
      Dot_Product_Computation:
      for I in Left'RANGE
      loop
        Sum := Sum + Left(I) * Right(I);
      end loop Dot_Product_Computation;
      Final_Value := Sum;
    else
      TEXT_IO.Put (Item => "Error - cannot compute Dot " &
                           "Product.");
      Final_Value := 0;
    end if;
    return Final_Value;
   end Dot_Product;

begin
  Distance := (others => 25);
  Time     := (others => 35);
  Total    := Dot_Product (Left  => Distance,
                           Right => Time);
  TEXT_IO.Put (Item => "The dot product is ");
  TEXT_IO.Put (Item => Total);
end Vector_Operations;
```

This example is somewhat contrived to illustrate the use of the function Dot_Product, but it meets the need. Two vectors, Distance and Time are given initial values, and then passed as parameters to the function Dot_Product. Dot_Product computes the dot product of the two vectors and returns the value that is assigned to the object Total. This value is then written out with an appropriate message.

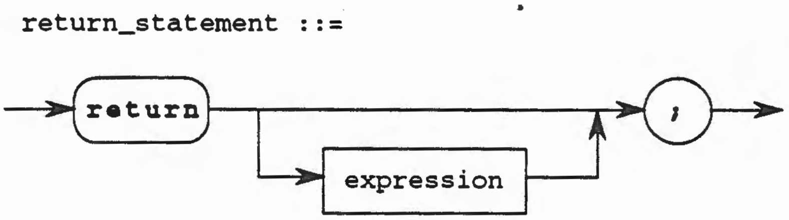
8.3.4 Return Statements

A **return** statement consists of the reserved word **return** followed by an expression, followed by a semicolon (;). The **return** statement has different meanings and rules depending upon whether it is used in a function subprogram or a procedure subprogram. In this section we will only discuss the use of the **return** statement for function subprograms. The syntax of the **return** statement for both functions and procedures is given in Syntax Definition 8.6.

```
return_statement ::= return [expression];
```

Return Statement
Syntax Definition 8.6

In graphic form this syntax can be expressed as in Syntax Chart 8.6.



Return Statement
Syntax Chart 8.6

For a function, the purpose of the **return** statement is to identify what value will be returned as the result of a function call. There can be as many **return** statements in a function as desired, but the first one encountered in the normal flow of control will be the one executed. While this is true as far as the syntax of the language, it is considered bad engineering practice to have more than a single entry and a single exit for each subprogram. Thus, we should not have more than a single return statement in any function. The **return** statement terminates the function normally by returning the value computed in the expression after the reserved word **return**. Any other statements that exist after that point are never executed. Each function *must* have at least one **return** statement. If a function should ever reach the **end** statement in its body without having encountered a **return** statement, then the exception Program_Error is raised.

For example, consider the following function,

```
function Compare_Grades (My_Grade   : CHARACTER;
                         Your_Grade : CHARACTER) return BOOLEAN
is
begin
  return My_Grade > Your_Grade;
end Compare_Grades;
```

This function takes two parameters and returns a BOOLEAN result indicating whether or not the first parameter is greater than the second. Note that the **return** statement itself computes the expression to be returned.

Now consider a slightly more complex function using the following type declarations which are contained in an outer scope,

```
type All_Colors is (RED, YELLOW, ORANGE, UNDEFINED);
subtype Colors is All_Colors range RED .. ORANGE;

function Mix_Colors (First_Color  : Colors;
                     Second_Color : Colors) return All_Colors
is
begin
  if First_Color = RED
  then
    if Second_Color = YELLOW
    then
      return ORANGE;  -- RED and YELLOW make ORANGE
    else
      if Second_Color = RED
      then
        return RED;  -- RED and RED make RED
      else   -- Second_Color must be ORANGE
        return UNDEFINED; -- not a possible combination
      end if;
    end if;
  else
    if First_Color = YELLOW
    then
      if Second_Color = RED
      then
        return ORANGE;  -- YELLOW and RED make ORANGE
      else
        if Second_Color = YELLOW
        then
          return YELLOW;  -- YELLOW and YELLOW make YELLOW
        else -- Second_Color must be ORANGE
          return UNDEFINED;  -- not possible
        end if;
      end if;        else  -- First_Color must be ORANGE
      return UNDEFINED;  -- must be RED or YELLOW to mix
     end if;
  end if;
end Mix_Colors;
```

This function mixes two primary colors, *i.e.*, RED and YELLOW, to obtain a secondary color, ORANGE. If either of the parameters are not a primary color, then the mix is undefined. If both of the parameters are the same primary color, say RED, then the mix will also be RED. Otherwise, if both parameters are different primary colors, then the resultant mix will be ORANGE. Note that the function has several **return** statements embedded in it. This is permissible since there is no limit on the number of **return** statements allowed. However, there will only be one **return** statement executed and it will be the first one encountered in the normal flow of control. Thus, a portion of the code above, reproduced and modified below,

```
if First_Color = RED
then
  if Second_Color = YELLOW
  then
    return ORANGE;  -- RED and YELLOW make ORANGE
    return RED;  -- never executed
  else ...
```

would be syntactically correct, but the second **return** statement, return RED; would never be executed since the previous statement is a **return** and that would terminate normally the execution of the function such that the second **return** statement would never be in the flow of control.

Further note that there are multiple return statements in this function. Given that this is bad engineering practice, how can we rewrite it so that there is but a single exit? Consider the following version of the subprogram, rewritten so as to contain a single exit,

```
function Mix_Colors (First_Color  : Colors;
                     Second_Color : Colors) return All_Colors
is
  Color_Mix : All_Colors;
begin
  if First_Color = RED
  then
    if Second_Color = YELLOW
    then
      Color_Mix := ORANGE;  -- RED and YELLOW make ORANGE
    else
      if Second_Color = RED
      then
        Color_Mix := RED;   -- RED and RED make RED
      else   -- Second_Color must be ORANGE
        Color_Mix := UNDEFINED; -- not a possible combination
      end if;
    end if;
  else
    if First_Color = YELLOW
    then
      if Second_Color = RED
      then
        Color_Mix := ORANGE;  -- YELLOW and RED make ORANGE
      else
        if Second_Color = YELLOW
        then
          Color_Mix := YELLOW;  -- YELLOW and YELLOW make YELLOW
        else -- Second_Color must be ORANGE
          Color_Mix := UNDEFINED;  -- not possible
        end if;
      end if;        else  -- First_Color must be ORANGE
      Color_Mix := UNDEFINED;  -- must be RED or YELLOW to mix
    end if;
  end if;
  return Color_Mix;
end Mix_Colors;
```

It is imperative that a function have a result to return. Thus, if a function ever reaches the end of its body without encountering a **return** statement, something is wrong. Therefore, an exception, namely `Program_Error`, is raised. For example, consider the original color mixing function again, a modified portion of which is given next,

```
function Mix_Colors (First_Color  : Colors;
                     Second_Color : Colors) return All_Colors
is
begin
  if First_Color = RED
  then
    -- as before, removed to conserve space
  else
    if First_Color = YELLOW
    then
      -- same as before except the else has been removed
    end if;
  end if;
end Mix_Colors;
```

Note that in this skeleton of the function the only values against which `First_Color` is tested are `RED` and `YELLOW` (the else clause of the inner if statement has been removed for this discussion). Now suppose that the value of `First_Color` happens to be `ORANGE`. The flow of control will be to compare `First_Color` to `RED` in the outer **if** statement, then go into the **else** clause because `ORANGE` is not `RED`. The **else** clause of the outer **if** statement contains another **if** statement that compares `First_Color` to `YELLOW`. This will also be FALSE since `ORANGE` is not `YELLOW`. But there is no **else** clause provided for the inner **if** statement. As a consequence, the flow of control will leave the inner **if** statement, then leave the outer **if** statement and come to the end of the function body without encountering a **return** statement. At this point an error has obviously occurred and the run time system will raise the exception `Program_Error`.

Finally, note that there is an expression after each of the **return** statements. For a function, this expression is not optional, *i.e.*, one must be provided for each **return** statement. This is necessary so that the function has a value to return. Thus, for all **return** statements in a function there must be an expression of the type specified to be returned by the function.

In summary, for a function, multiple **return** statements are permissible but considered bad engineering practice; in any case, only the first one encountered in the normal flow of control will be executed. In addition, at least one **return** statement must be encountered before the end of the function body or the exception `Program_Error` will be raised. Also, an expression of the appropriate type must appear in each **return** statement.

8.3.5 Exercises

1. What is the result of the call to the function `Mystery` in the following procedure?

```
with TEXT_IO;
  procedure Mystery_Procedure
  is
     Letter : CHARACTER;

     function Mystery (Old : CHARACTER) return CHARACTER
     is
       return CHARACTER'SUCC (Old);
     end Increment;
```

```
    begin
      TEXT_IO.Get (Item => Letter);
      TEXT_IO.Put (Item => "The original value is ");
      TEXT_IO.Put (Item => Letter);
      Letter := Increment (Old => Letter);
      TEXT_IO.Put (Item => "The new value is ");
      TEXT_IO.Put (Item => Letter);
    end Mystery_Procedure;
```

2. What is the specification of the following function?

```
with TEXT_IO;
  function Decrement (The_Old_Value : INTEGER) return INTEGER
  is
    return The_Old_Value - 1;
  end Decrement;
```

3. Why is it permissible to have a function body without providing a function specification, but it is not permissible to have a function specification without providing a body?

4. What is wrong with the following function specification?

```
function Reverse_Mode (How_Far : out INTEGER) return INTEGER;
```

5. What would be the result stored in the object `New_Value` be if the function given below were called by the statement

```
New_Value := Exercise_5 (Old_Value => 7) + 3; ?

function Exercise_5 (Old_Value : INTEGER) return INTEGER
is
begin
  if Old_Value < 10
  then
    return Old_Value + 10;
  else
    return Old_Value - 10;
  end if;
end Exercise_5;
```

6. What would the value of `New_Value` be if the following statement was used in a procedure, referencing the function given in Exercise 5?

```
New_Value := Exercise_5 (Old_Value => 10) * 5 +
             Exercise_5 (Old_Value => 11);
```

7. Suppose the following call were made to the function in Exercise 5. What you be the value, if any, in `New_Value`? If no value can be computed, explain why not.

```
New_Value := Exercise_5 (Old_Value =>
             Exercise_5 (Old_Value => 2));
```

8. What is the output of the following program?

```
with TEXT_IO;
  procedure Exercise_8
  is
    The_Value : CHARACTER := 'C';

    function Mystery (Old_Value : CHARACTER) return CHARACTER
    is
    begin
      return CHARACTER'PRED (Old_Value);
    end Mystery;
  begin
    TEXT_IO.Put (Item => The_Value);
    The_Value := Mystery (Old_Value => The_Value);
    TEXT_IO.Put (Item => The_Value);
    The_Value := 'G';
    The_Value := Mystery (Old_Value =>
                          Mystery (Old_Value => The_Value));
    TEXT_IO.Put (Item => The_Value);
  end Exercise_8;
```

8.4 Parameters of Ada Subprograms

Subprograms are abstractions of operations and can perform whatever action the software engineer decides needs to be done. Some actions are very simple and are always the same. These procedures would not need to be parameterized. Consider the procedure `Draw_Line` below.

```
with TEXT_IO;
procedure Draw_Line
is
begin
  Put_Line_Elements:
  for Count in 1 .. 80
  loop
    TEXT_IO.Put (Item => '-');
  end loop Put_Line_Elements;
end Draw_Line;
```

This procedure has a simple function, namely to draw a line. The line will always be 80 characters long and it will always be a sequence of '-' characters. If that is all that is ever needed, then this procedure will suffice. Note that a **while** loop could have been used, but the **for** loop is probably more appropriate in this case since we know precisely how many iterations that we want.

However, it is possible to make this procedure more general. Suppose that it is desired to use a '-' character to draw the line sometimes, but another character, say '=', to draw the line at other times. How can the procedure be make more flexible and versatile? The answer is via the use of a parameter. Thus, observe the changes in the `Draw_Line` procedure to allow for the parameter mechanism,

```
with TEXT_IO;
procedure Draw_Line (The_Character : CHARACTER)
is
begin
  Put_Line_Elements:
  for Count in 1 .. 80
  loop
    TEXT_IO.Put (Item => The_Character);
  end loop Put_Line_Elements;
end Draw_Line;
```

Now the procedure Draw_Line has a parameter called The_Character which is an object of type CHARACTER. It is not actually an object, but is really placeholder for an object. This placeholder is called a formal parameter. The procedure Draw_Line may refer to the formal parameter anywhere, and in any manner, appropriate for an object of type CHARACTER. A call to the procedure Draw_Line must include an object of the same type as the formal parameter of Draw_Line, i.e., a call must include a CHARACTER object. This object supplied with the call is termed an actual parameter. Thus, a call to Draw_Line such as

```
Draw_Line (The_Character => '+');
```

would cause the formal parameter The_Character in the procedure Draw_Line to be replaced (since the formal parameter is just a placeholder) by the value of the actual parameter supplied as part of the call to Draw_Line. In this case, a '+' character would be, in effect, substituted for The_Character in Draw_Line; in particular, the statement

```
TEXT_IO.Put (Item => The_Character);
```

would be handled as if it had been written

```
TEXT_IO.Put (Item => '+');
```

for this call to the procedure Draw_Line. Yet a call to the same procedure using the actual parameter '/' would yield a line of '/' characters, since the actual parameter would cause the procedure to behave as if it had been written

```
TEXT_IO.Put (Item => '/');
```

substituting the actual parameter '/' for the formal parameter The_Character.

Thus, the increased flexibility of procedures is provided by the parameter mechanism. When the software engineer designs a procedure, he/she should always look for opportunities to make the procedure more general and more flexible. This will lead to increased reuse of procedural abstractions.

Consider once again the procedure Draw_Line. Is there a way to increase its utility by making it more adaptive to different situations? Consider that the procedure Draw_Line, as revised, is flexible enough to allow any character to be used to draw the line, but that a line is always 80 characters long. Can the flexibility of this procedure be increased by parameterizing the length of the line? The new version of Draw_Line is thus,

```
with TEXT_IO;
procedure Draw_Line (The_Character      : CHARACTER;
                     The_Desired_Length : NATURAL)
is
begin
  Put_Line_Elements:
  for Count in 1 .. The_Desired_Length
  loop
    TEXT_IO.Put (Item => The_Character);
  end loop Put_Line_Elements;
end Draw_Line;
```

where there are two formal parameters, namely The_Character and The_Desired_Length. Note that different formal parameters are separated by a semicolon. A call to the procedure Draw_Line as currently revised would be of the form,

```
Draw_Line (The_Character => '-',
           The_Desired_Length => 80);
```

which would procedure a line 80 characters long consisting of the character '-'. Note that the actual parameters are separated by a comma.

8.4.1 Parameter Modes In Ada Procedures

Parameters are means for tailoring a procedure and therefore increase the flexibility and adaptability of procedures. Examples already discussed have shown how parameters can provide this additional capability. However, one item relating to parameters needs to be presented more fully and that is that parameters are directional. This directional nature of parameters is represented in Ada by a parameters mode. Parameter modes are either **in**, **out**, or **in out**.

The mode of a parameter is an indication of the direction of information flow relative to the procedure being called. **In** parameters carry information into the procedure, **out** parameters carry information out of the procedure, and **in out** parameters carry information in both directions. The parameter mode is specified in the parameter list when a procedure specification is written. It appears between the colon and the type mark. For example,

```
procedure Sum (First_Number, Second_Number : in INTEGER;
               Result : out INTEGER);
```

declares the procedure Sum to have two parameters of mode **in**, namely First_Number and Second_Number, and one parameter of mode **out**, namely Result. If a parameter mode is not specified, it is by default taken to be mode **in**. The syntax for parameter specifications in which the mode is indicated is given in **Syntax Definition 8.6 and Syntax Chart 8.6**. The use of parameter modes is explained next.

Suppose that you call the procedure Draw_Line from the previous section. The call to the procedure must include the character to be used to draw the line and the length of the line (number of characters). The procedure cannot do its job without this information. Thus, these parameters are needed in the procedure and are called **in** parameters. **In** parameters act as a constant within the procedure and can only be referenced or read. The value of an **in** parameter cannot be changed. As a consequence of this, **in** parameters can only appear on the right hand side of assignment statements in the procedure and must not have a use other than one in which the value of the parameter is inspected or read.

Imagine now that the procedure Sum is called. Recall that the specification of this procedure is,

```
procedure Sum (First_Number, Second_Number : in INTEGER;
               Result : out INTEGER);
```

and the body is,

```
procedure Sum (First_Number, Second_Number : in INTEGER;
               Result : out INTEGER)
is
begin
  Result := First_Number + Second_Number;
end Sum;
```

Note that the first two parameters are of mode in and thus can only provide values that may be used by the procedure. The last parameter is of mode out and has a different meaning. Out parameters are objects that are supplied by the calling program unit for the procedure to use to return values computed by the procedure. Since these parameters are only for the purpose of providing values back to the calling program unit, they can never be read inside of the procedure. As a consequence they may only appear on the left hand side of an assignment statement, or in places where their values are not read. Note that this rule applies even after a value has been supplied to an out parameter inside of the procedure. Thus, in the Sum procedure above, if another statement were added to this procedure after the assignment to Result that attempted to read the value in Result, an error would occur. Thus, a statement such as,

```
if Result < 0
then
  Result := 0;
end if;
```

after the assignment of a value to Result would not be allowed since an attempt is being made to inspect the value in Result by the conditional portion of the if statement Result < 0.

Now suppose that you desired to modify the procedure Sum to allow for an examination of the value in Result after the assignment. The revised version of Sum would then be,

```
procedure Revised_Sum (First_Number,
                       Second_Number : in INTEGER;
                       Result : out INTEGER)
is
begin
  Result := First_Number + Second_Number;
  if Result < 0
  then
    Result := 0;
  end if;
end Sum;
```

Unfortunately, as discussed above, this procedure will not compile because of the error in the mode of the parameter Result.

The solution is to modify the specification of the procedure to redefine the mode of the parameter Result from an out parameter to an in out parameter. An in out parameter provides a bidirectional capability for a parameter, *i.e.*, it provides for information flow between the calling program unit and the called procedure via parameters to be in both directions. The procedure then receives information in the form of a value provided with the parameter when the procedure is called and it returns a value when it completes. Thus, the new revised version of Sum becomes,

```
procedure New_Revised_Sum (First_Number,
                           Second_Number : in INTEGER;
                           Result : in out INTEGER)
is
begin
  Result := First_Number + Second_Number;
  if Result < 0
  then
    Result := 0;
  end if;
end Sum;
```

The New_Revised_Sum will properly compile and execute the statements indicated.

Since the **in out** form of parameter mode seems to be the most powerful, why not declare all of the parameters to be of mode **in out**? The reason is that the mode of a parameter should match its intended use. In this manner, the compiler and/or runtime system can assist in detecting errors by noting improper use of parameters. In general, it is a bad practice to use **in out** parameters unless their intended use requires this parameter mode.

8.4.2 Parameters of Ada Functions

Parameters of functions afford the same benefits as those used for procedures. The additional flexibility and versatility provided by parameters make them very useful for software engineers. There is a difference between the parameter modes that were available for procedures and those that allowed for functions.

Whereas procedures had three possible modes for their parameters, functions may only have parameters of mode **in**. This is necessary to try to prevent the creation of functions that have side-effects. Recall that a function is intended to compute a single logical operation and return a value. In some cases, software engineers using other languages have used functions to not only return a value, but also to modify the parameters used in the computation of the value to be returned by the function. This additional effect, changing the value of one or more parameters, is usually not a good idea. In general, such side-effects make proving functions correct very difficult. Thus, Ada was designed to prevent the software engineer from changing any of the parameters, allowing only the computed value to be returned from a function call and leaving all other aspects of the function and its parameters unaffected. In order to insure this limitation on the software engineer, all Ada parameters are required to be of mode **in**.

Recall that parameters of mode **in** act as constants within the subprogram. Thus, all function parameters act as if they are nothing more than initialized constants within the function. Recall also that in the absence of an explicit mode for a parameter, the mode **in** is assumed, *i.e.*, it is the default. Thus, in functions it is not necessary to provide a mode because the default mode is the only permissible mode.

8.4.3 Exercises

1. Why does a procedure have both a specification and a body? What is the purpose of each?

2. What is the specification for the following procedure?

```
procedure Demo (First_Num : in out INTEGER)
  is
  begin
    First_Num := First_Num / 2;
  end Demo;
```

3. Consider the following revision of the procedure `Traffic_Signals` described earlier. What is the purpose of the parameter?

```
procedure Traffic_Signals (Signal_Length : DURATION)
  is
    type Color_Type is (RED, YELLOW, GREEN);
    Signal : Color_Type := GREEN;
    Signal_Light_Activated : BOOLEAN := TRUE;
  begin
    Traffic_Control_Processing:
    while Signal_Light_Activated loop
      delay Signal_Length;  -- seconds
      Signal := YELLOW;
      delay 5.0;   -- seconds
      Signal := RED;
      delay Signal_Length;  -- seconds
      Signal := GREEN;
    end loop Traffic_Control_Processing;
  end Traffic_Signals;
```

4. What is the mode of the parameter in Exercise 3? Is it appropriate for the use of this parameter within the procedure?

5. What is wrong with the following procedure?

```
procedure Another_Mystery (My_Value  : out CHARACTER;
                           New_Value : in  CHARACTER)
is
  begin
    if New_Value /= 'Z'
    then
      My_Value := CHARACTER'SUCC (New_Value);
    else
      My_Value := 'A'
    end if;
    if My_Value = 'Z'
    then
      New_Value := 'A';
    end if;
end Another_Mystery;
```

6. What is the purpose of the following procedure?

```
with TEXT_IO;
  procedure Mystery_Again
  is
    A_Value : CHARACTER;
    Count   : INTEGER := 0;
  begin
    Mystery_Loop:
    while not TEXT_IO.End_Of_File
    loop
      TEXT_IO.Get (Item => A_Value);
      Count := Count + 1;
    end loop Mystery_Loop;
    TEXT_IO.Put (Item => Count);
  end Mystery_Again;
```

7. According to **Syntax Definition 8.6**, what is the maximum number of parameters that any procedure may have?

8. What is wrong with the following procedure? Is the problem that you identified an error in the syntax or an error in the design?

```
procedure Error (Name   : in out STRING;
                 Number : in out INTEGER;
                 Value  : in out CHARACTER)
  is
  begin
    if Name(1) < Value
    then
      Value  := Name(1);
      Number := 0;
    end if;
  end Error;
```

8.5 Ada Program Execution Structure

In a previous chapter we discussed the terms scope and visibility and gave you a preliminary introduction to what these terms mean. In this section we will expand upon these ideas to include the new concepts of procedures, functions, and packages. In addition, we will examine the need for the Ada program library and what kinds of information you should expect to find there.

8.5.1 Scope and Visibility

Recall that visibility was the ability of the program to directly access any given identifier. If the program could access the identifier at that point in the program, then it is said to have visibility on that identifier at that point. Scope was the area of potential visibility. The scope of an identifier is that region of the program from the point where the identifier is declared until the end of the executable region of the declarative part where the identifier was declared. For example, in the following code segment

```
with TEXT_IO;
procedure Outer
is
  My_Number : INTEGER := 10; -- Point A

  procedure Inner
  is
    My_Number : CHARACTER := 'A'; -- Point B
  begin
    TEXT_IO.Put (My_Number); -- Point C
  end Inner;
begin
  while My_Number > 0
  loop
    My_Number := My_Number - 1; -- Point D
  end loop;
end Outer;
```

the scope of the identifier `My_Number` declared at Point A is from Point A until the end of the procedure `Outer`. However, the visibility of this identifier does not include the same region. At Point B the identifier `My_Number` is redeclared to be of type CHARACTER and the scope of that identifier is from Point B until the end of the procedure `Inner`. During this region, the meaning of the identifier `My_Number` is the one with a type of CHARACTER. Thus, the outer identifier `My_Number`, whose type is INTEGER is hidden by the declaration of the inner identifier with the same name. In this example, the scope of the inner declaration of `My_Number` (of type CHARACTER) is the same as its visibility. In the case of the outer declaration of `My_Number` (of type INTEGER) the scope is from Point A until the end of procedure `Outer`, but the visibility is restricted during the scope of the inner `My_Number` (of type CHARACTER). This is the reason that at Point C the meaning of the identifier is `My_Number` is the one whose type is CHARACTER, while at Point D meaning of the same identifier is of type INTEGER. It is the block structure of the language that allows us to recognize the proper meaning as defined by the software engineer. Thus, the scope and the visibility of an identifier will be identical except in cases where the same identifier is used in an inner scope to refer to another declaration, hiding the meaning of the outer declaration. In actuality, in Ada there is a way to reference identifiers that are within scope, but that are not directly visible. We can use extended notation to name the scope level where the identifier was declared, thereby providing visibility where direct visibility is not possible. Details of this will be deferred until a later discussion.

8.5.2 Scope of Parameters

Now that we have studied the concept of parameters to subprograms, it is reasonable to ask how these entities are affected by scope and visibility. What is the scope of the parameter of a subprogram? The answer is simply that the same rules apply. From the point of the declaration of the parameter until the end of the subprogram we have the scope of the parameter. Its visibility is determined by whether or not the same identifier is redefined within the subprogram as illustrated in 8.5.1. Note that the name of the subprogram itself is at a different scoping level than the parameters and local declarations of the subprogram. This must be case so that the name of the subprogram is visible to other subprograms in the same scope, allowing it to be referenced and called. However, the parameters and local declarations are at a different, inner, scope preventing their being referenced outside the subprogram. For example, given the following subprogram:

```
with TEXT_IO;
procedure MugWumps
is
  Universe_Size : constant := 9;
  subtype Lifeforms is NATURAL range 0 .. 25;
  type Habitat is array (Universe_Size, Universe_Size)
                         of Lifeforms;
  The_World : Habitat := (others => (others => 0));
                      -- initialize the world to be empty

  procedure MugWump_Display (Universe : Habitat)
  is
  begin
    TEXT_IO.Put (Item => "MugWump Land looks like this:");
    TEXT_IO.New_Line;
    Row_Loop:
    for Row in Universe_Size
    loop
      Column_Loop:
      for Column in Universe_Size
      loop
        if Universe(Row, Column) = 0
        then
          TEXT_IO.Put (Item => " . ");
        else
          TEXT_IO.Put (Item => " M ");
        end if;
      end loop Column_Loop;
      TEXT_IO.New_Line;
    end loop Row_Loop;
    TEXT_IO.New_Line;
  end MugWump_Display;
begin
  The_World(2, 4) := 3; -- Put some Mugwumps in the World
  The_World(8, 1) := 1;
  The_World(5, 6) := 2;
  The_World(9, 7) := 7;
  The_World(3, 5) := 4;
  MugWump_Display (Universe => The_World);
end MugWumps;
```

In this example, each identifier is unique and consequently the scope of each identifier is also its region of visibility. However, to illustrate what is meant by the various scoping levels in this program, let us analyze the identifier `Universe` which is a parameter to the procedure `MugWump_Display`. Its scope is from its point of declaration until the end of the procedure `MugWump_Display`. It may be referenced anywhere in the procedure, but it may not be referenced outside the procedure. This is true because of the scope and visibility rules. You may always "see" identifiers that are declared at an outer scope (and that are not hidden by being redeclared at some inner scope), but you may never "see" identifiers inside a nested scope, *i.e.*, a scope that is wholly contained within another scope.

The best way to prevent confusion on this point is to learn a little trick that helps to resolve scope and visibility issues. Simple draw a box around each procedure, including in the box the parameters to the procedure and all of the procedure except the procedure name. when this is done, we have something that looks like the following:

```
with TEXT_IO;
procedure MugWumps
is                                                          Scope A
  Universe_Size : constant := 9;
  subtype Lifeforms is NATURAL range 0 .. 25;
  type Habitat is array (Universe_Size, Universe_Size)
                        of Lifeforms;
  The_World : Habitat := (others => (others => 0));
                      -- initialize the world to be empty

  procedure MugWump_Display (Universe : Habitat)     Scope B
  is
  begin
    TEXT_IO.Put (Item => "MugWump Land looks like this:");
    TEXT_IO.New_Line;
    Row_Loop:
    for Row in Universe_Size
    loop
      Column_Loop:
      for Column in Universe_Size
      loop
        if Universe(Row, Column) = 0
        then
          TEXT_IO.Put (Item => " . ");
        else
          TEXT_IO.Put (Item => " M ");
        end if;
      end loop Column_Loop;
      TEXT_IO.New_Line;
    end loop Row_Loop;
    TEXT_IO.New_Line;
  end MugWump_Display;
begin
  The_World(2, 4) := 3; -- Put some Mugwumps in the World
  The_World(8, 1) := 1;
  The_World(5, 6) := 2;
  The_World(9, 7) := 7;
  The_World(3, 5) := 4;
  MugWump_Display (Universe => The_World);
end MugWumps;
```

The boxes provide an easy mechanism to clearly see the scope and visibility of each identifier. We can see that the identifier `Universe` belongs to Scope B and is only available within this scope. On the other hand, `The_World` is available in Scope A, as well as Scope B, since its scope includes the nested procedure. Further, it is not hidden by another declaration of the same identifier within the inner scope, so it is also visible within Scope B. The identifier `MugWump_Display`, the name of the nested procedure, is in Scope A, not Scope B. This must be the case so that this procedure can be called from the body of the outer procedure `MugWumps`. However, everything else about the nested procedure is completely within Scope B and is not visible or available to the outer scope, at least not directly.

8.5.3 Ada Libraries

We have already seen that Ada provides us with a very powerful mechanism to verify the correctness of the software system statically. It is able to do this because of its ability to allow separate compilation. Separate compilation refers to the fact that we can compile the specifications and bodies of all of the program units separately. When we do so, we are guaranteed that we will have cross-module consistency checking performed at compile time. Cross-module consistency checking is nothing more than assuring that if one program unit refers to another one, such as by calling a procedure within a package that was `withed` by the unit being compiled, the number, order, and type of the parameters to the called procedure are appropriate. This information will be checked at compile time so as to prevent errors later when the system is linked or executed.

Separate compilation is very powerful, but it requires that the compiler have information available to it at compile time that ordinarily would not be needed. For example, in the illustration previously given, the number, order, and type of the parameters to the called procedure were checked. How did the compiler know if there were the proper number, order, and type of parameters? The answer is that it had to find the information somewhere. Ada places all of this necessary information into a library. Then, by requiring a partial order in the compilation of the system, it can guarantee that when it needs to check something in another unit, the information is already available and is stored in the library.

Thus, we can see that the Ada library acts as a repository of information for the compiler. When a program unit is compiled, all of the information about that unit is placed into the library. Then, when other program units are compiled, the compiler can check to see that if they use any of the operations from the first program unit, the operations (subprograms) are used correctly, as verified by checking the relevant information in the library.

Ada libraries are simple to understand and they are what gives Ada the power to provide separate compilation to software systems. Later, we will return to the concept of a program library and see what other things we can learn about how to use the library more effectively. For now, suffice it to say that the library is a powerful tool that makes it possible to build large programming systems using separate compilation.

8.5.4 Exercises

1. Show by example that it is *not* possible to have a region of visibility that is larger than the scope of an identifier.

2. Draw an arrow to show the scope of the identifier `The_Name` in the following code segment:

```
procedure Exercise_2
is
  The_Age : INTEGER := 0;
  procedure Write_Answer
  is
    The_Name : STRING(1 .. 15);
  begin
    null;
  end Write_Answer;
begin
  null;
end Exercise_2;
```

3. Draw an arrow to show the scope of the identifier The_Age in the following code segment. Use a different color to draw an arrow to show the region of visibility of the same identifier.

```
procedure Exercise_3
is
  The_Age : INTEGER := 0;
  procedure Write_Answer
  is
    The_Age : STRING(1 .. 5);
  begin
    null;
  end Write_Answer;
begin
  null;
end Exercise_3;
```

4. Draw boxes to show the scope regions in the following code segment:

```
with TEXT_IO;
procedure Exercise_4
is
  Universe_Size : constant := 9;
  subtype Lifeforms is NATURAL range 0 .. 25;
  type Habitat is array (Universe_Size, Universe_Size)
                        of Lifeforms;
  The_World : Habitat := (others => (others => 0));
                      -- initialize the world to be empty

  procedure MugWump_Display (Universe : in Habitat)
  is
  begin
    TEXT_IO.Put (Item => "MugWump Land looks like this:");
    TEXT_IO.New_Line;
    Row_Loop:
    for Row in Universe_Size
    loop
      Column_Loop:
      for Column in Universe_Size
      loop
        if Universe(Row, Column) = 0
        then
          TEXT_IO.Put (Item => " . ");
        else
          TEXT_IO.Put (Item => " M ");
        end if;
      end loop Column_Loop;
      TEXT_IO.New_Line;
    end loop Row_Loop;
    TEXT_IO.New_Line;
  end MugWump_Display;
  procedure Mate_MugWumps (The_Universe : in out Habitat)
  is
    -- MugWumps mate ONLY when there are exactly two
    -- in a cave at the beginning of a MugWump turn.
  begin
    TEXT_IO.Put (Item => "Initially ");
    MugWump_Display (Universe => The_Universe);
    Rows:
    for Row in Universe_Size
    loop
      Columns:
      for Column in Universe_Size
      loop
        if The_Universe(Row, Column) = 2
        then
          The_Universe (Row, Column) := 3;
        end if;
      end loop Columns;
    end loop Rows;
    TEXT_IO.Put (Item => "After the mating ritual ");
    MugWump_Display (Universe => The_Universe);
  end Mate_MugWumps;

begin
  The_World(2, 4) := 3; -- Put some Mugwumps in the World
  The_World(8, 1) := 1;
  The_World(5, 6) := 2;
  The_World(9, 7) := 7;
  The_World(3, 5) := 4;
  The_World(6, 3) := 2;
  The_World(2, 9) := 2;
  Mate_MugWumps (The_Universe => The_World);
end MugWumps;
```

5. What kinds of information are likely to be stored in an Ada library and why? State your answer as explicitly as possible after consulting your compiler vendor's user's guide.

Chapter 9

Linked Lists

Thus far in our study we have concentrated on program analysis, program correctness, and program design. In this chapter we will focus on data analysis and data structuring techniques. We will begin this analysis by introducing two new data types, records and access types. Then we will investigate methods of combining these types into structures called linked lists.

9.1 Introduction to Records

In Chapter 5 we introduced a composite data type called an array. Array objects are used to store lists of component values that are all of the same type. For example, a list of quiz grades for 100 students could easily be handled by an array object. However, arrays are not appropriate in cases where the data components to be stored are of different types. To handle these cases, we now introduce the concept of a record.

9.1.1 Records

A record is a list of component values that may be of different types. Records are used to store different, but related values as a single unit. For example, a student record might consist of an id number, a major code, and a year level. As another example, consider the date today. This composite value usually consists of a month name, a day number, and a year number. Thus it seems natural to define a single object, say `Date`, with three related components for the month, day, and year.

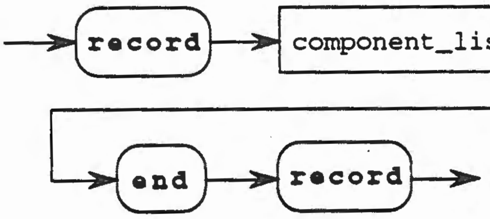
The formal syntax for a record data type in Ada is given by the following syntax productions.

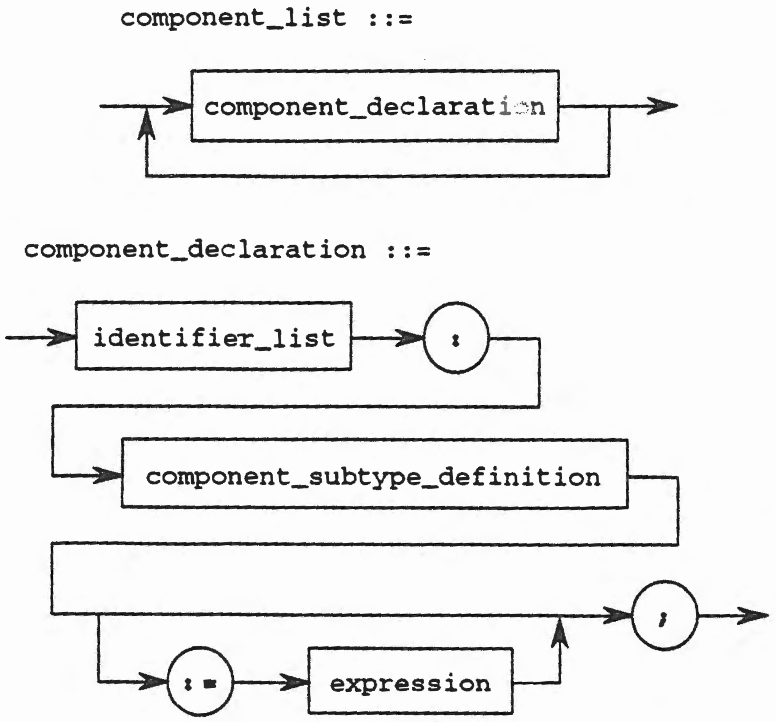
```
record_type_definition ::=
  record
    component_list
  end record

component_list ::= component_declaration {component_declaration}

component_declaration ::=
  identifier_list : component_subtype_indication [:= expression];
```

Record Data Type
Syntax Definition 9.1

The syntax can also be expressed in graphic form as indicated in Syntax Chart 9.1

record_type_definition ::=





Record Data Type
Syntax Chart 9.1

Suppose we wish to define a record type with a suitable structure for the student record mentioned earlier. We will begin by analyzing the component values. For the student id number we will use values between 0 and 9999. For the major code we will use values of CS, CP, EE, or MA. Finally, for year level we will use values of 1, 2, 3, or 4. Thus the full record type definition looks like the following:

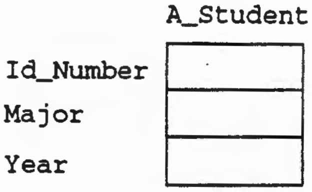
```
type Student_Id_Number is range 0 .. 9999;
type Major_Code is (CS, CP, EE, MA);
type Year_Level is range 1 .. 4;
type Student_Record is
record
  Id_Number : Student_Id_Number;
  Major : Major_Code;
  Year : Year_Level;
end record;
```

Note that `Student_Record` now describes a type with three components, `Id_Number`, `Major`, and `Year` of differing types. Note also that the type of each component must be indicated. Any data type may be used as the component of a record.

Now that we have described the type `Student_Record`, we are free to create objects of that type. For example,

```
A_Student : Student_Record:
```

declares an object named `A_Student` with enough contiguous storage for the three component values desired. Pictorially we have the following object:

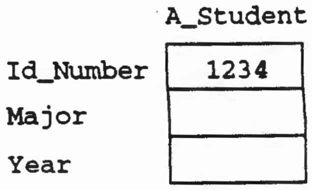


A_Student Object
Figure 9.1

Each component location within the record structure is called a field. We may access the various fields of a record directly through the use of a "dot notation". For example,

```
A_Student.Id_Number := 1234;
```

places the value 1234 into the `Id_Number` field of the record `A_Student`. That is, our picture now looks like this:

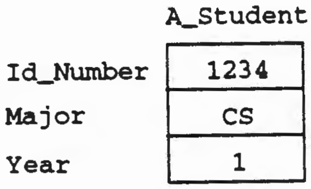


Initializing Id_Number
Figure 9.2

To continue with our initialization suppose we perform the following assignments:

```
A_Student.Major := CS;
A_Student.Year := 1;
```

Now our picture is:



Initialized A_Student Object
Figure 9.3

9.1.2 Record Operations

The operations available for records include assignment, membership, component indication, relational, and explicit conversion.

Aggregate Assignment	:=	
Membership	in	not in
Relational	=	/=

Aggregate assignment for records is similar to aggregate assignment for arrays. It provides the ability to assign values to all fields of a record at the same time. For example, if we define a record structure and create objects suitable for various dates:

```
type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN, JUL,
  AUG, SEP, OCT, NOV, DEC);
type Day_Number is range 1 .. 31;
type Year_Number is range 1900 .. 2100;
type Date_Record is
record
  Month : Month_Name;
  Day : Day_Number;
  Year : Year_Number;
end record;

Today : Date_Record;
Birthday, Holiday : Date_Record;
```

we could assign values to the objects by the following aggregate assignment statements.

```
Today := (MAR, 30, 1992);
Birthday := (Month => MAY, Day => 15, Year => 1900);
Holiday := (DEC, 25, 1992);
```

The statement

```
Today := (MAR, 30, 1992);
```

is equivalent to the three statements:

```
Today.Month := MAR;
Today.Day := 30;
Today.Year := 1992;
```

The statement

```
Birthday := (Month => MAY, Day => 15, Year => 1990);
```

uses named association of field and value. That is, `Month => MAY` means for the `Month` field use (or assign) the value `MAY`. Named association permits field values to be given in orders that may differ from the order specified in the type definition. Thus an equivalent assignment is:

```
Birthday := (Day => 15, Year => 1990, Month => MAY);
```

Aggregate assignment may also be used to initialize a record object as it is created. For example,

```
Today : Date_Record := (MAR, 30, 1992);
```

creates the object `Today` and initializes the three fields with the values indicated.

The *membership* operation allows us to test a record to determine if a particular value exists as one of the components of the record.

The *relational* operations of "=" and "/=" allow two record objects to be compared. For example,

```
if Today = Birthday
then
  TEXT_IO.Put (Item => "Happy Birthday!");
else
  TEXT_IO.Put (Item => "Just another day.");
end if;
```

will compare the corresponding fields of the two record objects Today and Birthday. If the corresponding fields contain identical values, then the message "Happy Birthday!" will be printed. Otherwise the message "Just another day." will appear.

9.1.3 Default Initial Values

In addition to aggregate assignment and single component assignment, record objects may be initialized through the use of default initial values given in the declaration of the record type. These values are specified using the ":=" operator in the type definition. Some or all of the components in a record may be given a default value. For example,

```
type Gender is (M, F);
type Age_Range is range 0 .. 150;
type Person is
record
  Name : STRING (1 .. 5);
  Sex : Gender := M;
  Age : Age_Range := 20;
end record;

First_Person : Person;
Second_Person : Person := ("Linda", F, 30);
```

the record type Person specifies default initial values of M and 20 for the fields Sex and Age respectively. The declaration of First_Person makes use of these default values while leaving the Name field uninitialized. On the other hand, the declaration of Second_Person overrides the default values by placing the string "Linda" in the Name field, F in the Sex field, and 30 in the Age field.

9.1.4 Record Input and Output

Since a record is a collection of values of differing types, it is not possible to put or get an entire record in the same manner that a single value is read or written into an object. Instead each component must be treated as an object of a particular type and the corresponding methods of input and output employed. See the following example.

```
with TEXT_IO;
procedure Test_Days
is
  type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN, JUL,
    AUG, SEP, OCT, NOV, DEC);
  type Day_Number is range 1 .. 31;
  type Year_Number is range 1900 .. 2100;
  type Date_Record is
  record
    Month : Month_Name := JAN;
    Day : Day_Number := 1;
    Year : Year_Number := 1993;
  end record;
```

```
    package Month_IO is new TEXT_IO.ENUMERATION_IO (Month_Name);
    package Day_IO is new TEXT_IO.INTEGER_IO (Day_Number);
    package Year_IO is new TEXT_IO.INTEGER_IO (Year_Number);

    Today : Date_Record := (MAR, 30, 1992);
    Birthday, Holiday : Date_Record;

  begin
    TEXT_IO.Put (Item => "In what month were you born?");
    Month_IO.Get (Item => Birthday.Month);
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "On what day?");
    Day_IO.Get (Item => Birthday.Day);
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "In what year?");
    Year_IO.Get (Item => Birthday.Year);
    TEXT_IO.New_Line;
    Holiday := (JUL, 4, 1992);
    if Today = Birthday
    then
      TEXT_IO.Put (Item => "Happy Birthday!");
    elsif Today = Holiday
    then
      TEXT_IO.Put (Item => "Hurray - a holiday!");
    else
      TEXT_IO.Put (Item => "It's just ");
      Month_IO.Put (Item => Today.Month);
      Day_IO.Put (Item => Today.Day);
      Year_IO.Put (Item => Today.Year);
    end if;
    TEXT_IO.New_Line;
  end Test_Days;
```

9.1.5 Exercises

1. Describe the difference between a record and an array.

2. Define a record type named `Person_Record` with fields for a person's name, age, height, and weight.

3. Suppose you had a record object named `A_Person` of type `Person_Record` (see Exercise 2). How can you initialize this object?

4. Write a segment of Ada code to output the values in the object `A_Person` (see Exercise 3).

5. Consider the record objects `Today`, `Birthday`, and `Holiday`. Describe how the same information could be stored using parallel arrays.

9.2 Introduction to Access Types

The data objects that we have seen thus far have all been static objects. That is, each object was created as a result of a declaration, and associated with a unique identifier. Unfortunately this requires that the number of such objects be known in advance. In situations where this is not the case, Ada provides a mechanism to dynamically allocate data objects.

9.2.1 Access Types

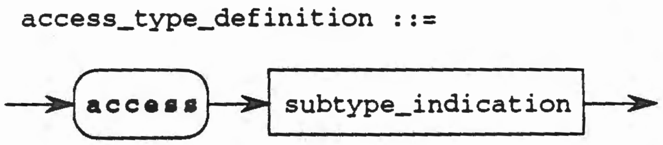
Suppose you were moving to a new location and wanted to make sure that your mail followed you to your new address. More than likely you would go to your post office and fill out a change of address form. Then when mail arrived your postal service could forward it to you at your new home. Thus they could provide access to you through the address you left behind. This is the basic idea behind Ada's access type.

Ada provides a method of defining variables capable of containing the address of other data objects. Such access variables are defined in terms of two other types, the access type and the type of the object being accessed (also called the designated type). The formal syntax for access types in Ada is given in the following syntax production. The `subtype_indication` defines the type of the object being accessed.

```
   access_type_definition ::= access subtype_indication
```

Access Type
Syntax Definition 9.2

This syntax can also be expressed in graphic form as indicated in Syntax Chart 9.2

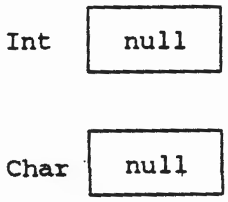


Access Type
Syntax Chart 9.2

Let us first consider the concept of accessing objects by studying the following definitions and declarations.

```
   type Integer_Access is access INTEGER;
   type Char_Access is access CHARACTER;
   Int : Integer_Access;
   Char : Char_Access;
```

Here we have defined an access type called `Integer_Access` capable of providing access to INTEGER objects. Similarly `Char_Access` defines a type capable of providing access to CHARACTER objects. The two object declarations, `Int` and `Char`, create access variables whose initial values are both **null**. That is, `Int` and `Char` currently give access (or point) to nothing. Access variables are the only objects that Ada automatically initializes for you. Thus our current situation could be viewed as:



Int and Char Initial Values
Figure 9.4

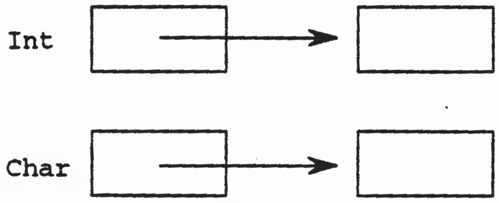
Note carefully that we have created only two variables and that both variables have the value **null**. Note also that `Int` and `Char` are not of compatible types since one, namely `Int`, will give access to INTEGER objects while the other will give access only to CHARACTER objects.

9.2.2 Dynamic Allocation

Suppose we want to create an INTEGER object and a CHARACTER object and make `Int` and `Char` (defined above) our means of accessing these objects. One method of doing this is

```
Int := new INTEGER;
Char := new CHARACTER;
```

These statements create objects of the indicated types, return access values (addresses), and store these access values in the appropriate access variables. As a result, our picture becomes

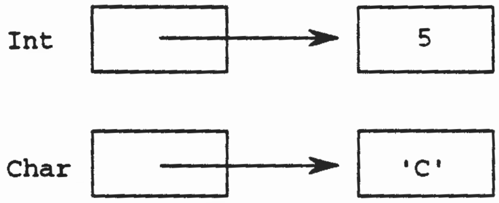


Creating INTEGER and CHARACTER Objects
Figure 9.5

Note that the actual INTEGER and CHARACTER objects have no names and can only be accessed through `Int` and `Char` respectively. Note also that the actual INTEGER and CHARACTER objects were given no values. Let us correct that situation by giving both objects values.

```
Int.all := 5;
Char.all := 'C';
```

It is important to realize that an access variable such as `Int` contains an access value (address). Appending .all to an access variable name references the dynamic variable that it is pointing to. So the statements above place the values 5 and 'C' in the corresponding dynamic variables. Our picture now is

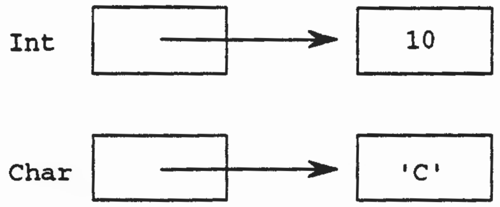


Assigning Dynamic Variables
Figure 9.6

Suppose we want to change the value currently stored in the object designated by `Int`. All we must do is to execute the following statement:

```
Int.all := 10;
```

Then our picture becomes

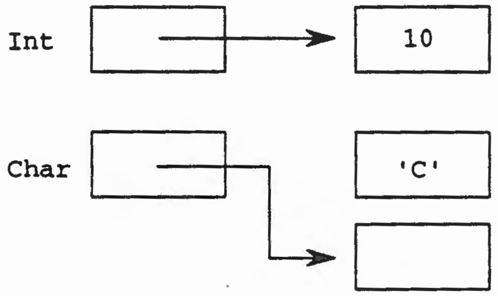


Changing the Value of a Dynamic Variable
Figure 9.7

Now suppose we want Char to designate a completely different object. To do this we execute the statement

```
Char := new CHARACTER;
```

Each time we use the **new** designator, we dynamically create a new object of the indicated type and place the access value in the access variable. Thus our picture changes to



Creating a New Object
Figure 9.8

Note that the CHARACTER object containing 'C' still exists but that there is now no way to access it as Char now points to a different object. (We will talk about how to explicitly free this inaccessible data object at a later time.)

9.2.3 Access Type Operations

The operations available for access variables include assignment, allocation, membership tests, explicit conversion, and the literal **null**.

Assignment	:=	
Allocator	new	
Membership	in	not in

Let us consider a more complete example in which to study the operations available.

```
with TEXT_IO;
procedure Show_Access
is
  type Int_Access is access INTEGER;
  type Char_Access is access CHARACTER;
  Int_1, Int_2 : Int_Access;
  Char_1, Char_2 : Char_Access;
```

```
begin
  Int_1 := new INTEGER;
  Int_1.all := 14;
  Int_2 := Int_1;  -- Int_1 and Int_2 access the same object
  Char_1 := new CHARACTER;
  Char_1.all := 'C';
  Char_2 := new CHARACTER;
  Char_2.all := 'C'; -- Char_1 and Char_2 access different objects
  if Char_1.all = Char_2.all
  then
    TEXT_IO.Put (Item => "Same values.");
  end if;
  Int_1 := null; -- Int_1 accesses nothing
end Show_Access;
```

The lines

```
Int_1 := new INTEGER;
Int_1.all := 14;
Int_2 := Int_1;
```

create one INTEGER object, place 14 in the object, and place the access value (address) of the object in both Int_1 and Int_2. There is no restriction on the number of access variables through which an object can be accessed.

The lines

```
Char_1 := new CHARACTER;
Char_1.all := 'C';
Char_2 := new CHARACTER;
Char_2.all := 'C';
```

create two CHARACTER objects, place 'C' in both, and place the access values in the two access variables Char_1 and Char_2. Thus Char_1 and Char_2 point to different objects that happen to contain the same value.

What does the following do?

```
if Char_1.all = Char_2.all
then
  TEXT_IO.Put (Item => "Same values.");
end if;
```

Finally, the line

```
Int_1 := null;
```

forces Int_1 to point at nothing. However, the INTEGER object that Int_1 originally accessed is still accessible through Int_2.

9.2.4 Designated Types

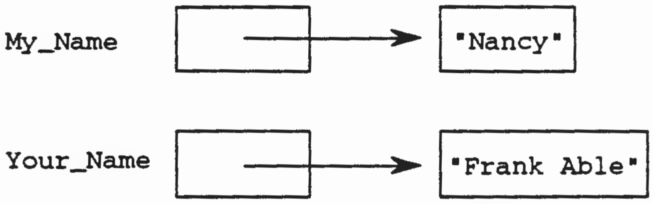
In Section 9.2.1 we stated that access variables are defined in terms of the access type and the designated type. You may begin to wonder what types can be used as designated types. The answer is that any of the types that we have discussed may be used in this capacity. That means that you can create access variables that provide access to scalar data objects, STRING data objects, array data objects, and record data objects. All that is required is that the designated type be defined before being used in an access type definition. For example suppose we want to create access variables for STRING objects.

```
type String_Access is access STRING;
My_Name : String_Access := new STRING (1..5);
Your_Name : String_Access := new STRING (1..10);
```

We could then give our STRING objects values through the statements

```
My_Name.all := "Nancy";
Your_Name.all := "Frank Able";
```

A picture of our situation is shown below.



Assigning Values to String Objects
Figure 9.9

As another example, suppose we want to create access variables for an array object. Note that we must first define the array type before using the array type name in the definition of our access type.

```
package Int_IO is new TEXT_IO.INTEGER_IO (INTEGER);
type Grade_Array is array (1..10) of INTEGER;
type Array_Access is access Grade_Array;
My_Grades : Array_Access;
```

Now that we have our types defined and an access variable declared, we are able to create and manipulate array objects in the following block of code.

```
My_Grades := new Grade_Array;
Fill_Array:
for Index in 1..10
loop
  TEXT_IO.Put (Item => "Enter grade => ");
  Int_IO.Get (Item => My_Grades.all(Index));
  TEXT_IO.New_Line;
end loop Fill_Array;
```

The line

```
My_Grades := new Grade_Array;
```

creates the dynamic array object and places the address in My_Grades. Within the body of the for loop My_Grades.all(Index) accesses a specific element in the array object. For example, when Index has value 1, My_Grades.all(Index) will refer to the first array element in the array object. As Index changes values within the loop, the corresponding array element is addressed. The loop will fill the array elements with the user input values.

As a final example, let us write a short Ada program that illustrates access to record objects.

```
with TEXT_IO;
procedure Record_Access
is
  type Name_Record is
  record
    First : STRING (1..5);
    Last : STRING (1..10);
  end record;
  type Name_Access is access Name_Record;
  My_Name, Your_Name : Name_Access;
begin
  My_Name := new Name_Record;
  Your_Name := new Name_Record;
  My_Name.First := "Sally"
  My_Name.Last := "Franklin  ";
  TEXT_IO.Put (Item => "What is your first name? ");
  TEXT_IO.Get (Item => Your_Name.First);
 .TEXT_IO.New_Line;
  TEXT_IO.Put (Item => "What is your last name? ");
  TEXT_IO.Get (Item => Your_Name.Last);
  TEXT_IO.New_Line;
  if (My_Name.First = Your_Name.First) and
    (My_Name.Last = Your_Name.Last)
  then
    TEXT_IO.Put (Item => "We have the same name.");
  else
    TEXT_IO.Put (Item => "I like your name.");
  end if;
  TEXT_IO.New_Line;
end Record_Access;
```

As you analyze the program `Record_Access` you will discover that we did not use the **`.all`** selector as in other examples. Instead we referred to `My_Name.First` and `Your_Name.Last`. Since the designated objects contain named fields, a shorthand notation such as `My_Name.First` can be employed instead of the more cumbersome yet correct notation `My_Name.all.First`

9.2.5 Exercises

1. What is the relationship between access variables and dynamic variables?

2. Define a type called `Bool_Access` whose designated type is BOOLEAN.

3. Create an access variable called `My_Bool` of the type defined in Exercise 2.

4. Create a dynamic variable with value TRUE and place its address in `My_Bool`. (See Exercise 3.)

5. Given the following definitions and code segment:

```
     type Int_Access is access INTEGER;
     Int_1, Int_2 : Int_Access;
   begin
     Int_1 := new INTEGER;
     Int_1.all := 10;
     Int_2 := Int_1;
     Int_2.all := 20;
```

a. How many dynamic variables have we created?
b. What value is in `Int_1`?
c. What value is in `Int_2`?
d. What value is in `Int_1.all`?
e. What value is in `Int_2.all`?

6. Given

```
   package Int_IO is new TEXT_IO.INTEGER_IO (Grade_Range);
   type Grade_Array is array (1..50) of Grade_Range;
   type Grade_Access is access Grade_Array;
```

a. How can the fifth array element be accessed?
b. Write a code segment that will initialize each array element to 0.

9.3 Introduction to Linked Lists

In the last section, we saw that Ada has a mechanism for creating dynamic variables. This means that we can define an access type but defer creating objects of the designated type until such time as they are needed. This dynamic allocation (and deallocation) allows us to save space and machine time. In this section we will consider the use of dynamic variables in a common application problem–list processing.

9.3.1 List Processing

Many problems in software development require the manipulation of a list of data values. The first step in the design of a solution involving such a list is the choice of an appropriate data structure in which to store the list. This choice will determine the maximum number (if any) of list elements permissible, and the speed and complexity of the operations to be applied to the list elements. To facilitate a discussion of desired list operations, suppose that we have an ordered list of names of unknown length such as

```
Adams
Baker
Claus
Davis
  :
Zachs
```

What operations are we likely to want to apply to such a list? The most fundamental operation is the creation and initialization of the list internally. For this we must consider how to read and store the data values in a data structure. Once the list has been initialized we will want the ability to traverse the list. That is, we will want a mechanism whereby we can start at the beginning of the list and visit each element on our way to the end of the list. Another desired operation is the ability to insert a name in the proper place in the list. For example, suppose we want to add the name `Blatt` to the list given above. Similarly, suppose we want to delete the name `Claus` from the list. Although these are the most common list processing operations, we may be asked to support additional operations in order to satisfy the program specification.

9.3.2 List Processing using Static Structures

We can implement the operations described in Section 9.3.1 by using a static array. Conceptually we can store the list elements in the following way.

Name_Array	Adams	Baker	Claus	...	Zacks	...	
	(1)	(2)	(3)		(Last)		(Max)

Storing Elements in a Static Array
Figure 9.10

This array will be a static structure, created in a declarative section with a fixed maximum size–`Max`. The array index `Last` will be used to denote the location of the last list element. Note that we will want to allocate more array locations than we plan to use since we expect our list size to change during the program. Filling the array data structure with our list values and traversing the array will be straight forward. However, insertion of new values or deletion of existing values will be more complicated. Suppose we must insert the name `Blatt` into the array. We can readily see that the new name must be placed in the array at position 3 while the value currently in position 3 must be copied into position 4. The value currently in position 4 must be copied into position 5, and so on. A similar situation exists for deletion. An implementation of this solution is shown next.

```
package List_Processing
is
  procedure Create_List;
  procedure Show_List;
  procedure Insert_In_List;
  procedure Delete_From_List;
end List_Processing;

with TEXT_IO;
package body List_Processing
is
  subtype Index_Type is INTEGER range 1 .. 200;
  subtype Names is STRING (1 .. 5);
  type List_Array is array (Index_Type) of Names;
  Name_Array : List_Array;
  Last : INTEGER := 0;

  procedure Create_List
  is
    Count : Index_Type := 1;
    More_Names : CHARACTER := 'y';
```

```
begin
  TEXT_IO.Put (Item => "Ready to create your list. ");
  TEXT_IO.New_Line;
  TEXT_IO.Put (Item => "Enter each name at the prompt.");
  TEXT_IO.New_Line;
  Get_Names:
  while (More_Names = 'Y') or (More_Names = 'y')
  loop
    Insert_In_List;
    TEXT_IO.Put (Item => "More names - y or n? ");
    TEXT_IO.Get (Item => More_Names);
    TEXT_IO.New_Line;
  end loop Get_Names;
end Create_List;

procedure Show_List
is
begin
  TEXT_IO.Put (Item => "Your list contains ");
  TEXT_IO.New_Line;
  if Last > 0
  then
    Put_Names:
    for Count in 1 .. Last
    loop
      TEXT_IO.Put (Item => Name_Array (Count));
      TEXT_IO.New_Line;
    end loop Put_Names;
  end if;
end Show_List;

procedure Insert_In_List
is
  New_Name : Names;
  Count : Index_Type := 1;
begin
  TEXT_IO.Put (Item => "Enter value to be inserted => ");
  TEXT_IO.Get (Item => New_Name);
  TEXT_IO.New_Line;
  if (Last > 0) and (Last < Index_Type'LAST)
  then
    Find_Place:
    while (Name_Array (Count) < New_Name) and (Count <= Last)
    loop
      Count := Count + 1;
    end loop Find_Place;
    Move_Names:
    for Index in reverse Count .. Last
    loop
      Name_Array (Index + 1) := Name_Array (Index);
    end loop Move_Names;
    Last := Last + 1;
    Name_Array (Count) := New_Name;
    TEXT_IO.Put (Item => "Value has been inserted.");
    TEXT_IO.New_Line;
    elsif Last = 0
    then
      Name_Array (Count) := New_Name;
      Last := Last + 1;
    else
      TEXT_IO.Put (Item => "Element can not be added. ");
      TEXT_IO.Put (Item => "Your list is full.");
      TEXT_IO.New_Line;
    end if;
  end Insert_In_List;

  procedure Delete_From_List
  is
    Count : Index_Type := 1;
    Old_Name : Names;
  begin
    TEXT_IO.Put (Item => "Enter value to be deleted => ");
    TEXT_IO.Get (Item => Old_Name);
    TEXT_IO.New_Line;
    if Last > 0
    then
      Find_Value:
      while (Name_Array (Count) /= Old_Name) and (Count < Last)
      loop
        Count := Count + 1;
      end loop Find_Value;
      if Count < Last
      then
        Move_Names:
        for Index in (Count + 1) .. Last
        loop
          Name_Array (Index - 1) := Name_Array (Index);
        end loop Move_Names;
        Last := Last - 1;
      elsif Name_Array (Last) = Old_Name
      then
        Last := Last - 1;
      else
        TEXT_IO.Put (Item => "Name not found.");
        TEXT_IO.New_Line;
      end if;
    else
      TEXT_IO.Put (Item => "Your list is empty.");
      TEXT_IO.New_Line;
    end if;
  end Delete_From_List;
end List_Processing;
```

As you read the implementation given above you should note several things. First, the package specification only describes the four operations that we wish to support. No data structures are actually mentioned in this specification. That is, no types or objects have been defined by this specification. Users of this package will not be able to create or operate on several lists at the same time. (We will discuss an alternate approach to the package design in 9.5). Therefore, since the using program is limited to one list, the list name does not need to be provided to the various routines in the package. You may begin to wonder where this one list is defined. The answer is in the package body. The declaration

```
Name_Array : List_Array;
```

creates this one and only list that will be maintained by the routines in the package body. The advantage of this approach is to hide the actual implementation of the list from the user. Remember, only those items in the package specification are accessible to a using program. By placing this list object declaration in the package body we are in effect telling the user that they do not need to worry about how the list is actually maintained. This also makes the four procedures easier to call since no parameters are required.

Now, let us look more closely at the procedure `Create_List`. The purpose of this procedure is to build a sorted list as the user provides the names that are to reside in the list. Although not apparent in this procedure, we have chosen to maintain the names in increasing order. As you read this procedure you should note that it repeatedly calls the procedure `Insert_In_List` and thus is dependent upon that routine to prompt for and insert the name into the proper position in the list.

Now look at `Show_List`. This procedure is intended to print out the contents of the list. What happens if the list is empty? In this case the object `Last` will have value 0 (that is, `Last` will always contain the length of the list). If `Last` is 0 only the message "`Your list contains` " will be printed.

Now consider `Insert_In_List`. This procedure is designed to prompt for the name to be inserted and insert the name at the appropriate location in the list. If the list is currently empty, the given name will be inserted into the first array element and `Last` will be incremented to 1. If the list currently contains 200 names (our maximum amount) the message "`Element can not be added. Your list is full.`" will appear. What happens if the name given already exists in the the list?

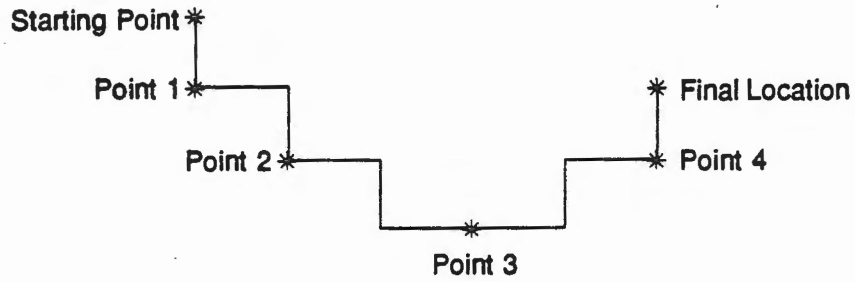
Finally, consider `Delete_From_List`. This procedure prompts for the name to be removed from the list and removes that name from the list. What happens if the name given is not in the list? What happens if the name given appears more than once in the list? What happens if the list is empty?

The package presented above was designed for simplicity to illustrate list handling operations. In a later section we will discuss the package behavior in more detail. We will also present another list handling package that illustrates alternate design decisions.

9.3.3 Dynamic Structures

In the preceding section we considered how to implement list processing operations using a static array data structure. There is another technique for representing a list using dynamic variables that are created only as they are needed. Using this technique the list elements are logically next to each other rather than physically next to each other as in our array representation. Each element must now include instructions on how to find the next element in the list. This may sound like a lot of additional overhead, however, as we will see in the next section, this permits our list to grow (or shrink) dynamically as the program executes. We will not need to guess the maximum size of the list. The only limitation that we must be aware of is the amount of available memory space. Should we exceed this value, STORAGE_ERROR will be raised.

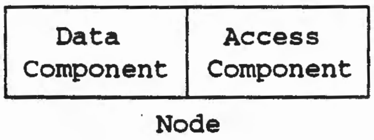
How many of you have ever participated in a road rally? Road rallies are typically organized in the following fashion. All participants meet at a predefined starting point where they receive a set of clues intended to lead them to the next point on the route. Upon reaching the next point, each team picks up an item that proves they physically visited this point along with clues leading them to the next point. Eventually the process terminates at a final location and the first team to successfully complete the route is declared the winner. The complete route might look something like



Road Rally Route
Figure 9.11

More than likely the points along the route are separated by some distance.

We can implement this type of linked structure using a dynamic data structure commonly called a linked list. Each element in the linked list is a **node.** A node is made up of a data component and an access component.



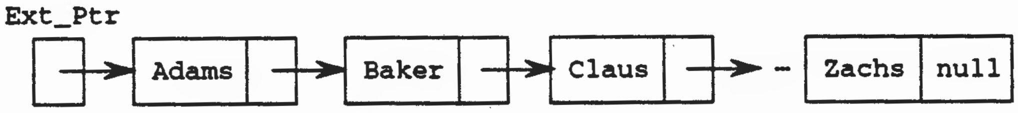
Node Element in a Linked List
Figure 9.12

The data component contains the value for that location in the list while the access component contains the instructions for (address of) the next node in the list. The list itself is accessed through an access variable that contains the address of the first node in the list. This access variable is often called the **external pointer.** Thus the first node in the list is accessed through the external pointer while every other node is accessed through the access component of the node before it.

Recall the list of names from Section 9.3.1.

```
Adams
Baker
Claus
Davis
  :
Zachs
```

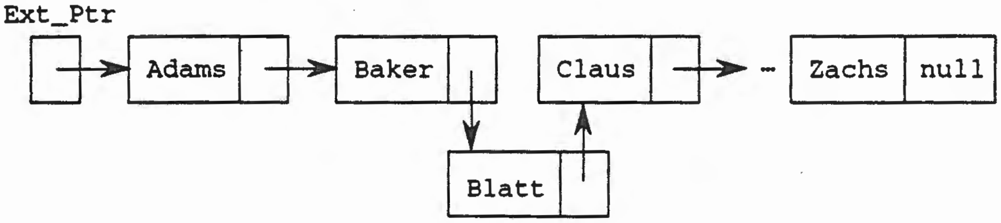
As a linked list we would have



Linked List
Figure 9.13

where our external pointer Ext_Ptr points to the first element in the list, the first element points to the second, and so on. The access component of the last element contains **null** since it is the last element.

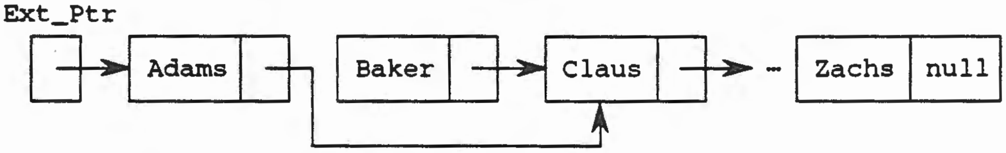
Now suppose we want to insert the name Blatt into the list.



Node Insertion
Figure 9.14

All we must do is to create a new node with data component Blatt and access component giving access to the node for Claus and change the access component of the node for Baker to point to our new node.

To delete a node we simply change the access component of the preceding node to point to the successor node. For example, to delete Baker from our original list:



Node Deletion
Figure 9.15

In the next section we will describe the actual implementation details of linked lists.

9.3.4 Advantages and Disadvantages of Linked Lists

We have talked briefly about two methods to handle list processing - using a static array and using a dynamic linked list. Which is actually better? In answer let us focus on the operations that we said we wanted to support.

List Creation and Initialization
Inputing values into a linked list is actually slower since a new node must be created and added to the list for each input value.

List Traversal
Accessing each value sequentially in the list takes approximately the same amount of time with both structures.

Element Insertion
Inserting an element as the first in the list is faster using a linked list representation. Inserting an element as the last in the list is faster using an array representation. Otherwise the amount of time required is approximately the same for the two structures.

Element Deletion

Deleting the first element of the list is faster using a linked list representation. Deleting the last element in the list is faster using an array representation. Otherwise the amount of time required is approximately the same for the two structures.

Storage Required

One additional point to consider when deciding whether to use an array or a linked list is how efficiently can you utilize the storage space required? How well can you guess the maximum number of elements in the list? How much fluctuation will take place? If you know the maximum size and the number of elements does not vary by much you may want to use an array. Otherwise use a linked list.

9.3.5 Exercises

1. What distinguishes a linked list from an array?

2. Describe a situation (other than the one illustrated in this section) in which list manipulation is required.

3. What list operations are necessary in your answer to Exercise 2?

4. Describe the steps necessary to remove the first element of a list that uses an array structure. Now describe the steps necessary to remove the first element of a list that uses a linked list structure. Which is easier and why?

5. Describe the steps necessary to remove the last element of a list that uses an array structure. Now describe the steps necessary to remove the last element of a list that uses a linked list structure. Which is easier and why?

9.4 Linked List Implementation

In the last section we introduced the idea of a linked list and provided a static array implementation. This section will investigate the alternative implementation using dynamic variables.

9.4.1 Incomplete Types

As we begin our investigation of linked lists, we are immediately faced with a dilemma. Recall that access variables must be defined in terms of two types, the access type and the designated type. We desire to build a linked list of nodes with a data component part and an access component part. This access component part must point to the next node (if any) in the list. How do we define such a situation? Let us consider the list of names from the last section and attempt to define a suitable node structure.

```
type Node is
record
  Data : STRING (1..5);
  Link : Node_Ptr;
end record;
```

It should seem reasonable that our node type will be a record with a field capable of storing one list name and a second field capable of pointing to the next node in the list. Our problem is how to define the type of that second field. In our definition above, we have used the type name `Node_Ptr`. Unfortunately we have not defined the type `Node_Ptr` and the compiler would flag this as an error. To correct this oversight, we would need the following type definition.

```
type Node_Ptr is access Node;
```

However, since Ada requires that names be defined before their use, we cannot place this type definition after the type definition of `Node` since `Node` references `Node_Ptr`. On the other hand, if we place the definition of `Node_Ptr` before that of `Node` we have a similar situation. Ada solves this "chicken and egg" problem by allowing us to use incomplete type definitions. The solution for our example is as follows:

```
type Node;      -- incomplete type definition
type Node_Ptr is access Node;
type Node is
record
  Data : STRING (1..5);
  Link : Node_Ptr;
end record;
```

An incomplete type definition is intended to tell the compiler that a full definition of the type will soon follow. This enables the compiler to acknowledge definition of the type name without knowing specific details of the type. Incomplete type definitions should only be used in access type definitions.

9.4.2 Singly Linked Lists

Now that we have the basic building blocks in place, let us actually create a linked list that contains the following names in the order shown. We consider a shortened form of the list to simplify the code segments presented. Full implementation will be discussed in Section 9.5.

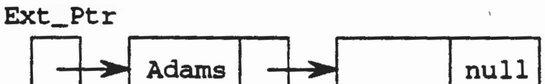
```
Adams
Baker
Claus
Zachs

procedure Build_List
is
  type Node;      -- incomplete type definition
  type Node_Ptr is access Node;
  type Node is
  record
    Data : STRING (1..5);
    Link : Node_Ptr;
  end record;
  External_Ptr, Current_Ptr : Node_Ptr;
begin
  -- create and initialize the first node
  External_Ptr := new Node;
  External_Ptr.Data := "Adams";
  External_Ptr.Link := new Node;
```

```
    -- initialize the second node
    Current_Ptr := External_Ptr.Link;
    Current_Ptr.Data := "Baker";
    Current_Ptr.Link := new Node;
    -- initialize the third node
    Current_Ptr := Current_Ptr.Link;
    Current_Ptr.Data := "Claus";
    Current_Ptr.Link := new Node;
    -- initialize the final node
    Current_Ptr := Current_Ptr.Link;
    Current_Ptr.Data := "Zachs";
    Current_Ptr.Link := null;
  end Build_List;
```

Let us analyze small segments of this program to see how the list will actually be built.

```
  -- create and initialize the first node
  External_Ptr := new Node;
  External_Ptr.Data := "Adams";
  External_Ptr.Link := new Node;
```

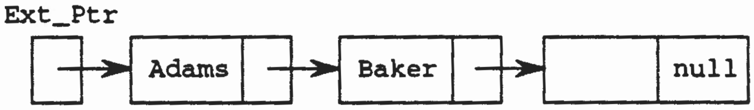
This code segment creates a new node, returns the access value to the node and places this value in the access variable `External_Ptr`. It then places the name `Adams` in the Data field of the node. Finally, a second node is created and its access value is placed in the Link field of the first node. As a result we have the following picture.



A Two Node Linked List
Figure 9.16

```
  -- initialize the second node
  Current_Ptr := External_Ptr.Link;
  Current_Ptr.Data := "Baker";
  Current_Ptr.Link := new Node;
```

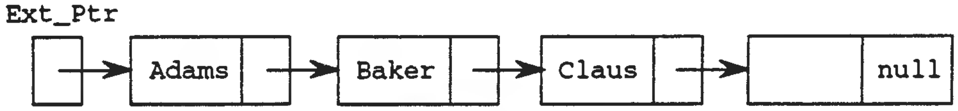
The code segment shown above first sets `Current_Ptr` to the access value of the second node. It then places the name `Baker` in the Data field of this node. Finally, it creates a third node and places its access value in the Link field of the second node. Our picture now becomes



A Three Node Linked List
Figure 9.17

```
  -- initialize the third node
  Current_Ptr := Current_Ptr.Link;
  Current_Ptr.Data := "Claus";
  Current_Ptr.Link := new Node;
```

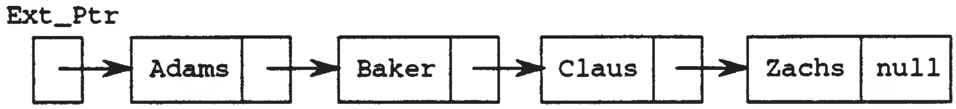
The next code segment moves `Current_Ptr` to point at the third node, places the name `Claus` in the Data field of the third node, and creates the fourth and final node. Now the picture is



A Four Node Linked List
Figure 9.18

```
-- initialize the final node
Current_Ptr := Current_Ptr.Link;
Current_Ptr.Data := "Zachs";
Current_Ptr.Link := null;
```

The final code segment finishes the task by setting `Current_Ptr` to the access value of the fourth node, inserting the name "`Zachs`" and setting the Link field to null. Thus our completed picture is as desired.



Initializing the Fourth Node
Figure 9.19

Now suppose that we want to traverse a linked list for the purpose of data extraction. That is, we want to sequentially visit each node in the structure and write out the contents of the data component. As an illustration we will modify our `Build_List` procedure to write the list back out.

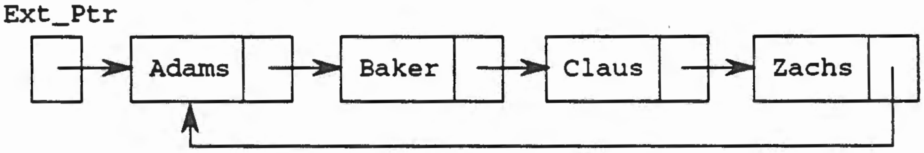
```
with TEXT_IO;
procedure Build_and_Traverse_List
is
  type Node;     -- incomplete type definition
  type Node_Ptr is access Node;
  type Node is
  record
    Data : STRING (1..5);
    Link : Node_Ptr;
  end record;
  External_Ptr, Current_Ptr : Node_Ptr;
begin
  -- create and initialize the first node
  External_Ptr := new Node;
  External_Ptr.Data := "Adams";
  External_Ptr.Link := new Node;
  -- initialize the second node
  Current_Ptr := External_Ptr.Link;
  Current_Ptr.Data := "Baker";
  Current_Ptr.Link := new Node;
```

```
    -- initialize the third node
    Current_Ptr := Current_Ptr.Link;
    Current_Ptr.Data := "Claus";
    Current_Ptr.Link := new Node;
    -- initialize the final node
    Current_Ptr := Current_Ptr.Link;
    Current_Ptr.Data := "Zachs";
    Current_Ptr.Link := null;
    -- now write out the list
    -- initialize the Current_Ptr to the first node
    Current_Ptr := External_Ptr;
    -- repeat until there are no more nodes
    Visit_Nodes:
    while Current_Ptr.Link /= null
    loop
      TEXT_IO.Put (Item => Current_Ptr.Data);
      TEXT_IO.New_Line;
      Current_Ptr := Current_Ptr.Link;
    end loop Visit_Nodes;
  end Build_and_Traverse_List;
```

9.4.3 Circularly Linked Lists

Although a singly linked list is an efficient storage mechanism, it has one drawback. Given a pointer to some node in the list, we can access all of the nodes that follow, but none of the nodes that precede it. We must always have access to the beginning of the list in order to access all of the nodes in the list. Suppose we modify our list structure slightly be making the last node in the list point back to the first node in the list. Now we have a circular linked list. The external pointer can now point to any node in the list and still provide access to the entire list.

Let us revisit our example from the last section and modify it to illustrate a circular linked list. Pictorially we have



Circular Linked List
Figure 9.20

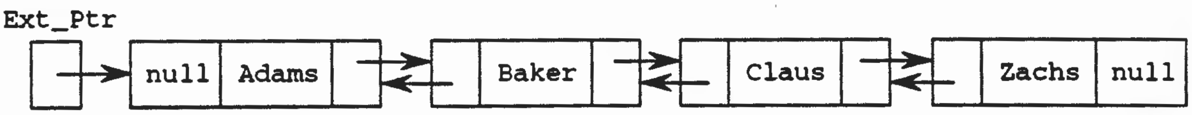
Modification of our example program is straightforward and involves changing only two lines. First, the Link field of the last node must be to point back to the first node. Second, the while test must be modified so that we exit the loop when the `Current_Ptr` once again points to the first node in the list. The modified program is shown below.

```
  with TEXT_IO;
  procedure Build_and_Traverse_Circular_List
  is
    type Node;      -- incomplete type definition
    type Node_Ptr is access Node;
    type Node is
  record
    Data : STRING (1..5);
    Link : Node_Ptr;
  end record;
  External_Ptr, Current_Ptr : Node_Ptr;
begin
  -- create and initialize the first node
  External_Ptr := new Node;
  External_Ptr.Data := "Adams";
  External_Ptr.Link := new Node;
  -- initialize the second node
  Current_Ptr := External_Ptr.Link;
  Current_Ptr.Data := "Baker";
  Current_Ptr.Link := new Node;
  -- initialize the third node
  Current_Ptr := Current_Ptr.Link;
  Current_Ptr.Data := "Claus";
  Current_Ptr.Link := new Node;
  -- initialize the final node
  Current_Ptr := Current_Ptr.Link;
  Current_Ptr.Data := "Zachs";
  Current_Ptr.Link := External_Ptr;
  -- now write out the list
  -- initialize the Current_Ptr to the first node
  Current_Ptr := External_Ptr;
  -- repeat until we return to the start
  Visit_Nodes:
  while Current_Ptr.Link /= External_Ptr
  loop
    TEXT_IO.Put (Item => Current_Ptr.Data);
    TEXT_IO.New_Line;
    Current_Ptr := Current_Ptr.Link;
  end loop Visit_Nodes;
end Build_and_Traverse_Circular_List;
```

9.4.4 Doubly Linked Lists

Circularly linked lists enable us to reach any node in the list from any starting point, an obvious advantage over singly linked lists. However, even circularly linked lists are restrictive. Consider attempting to traverse a list in reverse. In this case it would be helpful to have direct access to the node that precedes a given node in the list. We can provide this capability by including backward links as well as forward ones. This type of list structure is called a doubly linked list. The nodes in such a list are linked in both directions. Each node in a doubly linked list contains three components–the data component, the forward access component, and the backward access component.

Let us again modify our example from Section 9.4.2 to illustrate a doubly linked list. Our picture this time looks like

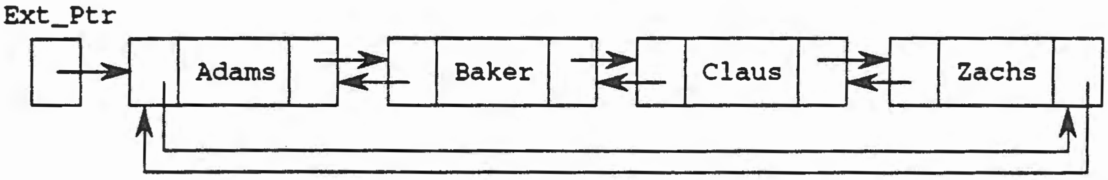


Doubly Linked List
Figure 9.21

```
with TEXT_IO;
procedure Build_and_Traverse_Double_List
is
  type Node;     -- incomplete type definition
  type Node_Ptr is access Node;
  type Node is
  record
    Data : STRING (1..5);
    Next : Node_Ptr;
    Back : Node_Ptr;
  end record;
  External_Ptr, Current_Ptr, Prior_Ptr : Node_Ptr;
begin
  -- create and initialize the first node
  External_Ptr := new Node;
  External_Ptr.Data := "Adams";
  External_Ptr.Next := new Node;
  External_Ptr.Back := null;
  Prior_Ptr := External_Ptr;
  -- initialize the second node
  Current_Ptr := External_Ptr.Next;
  Current_Ptr.Data := "Baker";
  Current_Ptr.Next := new Node;
  Current_Ptr.Back := Prior_Ptr;
  -- initialize the third node
  Prior_Ptr := Prior_Ptr.Next;
  Current_Ptr := Current_Ptr.Next;
  Current_Ptr.Data := "Claus";
  Current_Ptr.Next := new Node;
  Current_Ptr.Back := Prior_Ptr;
  -- initialize the final node
  Prior_Ptr := Prior_Ptr.Next;
  Current_Ptr := Current_Ptr.Next;
  Current_Ptr.Data := "Zachs";
  Current_Ptr.Next := null;
  Current_Ptr.Back := Prior_Ptr;
  -- now write out the list in reverse
  -- set the Current_Ptr to the last node
  Current_Ptr := External_Ptr;
  Find_Back:
  while Current_Ptr.Next /= null
  loop
    Current_Ptr := Current_Ptr.Next;
  end loop Find_Back;
  -- repeat until we return to the start
  Visit_Nodes:
  while Current_Ptr.Back /= null
  loop
    TEXT_IO.Put (Item => Current_Ptr.Data);
    TEXT_IO.New_Line;
    Current_Ptr := Current_Ptr.Back;
  end loop Visit_Nodes;
end Build_and_Traverse_Double_List;
```

Note that our new program has many more statements. The reason is that there are now more pointers to take care of. You should work through the example on your own to verify that the access values are properly set.

Although we used a simple doubly linked list in our example, a better choice might have been a circular doubly linked list. That is, we might have employed the following structure.



Circular Doubly Linked List
Figure 9.22

We leave this implementation as an exercise.

9.4.5 Exercises

1. Given the declarations

```
type Node;      -- incomplete type definition
type Node_Ptr is access Node;
type Node is
record
  Data : STRING (1..5);
  Next : Node_Ptr;
end record;

Current_Node : Node_Ptr;
```

identify the type of each of the following expressions

a. `Current_Node`
b. `Current_Node.Data`
c. `Current_Node.Data(1)`
d. `Current_Node.Next`

2. What is an incomplete type definition?

3. What is the purpose the the variable `Current_Ptr` in `Build_And_Traverse_List`?

4. How do you know when you have reached the end of a singly linked list?

5. How does a circularly linked list differ from a singly linked list?

6. How does a doubly linked list differ from a singly linked list?

7. How do you know when you have reached the end of a circularly linked list?

8. Draw a picture of a circular doubly linked list.

9. Modify the procedure `Build_And_Traverse_Double_List` to use a circular doubly linked list.

```
package Singly_Linked_List
is
  subtype Element_Type is STRING (1 .. 5);
  type Node;
  type Node_Access is access Node;
  type Node is
  record
    Data : Element_Type;
    Link : Node_Access;
  end record;
  List_Underflow : exception;
  List_Overflow : exception;
  procedure Create_List (The_List : in out Node_Access);
  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Delete_From_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Show_List (The_List : in Node_Access);
end Singly_Linked_List;
```

You will see that we have encapsulated our types and operations in the package Singly_Linked_List. Additionally we have defined two user-defined exception names, List_Underflow and List_Overflow. List_Underflow is provided to alert a using program of an attempt to remove an element from an empty list. List_Overflow is provided to alert a using program that the available storage space has been exceeded.

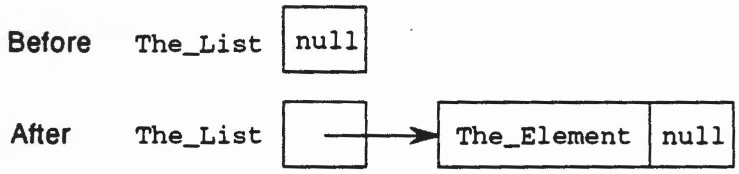
Now that we have our package specification, we can compile it and enter the compilation information into our Ada library. We now need to construct the package body thereby providing the implementation of each of our procedures.

Let us begin our implementation analysis by developing suitable algorithms for our procedures.

```
Create_List (The_List):
  set The_List to null

Insert_In_List (The_List, The_Element):
  if The_List is empty,
  then
    make a node containing The_Element
    set The_List to point to the new node
  else
    if The_Element <= the value in the first list node
    then
      make a node containing The_Element
      change the pointers to make the new node the first
        node in the list
    else
      let Current_Ptr point to the first element
      let Previous_Ptr be null
      while Current_Ptr.Data < The_Element and
        Current_Ptr.Link /= null
      loop
        advance both Current_Ptr and Previous_Ptr
      make a node containing The_Element
      change the pointers to insert the new node
```

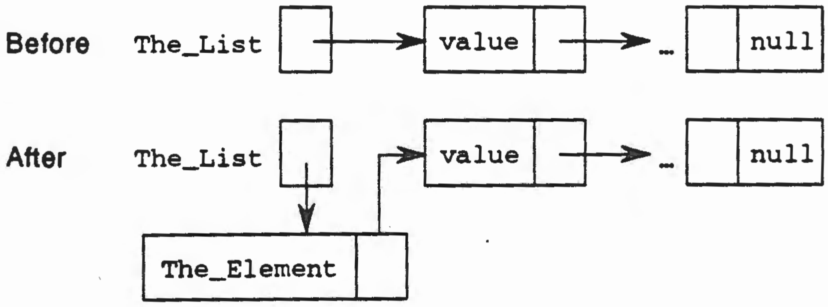
```
Delete_From_List (The_List, The_Element):
  if The_List is empty,
  then
    signal List_Underflow
  else
    if The_Element is in the first node
    then
      change The_List to point to the second node
    else
      let Current_Ptr point to the first element
      while Current_Ptr.Data /= The_Element and
        Current_Ptr.Link /= null
      loop
        advance both Current_Ptr and Previous_Ptr
      if The_Element has been found
      then
        change the pointers to delete the node
      else
        signal node not found

Show_List (The_List):
  set Current_Ptr to The_List
  while Current_Ptr /= null
  loop
    write out Current_Ptr.Data
    advance Current_Ptr
```

Since the insertion and deletion algorithms are the most difficult, let us consider their operation in more detail. For example, the insertion algorithm covers three cases–an empty list, insertion at the first node, and insertion later in the list. The first case is trivial. We just make The_List point to our new node as shown below.



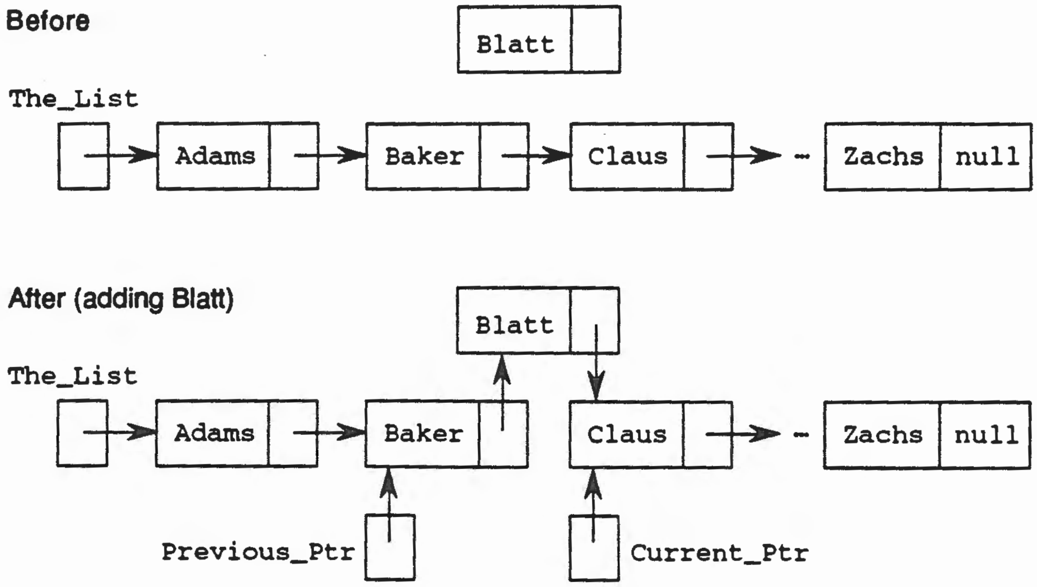
Inserting Into an Empty List
Figure 9.23

The second case is only slightly more complicated. We must make The_List point to the new node and make the new node point to the rest of the list.



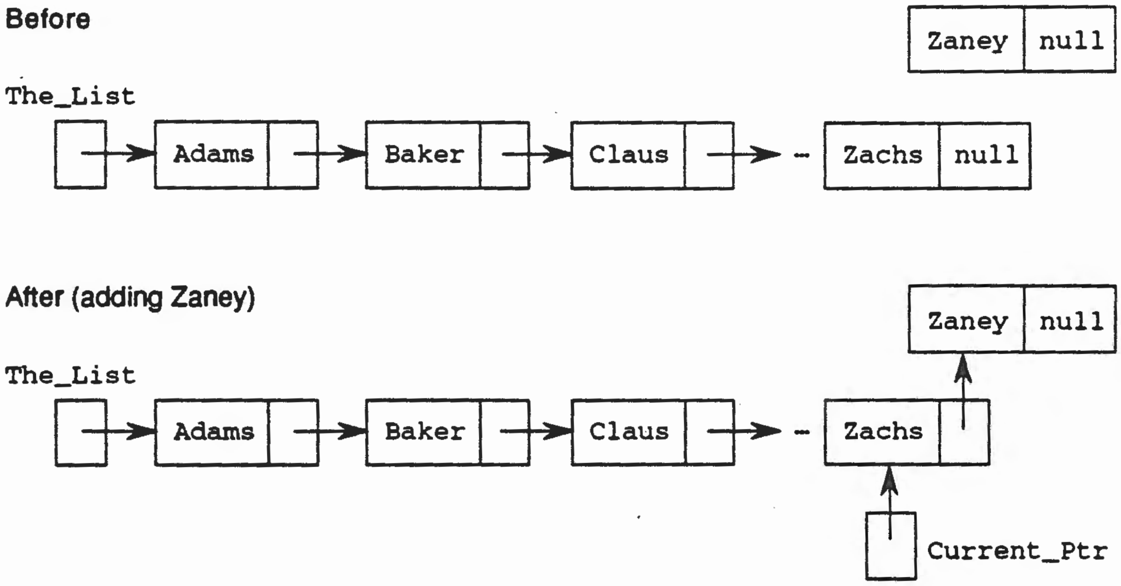
Inserting at the Beginning of List
Figure 9.24

The third case requires that we must traverse the list until we find the proper point at which to insert. We use Current_Ptr to point to the node against which we are current testing for position and Previous_Ptr to point to the node we just visited. The intention is that the new node will be inserted between Current_Ptr and Previous_Ptr. This situation is illustrated below.



Inserting in the Middle of List
Figure 9.25

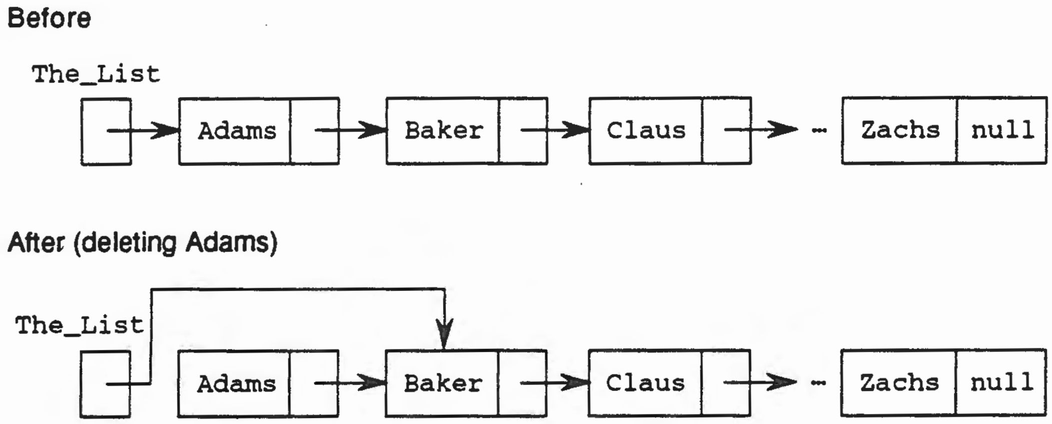
However, it is possible that the proper position for The_Element is in a new last node. This situation will arise when Current_Ptr points to the last node and Current_Ptr.Data < The_Element. When this occurs, we must make Current_Ptr.Link point to the new node as shown below.



Inserting at the End of List
Figure 9.26

The deletion algorithm covers four cases–an empty list, deleting the first node, deleting later in the list, and element not present. The first and last cases are straightforward. Let us look closely at the other two cases.

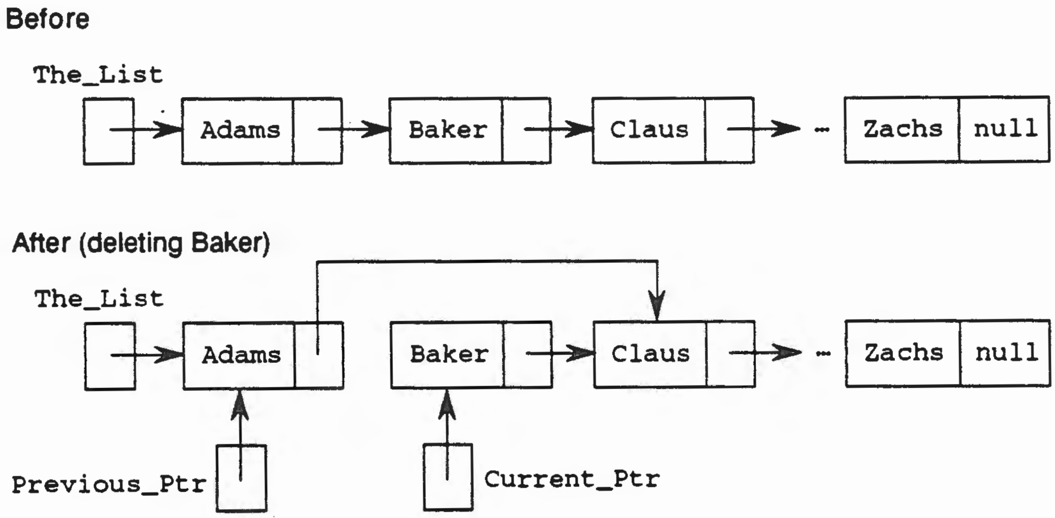
Deleting the first element in the list requires that we change `The_List` to point to the second element in the list if there is one.


Before

After (deleting Adams)

Deleting from the Beginning of List
Figure 9.27

Deleting from later in the list requires that we must traverse the list until we find the desired node. We use `Current_Ptr` and `Previous_Ptr` as before with the intention that `Current_Ptr` will stop on the node that we are to delete. We then change pointers to remove the node indicated by `Current_Ptr`.


Before

After (deleting Baker)

Deleting from the Middle of List
Figure 9.28

However, it is possible that the value that we wish to delete is not in the list. In this scenario, `Current_Ptr` will stop at the last node and `Current_Ptr.Data /= The_Element`. If this is the case, we issue a message stating the value was not found.

Now we are ready to proceed with our implementation by presenting the package body.

```
with TEXT_IO; -- needed for the list traversal
package body Singly_Linked_List
is
  procedure Create_List (The_List : in out Node_Access)
  is
  begin
    The_List := null;
  end Create_List;

  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type)
  is
    Previous_Ptr : Node_Access;
    New_Node, Current_Ptr : Node_Access;
  begin
    -- test for an empty list
    if The_List = null
    then
      -- build the first node
      The_List := new Node;
      The_List.Data := The_Element;
      The_List.Link := null;
    -- test against the first element
    elsif The_Element <= The_List.Data
    then
      -- insert the value as the first node
      Current_Ptr := new Node;
      Current_Ptr.Data := The_Element;
      Current_Ptr.Link := The_List;
      The_List := Current_Ptr;
    else
      -- initialize two pointers
      Current_Ptr := The_List;
      Previous_Ptr := null;
      -- search for the proper place
      Find_Place:
      while (Current_Ptr.Data < The_Element) and
            (Current_Ptr.Link /= null)
      loop
        Previous_Ptr := Current_Ptr;
        Current_Ptr := Current_Ptr.Link;
      end loop Find_Place;
      -- insert the new value
      New_Node := new Node;
      New_Node.Data := The_Element;
      if Current_Ptr.Data >= The_Element
      then
        New_Node.Link := Current_Ptr;
        Previous_Ptr.Link := New_Node;
      else
        Current_Ptr.Link := New_Node;
        New_Node.Link := null;
      end if;
    end if;
  exception
    when STORAGE_ERROR => raise List_Overflow;
  end Insert_In_List;
  procedure Delete_From_List (The_List : in out Node_Access;
    The_Element : in Element_Type)
  is
    Previous_Ptr : Node_Access;
    Current_Ptr : Node_Access := The_List;
  begin
    -- test for an empty list
    if The_List = null
    then
      raise List_Underflow;
    end if;
    -- test against the first element
    if The_List.Data = The_Element
    then
      The_List := The_List.Link;
    else -- search the list
      Find_Element:
      while (Current_Ptr.Data /= The_Element) and
            (Current_Ptr.Link /= null)
      loop
        Previous_Ptr := Current_Ptr;
        Current_Ptr := Current_Ptr.Link;
      end loop Find_Element;
      -- remove the value if it was found
      if Current_Ptr.Data = The_Element
      then
        Previous_Ptr.Link := Current_Ptr.Link;
      else
        TEXT_IO.Put (Item => "Element not found.");
        TEXT_IO.New_Line;
      end if;
    end if;
  end Delete_From_List;

  procedure Show_List (The_List : in Node_Access)
  is
    Current_Ptr : Node_Access := The_List;
  begin
    TEXT_IO.Put (Item => "Here is your list ");
    TEXT_IO.New_Line;
    Travel_List:
    while Current_Ptr /= null
    loop
      TEXT_IO.Put (Item => Current_Ptr.Data);
      TEXT_IO.New_Line;
      Current_Ptr := Current_Ptr.Link;
    end loop Travel_List;
  end Show_List;
end Singly_Linked_List;
```

As you read the package body you should compare it to the package specification and to the various algorithms. Does it provide the functionality that we wanted?

We can now compile the package body and place the compilation information in our Ada library. In order to test our new package, suppose we construct a sample test program.

```
with TEXT_IO, Singly_Linked_List;
procedure Build_List
is
  Name : STRING (1 .. 5);
  Name_List : Singly_Linked_List.Node_Access;
  Choice : CHARACTER := ' ';
begin
  TEXT_IO.Put (Item => "This program tests singly linked list ");
  TEXT_IO.Put (Item => "package.");
  TEXT_IO.New_Line;
  -- first create the list
  Singly_Linked_List.Create_List (Name_List);
  -- now exercise the options
  Test_List:
  while Choice /= 'E'
  loop
    TEXT_IO.Put (Item => "Choose your next action ");
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "A)dd a name");
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "D)elete a name");
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "S)how the list");
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "E)xit");
    TEXT_IO.New_Line;
    TEXT_IO.Put (Item => "Your choice ? ");
    TEXT_IO.Get (Item => Choice);
    TEXT_IO.New_Line;
    case Choice
    is
      when 'A' | 'a' =>
        TEXT_IO.Put (Item => "Next name => ");
        TEXT_IO.Get (Item => Name);
        TEXT_IO.New_Line;
        Singly_Linked_List.Insert_In_List (Name_List, Name);
      when 'D' | 'd' =>
        TEXT_IO.Put (Item => "Name to delete => ");
        TEXT_IO.Get (Item => Name);
        TEXT_IO.New_Line;
        Singly_Linked_List.Delete_From_List (Name_List, Name);
      when 'S' | 's' =>
        Singly_Linked_List.Show_List (Name_List);
      when 'E' | 'e' =>
        Choice := 'E';
      when others =>
        TEXT_IO.Put (Item => "Try again.");
        TEXT_IO.New_Line;
    end case;
  end loop Test_List;
  TEXT_IO.Put (Item => "Thank you.");
  TEXT_IO.New_Line;
```

```
exception
  when Singly_Linked_List.List_Overflow =>
    TEXT_IO.Put (Item => "Not enough memory space.");
    TEXT_IO.New_Line;
  when Singly_Linked_List.List_Underflow =>
    TEXT_IO.Put (Item => "Your list is empty.");
    TEXT_IO.New_Line;
end Build_List;
```

The procedure shown above is designed to allow us to interactively test our new package by creating an access variable for a list and then allowing additions or deletions of as many elements as we desire. We establish visibility to our package through the "with" clause.

```
with TEXT_IO, Singly_Linked_List;
```

We then proceed to call the appropriate operations from our package by referring to the package name "." operation name. For example,

```
Singly_Linked_List.Insert_In_List (Name_List, Name);
```

calls the `Insert_In_List` operation from the package `Singly_Linked_List`. This is the same notation that you use in the statement

```
TEXT_IO.Put (Item => "Hi there!");
```

Study the sample program. How could you use it to produce the following list?

```
adams
baker
claus
zachs
```

9.5.2 Operations of a Circular Linked List

Recall that a circular linked list differs from a singly linked list in only one access value–that of the last node in the list. For this node, the access field contains the access value of the first node in the list. How would we modify our package from the last section to apply to a circular linked list?

Our data types need not change to accommodate a circular linked list but some of our algorithms must change slightly. We present the modified algorithms below.

```
Insert_In_List (The_List, The_Element):
  if The_List is empty,
  then
    make a node containing The_Element
    set The_List to point to the new node
    set the access field of the new node to point to itself
  else
    if The_Element <= the value in the first list node
    then
      make a node containing The_Element
      change the pointers to make the new node the first
        node in the list
      traverse the list until the last node is found
      make the last node point to the new first node
    else
      let Current_Ptr point to the first element
      let Previous_Ptr be null
      while Current_Ptr.Data < The_Element and
        Current_Ptr.Link /= The_List
      loop
        advance both Current_Ptr and Previous_Ptr
      make a node containing The_Element
      change the pointers to insert the new node

Delete_From_List (The_List, The_Element):
  if The_List is empty,
  then
    signal List_Underflow
  else
    if The_Element is in the first node
    then
      if this is the only node
      then
        make The_List null
      else
        change The_List to point to the second node
        make the last node point to the second node
    else
       let Current_Ptr point to the first element
       while Current_Ptr.Data /= The_Element and
         Current_Ptr.Link /= The_List
       loop
         advance both Current_Ptr and Previous_Ptr
       if The_Element has been found
       then
         change the pointers to delete the node
       else
         signal node not found

Show_List (The_List):
  if The_List is not null
  then
    write out The_List.Data
    set Current_Ptr to The_List.Link
    while Current_Ptr /= null
    loop
      write out Current_Ptr.Data
      advance Current_Ptr
```

Our new package specification and body are shown below.

```
package Circular_Linked_List
is
  subtype Element_Type is STRING (1 .. 5);
  type Node;
  type Node_Access is access Node;
  type Node is
  record
    Data : Element_Type;
    Link : Node_Access;
  end record;
  List_Underflow : exception;
  List_Overflow : exception;
  procedure Create_List (The_List : in out Node_Access);
  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Delete_From_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Show_List (The_List : in Node_Access);
end Circular_Linked_List;

with TEXT_IO;
package body Circular_Linked_List
is
  procedure Create_List (The_List : in out Node_Access)
  is
  begin
    The_List := null;
  end Create_List;

  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type)
  is
    Previous_Ptr, Travel_Ptr : Node_Access;
    New_Node, Current_Ptr : Node_Access;
  begin
    -- test for an empty list
    if The_List = null
    then
      The_List := new Node;
      The_List.Data := The_Element;
      The_List.Link := The_List;
    -- test against the first element
    elsif The_element <= The_List.Data
    then
      Current_Ptr := new Node;
      Current_Ptr.Data := The_Element;
      Current_Ptr.Link := The_List;
      -- find the last element in the list
      Travel_Ptr := The_List;
      Find_End:
      while Travel_Ptr.Link /= The_List
      loop
        Travel_Ptr := Travel_Ptr.Link;
      end loop Find_End;
      -- make the last element point to the new first element
      Travel_Ptr.Link := Current_Ptr;
      The_List := Current_Ptr;
    else
      Current_Ptr := The_List;
      Previous_Ptr := null;
      Find_Place:
      while (Current_Ptr.Data < The_Element) and
            (Current_Ptr.Link /= The_List)
      loop
        Previous_Ptr := Current_Ptr;
        Current_Ptr := Current_Ptr.Link;
      end loop Find_Place;
    -- insert the new value
    New_Node := new Node;
    New_Node.Data := The_Element;
    if Current_Ptr.Data >= The_Element
    then
      New_Node.Link := Current_Ptr;
      Previous_Ptr.Link := New_Node;
    else
      Current_Ptr.Link := New_Node;
      New_Node.Link := The_List;
    end if;
  end if;
exception
  when STORAGE_ERROR => raise List_Overflow;
end Insert_In_List;

procedure Delete_From_List (The_List : in out Node_Access;
  The_Element : in Element_Type)
is
  Previous_Ptr, Travel_Ptr : Node_Access;
  Current_Ptr : Node_Access := The_List;
begin
  -- test for an empty list
  if The_List = null
  then
    raise List_Underflow;
  end if;
  -- test against the first element
  if The_List.Data = The_Element
  then
  -- test to see if this is the only element
  if The_List.Link = The_List
  then
    The_List := null;
  else
    -- find the end of the list
    Travel_Ptr := The_List;
    Find_End:
    while Travel_Ptr.Link /= The_List
    loop
      Travel_Ptr := Travel_Ptr.Link;
    end loop Find_End;
    -- delete the element
    The_List := The_List.Link;
    Travel_Ptr.Link := The_List;
  end if;
  else -- search the list
    Find_Element:
    while (Current_Ptr.Data /= The_Element) and
          (Current_Ptr.Link /= The_List)
    loop
      Previous_Ptr := Current_Ptr;
      Current_Ptr := Current_Ptr.Link;
    end loop Find_Element;
```

```
      -- remove the value if it was found
      if Current_Ptr.Data = The_Element
      then
        Previous_Ptr.Link := Current_Ptr.Link;
        TEXT_IO.Put (Item => "Element removed.");
        TEXT_IO.New_Line;
      else
        TEXT_IO.Put (Item => "Element not found.");
        TEXT_IO.New_Line;
      end if;
    end if;
  end Delete_From_List;

  procedure Show_List (The_List : in Node_Access)
  is
    Current_Ptr : Node_Access;
  begin
    TEXT_IO.Put (Item => "Here is your list ");
    TEXT_IO.New_Line;
    if The_List /= null
    then
      TEXT_IO.Put (Item => The_List.Data);
      TEXT_IO.New_Line;
      Current_Ptr := The_List.Link;
      Travel_List:
      while Current_Ptr /= The_List
      loop
        TEXT_IO.Put (Item => Current_Ptr.Data);
        TEXT_IO.New_Line;
        Current_Ptr := Current_Ptr.Link;
      end loop Travel_List;
    end if;
  end Show_List;
end Circular_Linked_List;
```

How would you modify our testing program from 9.5.1 to exercise this new package?

9.5.3 Operations on Doubly Linked Lists

We have one more situation to consider, that of a doubly linked list. As you recall from 9.4, a doubly linked list provides access to both the successor and predecessor nodes of a given node. Therefore we must augment our type definitions to include the predecessor access field as follows.

```
subtype Element_Type is STRING (1 .. 5);
type Node;
type Node_Access is access Node;
type Node is
record
  Data : Element_Type;
  Last : Node_Access; -- the predecessor
  Next : Node_Access; -- the successor
end record;
```

We leave the algorithm analysis as an exercise and present the package specification and body below.

```
package Doubly_Linked_List
is
  subtype Element_Type is STRING (1 .. 5);
  type Node;
  type Node_Access is access Node;
  type Node is
  record
    Data : Element_Type;
    Last : Node_Access;
    Next : Node_Access;
  end record;
  List_Underflow : exception;
  List_Overflow : exception;
  procedure Create_List (The_List : in out Node_Access);
  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Delete_From_List (The_List : in out Node_Access;
    The_Element : in Element_Type);
  procedure Show_List (The_List : in Node_Access);
end Doubly_Linked_List;

with TEXT_IO;
package body Doubly_Linked_List
is
  procedure Create_List (The_List : in out Node_Access)
  is
  begin
    The_List := null;
  end Create_List;

  procedure Insert_In_List (The_List : in out Node_Access;
    The_Element : in Element_Type)
  is
    New_Node, Current_Ptr : Node_Access;
  begin
    -- test for an empty list
    if The_List = null
    then
      The_List := new Node;
      The_List.Data := The_Element;
      The_List.Last := null;
      The_List.Next := null;
      -- test against the first element
    elsif The_Element <= The_List.Data
    then
      Current_Ptr := new Node;
      Current_Ptr.Data := The_Element;
      Current_Ptr.Last := null;
      Current_Ptr.Next := The_List;
      The_List := Current_Ptr;
    else
      Current_Ptr := The_List;
      Find_Place:
      while (Current_Ptr.Data < The_Element) and
            (Current_Ptr.Next /= null)
      loop
        Current_Ptr := Current_Ptr.Next;
      end loop Find_Place;
    -- insert the new value
    New_Node := new Node;
    New_Node.Data := The_Element;
    if Current_Ptr.Data >= The_Element
    then
      New_Node.Next := Current_Ptr;
      New_Node.Last := Current_Ptr.Last;
      Current_Ptr.Last := New_Node;
      Current_Ptr := New_Node.Last;
      Current_Ptr.Next := New_Node;
    else
      Current_Ptr.Next := New_Node;
      New_node.Last := Current_Ptr;
      New_Node.Next := null;
    end if;
  end if;
exception
  when STORAGE_ERROR => raise List_Overflow;
end Insert_In_List;

procedure Delete_From_List (The_List : in out Node_Access;
  The_Element : in Element_Type)
is
  Previous_Ptr : Node_Access;
  Current_Ptr : Node_Access := The_List;
begin
  -- test for an empty list
  if The_List = null
  then
    raise List_Underflow;
  end if;
  -- test against the first element
  if The_List.Data = The_Element
  then
    The_List := The_List.Next;
    if The_List /= null
    then
      The_List.Last := null;
    end if;
  else -- search the list
  Previous_Ptr := null;
  Find_Element:
  while (Current_Ptr.Data /= The_Element) and
        (Current_Ptr.Next /= null)
  loop
    Previous_Ptr := Current_Ptr;
    Current_Ptr := Current_Ptr.Next;
  end loop Find_Element;
  -- remove the value if it was found
    if Current_Ptr.Data = The_Element
    then
      Previous_Ptr.Next := Current_Ptr.Next;
      if Current_Ptr.Next /= null
      then
        Current_Ptr := Current_Ptr.Next;
        Current_Ptr.Last := Previous_Ptr;
      end if;
```

```
        TEXT_IO.Put (Item => "Element removed.");
        TEXT_IO.New_Line;
      else
        TEXT_IO.Put (Item => "Element not found.");
        TEXT_IO.New_Line;
      end if;
    end if;
  end Delete_From_List;

  procedure Show_List (The_List : in Node_Access)
  is
    Current_Ptr : Node_Access := The_List;
  begin
    TEXT_IO.Put (Item => "Here is your list ");
    TEXT_IO.New_Line;
    Travel_List:
    while Current_Ptr /= null
    loop
      TEXT_IO.Put (Item => Current_Ptr.Data);
      TEXT_IO.New_Line;
      Current_Ptr := Current_Ptr.Next;
    end loop Travel_List;
  end Show_List;
end Doubly_Linked_List;
```

You should take the time to convince yourself that the operations shown are indeed correct.

9.5.4 Exercises

1. Draw a picture that would represent the list

```
Adams
Baker
Claus
Zachs
```

as a singly linked list.

2. Modify your picture from Exercise 1 to insert `Davis`.

3. Draw a picture that would represent the list in Exercise 1 as a circular linked list.

4. Modify your picture from Exercise 3 to delete `Zachs`.

5. Draw a picture that would represent the list in Exercise 1 as a doubly linked list.

6. Modify your picture from Exercise 5 to delete `Adams`.

7. Modify the insertion and deletion algorithms for doubly linked lists.

8. Write an Ada procedure called `Show_Reverse` that could be added to the `Circular_Linked_List` package. This procedure should print out the elements in a circularly linked list in reverse.

9. Write an Ada procedure called `Size_Of` that could be added to the linked list package of your choice. This procedure should return the number of elements in the list.

Chapter 10

Ada Package Design with Box Structures

We have seen Ada packages used from the very start in Ada programs. Ada packages represent a new capability not present in other widely used programming languages to combine type and object definitions with processing. An Ada package is defined in two separate parts, namely the *package specification* which describes the external capabilities for users, and the *package body* which describes the internal means of meeting those capabilities. The package specification may contain type and object declarations and external information on subprograms or other packages. The package body may contain more type and object declarations and the internal completion of all the subprograms and packages referenced in the package specification. It defines a new program unit in Ada with the potential capability of creating abstract data types and abstract state machine behavior, beyond the simply functional behavior as a subprogram procedure or function. The subprograms of a package are connected with its data between calls on the subprogram procedures and functions.

For example, TEXT_IO is a standard Ada package that creates abstract data types of wide use in Ada. The files and parameters accessed during any TEXT_IO procedure or function continue to exist after their execution is completed for access by the next procedure or function, on until the TEXT_IO package itself is terminated. Section 14.3.10, "Specification of the Package Text_IO" of the Ada Programming Language, ANSI/MIL/STD-1815A, defines the Ada package (specification) TEXT_IO.

In the preceding Chapter 9, packages `List_Processing`, `Singly_Linked_List`, `Circular_Linked_List`, `Doubly_Linked_List` describe abstract state machine behavior in dealing with linked lists in various ways.

In this Chapter 10, we go from making good use of packages to creating them as needed as in Chapter 9. Packages are a critical and necessary part of good program design in Ada. They allow the combination of stored data with operations on them for various purposes. They permit the discipline of explicit type and object operations and no other ways to reach the data. This allows the optimization of the data and processes to meet program needs. Such optimization may be quite different in different circumstances, depending on what hardware is being used and what performance is required. In some cases available computer storage space is limited. For example, storing data in an array which is mostly filled with zeroes may allow easy descriptions of operations on the data but require a lot of storage space not really required. In other cases the available throughput over time is limited. For example, storing data in CHARACTER form may allow easy treatment of IO operations but require more time to deal with arithmetic operations than practical. A good solution in one applied situation may be quite wrong in another.

The good use of Ada packages permits software to be *developed* and *certified correct* under *statistical quality control* to well formed *specifications* of user requirements given in *construction increments*. This capability requires a sound development methodology to create well testable software solely by *design* and *verification*, in particular with *no unit testing* by the developers. Unit testing and fixing is the most error prone activity in software development today, leading to deeper failures in fifteen per cent or more of the fixes in large programs. Ada packages are even more difficult to test well than subprogram procedures and functions. Testing subprograms involves independent executions of the procedures or functions at each step. But testing the subprograms of a package involves dependent executions of the procedures and functions for each period of existence of the package, possibly with data stored and updated during that time.

This capability also requires a test design based not only on the behavioral and performance specifications, but also on usage specifications and how critical each test case is to system behavior. Such a test design is based on a *stratified statistical strategy* derived from the statistics of usage expected for the software. For an important case, a stratus may consist of the single case (with probability 1), or a small subset of cases, on out to strata containing large sections of the software. A total test design defines each stratus (possibly hundreds or thousands) and the number of tests in each strata.

Successful testing of packages without any failures found leads to a *certification of correctness* of the software. If failures are later found the certification is *negated*. If failures are fixed the certification process can be started again. The probability of failures can be described in terms of statistics. The Greek letter sigma (σ) defines the standard deviation of a measurement, in this case a failure or not, under the curve of the normal distribution. Certification continues with software release to users, moving with confidence from typical 3 sigma[1] at release to and beyond 6 sigma[2] with sufficient usage without failures. As noted, if software is entirely correct, there is no way to be sure of that except by testing and usage without failures. However, the longer testing and usage goes on without failure, the greater the subjective confidence can be in that correctness.

10.1 Package Uses In Program Design

A *package design* consists of two separate parts, first the *package specification* which is always required, of the form

```
package <name> is
  -- declarations of objects, types, subtypes,
    -- subprogram specifications, package specifications
end <name>;
```

and second the *package body* which is required for using subprograms and other packages, of the form

```
package body <name> is
  -- declarations of objects, types, subtypes, subprogram bodies,
    -- package bodies
end <name>;
```

The package specification declares the resources that may be referenced outside the package. The package is said to export these entities, including objects, types, subtypes, subprograms, even other packages. In summary, the package specification provides for declarations of all types, including subprograms and packages, but not for execution bodies of subprograms or packages.

The package body has a form somewhat similar to the body of a subprogram, with declarations followed by an optional block of statements for an initialization step when first invoked, and by an optional exception handler. Subprograms and packages whose external references appear in the package specification must be completed in the package body. The initialization, if it appears, is carried out as a continuation of the declaration process, following the elaboration of the declarations, as an initial execution that is never repeated or reachable thereafter. This initialization is optional, needed in some cases, not needed in others. In summary, the package body completes the definition of a package specification and body pair, giving subprogram internals and possible initialization and exception handling for the subprograms introduced in the package specification.

Note here that the term *package specification* has a specific Ada meaning, namely the external references and resources associated with the package, and not the complete behavior of the package. As already introduced, the term *specification*, more specifically *behavior specification*, refers to the entire behavior of the package as defined by the complete code of the package specification and package body (if defined). In this case, the behavior of a package is defined by the declarations of the package specification, the initialization section, the declarations of the package body, and the subprograms and packages possibly named in the package specification and completed in the package block. The subprograms of packages can be called just as any other subprograms, but the behavior of package subprograms are dependent. In particular, the package data continues to exist after any specific subprogram execution is completed, and will be available to any other subprogram executed, as long as the package continues to exist.

For example, a simple package to return Fibonacci values with an Ada function can be provided as a package specification and package body as shown below. Recall that Fibonacci values are integers associated with integers 0, 1, ... which are 1 for integers 0 and 1, and then the sum of the two preceding Fibonacci values from then on.

```
package Fibonacci_Package
is
  subtype Argument is NATURAL range 0 .. 20;
  function Fibonacci (Place : Argument) return POSITIVE;
end Fibonacci_Package;

package body Fibonacci_Package
is
  type Fibonacci_Table is array (Argument) of POSITIVE;
  Table : Fibonacci_Table;
  function Fibonacci (Place : Argument) return POSITIVE
  is
  begin
    return Table (Place);
  end Fibonacci;
begin -- Initialization of array Table
  Table (0) := 1;
  Table (1) := 1;
  Define_Fibonacci_Table:
  for Next in 2 .. Argument'LAST
  loop
    Table (Next) := Table (Next - 2) + Table (Next - 1);
  end loop Define_Fibonacci_Table;
end Fibonacci_Package;
```

First, package `Fibonacci_Package` specification defines the way into the package for users, using the function `Fibonacci` with `Place`, a NATURAL Argument. Next, on elaboration of the package specification and package body, the initial section of the package body (begin ... end) will be executed immediately after the elaborations, setting values for `Table` as

```
Table (0) = 1, Table (2) = 1, Table (3) = 2, Table (4) = 3, ...
```

This initial section will never be executed again. But now, values for `Table` from 0 to 20 have been computed and stored between calls on the function `Fibonacci` of the package `Fibonacci_Package`. From then on, any calls made on function `Fibonacci` with an integer argument between 0 and 20, will return the appropriate value from `Table`. Note that `Table` is invisible to the user of this package except through calls on the function `Fibonacci`.

As can be seen in this example, good packages begin with good and explicit behavior specifications of behavior required, then continue with appropriate designs, and finally conclude with verifications and certifications of their correctness. The extension of subprograms to packages is in some ways very small, in other ways very large. Packages are simply made up of declarations, subprograms, other packages, with no new ideas besides that in the terms package (specification) and package body. But the use of packages by other parts of a program can be very powerful, in combining data and processes. The behavior specifications of packages can be *abstract data types*, that recognize operations on a set of data, or *abstract state machines*, that recognize the storage of data between uses of a package, and the power of subprograms and other packages to deal with that data each time the package is called on through them. The example above is a simple form of an abstract state machine with all data constant. All stored data is defined when the package is declared and never modified from then on. After initialization, only one entry to the package will return data, but no other entry can modify the data. The linked list package examples of Chapter 9 are also abstract state machines with variable internal data. But, as we see next, packages can also define much simpler but useful operations than abstract data types and abstract state machines.

10.1.1 Booch Package Categories

The definition of packages is clearly motivated to deal with handling data as abstract state machines. But packages turn out to have many other simpler uses. Mr. Grady Booch[3] first identified four specific kinds of package uses, namely

1) **Declaration groups:** those defined entirely in package specifications, that export objects and types only, and export no subprograms or other packages,

2) **Subprogram groups:** those defined in package specifications and bodies, that export no objects or types, and export subprograms and other packages only, but do not maintain any state information,

3) **Abstract data types:** those defined in package specifications and bodies, that export objects and types, and export subprograms and other packages, but maintain no state information in the package body,

4) **Abstract state machines:** those defined in package specifications and bodies, that export subprograms and other packages, and maintain state information in the package body.

The first two kinds of packages are straightforward to define and recognize. But the latter two kinds of packages expand the usual uses, but not the logic, of abstract data types and abstract state machines in software design and architecture. The major understanding comes from expanding the usual terms for arguments in data types and state machines. In Ada packages, all references to their facilities are through their subprograms, both procedures and functions, with their parameters. The point to recognize is that the names of these procedures and functions are parameters, just as the data they may call. That is, any call by a using program on a package will name the subprogram to be invoked as well as giving the data for that particular use. It is the package that behaves like a data type or a state machine, and it is its procedures and functions that provide entries to that data type or state machine.

There are very good uses for all four kinds of packages. Declaration group packages can be used to store data declarations for related objects together that can become a part of subprograms or packages, or even part of larger declaration sections. Subprogram group packages can be used to store subprograms and other packages together that become parts of larger sections. Abstract data type packages with both data declarations in package specifications and subprograms and other packages, provide powerful capabilities to track and operate on data in multiple and successive calls to subprograms. Finally, abstract state machine packages with both package body data declarations and subprograms and other packages, provide powerful capabilities to track and operate on internally stored data in multiple and successive calls to subprograms.

In the most general form, abstract state machine packages provide a basis for storing data between the uses of subprograms–procedures and functions–and the calling of these subprograms as required. The individual procedures and functions are literally part of the operation of the package, not independent procedures and functions. That is, the package itself behaves as an abstract state machine with entries through the various procedures and functions of the package. The internal state of the abstract state machine is defined by the data declared in the package body, which is updated and preserved from call to call on the abstract state machine through its procedures and functions.

We begin with specific illustrations of all four kinds of these packages, then go into their behavior specifications, program design, verification and certification as packages in the rest of the Chapter. In this process we introduce the concept of box structured design of packages. Box structures consist of three separate descriptions of a package, namely a *black box*, a *state box,* and a *clear box*. The first step of box structured design begins with identifying *black box specifications*, that map stimuli histories into the next response with no reference to, or use of, internal storage. In the next step, the *state box design* defines the internal data and the subprogram specifications that map the current stimulus and the state into the response and next state. Then in the final step, the *clear box design* expands the state box specification into the next level design of connected black boxes.

In summary, there are four general kinds of package uses, dealing with the presence or not of exportable data declarations and of subprograms and other packages. *Declaration group* packages can be used to collect declarations of related data, with no executable parts to deal with that data. In this case, to be of value, such a package must be associated with executable code that makes use of the declarations. But there are, indeed, many important uses of this form in complex program design. *Subprogram group* packages can also be used to collect subprograms and other packages, with no declarations of data between them. Again, there are many important uses of this form in complex program design. *Abstract data type* packages can be used to both store and process data using data declarations in package specifications and subprograms and other packages. Abstract data types are especially useful in organizing programs with applied data types. Finally, *abstract state machine* packages can be used to both store and process data using data declarations in package bodies as well as specifications and subprograms and other packages. Abstract state machines give the full power of packages.

10.1.2 Using Declaration Group Packages

Declaration group packages provide a name and library storage for a set of data declarations that appear useful to store and make use of together. They can be retrieved either in with clauses or in package declarations. In illustration, declarations for geometric objects such as circles, squares, equilateral and right triangles, etc. can be provided for use in a single package. The more declarations assigned to a package name, the easier the individual declarations are to retrieve, but the larger are the units of retrieval. The same declarations may appear in different packages, and it may make sense to have several declaration group packages in an area to give a choice of how large a package is needed for a specific program.

For example declarations of a set of simple geometric objects can be formed into a package as follows.

```
package Geometric_Objects -- simple two dimensional objects
is
  Circle                : NATURAL := 1;
  Equilateral_Triangle : NATURAL := 1;
  Square                : NATURAL := 1;
  type Ellipse_Type is array (1 .. 2) of NATURAL;
  type Triangle_Type is array (1 .. 3) of NATURAL;
  type Isosceles_Triangle_Type is array (1 ..2) of NATURAL;
  type Right_Triangle_Type is array (1 .. 2) of NATURAL;
  type Rectangle_Type is array (1 .. 2) of NATURAL;
  type Parallelogram_Type is array (1 .. 3) of NATURAL;
  type Quadrilateral_Type is array (1 .. 6) of NATURAL;
  Ellipse               : Ellipse_Type := (2, 1);
  Triangle              : Triangle_Type := (3, 4, 5);
  Isosceles_Triangle    : Isosceles_Triangle_Type := (1, 1);
  Right_Triangle        : Right_Triangle_Type := (3, 4);
  Parallelogram         : Parallelogram_Type := (1, 1, 1);
  Quadrilateral         : Quadrilateral_Type := (others => 1);
end Geometric_Objects;

-- no package body required
```

These objects are not arbitrary as geometric figures. Additional documentation is needed to line up data with the object described. For example, in Isosceles_Triangle which is the unique side, which the paired side, in Right_Triangle, which two sides are listed? As an alternative, the declarations of Circle, Equilateral_Triangle and Square as arrays might seem strange, but are quite legal, to go along with the other elliptic, triangle and quadrilateral cases. In this case, the package would become

```
package Geometric_Objects_1 -- simple two dimensional objects
is
  type Circle_Type is array (1 .. 1) of NATURAL;
  type Equilateral_Triangle_Type is array (1 .. 1) of NATURAL;
  type Square_Type is array (1 .. 1) of NATURAL;
  type Ellipse_Type is array (1 .. 2) of NATURAL;
  type Triangle_Type is array (1 .. 3) of NATURAL;
  type Isosceles_Triangle_Type is array (1 ..2) of NATURAL;
  type Right_Triangle_Type is array (1 .. 2) of NATURAL;
  type Rectangle_Type is array (1 .. 2) of NATURAL;
  type Parallelogram_Type is array (1 .. 3) of NATURAL;
  type Quadrilateral_Type is array (1 .. 6) of NATURAL;
  Circle                : Circle_Type := (1);
  Equilateral_Triangle : Equilateral_Triangle_Type := (1);
  Square                : Square_Type := (1);
  Ellipse               : Ellipse_Type := (2, 1);
  Triangle              : Triangle_Type := (3, 4, 5);
  Isosceles_Triangle    : Isosceles_Triangle_Type := (1, 1);
  Right_Triangle        : Right_Triangle_Type := (3, 4);
  Parallelogram         : Parallelogram_Type := (1, 1, 1);
  Quadrilateral         : Quadrilateral_Type := (others => 1);
end Geometric_Objects_1;

-- no package body required
```

Another example of playing card declarations that might be often used is as follows.

```
package Playing_Cards
is
  type Suit_Type is (Clubs, Diamonds, Hearts, Spades);
  Jack   : CONSTANT := 11;
  Queen  : CONSTANT := 12;
  King   : CONSTANT := 13;
  Ace    : CONSTANT := 14;
  type Card_Type_Set is array (2 .. Ace, Suit_Type) of BOOLEAN;
end Playing_Cards;

-- no package body required
```

In this case each card is represented first by its level, from 2 to 14, and second by its suit, from `Clubs` to `Spades`. That information is now available in the package `Playing_Cards` rather than in all its details.

One more example is in dates, as introduced in Chapter 9, and organized into a package as follows.

```
package Calendar_Dates
is
  type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN,
                      JUL, AUG, SEP, OCT, NOV, DEC);
  type Day_Number is range 1 .. 31;
  type Year_Number is range 1900 .. 2100;
  record
    Month : Month_Name;
    Day   : Day_Number;
    Year  : Year_Number;
  end record;
end Calendar_Dates;

-- no package body required
```

10.1.3 Using Subprogram Group Packages

Subprogram group packages provide a name and library storage for a set of independent subprograms that appear useful to store and make use of together. They can be retrieved using with clauses. In illustration, a set of subprograms permit calculations of geometric properties such as area, volume, etc. with no storage of results from call to call. The more subprograms contained in a package, the easier the individual declarations are to retrieve, but the larger are the units of retrieving. The same declarations may appear in different packages, and it may make sense to have several subprogram group packages in an area to give a choice of how large a package is needed for a specific program. On the other hand, better implementations employ a "smart" linker, causing only those entities needed from a package to be linked. Thus, you may not need to provide these alternative possibilities.

For example, a package specification and package body to deal with various geometric objects is as follows.

```
package Geometric_Properties
is
  type Fraction
  is
    record
      Numerator : INTEGER;
      Denominator : INTEGER;
    end record;
  function Square_Area (Side : in Fraction) return Fraction;
  function Rect_Area (Side_1, Side_2 : in Fraction) return Fraction;
  function Square_Perimeter (Side : in Fraction) return Fraction;
  function Rect_Perimeter (Side_1, Side_2 : in Fraction)
    return Fraction;
end Geometric_Properties;

package body Geometric_Properties
is

  function Square_Area (Side : in Fraction) return Fraction
  is
  begin
    return (Side.Numerator * Side.Numerator,
            Side.Denominator * Side.Denominator);
  end Square_Area;

  function Rect_Area (Side_1, Side_2 : in Fraction) return Fraction
  is
  begin
    return (Side1.Numerator * Side2.Numerator,
            Side1.Denominator * Side2.Denominator);
  end Rect_Area;

  function Square_Perimeter (Side : in Fraction) return Fraction
  is
  begin
    return (4 * Side.Numerator, Side.Denominator);
  end Square_Perimeter;

  function Rect_Perimeter (Side_1, Side_2 : in Fraction) return
    Fraction
  is
  begin
    return (2 * Side1.Numerator + Side2.Numerator,
            Side1.Denominator + Side2.Denominator);
  end Rect_Perimeter;

end Geometric_Properties;
```

The four functions of the package are independent of one another, but related in what they provide. It may be convenient to give them a single name for storage and recall. While correct, the results may not be in simplest terms, particularly for function `Square_Perimeter` and function `Rect_Perimeter`. So an additional reduction will be called for and shown later.

As a second example, consider a package specification dealing with dates.

```
package Date_Calculation
is
  type Month_Name is (JAN, FEB, MAR, APR, MAY, JUN,
                      JUL, AUG, SEP, OCT, NOV, DEC);
  type Day_Number is range 1 .. 31;
  type Year_Number is range 1900 .. 2100;
  type Day_Of_Week is (MON, TUE, WED, THU, FRI, SAT, SUN);
  type Date_Form is
  record
    Month : Month_Name;
    Day   : Day_Number;
    Year  : Year_Number;
  end record;
  function Which_Day_Of_Week (Date : Date_Form) return Day_Of_Week;
end Date_Calculation;
```

How might the package body be completed? Should the package specification be modified? How will actual Date_Form data be discovered and stored? The package body can be started as follows, but how completed?

```
package body Date_Calculation
is
  function Which_Day_Of_Week (Date : Date_Form) return Day_Of_Week;
  is
  begin
    ...
  end;
end Date_Calculation;
```

10.1.4 Using Abstract Data Type Packages

Abstract data type packages provide a name and library storage for a set of package specification data declarations and subprograms that appear useful to store and make use of together for abstract data types. They can be retrieved either in with clauses or in package declarations. For example a set of data declarations and subprograms in dealing with TEXT input and output, such as TEXT_IO or more special forms provide the capability of an abstract data type. The more declarations and subroutines assigned to a package name, the easier the individual declarations are to retrieve, but the larger are the units of retrieving. The same declarations and subprograms may appear in different packages, and it may make sense to have several packages in an area to give a choice of how large a package is needed for a specific program.

For example, there are several IO packages more special than TEXT_IO, such as INTEGER_IO, FLOAT_IO, ENUMERATION_IO. Consider TEXT_IO which tracks and operates on data in external TEXT files. The general form of TEXT_IO begins as follows. Its complete form is in Section 14.3.10 of the *Reference Manual for the Ada Programming Language.*

```
with IO_EXCEPTIONS;
package TEXT_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);

  type COUNT is range 0 .. implementation_defined;
  subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
  UNBOUNDED : constant COUNT := 0; -- line and page length

  subtype FIELD       is INTEGER range 0 .. implementation_defined;
  subtype NUMBER_BASE is INTEGER range 2 .. 16;

  type TYPE_SET is (LOWER_CASE, UPPER_CASE);

  -- File Management

  procedure CREATE  (FILE  : in out FILE_TYPE;
                     MODE  : in FILE_MODE := OUT_FILE;
                     NAME  : in STRING    := "";
                     FORM  : in STRING    := "");

  procedure OPEN    (FILE  : in out FILE_TYPE;
                     MODE  : in FILE_MODE;
                     NAME  : in STRING;
                     FORM  : in STRING    := "");

  procedure CLOSE   (FILE : in out FILE_TYPE);
  procedure DELETE  (FILE : in out FILE_TYPE);
  procedure RESET   (FILE : in out FILE_TYPE; MODE : in FILE_MODE););
  procedure RESET   (FILE : in out FILE_TYPE);

  function  MODE    (FILE : in FILE_TYPE) return FILE_MODE;
  function  NAME    (FILE : in FILE_TYPE) return STRING;
  function  FORM    (FILE : in FILE_TYPE) return FILE_MODE;

  function  IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

  -- Control of default input and output files

   ..

  -- Specification of line and page lengths

  ...

  -- Column, Line, and Page Control

  ...

  -- Character Input-Output

  ...
```

```
-- Generic package for Input-Output of Integer Types

generic
  type NUM is range <>;
package INTEGER_IO is

  DEFAULT_WIDTH  :  FILED := NUM'WIDTH;
  DEFAULT_BASE   :  NUMBER_BASE := 10;

  ...

end INTEGER_IO;

...

-- Generic packages for Input-Output of Real Types

...

generic
  type NUM is digits <>;
package FLOAT_IO is

  DEFAULT_FORE   : FIELD := 2;
  DEFAULT_AFT    : FIELD := NUM'DIGITS-1;
  DEFAULT_EXP    : FIELD := 3;

  ...

end FLOAT_IO;

generic
  type NUM is delta <>;
package FIXED_IO is

  DEFAULT_FORE   : FIELD := NUM'FORE;
  DEFAULT_AFT    : FIELD := NUM'AFT;
  DEFAULT_EXP    : FIELD := 0;

  ...

end FIXED_IO;

-- Generic package for Input-Output of Enumeration Types

generic
  type ENUM is (<>);
package ENUMERATION_IO is

  DEFAULT_WIDTH   :  FIELD := 0;
  DEFAULT_SETTING :  TYPE_SETTING := UPPER_CASE;

  ...

end ENUMERATION_IO;
```

```
 -- Exceptions

 STATUS_ERROR   : exception renames IO_EXCEPTIONS.STATUS_ERROR;

 ...

 private
   -- implementation-dependent
 end TEXT_IO;
```

As noted, the package body of TEXT_IO will be implementation dependent, depending on the hardware.

10.1.5 Using Abstract State Machine Packages

Abstract state machine packages provide a name and library storage for a set of package body data declarations and subprograms that appear useful to store and make use of together as abstract state machines. They can be retrieved by a with clause. For example a set of data declarations and subprograms in dealing with the behavior of an airplane or a large company provide the capability of a abstract state machine. The more declarations and subroutines contained within a package name, the easier the individual declarations are to retrieve, but the larger are the units of retrieving.

Abstract state machine packages differ from abstract data type packages in that no data type is exported, only the operations on a type are exported. For example, in the following package specification, the data type STACKS is exported, as well as the operations for manipulating an object of type STACKS.

```
package STACK_PACKAGE
is

  type STACKS is
  record
    Values : array (1 .. 100) of INTEGER;
    Top_Of_Stack : NATURAL;
  end record;

  procedure PUSH (The_Item : in INTEGER; The_Stack : in out STACKS);
    -- places an item onto the stack
  procedure POP (The_Item : out INTEGER; The_Stack : in out STACKS);
    -- removes an item from the stack
  function DEPTH (The_Stack : STACKS) return NATURAL;
    -- returns the number of items on the stack
  function TOP (The_Stack : STACKS) return INTEGER;
    -- returns the top item on the stack
  procedure CLEAR (The_Stack : in out STACKS);
    -- clears the stack (removes and discards all items)

end STACK_PACKAGE;
```

This is an abstract data type package. Note that we are exporting the data type, thereby allowing the user to declare multiple objects of this type. This, in turn, necessitates that each call to an operation on this type provides the stack object upon which we desire to operate. Thus, a call to PUSH must provide not only the value to be placed on the stack, but also the particular stack object onto which we will push it.

An abstract state machine package, on the other hand, merely provides a set of operation on an object. The state of the objet is maintained exclusively by the package, hidden from the user except via the operations that are provided. Thus, abstract state machine packages are a powerful means of implementing information hiding as well as abstraction using Ada. As an example, consider the following package which exports the operations on a stack.

```
package STACK_PACKAGE
is

  procedure PUSH (The_Item : in INTEGER);
    -- places an item onto the stack
  procedure POP (The_Item : out INTEGER);
    -- removes an item from the stack
  function DEPTH return NATURAL;
    -- returns the number of items on the stack
  function TOP return INTEGER;
    -- returns the top item on the stack
  procedure CLEAR;
    -- clears the stack (removes and discards all items)

end STACK_PACKAGE;
```

This is an abstract state machine package. Note that we do not export the data type, merely the operations that will be performed on the single object that this package will provide. The declaration of this single stack object will be in the body of the abstract state machine package. Note that none of the operations require that we specify which stack, because in this case there is only one stack object, and it is hidden from our inspection except through the subprograms that we export.

10.1.6 Exercises

1. What are the parts of packages that can be stored separately?

2. For any subprogram contained in a package, how are its parts divided between package specification and package body?

3. What is the distinction between data declarations in package specification and data declarations in package body?

4. What are the four Booch package categories?

5. Define a package declaration group of use to you.

6. Define a package subprogram group of use to you.

7. Define a package abstract data type of use to you.

8. Define a package abstract state machine of use to you.

10.2 Package Behavior Specifications

Package behavior specifications are another step beyond subprogram behavior specifications, either in collecting data declaration behavior specifications or subprogram behavior specifications, or in defining behavior specifications that combine data declarations and subprograms into abstract data types or abstract state machine behavior. In the first two cases the ideas of behavior specifications can be simply expanded beyond that understood in subprograms. But in the latter two cases, the idea of implemented abstract data types or abstract state machines with data stored between calls on the subprograms of the package brings a new dimension to the behavior specification process.

10.2.1 The Basis for Behavioral Specification of Packages

As noted, any package is made up of data declarations in the package specification or the package body, and subprograms declared jointly between the package specification and package body. Any single subprogram is a *rule* for *behavior*, describing the effect of either a procedure as a statement or an Ada function as an expression. The collection of these package subprograms makes up the package behavior, as called through procedures and functions. The sequence in which these procedures and functions are called in any execution is defined by the host program, itself a procedure or a function.

In well structured Ada subprograms defined in package bodies, the rules are direct in form. Designers build program rules out of just two behavior building operations, first, *behavior composition* which corresponds to sequential execution of program parts, and second, *disjoint behavior union* which corresponds to alternative execution of one program part or another, say in **if** or **case** statements. Program *iteration*, say in **while loop** statements, uses no more than these two operations together, and *recursion* provides a useful view of an iteration process.

As noted, any package part, subprogram part or total program defines a unique, possibly complex behavior. The behavior is seldom a numerical function in classical terms. Even so called numerical programs must deal with finite sets of numbers in which overflow departs from classical number systems. Any INTEGER variable is subject to overflow if not treated with care.

Given the text or name of a subprogram or program part in a package, for example a *procedure* called `Alpha`

```
Alpha =
  procedure Beta
  is
  ...
  begin
  ...
  end Beta;
```

it has already been established that the *procedure behavior* is denoted by brackets `[, ]` around the name or text and defines a mathematical function or relation, as

```
[Alpha] =
  procedure Beta
  is
  ...
  begin
  ...
  end Beta;]
```

Since Alpha is part of a package, its specification part will be part of the package specification, its body part of the package body, as already seen. The procedure behavior will be determined by the behavior of the joint specification and body.

In this case, as before for independent procedures, [Alpha] is a set of ordered pairs

```
[Alpha] = {<X, Y>| Given initial data X,
                   Alpha will produce final data Y}
```

where X and Y are defined by sets of declared objects, possibly different sets. The behavior [Alpha] is determined by Ada text, but is independent of the language Ada. The same behavior can be defined in Pascal text, C text, etc.

The other choice for a package member is an Ada function, which defines an Ada expression rather than a statement. In this case the expression is evaluated rather than executed.

10.2.2 Behavioral Specification of Package Parts

As already noted, behavioral package specification is a step beyond subprogram specification, dealing with a collection of declarations and subprograms. A package specification defines the combined behavior of all declarations and subprograms as called on by a using subprogram. The behavior of a subprogram is defined by a single execution of the subprogram. But the behavior of a package is defined by any sequence of calls of the declarations and subprograms of the package. For the declaration groups and subprogram groups, the specification of packages are the union of the specifications of their declarations or subprograms. In this case package specification is an accumulation of specifications of their parts. But in abstract data types and state machines, the specification connects subprogram behaviors with data stored between calls. Specification must recognize any possible sequence of calls on the package parts, and determine the results of each call in the sequence.

It is already known that subprograms have behavior as mathematical functions or relations. In going to Ada packages, only a single step needs be made to subprograms to program parts. With packages, any number of calls can be made, with data stored between calls, so packages have behavior of mathematical state machines, being initialized at package declaration, accessed with each call of a subprogram, and terminated when the package declaration is closed out. Every access to a subprogram behaves exactly as the ordinary subprogram except that package data may be available from previous package calculations.

We review subprogram part behavior that make up package behavior next. Starting with simple assignment statements, such as

```
First := Second;
```

the *package part behavior*

```
[First := Second;]
```

takes its initial data state to its final data state. If legal, it will change the value of First in the final state to the value of Second in the initial state and change no other values of variables in the initial state. If illegal, the final state may be quite different than the initial state, possibly with both First and Second disappearing, as well as other variables, in terminating the entire program execution. So assignment statements have simple behavior parts when legal, but possibly more complex behavior parts when illegal. In summary, the behavior `[First := Second]` is a set of ordered pairs with second members determined by the first members

```
[First := Second;] = {<<First, Second, ...>, <Second, Second, ...>> |
                        First := Second; is legal}
       ∪ {<<First, Second, ...>, <abnormal_state>> |
              First := Second; is illegal}
```

where *abnormal_state* will be determined by other aspects of the initial state. Illegal situations will be suppressed in what follows for sake of time. In more direct behavior notation, dealing only with the legal situation,

```
[First := Second;](<First, Second, ...>) = <Second, Second, ...>
```

in which the *behavior* `[First := Second;]` with behavior *argument* `<First, Second, ...>` produces the behavior *value* `<Second, Second, ...>`.

Next, for example, with a sequence of statements, such as

```
First := Second; Second := Third; Third := First;
```

the part behavior

```
[First := Second; Second := Third; Third := First;]
```

will alter values of `First`, `Second`, `Third` as a composition of the three individual assignment behaviors

```
[First := Second;] ⊕ [Second := Third;] ⊕ [Third := First;].
```

That is, beginning with an initial state as argument, the first assignment behavior gives a new state as value

```
[First := Second;](<First, Second, Third, ...>)
        = <Second, Second, Third, ...>,
```

the second assignment behavior uses this value as an argument

```
[Second := Third;](<Second, Second, Third, ...>)
         = <Second, Third, Third, ...>,
```

and the third assignment behavior uses this last value as argument

```
[Third := First;](<Second, Third, Third, ...>)
        = <Second, Third, Second, ...>.
```

That is, the composition behavior is a nested set of simpler behaviors that evaluate as

```
([First := Second;]*[Second := Third;]*[Third := First;])
        (<First, Second, Third, ...>)
  =  [Third := First;]([Second := Third;]([First := Second;]
        (<First, Second, Third, .>)))
  =  [Third := First;]([Second := Third;]
        (<Second, Second, Third, ...>))
  =  [Third := First;](<Second, Third, Third, ...>)
  =  <Second, Third, Second, ...>
```

as worked out just above. In summary, this composition behavior will interchange the values of `Second` and `Third` and leave `First` with the initial value of `Second`, not changing any other data in the initial state.

Finally, for an alternation statement, such as

```
if First > Second then Second := Third; else First := Third; end if;
```

the part behavior will execute either the then part or else part, so that

```
[if First > Second then Second := Third; else First := Third;
    end if;]
  = (First > Second -> [Second := Third;] |
      First <= Second -> [First := Third;])
  = [Second := Third; | First > Second] ∪ [First := Third; |
      First <= Second]
```

where the expression `[Second := Third; | First > Second]` means the behavior `[Second := Third;]` with its domain restricted to the condition `First > Second`. That is, the part behavior is a union of disjoint behaviors.

As already noted, the step from subprogram behavior to package behavior is the step from a single call, each independent of one another, to a sequence of calls while a package is active, possibly with data stored between calls. For the using subprogram, each call of a package subprogram is just like a call of such a subprogram except possibly for data stored in the package.

10.2.3 Declaration Group Package Behavior Specifications

Declaration group package behavior specifications are simple extensions of declarations in subprograms. They include declarations of types, subtypes, and objects that will be useful to have collected as a package. They contain no executable part, only data declarations. Therefore, package specifications are sufficient to define declaration group package specifications, so no package bodies will be formed.

As already noted in subprogram semantics, declarations expand the data space by adding types, subtypes, and objects to the data space already defined. They also can give objects specific values at declaration. At the close of a package for execution, these declarations are reversed so the data definitions no longer exist.

The value of declaration group packages is to summarize and collect groups of declarations that are useful together, based on the knowledge of the software engineer and the subjects involved. For example, in geometric problems, ellipses, triangles, quadrilaterals will likely all be used. But these definitions are not so likely to be used together. There are many groups of declarations from engineering, science, finance that are useful to know about. Earlier in this chapter, several examples have been given, and no technical problems exist in expanding them.

10.2.4 Subprogram Group Package Behavior Specifications

Subprogram group package behavior specifications represent another kind of package containing subprograms of likely common use. The subprograms are independent and could well be given as individual subprograms, not as part of packages. But the subprograms will often be used as part of a group, so putting them into a subprogram group package may be very useful and convenient. It can reduce the amount of work to identify and define the subprograms, but even more it can help many people to use identical declarations and keep work consistent.

The question of how many subprograms should appear in a subprogram group package doesn't have a single answer. There may be several groups of different sizes or different focuses that involve the same set of subprograms.

In illustration, consider a package called `Fraction_Package` that deals with various operations on fractions. This package will deal with the need to reduce fractions to their lowest terms, as noted in the simple example above on the package called `Geometric_Properties`.

```
package Fraction_Package
is
  type Fraction_Type
  is
  record
    Numerator : NATURAL;
     Denominator : POSITIVE;
  end record;

  function "+" (Left, Right : Fraction_Type) return Fraction_Type;
  function "-" (Left, Right : Fraction_Type) return Fraction_Type;
  function "*" (Left, Right : Fraction_Type) return Fraction_Type;
  function "/" (Left, Right : Fraction_Type) return Fraction_Type;
end Fraction_Package;


package body Fraction_Package
is
  function GCD (Numerator : NATURAL; Denominator : POSITIVE)
    return POSITIVE
  is
    Reduced_Numerator : NATURAL := Numerator;
    Reduced Denominator : POSITIVE := Denominator;
    Temp : POSITIVE;
  begin
    -- GCD(Reduced_Numerator, Reduced_Denominator) =
    -- GCD (Numerator, Denominator)
    Common_Reduction:
    while Reduced_Numerator > 0
    loop
      Temp := Reduced Denominator;
      Reduced Denominator := Reduced_Numerator;
      Reduced_Numerator := Temp mod Reduced_Numerator;
        -- GCD(Reduced_Numerator, Reduced_Denominator) =
        -- GCD (Numerator, Denominator)
    end loop Common_Reduction;
    return Reduced Denominator;
  end GCD;
  function Reduce (Fraction : in Fraction_Type) return Fraction_Type
  is
    GCD_Now : POSITIVE := GCD (Fraction.Numerator,
                               Fraction.Denominator);
  begin
    return (Fraction.Numerator / GCD_Now,
            Fraction.Denominator / GCD_Now);
  end Reduce;

  function "+" (Left, Right : Fraction_Type) return Fraction_Type
  is
    Result : Fraction_Type;
  begin
    Result := (Left.Numerator * Right.Denominator +
               Right.Numerator * Left.Denominator,
               Left.Denominator * Right.Denominator);
    return Reduce (Result);
  end "+" ;

  function "-" (Left, Right : Fraction_Type) return Fraction_Type
  is
    Result : Fraction_Type;
    -- will raise CONSTRAINT_ERROR if Left < Right
  begin
    Result := (Left.Numerator * Right.Denominator -
               Right.Numerator * Left.Denominator,
               Left.Denominator * Right.Denominator);
    return Reduce (Result);
  end "-" ;

  function "*" (Left, Right : Fraction_Type) return Fraction_Type
  is
    Result : Fraction_Type;
  begin
    Result := (Left.Numerator * Right.Numerator,
               Left.Denominator * Right.Denominator);
    return Reduce (Result);
  end "*" ;

  function "/" (Left, Right : Fraction_Type) return Fraction_Type
  is
    Result : Fraction_Type;
    -- will raise CONSTRAINT_ERROR if Right.Numerator = 0
  begin
    Result := (Left.Numerator * Right.Denominator,
               Left.Denominator * Right.Numerator);
    return Reduce (Result);
  end "/" ;

end Fraction_Package;
```

10.2.5 Abstract Data Type Package Behavior Specifications

Abstract data type package groups carry the subprogram group one further step in permitting and exporting data declarations in the package specification, while permitting but *not* exporting them from the package body. As a result, the subprograms of the package are related by common data declared in the package specification.

Abstract data type packages arise in classical computation structures such as stacks and strings. For example, a stack specification may take the form of asking for six operations on stacks up to finite sizes, namely

`Clear:` Make the stack empty.

`Push:` If room, add an item to the top of the stack; if not room, destroy the stack.

`Pop:` If stack not empty remove top item, otherwise leave stack empty.

`Is_Empty:` If stack is empty return TRUE, if not empty return FALSE.

`Top:` If stack is not empty, return value of top element, if empty, return zero.

`Depth:` Return number of elements in the stack, including zero if the stack is empty.

There are other ways to handle stack overflow and underflow, and a direct and simple way is defined here in illustration. It will be up to the user to store the stack, including its number of elements between these operations. That is why the behavior is that of an abstract data type. In response to this specification the following package adds more formality in the design.

```
package Stack_Package
is
  type Stack is
  record
    Values : array (1..100) of INTEGER;
    Top_Of_Stack : NATURAL;
  end record;
  procedure Clear    (The_Stack : in out Stack);
  procedure Push     (The_Item  : in INTEGER;
                      The_Stack : in out Stack);
  procedure Pop      (The_Item  : out INTEGER;
                      The_Stack : in out Stack);
  function  Is_Empty (The_Stack : Stack) return BOOLEAN;
  function  Top      (The_Stack : Stack) return INTEGER;
  function  Depth    (The_Stack : Stack) return NATURAL;
  Stack_Overflow, Stack_Underflow : exception;
end Stack_Package;

package body Stack_Package
is
  procedure Clear    (The_Stack : in out Stack)
  is
  begin
    The_Stack.Top_Of_Stack := 0;
  end Clear;

  procedure Push     (The_Item  : in INTEGER;
                      The_Stack : in out Stack)
  is
  begin
    The_Stack.Top_Of_Stack := The_Stack.Top_Of_Stack + 1;
    The_Stack.Values (The_Stack.Top_Of_Stack) := The_Item;
  exception
    when CONSTRAINT_ERROR => raise Stack_Overflow;
  end Push;
```

```
    procedure Pop      (The_Item : out INTEGER;
                        The_Stack : in out Stack);
    is
    begin
      The_Item := The_Stack.Values (The_Stack.Top_Of_Stack);
      The_Stack.Top_Of_Stack := The_Stack.Top_Of_Stack - 1;
    exception
      when CONSTRAINT_ERROR => raise Stack_Underflow;
    end Pop;

    function  Is_Empty (The_Stack : Stack) return BOOLEAN
    is
    begin
      return The_Stack.Top_Of_Stack = 0;
    end Is_Empty;

    function  Top      (The_Stack : Stack) return INTEGER
    is
    begin
      return The_Stack.Values (The_Stack.Top_Of_Stack);
    end Top;

    function  Depth    (The_Stack : Stack) return INTEGER
    is
    begin
      return The_Stack.Top_Of_Stack;
    end Depth;

  end Stack_Package;
```

As already noted, it is up to the user to store and provide `The_Stack` and `Pointer`, which is not stored in the package. One advantage is that the package can be used concurrently for many stacks, the user keeping the data separate and using it as required.

10.2.6 Abstract State Machine Package Behavior Specifications

Abstract state machine package groups carry the abstract data type package a final step in permitting data declarations in the package body as well as package specifications. As a result, subprograms of the package are now related by common data stored between uses of the subprograms.

We have already seen several examples of abstract state machines, from the very simple `Fibonacci_Package` up to the list processing examples of Chapter 9. Their specifications allow data storage between executions. We will look at the specifications of list processing later in this chapter again in terms of box structures. It will become critical to specify abstract state machines as so called black boxes which map histories of stimuli to the next response without reference to stored data. That is taken up in the next section.

10.2.7 Exercises

1. Given an Ada procedure, separate it into its procedure specification and its procedure body. Identify where these two parts go in using the procedure in an Ada package.

2. Given an Ada function, separate it into its function specification and its function body. Identify where these two parts go in using the function in an Ada package.

3. Discuss the difference between the specification of an Ada package and the specifications of all the data declarations and subprograms in the package.

4. When is initialization required to be specified for an Ada package?

5. Provide a declaration group package specification of use.

6. Provide a subprogram group package specification of use.

7. Provide an abstract data type package specification of use.

8. Provide an abstract state machine package specification of use.

10.3 Box Structured Design with Packages

10.3.1 Background in Cleanroom Software Engineering

Cleanroom Software Engineering provides an engineering discipline to *develop* and *certify* software *correct* under *statistical quality control* to well formed *specifications* of user requirements given in *construction increments.*

The well formed specifications provide a set of *creatable increments* that *accumulate* into the complete system required. The increments must be of a size possible to develop *without testing* –say two thousand to twenty thousand lines of code–and be accessed entirely by user commands and data. The partial accumulations of increments are then tested and certified for correctness to the specifications. If more than five failures per thousand lines of code are discovered in an increment, that increment should come off and likely be redeveloped.

A *sound development methodology* is defined to create well testable software by design and verification, in particular with *no unit testing* by the developers. That is, in Cleanroom operations, developers do no testing at all in the development of the software. As already noted, unit testing and fixing is the most error prone activity in software development today, leading to deeper failures in fifteen per cent or more of the fixes. The design must be structured top down with behavioral verification to scale up to entire increments and accumulations developed. Failures due to lapses in behavioral verification are five times easier to find and fix than failures due to unit fixes, and are very unlikely to lead to deeper failures.

A *sound certification methodology* is defined for releasing the software with no known failures, especially no important failures. It requires a test design based not only on the behavioral and performance specifications, but also on how critical each test case is to system behavior. Such a test design is based on a *stratified statistical strategy* derived from the statistics of usage expected for the software. For an important case, the stratus may consist of a single case (with probability 1), or a small subset of cases, on out to strata containing large sections of the software until the entire system behavior is covered. A test design defines each stratus (possibly in the hundreds or thousands) and the number of tests in each strata. The power of the test design is derived from the analysis of failures possible in each stratus after testing.

Testing without any failures found leads to *certification of correctness* of the software to the level tested. If failures are later found the certification is negated. If these failures are fixed the certification can be started again. Certification continues with software release to users, with no failures moving with confidence from typical 3 sigma at release (.001 failures per usage) to and beyond 6 sigma (.000000002 failures per usage) with sufficient usage. With any failures after release, the process is identical in fixing and starting certification again.

10.3.2 Cleanroom Engineering Activities

Cleanroom Engineering achieves *statistical quality control* over software development in an industrial environment by strictly *separating* the *design* process from the *testing* process in a *pipeline* of *incremental software development.* There are three major engineering activities in the process:

> First, a *specification activity* creates an *incremental specification* for development and certification that defines a *pipeline of software increments* to accumulate into the final software product, which specification includes the *statistics of its use* as well as its *behavior* and *performance* requirements;
>
> Second, a *development activity* designs and codes increments specified using *box structured design* and *behavioral verification* of each increment, for delivery to certification with *no debugging* beforehand, and provides subsequent correction for any failures that may be uncovered later during certification or usage;
>
> Third, a *certification activity* uses *statistical testing and analysis* for the *certification* of the *software correctness* to the *usage specification,* notification to the development team of any failures discovered during certification or usage, and subsequent *recertification* as failures are corrected.

As noted, there is an *explicit feedback process* between certification and development on any failures found in statistical usage testing. This feedback process provides an *objective measure* of the *correctness* of the software as it matures in the development pipeline. It does, indeed, provide a *statistical quality control process* for software development that has not been available in this first human generation of trial and error programming.

10.3.2.1 Dealing with Human Fallibility

Humans are fallible, even in using sound mathematical processes, so software failures are possible during the certification process. (Indeed, people are fallible in doing long division, even though the process is perfect.) But there is a surprising *power* and *synergism* between *behavioral verification* and *statistical usage testing.* It turns out that the mathematical failures left are much easier to find and fix during testing than failures left behind in debugging, measured at a factor of five in practice. Mathematical failures usually turn out to be simple blunders in the software, whereas failures left behind or introduced in debugging are usually deeper in logic or wider in system scope than those fixed.

In Cleanroom Engineering a major discovery is the ability of well educated and motivated people to create nearly defect free software before any execution or debugging, well less than five defects per thousand lines of code. Such code is ready for usage testing and certification with no unit debugging by the designers. The result of statistical testing is to remove practically all these defects with no side effects.

In this first human generation of software development it has been counter intuitive to expect software with so few defects at the outset. Typical heuristic programming creates fifty defects per thousand lines of code, then reduces that number to five or less by debugging. And it seems impractical to reduce the number of defects to or near zero no matter how much effort goes into it. In the final state of the software, defects are put into the software as fast as they are removed.

10.3.2.2 Software Development Without Testing

In spite of the experiences and assumptions of this first human generation of software development, programs are strict rules for mathematical behaviors. There is nothing experimental about program behavior except their inventions by people. As mathematical objects, programs admit mathematical inspection and verification of whether they meet mathematical specifications. Of course mathematics does not mean numerical, and most programs are not strictly numerical. A simple sort program performs a mathematical behavior in mapping a set of objects into a sorted sequence of those very objects. In this first human generation, programs are repeatedly drafted, tested, fixed, retested, refixed, ... as experimental activity. In this process, intellectual control of programs is lost, ending with objects people hope are right, but which are frequently not quite right.

At first glance, software development without testing seems impossible–like not looking at the keys while typing to early typists. But since programs are strict rules for mathematical behaviors, their correctness can be determined by mathematical inspection and verification against specifications. On second thought this may still look very difficult because of all the details involved. Isn't unit testing and debugging still easier and better? Unit debugging and testing adds deeper failures in fifteen per cent or more of the fixes, failures that are often not found until actual use of the software. Neither the inherent failures from development testing nor the possibility of doing without it has been fully understood in this first generation.

Just as place notation and long division made correct operations in arithmetic more practical, methods now exist in software engineering to make software development without testing a practical reality. The mathematics is relatively simple, like long division, rather than nuclear physics. In large programs there is a lot of such simple mathematics to do. But good program organization into hierarchical structures, both in control and data, make this mathematics possible and practical by well disciplined software engineers. Box structures of software design and behavioral verification of programs provide human capability to inspect and verify software in development rather than testing it and adding deeper failures to it.

10.3.3 Cleanroom Experiences

The IBM COBOL Structuring Facility (COBOL/SF), a complex product of some 80K lines of PL/I source code, was developed in the Cleanroom discipline, with box structured design and behavioral verification but no debugging before usage testing and certification of its correctness. A version of the US AF HH60 (helicopter) flight control program of over 30 KLOC was also developed using Cleanroom. The Coarse/Fine Attitude Determination Subsystems (CFADS) of the UARS Attitude Ground Support System (AGSS) of some 30 KLOC has been developed with Cleanroom at NASA.

The IBM COBOL/SF converts an unstructured COBOL program into a structured one of identical behavior. It uses considerable artificial intelligence to transform a flat structured program into one with a deeper hierarchy that is much easier to understand and modify. The product line was prototyped with Cleanroom discipline at the outset, then individual products were generated in Cleanroom extensions. In this development, several challenging schedules were defined for competitive reasons, but every schedule was met.

The COBOL/SF products have high function per line of code. The prototype was estimated at 100 KLOC by an experienced language processing group, but the Cleanroom developed prototype was 20 KLOC. The software was designed not only in structured programming, but also in structured data access. No arrays or pointers were used in the design; instead, sets, queues, and stacks were used as primitive data structures. Such data structured programs are more reliably verified and inspected, and also more readily optimized with respect to size or performance, as required.

COBOL/SF, Version 2, consisted of 80 KLOC, 28 KLOC reused from previous products, 52 KLOC new or changed, designed and tested in a pipeline of five increments, the largest over 19 KLOC. A total of 179 corrections were required during certification, under 3.5 corrections per KLOC for new code with no developer execution, under 2 corrections per KLOC for all code before testing. The productivity of the development was 740 LOC per staff month, including all specification, development, certification, and management, in meeting a very short deadline.

The HH60 flight control program was developed on schedule. Programmers' morale went from quite low at the outset ("why us?") to very high on discovering their unexpected capability in accurate software design without debugging. The twelve programmers involved had all passed the pass/fail coursework in mathematical (behavioral) verification of the IBM Software Engineering Institute, but were provided a week's review as a team for the project. The testers had much more to learn about certification by objective statistics.

The subsystem Coarse/Fine Attitude Determination System (CFADS) of the NASA Attitude Ground Support System (AGSS) of some 30 KLOC was developed in FORTRAN. 62% of the subroutines, which averaged 258 source lines each, compiled correctly the first time the testers tried to compile it, and all but one of the rest compiled correctly on the second attempt. Compared with well measured related systems, the failure rate was down by a factor of 5 while the productivity was up by 70%.

V. R. Basili and F. T. Baker introduced Cleanroom ideas in an undergraduate software engineering course at the University of Maryland, assisted by R. W. Selby. As a result, a controlled experiment in a small software project was carried out over two academic years, using fifteen teams with both traditional and Cleanroom methods. The result, even on first exposure to Cleanroom, was positive in the production of reliable software, compared with traditional results.

Cleanroom projects have been carried out at the University of Tennessee, under the leadership of J. H. Poore and at the University of Florida under H. D. Mills. At Florida, seven teams of undergraduates produced uniformly successful systems for a common structured specification of three increments. It is a surprise for undergraduates to learn about software development as a serious engineering activity using mathematical verification instead of debugging, since software development is typically introduced primarily in universities today as a trial and error activity with no real technical standards.

10.3.4 Software Specification by Increments

A user's specification for a substantial software system will identify various classes of user commands and data for various parts of the system. For example, bringing up an interactive system at the beginning of the day will require and accept certain kinds of user commands and data that the ordinary interactive users may not even be aware of. But bringing the system up is an integral part of the system for a certain class of users concerned with the overall system operations. During the day, several distinct classes of users may be interacting simultaneously and independently, such as users adding data to the system, or users making inquiries, or users monitoring the system use and performance. Within each such class, several or many users may be interacting simultaneously and independently, as well.

Now, to specify a sequence of increments that accumulate into the system desired takes some more thinking. The first increment must clearly bring the system up to some extent, even though the increment does not respond to all the needs of layer users. The next increment might accept data for storage, yet not make it available for access. But now the first two increments together can carry out some actions needed. The third increment might make data available for access, so the accumulation of the first three increments allows data in and out of the system. Still another increment may make data inquiries possible. In all these increments, only part of the actions of a specific kind may be possible, so later increments may help fill out a given type of action. The specifications of these increments takes much thought, and they will not usually be simple parts of the overall behavior, more likely interacting pieces of several basic parts.

However, as simultaneously and concurrently as various users seem to interact with the system, the individual computers in the system each operate strictly sequentially in real time, shifting from one user to another so rapidly that each user gets almost immediate response, even though ten, or a thousand, other users may have been serviced between the system's last response and the user's stimulus. As a rule, users are separated from one another by operating in different, relatively protected, data spaces that represent the tasks they are doing. But users can interact, intentionally or not, as their tasks become more intertwined.

For example, in an airline reservation system, a ticket agent may inquire about availability of seats on a given flight and get the response that seats are available. Then when the seats are requested a moment later, the response is that no seats are available. Other users have interacted in picking up the seats in the previous moment. Such system behavior is designed. It would be conceivable to design an airline reservation system such that seats could be held from inquiry to request, but it would require entirely different levels of data storage and processing. In this way, it is clear that user independence is relative, with economic and technical issues involved with multiple users in systems.

This understanding that significant software systems have different kinds of uses applies whether there are single or multiple users. A single user may be using a system in different ways at different times, even within a single session. The design of the software will typically reflect such different uses by packaging similar operations in common modules. For example, various kinds of data searching may be handled in a search module, but data retrievals handled in a different retrieval module. It also makes similar sense to identify similar stimuli response operations in specifications, entirely from the user point of view and state of mind. In particular, complex specifications need to be designed as carefully as programs to reflect the natural structure of the problem being solved and to find effective specification structures that reflect user activities and understandings.

10.3.4.1 Software Usage as a Markov Process

As noted, software specifications deal with functional behavior and performance. Functional behavior is ordinarily decomposed into various subfunctions in ways understandable by users, and often obtained from users as requirements. Performance will usually affect design in fundamental ways. But expected usage of the software will have critical impacts on performance issues. For example, a data base system, with very much more querying than data addition or deletion, may call for a design with high performance queries at the expense of data addition and deletion performance. And such a design can be entirely unsatisfactory with different usage. Thus expected usage statistics can play a key role in software system design.

However there is another critical use for usage statistics as part of software specifications. It is to permit the certification of software. Software behavior depends not only on how correct the software is but also on how it is used. For every possible state of internally stored data, any command and input data is handled either correctly or incorrectly, denoted a failure in the latter case at some level of seriousness.

Now, with a statistical usage specification for each possible internal state, the probability of each selection of commands and input data in such a state will be known. Next, the behavioral specification will define what the new internal state will become, as well as the response to the user. These two facts define a Markov process, namely the set of all internal data states and the probability from getting from each member of the set to the next member. Of course, some members may be terminal when the process terminates.

10.3.5 Box Structures

Box structured design is based on a usage hierarchy of packages. Such packages, also known as *abstract data types* or *abstract state machines*, are described by a set of data declarations and subprograms that may define and access internally defined and stored data. In order to create and control such designs based on usage hierarchies in practical ways, their box structures provide standard, finer grained subdescriptions for any package of three forms, namely as *black boxes*, as *state boxes*, and as *clear boxes*, defined as follows.

> **Black Box:** External view of an Ada package specification and package body, describing its behavior as a mathematical function in Ada from historical sequences of stimuli through its subprograms to its next response in values of its data. Subprograms may be either procedures, which are Ada statements, or functions, which are Ada expressions.
>
> **State Box:** Intermediate view of an Ada package specification and package body, describing its behavior in Ada by use of an internal state of declared data and an internal black box with a mathematical function from historical sequences of stimuli and states through its data and subprograms to its next response and state in values of its data, and an initial internal state. Subprograms may be either procedures, which are Ada statements, or functions, which are Ada expressions. State box subprograms are derived from the black box subprograms, but are not the same as the black box subprograms.
>
> **Clear Box:** Internal view of an Ada package specification and package body, describing the internal black box of its state box in a usage control structure of Ada subprograms from other packages. Such a control structure may define sequential or concurrent use of the other packages, right down to individual variables. The new subprograms may be either procedures, which are Ada statements, or functions, which are Ada expressions. The new subprograms create together the state box subprograms as required.

Box structures enforce completeness and precision in design of software systems as usage hierarchies of Ada packages. Such completeness and precision can lead to pleasant surprises in human capabilities in software engineering and development. The surprises are in capabilities to move from software specifications to design in programs without the need for unit testing and debugging before delivery to certification for usage testing. In this first generation of software development, it has been widely assumed that trial and error programming, unit testing and debugging were necessary. But well educated, well motivated software professionals are, indeed, capable of developing software systems of arbitrary size and complexity without program debugging before usage testing.

10.3.5.1 Black Box Behavior Specifications

As noted, black box behavior is defined without any state data between calls on the black box. It is defined in terms of the history of Stimuli Stimulus_Star (written Stimulus* and meaning the history of Stimuli) to produce the next response Response. This may seem awkward in many cases when the states seem already defined naturally. But the experience shows the contrary with some thinking. What seems natural may be customary from past experience. Looking at the keys while typing seemed natural in the late 19th century, but was just customary from current experience. Dog paddle and breast stroke were also customary in the late 19th century, from ignorance about free style swimming. So data states are eventually necessary in software design, but initial specifications need to be state free for precise engineering design. In illustration of simple black box behavior, consider the following examples.

```
Copy.BB:  Return the last stimulus as the next response.
```

Black box Copy.BB needs no history earlier than the last stimulus, which is returned directly as the next response. In this case

```
Response := Stimulus;
```

each time a stimulus reaches Copy.BB.

```
Hist.BB:  Return the first stimulus as the next response.
```

Black box Hist.BB needs the entire history of stimuli to have access to the first stimulus. In this case

```
Response := Stimulus.1;
```

each time a stimulus reaches Hist.BB, where Stimulus.1 is the first stimulus received by Hist.BB.

```
Add.BB:  Return the first stimulus as the first response, and from
then on return the sum of the last two stimuli as the next response.
```

Black box Add.BB needs the first stimulus to return as the response and after that the last two stimuli whose sum is returned as the response. In this case

```
Response.1 := Stimulus.1;
Response.i := Stimulus.i-1 + Stimulus.i ( for i > 1);
```

each time a stimulus reaches Add.BB, where Stimulus.1 is the first stimulus received by Add.BB and thereafter the sum of the last two stimuli received make up the next response.

```
Sum.BB:  Return the first stimulus as the first response, and from
then on return the sum of the first and last stimuli as the next re-
sponse.
```

Black box Sum.BB needs the first stimulus to return as the response and after that the first and last stimuli whose sum is returned as the response. In this case

```
Response.1 := Stimulus.1;
Response.i := Stimulus.1 + Stimulus.i ( for i > 1);
```

each time a stimulus reaches Sum.BB, where Stimulus.1 is the first stimulus received by Sum.BB and thereafter the sum of the first and last stimuli received make up the next response.

```
    All.BB:  Return the sum of all stimuli as next response.
```

Black box All.BB needs all stimuli and their sum to return as the response. In this case

```
    Response.i := Stimulus.1 + ... + Stimulus.i
```

each time a stimulus reaches All.BB, where ... includes all stimuli between the first Stimulus.1 and the last Stimulus.i.

10.3.5.2 State Box Design

Once the black box behavior is specified, the state box can be designed in many ways. The state box design defines what is to be stored from stimulus to stimulus in the state, how the response is to be calculated from the last stimulus and the state, and how the new state is to be calculated from the last stimulus and old state. For example, to design states for the simple examples above, consider the following.

```
    Copy.SB:  Return the last stimulus as the next response.
```

In this case the simplest solution is to take the state as empty, so the black box and state box are identical.

```
    Hist.SB:  Return the first stimulus as the next response.
```

A direct solution is to give the state a variable with the value of the first stimulus, Stimulus.1, and never change it afterward, and make the response the value of Stimulus.1. But how will later stimuli be ignored for value? One design could add and initialize a state variable called Count to count stimuli, then update the stimulus variable only when Count = 1.

In this case the state has two variables, Count and Stimulus.1, and Count must be initialized.

```
    Add.SB:  Return the first stimulus as the first response, and from
    then on return the sum of the last two stimuli as the next response.
```

One design could add and initialize a state variable called Count to count stimuli, and add a state variable Stimulus.State to store the last Stimulus detected. Then with each Stimulus, increase Count; with Count = 1, return the Stimulus to Response and move the value of Stimulus into Stimulus.State; with Count > 1, return the sum of Stimulus + Stimulus.State to Response and move the value of Stimulus into Stimulus.State.

In this case the state has two variables, Count and Stimulus.State, and Count must be initialized.

```
    Sum.SB:  Return the first stimulus as the first response, and from
    then on return the sum of the first and last stimuli as the next
    response.
```

One design could add and initialize a state variable called Count to count stimuli, and add a state variable Stimulus.State to store the first Stimulus detected. Then with each Stimulus, increase Count; with Count = 1, return the Stimulus to Response and move the value of Stimulus into Stimulus.State; with Count > 1, return the sum of Stimulus + Stimulus.State to Response.

In this case the state has two variables, Count and Stimulus.State, and Count must be initialized.

```
All.SB:  Return the sum of all stimuli as next response.
```

State box All.SB needs the sum of all stimuli to return as the response. A design could add and initialize a state variable called Stimulus.Sum, initially set to zero. With each Stimulus received add to Stimulus.Sum and return the value to Response.

In this case the state has one variable, Stimulus.Sum, which must be initialized.

10.3.5.3 Clear Box Design

Once the state box is designed, the clear box can be designed in many ways. The clear box design defines how to meet the need from stimulus to stimulus in the state, how the response is to be calculated from the last stimulus and the state, and how the new state is to be calculated from the last stimulus and old state. For example, to design processes for the simple examples above, consider the following.

```
Copy.CB  Return the last stimulus as the next response.
```

Clear box Copy.CB is simply defined by the assignment given in the black box

```
Response := Stimulus;

Hist.CB  Return the first stimulus as the next response.
```

Clear box Hist.CB needs two state variables, Count and Stimulus.1, Count initialized, and processed as follows.

```
Initialize Count := 0;

Count := Count + 1;
if Count = 1
then
  Stimulus.1 := Stimulus;
end if;
Response := Stimulus.1;

Add.CB:  Return the first stimulus as the first response, and from
then on return the sum of the last two stimuli as the next response.
```

Clear box Add.CB needs two state variables, Count and Stimulus.State, Count initialized, and processed as follows.

```
Initialize Count := 0;

Count := Count + 1;
if Count = 1
then
  Response := Stimulus;
  Stimulus.State := Stimulus;
else
  Response := Stimulus.State + Stimulus;
  Stimulus_State := Stimulus;
end if;
```

```
Sum.CB:  Return the first stimulus as the first response, and from
then on return the sum of the first and last stimuli as the next
response.
```

Clear box `Sum.CB` needs two state variables, `Count` and `Stimulus.First`, `Count` initialized, and processed as follows.

```
Initialize Count := 0;

Count := Count + 1;
if Count = 1
then
  Response := Stimulus;
  Stimulus.First := Stimulus;
else
  Response := Stimulus.First + Stimulus;
end if;
```

```
All.CB:  Return the sum of all stimuli as next response.
```

Clear box `All.CB` needs a single state variable, `Stimulus.Sum`, initially set to zero.

```
Initialize Stimulus.Sum := 0;

Stimulus.Sum := Stimulus.Sum + Stimulus;
Response := Stimulus.Sum;
```

In this case the state has one variable, `Stimulus.Sum`, which must be initialized.

10.3.6 Box Structure Examples

10.3.6.1 Maximum and Minimum Analysis

Let a sequence of temperatures in a chemical operation be analyzed for maximum and minimum values encountered in the past twenty four hours, minute by minute, namely the past 1440 values. If either the maximum and/or minimum value is changed it should be reported in a message to the user. How might this problem be addressed with box structures in Ada packages? The package specification will describe the external basis for user communication, and the package body will give the internal basis for computation.

We begin with the black box, which describes the behavior of the package in terms of stimuli history mapping into responses. The black box is very straightforward, except for getting started. For the first 1439 values there are not 1440 past values. With the first value it turns out to be both the maximum and minimum value so far; a message is due the user. If the second value differs from the first, one is the maximum and the other is the minimum, and a message is due the user; otherwise the maximum and minimum continue to be the same value and no message is needed. And so it continues through the first 24 hours of measurement. From then on the maximum and minimum values of the past 24 hours are contained in the last 1440 values received. In a long run chemical operation this initial condition may be soon forgotten, but there is no other way to get started. Of course, even though the entire stimulus history exists, only a maximum of the most recent 1441 stimuli need be consulted in deciding what response is needed for the user. Note that both the maximum and the minimum values can go either up or down. The last value may push the maximum value up or the minimum value down. And the value 1441 ago being dropped may push the maximum value down or the minimum value up.

More formally, this black box can be described as follows.

At each minute, determine the maximum and minimum values in the past 1440 temperatures or as many as exist if less than 1440; report any new values in maximum and/or minimum from one minute ago to the user.

More precisely, the stimulus is the temperature value just received for the last minute, say INTEGER variable Temperature. The response is one of four messages

```
"No change in Maximum or Minimum values"
"New Maximum value is ", Maximum
"New Minimum value is ", Minimum
"New Maximum value is ", Maximum, "New Minimum value is ", Minimum
```

where `Maximum`, `Minimum` are INTEGER variables whose values are the maximum, minimum temperatures of the past 1440 minutes or as many as exist in the first 24 hours of execution. More specifically, in Ada the package specification for `Temperature_Monitor` with a single procedure `Temperature_Check` is as follows.

```
with TEXT_IO;
package Temperature_Monitor
is
  procedure Temperature_Check -- Black Box form
           (Temperature : in INTEGER;        -- from black box
            Maximum, Minimum : in out INTEGER); -- from black box
    -- With each stimulus Temperature, return one of the four
    -- messages above with Maximum and/or Minimum based on
    -- stimulus history
end Temperature_Monitor;
```

Next, a state box can be designed directly from the black box. The state can be simply taken to include the last 1440 or fewer temperature values received, regarded now as part of the state rather than stimuli history. In fact, the last 1439 values plus the current stimulus are sufficient, but the calculations can be simplified if 1440 values have been saved. We will store these values in an array `Temperature_Array` with index of 1 .. 1440 and INTEGER values. In order to record the number of values received so far, one new INTEGER variable can be used, say `Count`, initialized at one and augmented by one on each transaction until 1440 is reached, then remaining constant from then on. Now with a new stimulus a total of `Count` + 1 values are known. If `Count` = 1440 and 1441 values are known, the oldest value is to be discarded and the last stimulus is to be saved. However before that oldest value is discarded, it can be checked as to whether it was the maximum or minimum. In that case some different value from the 1440 will become the maximum or minimum. In a similar way, the last stimulus can be checked as to whether it is a new maximum or minimum. As a result, any new maximum and/or minimum is to be reported to the user. After that analysis, the new stimulus can replace that oldest value. One final analysis is on how the 1440 values should be stored. One strategy is to store them in array Temperature_Array in sequence of access, say the most recent in location 1, ..., the oldest in location 1440. That will certainly work, but will require moving every value back one place in the array with every new value. A better strategy is to leave values where first placed, but keep an additional index that points to the last value. In this way the 1440 values would wrap around the array. We will use that strategy and call the location of the last value an INTEGER variable Last_Location. At this point we are ready to give a state box design for Temperature_Check as below.

```
package body Temperature_Monitor
is
  type Index is range 1 .. 1440;                -- from state box
  Temperature_Array : array (Index) of INTEGER; -- from state box
  Count, Last_Location : INTEGER := 1;          -- from state box
  procedure Temperature_Check -- State Box form
           (Temperature : in INTEGER;           -- from black box
            Maximum, Minimum : in out INTEGER)  -- from black box
  is
  begin
    -- With each stimulus Temperature maintain Maximum and Minimum in
    -- updated Temperature_Array (1 .. Count) and respond with the
    -- correct one of the messages below:
    -- "No change in Maximum or Minimum values"
    -- "New Maximum value is ", Maximum
    -- "New Minimum value is ", Minimum
    -- "New Maximum value is ", Maximum, "New Minimum value is ",
    --  Minimum
    --  putting last Temperature received at Last_Location
  end Temperature_Check;
end Temperature_Monitor;
```

Finally, the clear box of Temperature_Check can be designed as follows. First, it will be useful to separate the first case from the rest. For this reason, the executable portion will start with an **if** statement. One pair of new working variables, Old_Maximum, Old_Minimum of obvious meaning will be useful.

```
package body Temperature_Monitor
is
  type Index is 1 .. 1440;                      -- from state box
  Temperature_Array : array (Index) of INTEGER; -- from state box
  Count, Last_Location : INTEGER := 1;          -- from state box
  procedure Temperature_Check -- Clear Box form
           (Temperature : in INTEGER;           -- from black box
            Maximum, Minimum : in out INTEGER); -- from black box
  is
    Old_Maximum, Old_Minimum : INTEGER;         -- from clear box
  begin
    -- With each stimulus Temperature maintain Maximum and Minimum in
    -- updated Temperature_Array (1 .. Count) and respond with the
    -- correct one of the messages below:
    -- "No change in Maximum or Minimum values"
    -- "New Maximum value is ", Maximum
    -- "New Minimum value is ", Minimum
    -- "New Maximum value is ", Maximum, "New Minimum value is ",
    --  Minimum
    --  putting last Temperature received at Last_Location
```

```
if Count = 1
then
  Temperature_Array (1) := Temperature;
  TEXT_IO.Put (Item => "New Maximum value is " &
    INTEGER'IMAGE (Temperature));
  TEXT_IO.New_Line;
  TEXT_IO.Put (Item => "New Minimum value is " &
    INTEGER'IMAGE (Temperature));
  TEXT_IO.New_Line;
  Old_Maximum := Temperature;
  Old_Minimum := Temperature;
  Count := 2;
  Last_Location := 2;
else
  Temperature_Array (Last_Location) := Temperature;
  Maximum := Temperature_Array (1);
  Minimum := Temperature_Array (1);
  Find_Maximum_And_Minimum:
  for Next in 2 .. Counter
  loop
    if Temperature_Array (Next) > Maximum
    then
      Maximum := Temperature_Array (Next);
    end if;
    if Temperature_Array (Next) < Minimum
    then
      Minimum := Temperature_Array (Next);
    end if;
  end loop Find_Maximum_And_Minimum;
  if (Maximum = Old_Maximum) and (Minimum = Old_Minimum)
  then
    TEXT_IO.Put (Item => "No change in Maximum, Minimum
      values.");
    TEXT_IO.New_Line;
  if Maximum /= Old_Maximum
  then
    TEXT_IO.Put (Item => "New Maximum value is " &
      INTEGER'image (Maximum));
    TEXT_IO.New_Line;
    Old_Maximum := Maximum;
  end if;
  if Minimum /= Old_Minimum
  then
    TEXT_IO.Put (Item => "New Minimum value is " &
      INTEGER'image (Minimum));
    TEXT_IO.New_Line;
    Old_Minimum := Minimum;
  end if;
end if;
if Count < 1440
then
  Count := Count + 1;
end if;
if Last_Location < 1440
then
  Last_Location := Last_Location + 1;
```

```
      else
        Last_Location := 1;
      end if;
    end Temperature_Check;
  end Temperature_Monitor;
```

This clear box can be optimized in several ways. For example, if values for `Old_Maximum` and `Old_Minimum` are not effected by discarding the oldest temperature nor by the latest temperature, there will be no change in `Maximum` or `Minimum`, so the **`for loop`** calculation is not necessary. If indices in `Temperature_Array` for `Old_Maximum` and `Old_Minimum` are retained, the calculations can also be shortened.

10.3.6.2 Historic Change in Inventory Policy

Box structure analyses have changed understandings of behavior in classic situations. One example is in inventory theory and practice in military organizations in the United States. From the 1870's up to the middle 1950's, a standard practice in inventory control in the Navy and Army was called the "K months of supply policy", meaning that an item of inventory was kept on hand and on order at the level of K months of usage, where usage was measured over the past year and updated each month. This policy applied to all kinds of inventory, from socks to anchors, and to all levels of storage, from local storage to national entry points of inventory. The value of K depended on the item, typically around a year, a little less for socks and a little more for anchors. In principle, the value of K depended on three main factors, namely 1) time to order the item from industry, 2) the variation in demand from month to month, and 3) the criticality of the item being available. A missing anchor might hold up a ship going to sea, but missing socks could be worked around.

The K months of supply policy is defined directly as a clear box, and its properties as a black box was not known until the 1950's. It seemed very reasonable, but all through the years, inventory people knew something was wrong in the inventory systems. For example an anchor made in Ohio would be sent half way across the country to reach its final usage on a ship, or so it would seem. But the actual experience was that an anchor would be sent one and a half way across the country on the average. Where ever it was sent would likely be the wrong place, and it would be needed somewhere else. Also, in going up the inventory hierarchy, it would seem that the relative variability of demand would go down as lower level fluctuations averaged out. But the actual experience was that high level inventory points had even larger relative fluctuations than local storage points.

It was finally discovered, in spite of how sensible the K months of supply policy seemed, that it amplified the variability in demand in the orders to the next level, when it was supposed that demand variability would be reduced. This amplification was not visible in the clear box, nor even the state box, but became obvious in the black box. Intelligent people had looked hard at the K months of supply policy, particularly its clear box form, for 80 years and never suspected its amplification behavior. This discovery changed the inventory policy not only across the government but also across industry in the 1950'6 and 1960's.

This story is typical of what box structures allow in analysis. The clear box and state box seem very sensible in dealing with inventory as a state and how the item should be ordered to keep up with the demand. But the derivation of the black box from the state box shows what is wrong. This black box behavior was not discovered until the 1950's. In this case a new inventory control policy was discovered with surprising and optimal properties, not pursued any further here.

More formally, the clear box for the K months of supply policy can be defined in terms of demand `D(m)`, inventory `I(m)`, and order `O(m)` for month m (inventory is measured here at the beginning of the month and includes both what is on hand and what is on order). Then in each month a demand `D(m)` occurs, and inventory is decreased by that amount but also increased by the order `O(m)`, so

```
I(m + 1) := I(m) - D(m) + O(m);
```

Next, the order `O(m)` is determined by the K months of supply policy, namely

```
O(m) := K * ((D(m - 1) + ... + D(m - 12)) / 12) - I(m);
```

In this analysis we assume the item has existed more than K months and that `O(m)` has a non-negative value each month. In summary, we have a clear box of the form

```
O(m) := K * ((D(m - 1) + ... + D(m - 12)) / 12) - I(m);
I(m + 1) := I(m) - D(m) + O(m);
```

Next, the state box is given by converting this clear box sequence into a single assignment for `O(m)` and `I(m + 1)`, which requires that `O(m)` found in `I(m + 1)` be replaced by the expression assigned to `O(m)`, so

```
O(m), I(m + 1) := K * ((D(m - 1) + ... + D(m - 12)) / 12) - I(m),
           I(m) - D(m) + ((D(m - 1) + ... + D(m - 12)) / 12) - I(m);
```

which simplifies to

```
O(m), I(m + 1) := K * ((D(m - 1) + ... + D(m - 12)) / 12) - I(m),
                  - D(m) + ((D(m - 1) + ... + D(m - 12)) / 12);
```

Finally, the black box is given by converting this state box function into a black box function that has no inventory `I(m)` in its description, so as a first step, consider

```
O(m) := K * ((D(m - 1) + ... + D(m - 12)) / 12)
        + D(m - 1_) - K * ((D(m - 2) + ... + D(m - 13)) / 12);
```

Now, a little rearrangement shows that

```
O(m) := D(m - 1) + K * (D(m - 1) - D(m - 13)) / 12;
```

which can also be written as

```
O(m) := (1 + K / 12) * D(m - 1) - (K / 12) * D(m - 13);
```

Now `I(m)` has been eliminated, but a surprising thing has happened. Order `O(m)` depends on just two previous demands, `D(m - 1)` and `D(m - 13)`, not on all 13 demands. Furthermore, while the coefficients add to one, one is larger than one, one less than zero. So `O(m)` will expand compared to `D(m)`, and its variant will be greater than the variant of `D(m - 1) + ... + D(m - 12) / 12`, for example as might have been expected.

As a simple illustration, suppose each demand `D(m)` is independent and of values 75, 100, 125 with probabilities 1/4, 1/2, 1/4. Suppose K = 12. Then Table 10.1 shows the calculations of `O(m)` as follows.

```
O(m) = 2 * D(m - 1) - D(m - 13)
```

		D(m - 13)		
		75: 1/4	100: 1/2	125: 1/4
D(m - 1)	75: 1/4	75: 1/16	50: 1/8	25: 1/16
	100: 1/2	125: 1/8	100: 1/4	75: 1/8
	125: 1/4	175: 1/16	150: 1/8	125: 1/16

Table 10.1
Values of O(m) for K = 12

In this form it is easy to see that orders O(m) will vary from 25 to 175 while demands vary only from 75 to 125. These orders have the distribution shown in Table 10.2.

Order	25	50	75	100	125	150	175
Probability	1/16	1/8	3/16	1/4	3/16	1/8	1/16

Table 10.2
Distribution of O(m) for K = 12

Suppose K = 18. Then Table 10.3 shows the calculations of O(m) as follows.

```
O(m) = (5 / 2) * D(m - 1) - (3 / 2) * D(m - 13)
```

		D(m - 13)		
		75: 1/4	100: 1/2	125: 1/4
D(m - 1)	75: 1/4	75: 1/16	37.5: 1/8	0: 1/16
	100: 1/2	137.5: 1/8	100: 1/4	62.5: 1/8
	125: 1/4	200: 1/16	162.5: 1/8	125: 1/16

Table 10.3
Values of O(m) for K = 18

In this form it is easy to see that orders O(m) will vary from 0 to 200 while demands still vary only from 75 to 125. These orders have the distribution shown in Table 10.4.

Order	0	37.5	62.5	75	100	125	137.5	162.5	200
Probability	1/16	1/8	1/8	1/16	1/4	1/16	1/8	1/8	1/16

Table 10.4
Distribution of O(m) for K = 18

As a final illustration, suppose each demand D(m) is independent and of values 50, 100, 150 with probabilities 1/4, 1/2, 1/4. Suppose K = 12. Then Table 10.5 shows the calculations of O(m) as follows.

```
O(m) = 2 * D(m - 1) - * D(m - 13)
```

		D(m - 13)		
		50: 1/4	100: 1/2	150: 1/4
D(m - 1)	50: 1/4	50: 1/16	0: 1/8	-50: 1/16
	100: 1/2	150: 1/8	100: 1/4	50: 1/8
	150: 1/4	250: 1/16	200: 1/8	150: 1/16

Table 10.5
Values of O(m) for K = 12

First, a negative order appears, which might be handled in several ways. For this example consider it a return. Now, in this form it is easy to see that orders O(m) will vary from -50 to 250 while demands vary only from 50 to 150. These orders have the distribution shown in Table 10.6.

Order	-50	0	50	100	150	200	250
Probability	1/16	1/8	3/16	1/4	3/16	1/8	1/16

Table 10.6
Distribution of O(m) for K = 12

This result is similar to the first with a change in demand variability. In each case, the behavior shows internal orders vary more than external demand. The examples here are for simple demand distributions, but the general results hold as more realistic distributions are applied.

Once understood, it was easy to change inventory control from this policy to others that did indeed reduce variation of ordering in the system. But as noted, it took some 80 years of intelligent concern to get that understanding. Finding the black box and its properties was not known for either the reason or the reasoning until the 1950's.

10.3.7 Exercises

1. Distinguish between software engineering and program debugging in reaching defect free software.

2. What is stratified statistical testing and how does it deal with the importance as well as the statistics of software under test?

3. Identify a software problem requiring specification activity.

4. How is development activity related to specification of software and what can a developer do with incomplete specifications?

5. How is certification activity related to specification of software and what can a certifier do with incomplete specifications?

6. How are box structures related to Ada packages?

7. State box verification shows that the designed state box defines the same external behavior as the black box specification. Show this verification is analogous to verifying solutions to differential equations.

8. Clear box verification shows that the designed clear box defines the same internal behavior as the state box design. Show this verification is part of software verification to specifications.

10.4 Verification and Certification of Packages

10.4.1 Background In Statistical Quality Control

Computer software has existed only little over a human generation old, and software development as it is practiced today has been worked out in just that short time. Think of accounting when a human generation old, whenever that may have been. It certainly did not have double entry bookkeeping, and not even sound arithmetic methods. Civil engineering did not have right triangles or methods of calculating areas at that age. Software has many more people than accounting and civil engineering at that time. But fundamental ideas still take time to find, even though people in the field are doing with what is available without them,

In another direction, *statistical quality control* (SQC) came into being about a human generation ago, with work of Dr. Edward Deming and others in manufacturing in the 1950's. However, American industry largely ignored the new ideas of SQC for manufacturing in that period, getting along with however they were dealing (or not) with quality. Statistics seemed odd and extra effort for work in quality control the industry was already doing. Of course, the rest is well known with Dr. Deming and others taking SQC to Japan with dramatic successes in Japanese industry creating products with entirely new levels of quality and productivity both. Ever since, American industry has worked to catch up with Japanese industry in manufacturing SQC. In fact, today we recognize the work of Dr. Deming by embracing his ideas under the umbrella term Total Quality Management (TQM).

It is now beginning to be known how to develop software under statistical quality control as well. But SQC in software development is almost never practiced in American industry today. To accomplish SQC it is required to eliminate the *private debugging* done almost universally by programmers, replacing it with *engineering verification* instead. The initial, intuitive programming practices are still widely used in industry, in large part because so many people have learned to develop software informally and intuitively. Software users have learned to expect software failures now and then, and management has learned to put up with the problems they cause.

There is a lesson in arithmetic and engineering in number systems. Roman numerals were used two thousand years ago because nothing better was known. Perfect arithmetic is possible in Roman numerals, but is so difficult that many errors were expected and lived with. The Arabic numbers and methods were developed a thousand years ago. Arabic arithmetic is so straight forward people can learn it directly and practically eliminate errors. It goes from an art to engineering, permitting arithmetic review step by step. Answers are objective, either correct or not, and not subjective. As a result people are glad to discover any errors in reviews, to get answers right.

10.4.1.1 Statistical Quality Control In Manufacturing

There is a considerable difference in SQC between manufacturing and software. But manufacturing SQC has been very informative and helpful in going to software. In manufacturing the *design* is considered *correct* and the SQC applies to creating physical products to the design. The design may be wrong for the product, but the job of manufacturing is to meet the design, right or wrong. That is a difficult discipline to enforce, when manufacturing people think they see a better design. But in the long run it is better to force the separation of design and manufacturing for quality.

The physical parts manufactured may be slightly incorrect but the product still meet the needs of the design on a physical basis. For example a wire cannot be cut to a 10 mm length exactly, but say within .001 mm, and still meet specifications in the product performance. If higher accuracy to design is necessary the manufacturing process may need radical improvement.

Manufacturing under SQC is *very different* than under previous controls. For example, in a 1950 assembly line of twenty work stations, each station generating parts and adding to the product was producing products at a rapid rate, but many of such products were then found defective in the testing that followed. The attempted answer to such problems was to improve the part making stations, because if each station was producing perfect parts the product would be satisfactory. But while some improvements were indeed made, new products ran into similar problems no matter how hard people tried.

Manufacturing under SQC used ideas that first seemed strange and not useful. In the assembly line of twenty work stations, first work out what each intermediate assembly should perform like; in many cases the stations must be redesigned to make this possible. Next provide statistical measurements for the performances of the intermediate assemblies at each station, and make them with each partial product coming down the line. Now, shocking as it may seem, *stop* the entire assembly line if any partial product fails its performance test. Fix the reason for the failure in whatever preceding stations needed. All the workers are idle now! What a dumb thing that seems. In the old assembly line every body worked hard all the time. But forcing all the parts to be right during assembly created a dramatic improvement in quality and productivity both. The idle workers were a clear motivation for getting the work stations working correctly to levels undreamed of before.

So in retrospect, SQC seemed very strange for manufacturing assembly lines in American industry. Who would think such ideas would be practical? No wonder American industry turned it down in the 50's. The people could see the *extra work*, but could not see the *extra benefits*. They did not see that the objective is not statistics, it is quality control. The reason for statistics is that it is the only way to get real quality control. And the improvement in productivity is a pleasant surprise, as well. But it becomes understandable when the amount of rework becomes known and now understood as *unnecessary* under better parts work and good management.

Another simple case of industrial improvement was in touch typing. When typewriters were first available, everyone looked at the keys while typing; there seemed no other way. When touch typing was introduced it seemed impractical at first. Key lookers could learn to type in a single day, just needing to type faster as they went along. Why bother with such a complex method as touch typing? But it was finally learned that touch typing greatly improved both quality and productivity, to the point where it brought typewriters into business offices in ways that would never have happened otherwise. Until touch typing and its benefits, secretaries didn't use typewriters to any extent.

10.4.1.2 Statistical Quality Control in Software

With this background, it is time to move SQC into software development. However it is the *design* that must be produced *correctly* to meet a software *specification*. Just as American manufacturing in the 50's, American software in the 90's is created in well intended ways without SQC. Its performance is low in both quality and productivity compared to what is possible. In a 1990 European conference in Oslo, a Japanese group stated that Japanese companies were going into SQC just as described in this book.

But American companies need not bring up the rear this time around in software. Just as in manufacturing SQC and touch typing, it is not easy for current managers and workers to move into software SQC. It requires new *engineering capabilities*, but capabilities potential in educated and disciplined people. For example, manufacturing workers discovered they could create parts that were orders of magnitude more accurate than imagined before. Touch typers could create written text three times as fast with one tenth as many errors. Right now, software programmers imagine that software must have a few failures–say 1 to 5 per thousand lines of code–on release, but cannot image a serious objective of creating software with no failures and higher productivity. It is not to ask programmers to work faster, but to work smarter with *real engineering discipline* under SQC.

Zero failure software is *not possible* with *heuristic methods* of programming used in this first human generation. It is *possible* with mathematics based *engineering discipline* made possible by work of Dijkstra, Parnas, and others. Such a discipline was taught in the eighties in a six course curriculum in Software Engineering across IBM with a faculty of over sixty well prepared teachers and over ten thousand students. Three examples of zero failure software follow.

First, the US 1980 Census was acquired by a nationwide network system of 20 miniprocessors. The system was controlled by a 25 KLOC program, which operated its entire ten months in production with *no failure* observed. It was developed by Paul Friday, of the US Census Bureau, using *behavioral verification* in Pascal. Mr. Friday was given the highest technical award of the US Department of Commerce for the achievement. Mr. Friday was a student at University of Maryland and credited computer science courses in rigorous software design and testing for his achievement.

Second, the IBM wheelwriter typewriter products released in 1984 are controlled by three microprocessors with a 65 KLOC program. It has had *millions of users* ever since with *no failures* ever detected. The IBM team creating this software completed the curriculum of six pass/fail courses and used *behavioral verification with extensive testing* in a well managed environment to achieve this result.

Third, the US space shuttle software of some 500 KLOC, while not completely zero failure, has been zero failure in flight. It had a well known failure in attempting to synchronize the five computers for liftoff in the first flight, which of course was corrected. The IBM team also used *behavioral verification*, and *extensive testing* to achieve that result. The space shuttle software is such a large, complex, and visible product that there are real lessons in it. All IBM managers and programmers in the shuttle software were required to complete the basic curriculum of six pass/fail courses in understanding programs as *rules* for mathematical behaviors, and *behavioral verification* to remain on the team.

10.4.2 Verification of Packages

10.4.2.1 Behavioral Verification of Packages

In describing the activities of the development team, no mention is made of testing or even of compilation. The Cleanroom development team does not test or even compile. They use behavioral verification to demonstrate the correctness of packages in the program increments. Testing and measuring failures by program execution is the responsibility of the certification team.

The mathematical foundations for behavioral verification come from the deterministic nature of computers themselves. As noted, a computer program is no more, no less than a rule for a mathematical behavior. Such a behavior need not be numerical, of course, and most programs do not define numerical behaviors. But for every legal input a program directs the computer to produce a unique output, whether correct as specified or not. And the set of all such input, output pairs is a mathematical behavior.

With these mathematical foundations, software development becomes a process of constructing rules for behavior that meet required specifications, which needs not be a trial and error programming process. In any package the behaviors of the procedures and Ada functions combine to make up the semantics of the package. As already noted, the name of the subprogram is part of the input data on each call. The behavioral semantics of the package allows repeated calls of various procedures and Ada functions in any sequence.

The behavioral semantics of a structured programming language can be expressed in an algebra of functions with function operations corresponding to program sequence, alternation, and iteration. The systematic top down development of programs is mirrored in describing behavior rules in terms of algebraic operations among simpler behavior, and their rules in terms of still simpler behavior until the rules of the programming language are reached.

10.4.2.2 State Box to Black Box Verification

Black box specifications deal entirely with external behavior of the box structure, describing the behavior of the software as a mapping from any history of stimuli to the next response. So the first need in defining a black box behavior is a *specification function* (or *relation*). This behavior may well have other requirements of performance in both space and time to be met.

A state box design to meet this black box specification will identify internal states to replace the stimulus history of the black box. It will also identify the process required, to map any stimulus and current state to a response and new state. With this process, the initial state is required, as well. The process required for the state box is another black box itself, which has both the external stimulus and current state as the joint stimulus and the external response and new state as the joint response.

Now the verification of the state box is to ensure that the responses of the state box are identical with the responses of the black box at every step. The internal black box of the state box is not the same as the black box behavior because it creates a new state as well as an external response at each step. The two black boxes are related, of course. In fact the internal black box calculates a response from the current stimulus and current state which must be identical with the response of the external black box from the history of stimuli. The internal black box also calculates a new state as well.

10.4.2.3 Clear Box to State Box Verification

State box designs become specifications for clear boxes. The states will be identical and the clear box will expand the state box internal black box into a design of lower level steps calling on other black boxes. In this case verification is exactly program verification of procedures and Ada functions that make up the clear box definitions.

10.4.3 Certification of Packages

10.4.3.1 Software Certification with Failure Free Testing

Software is either correct or incorrect in design to a well defined specification, in contrast to hardware which is reliable to a certain level in performing to a design assumed to be correct. For small and regular software, it may be possible to exhaustively test the software to determine its correctness. Even then, failures can be overlooked from human fallibility. But software of any size or complexity can only be tested partially, and typically a very small fraction of possible inputs actually tested. At first glance, the fractions are so small for systems of ordinary size that the task of testing looks impossible. But when combined with mathematical verification, getting correct software is indeed possible.

Certifying the correctness of such software requires two conditions, namely 1) statistical testing with inputs characteristic of actual usage, and 2) no failures in the testing. For interactive software, the statistical correlation of successive inputs must be treated, as well. If any failures arise in testing or subsequent usage, the software is incorrect, and the certification is invalid. If such failures are corrected, the certification process can be restarted, with no use of previous testing results. Such corrections may lead to additional failures, or may not. So certifying the correctness of software is an empirical process which is bound to succeed if the software is indeed correct and may succeed for some time if the software is incorrect.

While possibly frustrating at first glance, this is all humans can assert about the correctness of software. In both verification and testing, human fallibility is present. But on second glance, the sequential history of certification efforts provides a human basis for assessing the quality of the software and expectations for achieving future correctness.

10.4.3.2 Certification Process

Certification of software on a scientific basis requires a statistical usage specification as well as behavioral and performance specifications. The testing must be carried out by statistical selection of tests from these specifications. Some uses of the software may be much more important than other uses, and the statistical selections can be given in various levels of stratified sampling. Thus, not only basic statistical usage is to be defined, but the relative importance of correctness for each usage. An extreme form of stratified testing is important cases chosen with probability 1. Tests selected directly do not add statistical inference on the correctness of the software. But they represent very important cases that need to be tested. Other important sets can be used to define other strata, and so on until all possible cases are included in strata.

The balance between a few important cases and general cases takes good engineering design in the best use of test capabilities that is seldom explicitly addressed today. The number of tests defined for each stratus is a matter of test design, from which the reliability of software which passes the test design can be calculated. This is new information which is often not known now until the software is put to actual use, but should be generated with behavioral and performance specifications beforehand. Without usage specification, testing can be inadequate and current software often faces just that problem of a surprising number of failures in use because the software is used differently than expected.

Next, the actual statistical testing must be carried out when the software is available, as noted in stratified form for sizable systems. If a failure is found in testing, the software should be returned to the developers for correction before further testing. When the location of the needed correction is identified and the correction made, a new start of testing is begun. The Time To Failure (TTF) is recorded for each failure discovered in the user section of the specification. The Time Without Failure (TWF) is tracked when no failures have appeared. This TWF can be tracked after the software is distributed to users as part of the characterization of is correctness. If failures appear with users, the same rules of correction and restart of TWF should occur.

10.4.3.3 Certification Experience

As already noted, some failures can be expected for software never executed before, even though verified. Humans are fallible in behavioral verification, even though the methods are complete and correct. In illustration, people can make mistakes doing long division, for example saying 7 times 4 is 35 instead of 28. But such mistakes can be found by other people without insulting the dividers, because there is no private subjective value in the common long division process. In the same way, people can make mistakes creating software to meet explicit specifications using common design methods. Again, such mistakes can be found without insulting the creators. As noted, under five failures per KLOC can be expected, but discovered quickly on execution, typically two or three such failures per KLOC. These failures are very different from failures following unit debugging by the programmers. Verification failures are simple program errors due to wrong relations in tests, wrong variables in subprogram calls, very seldom due to wrong control logic. These verification failures turn out to be easily found and fixed, with very few, if any, follow on failures.

At first glance it seems strange and wasteful to not let the programmers find and correct the simple failures before sending the software over to full testing and certification. But on second thought that is precisely what they do now, with the secondary sources of failures created unintentionally. Redefining software development as an engineering design process, with an independent engineering test process, changes the entire perspective. As surprising as it turns out, good software engineers can define software almost correct without testing, in fact correct enough for separate testing. But software design becomes a precise engineering process rather than an experimental trial and error process. Many programmers today know no other way than experimental trial and error to write programs. Some may be able to learn precise engineering, some not.

For example, the COBOL/SF prototype had a 5K subprogram for generating COBOL test programs tested in a sequence of four accumulating increments. These accumulations were given 30 tests each, or a total of 120 tests. On the first increment, all failures found occurred in the first test, and the following 29 tests ran correctly. On the accumulated first and second increments, all failures occurred in the first two tests, and the following 28 tests ran correctly. On the final two accumulated increments, all failures occurred in the first test, and the following 29 tests ran correctly. In subsequent use no failures were ever found in the subprogram.

Those are very different statistical experiences than in unit debugged software. It reflects the trivialities of the failures due to faulty verification from the execution side. Failures in unit debugged software would have been sprinkled through the 120 tests, not jammed up in the first or second tests. And more failures would have likely appeared after the 120 tests. This experience is typical in Cleanroom testing. All failures are discovered very early in the testing, and no further failures show up later.

10.4.3.4 Difference between Correctness and Reliability

As already noted, there is a profound difference between the correctness of software and the reliability of hardware. When software has hundreds or thousands of failures, its behavior may seem to approximate hardware reliability. But when software has very few failures, possibly none, the statistics of hardware failures are not valid. In this first human generation, it has seemed impossible to create zero defect software, but it can be, and has been, done as discussed above. Part of the issue is discovering a new human possibility, with more engineering education and engineering management. Part of the issue is the economic feasibility. As surprising as it may seem, it requires less human effort to produce zero defect software with new methods than failure prone software with older methods. The human effort required is both engineering verification and statistical testing, and they complement each other in unexpected ways.

10.4.4 Statistical Usage of Packages

The good use of Ada packages requires a test design based not only on the behavioral and performance specifications, but also on usage specifications and how critical each test case is to system behavior.

Packages will be used in many ways. But the set of usage and their frequencies will define statistical usage. Unlike procedure or Ada function uses which are single executions, the procedures and functions of a package can be used many times in many sequences in a single execution of the package. Determining the statistical usage of a package is a substantial intellectual challenge, comparable to the design of the package itself. At first glance, the statistical usage of a package may seem quite arbitrary, and hardly necessary–why not just get the package programmed and try it out? But on deeper look, there needs to be a basis for testing. If testing is not statistically based, the package can be deficient in certain ways but not discovered.

There is still the problem of defining the statistical usage of a package for its testing. Any given user may not use the package exactly in the way the statistics is defined. But that is not a problem. The idea of statistical usage defines usage for a class of users. It is conceivable that more than one set of users might be identified, say using the same package for radically different purposes. In this case, more than one set of statistics might be defined, associated with different sets of users.

As already noted, not only statistics needs be considered, but importance of various uses examined. Certain cases may be so critical that they should be identified and tested either individually or in small sets.

For this reason, a Cleanroom test design is based on a *stratified statistical strategy* derived from the statistics of usage expected for the software. For an important case, a stratus may consist of the single case (with probability 1), or a small subset of cases, on out to strata containing large sections of the software. A total test design defines each stratus (possibly hundreds or thousands) and the number of tests in each strata.

10.4.4.1 Statistical Testing of Packages

Successful testing of packages without any failures found leads to a *certification of correctness* of the software. If failures are later found the certification is *negated*. If failures are fixed the certification process can be started again. Certification continues with software release to users, moving with confidence from typical 3 sigma at release (.001 failures per usage) to and beyond 6 sigma (.000000002 failures per usage) with sufficient usage without failures.

As noted, if software is entirely correct, there is no way to be sure of that except by testing and usage without failures. However, the longer testing and usage goes on without failure, the greater the subjective confidence can be in that correctness.

As noted above, in the case of a package, testing requires use and reuse of the subprograms of the package in random ways appropriate to the application. These random ways require the derivation of the statistical usage for testing. Such statistical usage implies time and effort to develop and design proper and relevant testing plans.

10.4.5 Exercises

1. What is statistical quality control?

2. What is statistical quality control in manufacturing?

3. What is statistical quality control in software?

4. Package verification is related how to subprogram verification?

5. How does failure free testing certify software?

6. What is certification process?

7. Why is debugging dangerous to quality?

8. What is difference between correctness and reliability?

10.5 Behaviors of Linked Lists

As introduced in Chapter 9, access types provide a new and powerful way to store data. Arrays and records provide simpler and more direct ways to store data as well. Arrays and records define data in fixed ways, but access types define data in flexible ways, using designators, to dynamically allocate data objects.

Linked lists are general and useful ways to store data, which can be stored in either arrays or with access types. As noted in Chapter 9, either method of storage can be best, depending on the usage of the data. Access types take more effort step by step to set up data storage, but arrays require more effort to add or delete data items with ongoing data.

The verification of linked lists with either arrays or access types to store and transform data is different, as well. With arrays a standard sequential list is provided that is maintained in sequential form at all times, perhaps with considerable effort, such as adding or deleting a element early in the list, so many other elements must be moved one step to the right or left. With access types, a linked list is defined with pointers from each element to the next, so adding or deleting a element changes only the pointers to and from the adjacent elements–not adjacent in storage, but adjacent in value.

10.5.1 Specifying a Package for List Processing

As already noted, to deal with linked lists using arrays, a package called `List_Processing` in Chapter 9 was developed which contains four procedures, namely `Create_List`, `Insert_In_List`, `Delete_From_List`, and `Show_List`. In order to verify this package, all the procedures must be verified collectively as called in any sequence by a user. As noted above, to verify a procedure, it must be shown that it runs correctly on a single call, and that is all. But to verify the package, it must be shown that its subprograms (in this case all procedures) may be called in any sequence of any length in any order and return the correct data on each call of the data stored between calls as well as returned to the user.

In order to verify the package `List_Processing`, one must first identify its behavior specification. In this behavior specification one needs to create the black box behavior. To verify a subprogram one needs a specification to describe what is required from a single call of the subprogram. But to verify a package, one needs a specification of arbitrary sequential use of the subprograms, not just a single call. In this case the black box behavior will describe the results of any and every sequence of calls on the subprograms without any reference to stored data between calls. This black box behavior will be given by the behavior specification which maps any sequence of calls on the package of subprograms to the next response.

As already introduced, we want to specify a package `List_Processing` which consists of four procedures, so the package specification will have the form of the package already seen above

```
package List_Processing -- black box, no stored data
is
  procedure Create_List;
  procedure Show_List;
  procedure Insert_In_List;
  procedure Delete_From_List;
end List_Processing;
```

But the behavior specification must describe precisely what this package does in all call sequences of the procedures as a black box–that is without any reference to stored data. At first glance this may seem quite artificial and difficult, because it seems so natural to think of the stored data. But at second glance it becomes obvious that it is not so artificial or difficult. The stored data will come later in designing the state box and clear box.

The List_Processing black box identifies the entire history of stimuli to the procedures which generate successive responses in return. The name of each procedure called is part of the history. The input data is the other part of the history. So the history is a sequence of interlaced procedure calls and data. For example, following each Create_List will be a sequence of one or more elements, separated by 'y' items and terminated by a final 'n' item. Show_List requires no additional data to complete. Insert_In_List is followed by a New_Name to be inserted, and Delete_From_List is followed by an Old_Name to be deleted. As noted, the entire history is an arbitrary sequence of just these four kinds of stimuli sequences. The responses to these stimuli are determined entirely by the entire history.

The term "List" describes the current status of all elements in sorted order still active, which have been introduced by Create_List, inserted by Insert_In_List, not yet deleted by Delete_From_List. These current elements are given in the history of stimuli to the black box. The state box design may define an array and its contents in terms of active elements, but that is later, literally as part of design. We still need to identify a limited length of active elements in the black box history, call it Max_Length. The history itself is of unlimited length, and the number of elements introduced by Create_List, inserted by Insert_In_List, deleted by Delete_From_List are unlimited. Elements introduced or inserted when Max_Length elements are already stored will be ignored.

In more detail, as noted, the call sequences of the procedures can take any form. In particular, the procedure Create_List can be called any time, whether a current list exists or not. We will specify that Create_List will start a new list, wiping out any previous list, although other solutions are possible, such as ignoring a Create_List command if a non-empty list already exists. Another specification decision is whether Create_List must be furnished at least one element or not. We assume at least one element will be required in this specification. Another question is whether the list of elements supplied must be in sorted form? We will suppose it need not be in sorted form. What if more active elements are supplied than Max_Length? In this case we will ignore them, but other decisions are possible as well. So Create_List raises several questions to be resolved in the specification.

Show_List seems pretty straight forward. We return whatever the list is at the current time.

Insert_In_List has the same problem as Create_List in trying to add a new element when the list is full already while holding Max_Length active elements. That will be handled as with Create_List; the element will be ignored. Another question comes up if a new element is added when there is already one of that value already in the list. Two obvious ways to handle this is first permit only unique elements, and second permit duplicate elements. We will permit duplicate elements. Still another possibility is to attach a count to each distinct element. That might be an implementation tactic if much duplication is expected, but is not considered for the moment.

Delete_From_List can find one of two conditions, first the list has the element named to delete, and second no such element exists in the list. In each case a message can be returned. Delete_From_List will have an open question if the list can have duplicate elements. Does delete take out one element or all elements of a given value? We will assume a single element is deleted.

With these considerations, the behavior specification for the package List_Processing takes the following form.

```
package List_Processing -- black box, no stored data
is

  procedure Create_List;
    -- Except for exceptions discussed below: create a list as
    --   provided by user, one or more elements one name at a time at
    --   a prompt; the list will be supplied in unsorted form;
    -- Exceptions are: no element is provided; if list is already
    --   full with Max_Length elements, ignore element;

  procedure Show_List;
    -- Except for exceptions discussed below: print the current list;
    -- Exceptions are: no exceptions;

  procedure Insert_In_List;
    -- Except for exceptions discussed below: insert element provided
    --   by user in sorted place in list;
    -- Exceptions are: no element is provided; if list is already
    --   full with Max_Length elements, ignore element;

  procedure Delete_From_List;
    -- Except for exceptions discussed below: erase one element of
    --   the value given or signify no such value found;
    -- Exceptions are: no element is provided;

end List_Processing;
```

10.5.2 Designing the Package List_Processing

In Chapter 9, the package List_Processing was given to introduce the use of arrays for list processing with the four procedures specified above. We recall the package specification and package body in List_Processing from Chapter 9 as follows.

```
package List_Processing
is
  procedure Create_List;
  procedure Show_List;
  procedure Insert_In_List;
  procedure Delete_From_List;
end List_Processing;
```

```
with TEXT_IO;
package body List_Processing
is
   subtype Index_Type is INTEGER range 1 .. 200;
   subtype Names is STRING (1 .. 5);
   type List_Array is array (Index_Type) of Names;
   Name_Array : List_Array;
   Last : INTEGER := 0;

   procedure Create_List
   is
      More_Names : CHARACTER := 'y';
   begin
      TEXT_IO.Put (Item => "Ready to create your list. ");
      TEXT_IO.New_Line;
      TEXT_IO.Put (Item => "Enter each name at the prompt.");
      TEXT_IO.New_Line;
      Get_Names:
      while (More_Names = 'Y') or (More_Names = 'y')
      loop
         Insert_In_List;
         TEXT_IO.Put (Item => "More names - y or n? ");
         TEXT_IO.Get (Item => More_Names);
         TEXT_IO.New_Line;
      end loop Get_Names;
   end Create_List;

   procedure Show_List
   is
   begin
      TEXT_IO.Put (Item => "Your list contains ");
      TEXT_IO.New_Line;
      if Last > 0
      then
         Put_Names:
         for Count in 1 .. Last
         loop
            TEXT_IO.Put (Item => Name_Array (Count));
            TEXT_IO.New_Line;
         end loop Put_Names;
      end if;
   end Show_List;

   procedure Insert_In_List
   is
      New_Name : Names;
      Count : Index_Type := 1;
   begin
      TEXT_IO.Put (Item => "Enter value to be inserted => ");
      TEXT_IO.Get (Item => New_Name);
      TEXT_IO.New_Line;
      if (Last > 0) and (Last < Index_Type'LAST)
      then
         Find_Place:
         while (Name_Array (Count) < New_Name) and (Count <= Last)
         loop
            Count := Count + 1;
         end loop Find_Place;
      Move_Names:
      for Index in reverse Count .. Last
      loop
         Name_Array (Index + 1) := Name_Array (Index);
      end loop Move_Names;
      Last := Last + 1;
      Name_Array (Count) := New_Name;
      TEXT_IO.Put (Item => "Value has been inserted.");
      TEXT_IO.New_Line;
   elsif Last = 0
   then
      Name_Array (Count) := New_Name;
      Last := Last + 1;
      TEXT_IO.Put (Item => "Value is only element.");
      TEXT_IO.New_Line;
   else
      TEXT_IO.Put (Item => "Element can not be added. ");
      TEXT_IO.Put (Item => "Your list is full.");
      TEXT_IO.New_Line;
   end if;
end Insert_In_List;

procedure Delete_From_List
is
   Count : Index_Type := 1;
   Old_Name : Names;
begin
   TEXT_IO.Put (Item => "Enter value to be deleted => ");
   TEXT_IO.Get (Item => Old_Name);
   TEXT_IO.New_Line;
   if Last > 0
   then
      Find_Value:
      while (Name_Array (Count) /= Old_Name) and (Count < Last)
      loop
         Count := Count + 1;
      end loop Find_Value;
      if Count < Last
      then
         Move_Names:
         for Index in (Count + 1) .. Last
         loop
            Name_Array (Index - 1) := Name_Array (Index);
         end loop Move_Names;
         Last := Last - 1;
         TEXT_IO.Put (Item => "Name deleted.");
         TEXT_IO.New_Line;
      elsif Name_Array (Last) = Old_Name
      then
         Last := Last - 1;
         TEXT_IO.Put (Item => "Name deleted.");
         TEXT_IO.New_Line;
      else
         TEXT_IO.Put (Item => "Name not found.");
         TEXT_IO.New_Line;
      end if;
```

```
         else
            TEXT_IO.Put (Item => "Your list is empty.");
            TEXT_IO.New_Line;
         end if;
      end Delete_From_List;

   end List_Processing;
```

We discuss this package in separate parts below.

In summary of unusual conditions, the following are possible.

> In executing Create_List: If no input is provided any time through the loop the execution is halted, waiting for the input. No further action can be taken.
>
> In executing Create_List: If more elements are supplied than Max_Length (200), ignore their values.
>
> In executing Insert_In_List: If no input is provided the execution is halted, waiting for the input. No further action can be taken.
>
> In executing Insert_In_List: If active elements are of Max_Length (200), ignore value of additional element.
>
> In executing Delete_From_List: If no input is provided the execution is halted, waiting for the input. No further action can be taken.

10.5.3 Verifying the Package List_Processing

Now that we have a design for the specification of List_Processing, we need to verify its correctness. The discussion just above discusses this design and its verification informally, and we need to carry it out more formally and completely. The specification discusses user input/output relations in the black box form, while the design provides those user input/output relations in the clear box form with more specifics, in particular identifying an array solution not mentioned in the black box. The specification is open on some details of the communication between the user and the software, specifically the exact messages returned to the user in various circumstances. This is typical in real life software engineering, and requires intelligent interpretation for verification.

As noted above, the first concern in the verification is the treatment of a package, rather than a single subprogram. The behavior of each subprogram in the package is part of the package behavior, including the subprogram name at each call as well as data that may follow. In List_Processing there are four procedures defined in the black box specification and repeated in the clear box design. But in the design are additional Ada types and objects, as well as filled out Ada procedures that make use of them. Furthermore, each use of a procedure is started by a user which leads into an interactive process between computer and user, terminated by the computer. As noted above, if the user fails to respond to a request for data the process is terminated by the computer and the procedure involved is never completed. But that is possible.

In what follows we incorporate the black box procedure specifications as comments into the clear box procedure designs, in order to identify the verification task.

Package Body List_Processing

Package body List_Processing consists of a with clause for TEXT_IO, five declarations and four procedures. The TEXT_IO clause and the five declarations are made available immediately on execution, whereas the four procedures are called, if ever, during execution.

```
with TEXT_IO;
package body List_Processing
is
   subtype Index_Type is INTEGER range 1 .. 200;
   subtype Names is STRING (1 .. 5);
   type List_Array is array (Index_Type) of Names;
   Name_Array : List_Array;
   Last : INTEGER := 0;

   -- four procedures

end List_Processing;
```

The with clause for TEXT_IO makes all the subprograms of TEXT_IO available. Their exact behavior is given in the Ada Programming Language. The subprograms used will include Put, Get, and New_Line, all procedures, used in very direct ways.

The first four declarations create an array Name_Array of type List_Array with Index_Type INTEGER indices and 5 character STRING elements. Nothing is initialized in Name_Array. The fifth declaration of INTEGER Last gives what will be the final index of the current Name_Array, namely the count of how many elements are currently stored in Name_Array. On initialization that count is zero. This use of Last must be verified in the procedures.

The four procedures can be called in arbitrary sequence and will be taken up separately, as they use and update Name_Array and Last.Name_Array will contain elements entered but not deleted in sorted order. As noted, elements can be duplicated and appear in that form. Last takes on values 0 .. Index_Type'LAST(200), counting the number of elements currently stored in Name_Array, which may be zero. Elements provided beyond Index_Type'LAST will be ignored.

Procedure Create_List

The procedure Create_List has the specification as discussed above, namely to

```
procedure Create_List;
  -- Except for exceptions discussed below: create a list as provided
  --   by user, one or more elements one name at a time at a prompt;
  --   the list will be supplied in unsorted form;
  -- Exceptions are: no element is provided; if list is already full
  --   with Max_Length elements, ignore element;
```

Procedure Create_List declares one object,

More_Names which is CHARACTER and initially 'y',

used as discussed below. The execution begins with two start up messages to the user, namely

```
TEXT_IO.Put (Item => "Ready to create your list. ");
TEXT_IO.New_Line;
TEXT_IO.Put (Item => "Enter each name at the prompt.");
TEXT_IO.New_Line;
```

whose meaning is obvious. The remainder of the procedure is a while loop named Get_Names to create a list as provided by the user. Now the execution of the loop in the while loop depends on More_Names being 'y' or 'Y'. More_Names is initialized 'y' in the declaration so the first loop will be exercised. The first step in the while loop is to call Insert_In_List. Its function is to add the next element provided by the user, if possible. Recall the specification for Insert_In_List is as follows.

```
procedure Insert_In_List;
  -- Except for exceptions discussed below: insert element provided
  --    by user in sorted place in list;
  -- Exceptions are: no element is provided; if list is already full
  --    with Max_Length elements, ignore element;
```

Its effect will depend on whether the user responds with an element and the value of Last. If the user fails to respond execution is stalled. If the user returns an element and if Last is less than 200, the element will be entered in sorted order in Name_Array and Last incremented by one. If Last is 200, the element will be ignored.

On return from Insert_In_List the loop execution checks on whether more names are available from the user, in

```
TEXT_IO.Put (Item => "More names - y or n? ");
TEXT_IO.Get (Item => More_Names);
TEXT_IO.New_Line;
```

The user can respond with 'y' or 'n', or not respond. If the user fails to respond execution is stalled. If and when the user returns 'n' for More_Names, the loop will terminate. With a return of 'y' the loop will continue. If and when the loop terminates the procedure terminates as well.

In summary, let user inputs be a sequence of the form

```
name-1, y, name-2, y, ... name-k, n
```

First, if 1 <= k <= 200, Name_Array will be filled from 1 to k with name-1, name-2, ... name-k in sorted order, and Last set to k.

Second, if k > 200, Name_Array will be filled from 1 to 200 with name-1, name-2, ... name-200 in sorted order, and Last set to 200.

Finally, let user inputs be a sequence of one of the forms

```
name-1, y, name-2, y, ... name-k, y, ...
name-1, y, name-2, y, ... name-k, y
name-1, y, name-2, y, ... name-k
Null
```

Then Create_List will not terminate.

Procedure Show_List

The procedure Show_List has the specification as discussed above, namely to

```
procedure Show_List;
  -- Except for exceptions discussed below: print the current list;
  -- Exceptions are: no exceptions
```

Procedure Show_List is not iterative, but following the user start up will begin with the message given in the first two statements

```
TEXT_IO.Put (Item => "Your list contains ");
TEXT_IO.New_Line;
```

and will follow that with all elements found (including duplicates) by checking first if Last > 0, and if so defining a for loop called Put_Names from 1 to Last, and returning all Names found in Name_Array at those locations, namely in

```
if Last > 0
then
  Put_Names:
  for Count in 1 .. Last
  loop
    TEXT_IO.Put (Item => Name_Array (Count));
    TEXT_IO.New_Line;
  end loop Put_Names;
end if;
```

If Last = 0, only the message "Your list contains " will appear for the user.

In summary, Show_List will put the message "Your list contains " followed by a list of all elements in the list on separate lines.

Procedure Insert_In_List

The procedure Insert_In_List has the specification as discussed above, namely to

```
procedure Insert_In_List;
  -- Except for exceptions discussed below: insert element provided
  --   by user in sorted place in list;
  -- Exceptions are: no element is provided; if list is already full
  --   with Max_Length elements, ignore element;
```

Procedure Insert_In_List begins by two declarations of New_Name of type Names, and Count of type Index_Type initially 1. It begins with three TEXT_IO operations to get a value to be inserted called New_Name, as follows.

```
TEXT_IO.Put (Item => "Enter value to be inserted => ");
TEXT_IO.Get (Item => New_Name);
TEXT_IO.New_Line;
```

If the user does not respond, execution is stalled. Otherwise New_Name will have a value to enter into Name_Array. The rest of the procedure is a single if statement, starting with

```
if (Last > 0) and (Last < Index_Type'LAST)
```

so that three conditions will be found, first in the then part, second if `Last` = 0 in an elsif part, and third if `Last` = `Index_Type'LAST` in the else part. We carry out these three cases next.

```
if (Last > 0) and (Last < Index_Type'LAST)
then
   Find_Place:
   while (Name_Array (Count) < New_Name) and (Count <= Last)
   loop
      Count := Count + 1;
   end loop Find_Place;
   Move_Names:
   for Index in reverse Count .. Last
   loop
      Name_Array (Index + 1) := Name_Array (Index);
   end loop Move_Names;
   Last := Last + 1;
   Name_Array (Count) := New_Name;
   TEXT_IO.Put (Item => "Value has been inserted.");
   TEXT_IO.New_Line;
elsif Last = 0
then
   Name_Array (Count) := New_Name;
   Last := Last + 1;
   TEXT_IO.Put (Item => "Value is only element.");
   TEXT_IO.New_Line;
else
   TEXT_IO.Put (Item => "Element can not be added. ");
   TEXT_IO.Put (Item => "Your list is full.");
   TEXT_IO.New_Line;
end if;
```

The first part finds a place for the `New_Name`, then moves the names above it one position, increments `Last` by one, inserts the `New_Name` into the right place in `Name_Array`, and finally returns a message of the work to the user. Finding a place called `Count` for `New_Name` is straight forward in searching for it in the while condition. Moving the names beyond it must be done in reverse order from `Last` down to `Count` to preserve the data. Now `Last` must be incremented, and `New_Name` can be inserted at place `Count`. Finally the message given by

```
TEXT_IO.Put (Item => "Value has been inserted.");
TEXT_IO.New_Line;
```

completes this first part.

The second part deals with the special case `Last` = 0 so the list is currently empty, so the two statements, since `Count` = 1,

```
Name_Array (Count) := New_Name;
Last := Last + 1;
```

insert `New_Name` at the start of `Name_Array`, and `Last` is incremented to 1. Finally the message given by

```
TEXT_IO.Put (Item => "Value is only element.");
TEXT_IO.New_Line;
```

completes this second part.

The third part returns a message to the user

```
TEXT_IO.Put (Item => "Element can not be added. ");
TEXT_IO.Put (Item => "Your list is full.");
TEXT_IO.New_Line;
```

to the effect that the element cannot be added to the list.

In summary, the user either returns a name on request or does not. If a name is returned, then

If `Last` < 200, the name is added to the end of `Name_Array` and `Last` incremented.

If `Last` = 200, the name is ignored.

If a name is not returned, execution is stalled.

Procedure Delete_From_List

The procedure `Delete_From_List` has the specification as discussed above, namely to

```
procedure Delete_From_List;
  -- Except for exceptions discussed below: erase one element of the
  --   value given or signify no such value found;
  -- Exceptions are: no element is provided;
```

Procedure `Delete_From_List` begins by two declarations

```
Count : Index_Type := 1;
Old_Name : Names;
```

Execution begins with three TEXT_IO operations to get a value to be deleted called `Old_Name`, as follows.

```
TEXT_IO.Put (Item => "Enter value to be deleted => ");
TEXT_IO.Get (Item => Old_Name);
TEXT_IO.New_Line;
```

If the user does not respond, execution is stalled. Otherwise `Old_Name` will have a value to remove from `Name_Array`. The rest of the procedure is a single if statement which checks if `Last` > 0 so that elements exist in `Name_Array`, and if so then looks for `Old_Name` in it, or that `Last` = 0 and the list is empty.

```
if Last > 0
then
  Find_Value:
  while (Name_Array (Count) /= Old_Name) and (Count < Last)
  loop
    Count := Count + 1;
  end loop Find_Value;
  if Count < Last
  then
    Move_Names:
    for Index in (Count + 1) .. Last
    loop
      Name_Array (Index - 1) := Name_Array (Index);
    end loop Move_Names;
```

```
      Last := Last - 1;
      TEXT_IO.Put (Item => "Name deleted.");
      TEXT_IO.New_Line;
    elsif Name_Array (Last) = Old_Name
    then
      Last := Last - 1;
      TEXT_IO.Put (Item => "Name deleted.");
      TEXT_IO.New_Line;
    else
      TEXT_IO.Put (Item => "Name not found.");
      TEXT_IO.New_Line;
    end if;
  else
    TEXT_IO.Put (Item => "Your list is empty.");
    TEXT_IO.New_Line;
  end if;
```

The then part consists of a while loop followed by an if statement. The while loop `Find_Value` will search `Name_Array` for `Old_Name` with variable `Count`. The search will begin at 1 and go no farther than `Last` - 1. It will stop if and when `Old_Name` is found. What follows is an if statement on condition `Count` < `Last`, with a then part, an elsif part, and an else part. In the then part, if `Old_Name` is found before `Count` reaches the current value of `Last` then a for loop called `Move_Names` will move all elements beyond `Old_Name` one unit up, overriding `Old_Name`, and `Last` is decremented by one with the message

```
  TEXT_IO.Put (Item => "Name deleted.");
  TEXT_IO.New_Line;
```

In the elsif part if `Old_Name` = `Name_Array (Last)`, `Old_Name` is deleted by decrementing `Last` by one with the same message

```
  TEXT_IO.Put (Item => "Name deleted.");
  TEXT_IO.New_Line;
```

If `Name_Array (Last)` is not `Old_Name` the message

```
  TEXT_IO.Put (Item => "Name not found.");
  TEXT_IO.New_Line;
```

is reported. Finally, in the outer else it is discovered that `Last` = 0 with the message

```
  TEXT_IO.Put (Item => "Your list is empty.");
  TEXT_IO.New_Line;
```

In summary, the procedure either deletes a name on request or does not.

If the name is found, then the name is deleted from `Name_Array` and `Last` decremented.

If the name is not found, no change to `Name_Array` or `Last` is made.

If a name is not returned, execution is stalled.

10.5.4 Exercises

1. Expand the specification for `List_Processing` to completely define the package already designed, keeping it in user terms rather than Ada.

2. Redesign `List_Processing` to list repeated elements only once, but record a count with each existing element. Is any change in specification implied?

3. The current specification and design for `List_Processing` defines `Create_List` to add to any list already there. How would the specification and design be changed to start always new lists with `Create_List`?

4. The current specification and design for `List_Processing` allows `Insert_In_List` to start a list before `Create_List` has been used. How would the specification and design be changed to force the use of `Create_List` to start a list?

5. Identify how this partial specification for `List_Processing` maps into the design, and what remains as good engineering judgment in its verification?

Endnotes

1. 3 sigma is a term used to express the quality of an object; it indicates .001 failures per usage.

2. 6 sigma is an expression of greater quality equal to .000000002 failures per usage.

3. G. Booch, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Co., 1983.

Chapter 11

Sequential Ada IV

The programming world that we have created so far is very artificial. In particular, all of the numeric values that we have encountered are whole counting numbers, *i.e.* integers. We know that in the real world not all numeric values can be represented in this manner. For example, consider a monetary system where only whole dollar amounts are allowed. Further, consider what happens when we divide two integer values that are not multiples of one another, say 5 and 2. The result of that division is 2.5, yet up to now we have only allowed whole numbers to represent the results of such a division and thus we would obtain a result of 2. Clearly, this lack of precision is intolerable if it is your money that is being lost!

Therefore, we now will expand our treatment of numeric values to include the rational numbers that are represented in Ada as having fractional parts. Thus, if we divide the integer 5 by the integer 2, we will still obtain the integer 2 as the quotient. However, when we divide the floating point value 5.0 by the floating point value 2.0 we will obtain the floating point value 2.5, which is more in keeping with our real world experiences.

Later in this chapter, we will discuss the parameterization of record types using a discriminant and the combination of the array and record types to create arrays of records and records with array components. Next we will investigate a new concept in Ada that will afford us the capability of enforcing our software engineering principle of information hiding, namely the private type. Finally, we will explore the use of file formats for input/output of textual data.

11.1 Real Types

Real values conceptually consist of an unbounded, infinitely long set of numeric values with unbounded and infinite precision. In other words, these values consist of all possible values of numeric quantities extending from negative infinity to positive infinity on the number line, with an infinite level of precision. This data type is called the `universal_real`. Unfortunately, there are no machines that can represent an infinite range of values nor can they represent even a portion of these values with anything approaching infinite precision. Therefore, we are forced by hardware constraints to limit `universal_real` to a subset of these values that are representable on any given machine. In consequence, each implementation of Ada will provide a machine dependent type called FLOAT that will consist of the representable values of floating point numbers with some machine limited precision. The details of how these values are stored are not important for our purposes, but suffice it to say that these floating point numbers are represented internally differently than the integer values that we have already discussed.

In addition, Ada provides us with the conceptual ability to specify either a relative bound on the error of the actual value and its internal representation, or an absolute bound on this error. What this means and why it is important will be explained in this section.

11.1.1 Floating Point Types and Subtypes

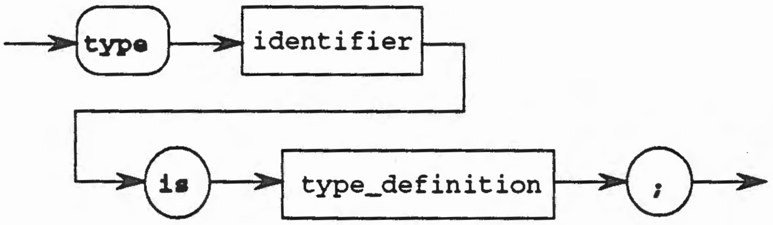
As was previously mentioned, the floating point values in any given implementation of Ada represent the precision and range limitations of the underlying computer. As in the integer types that we have already seen, there is a predefined type called FLOAT that is provided for every Ada implementation. In addition, the user is free to define other floating point types, specifying the relative precision, and optionally the range, to be used. The formal syntax for a floating point type and subtype is given next.

```
full_type_declaration ::= type identifier is type_definition;
type_definition ::= real_type_definition
real_type_definition ::= floating_point_constraint
floating_point_constraint ::= floating_accuracy_definition
                                  [range_constraint]
floating_accuracy_definition ::= digits static_simple_expression
subtype_declaration ::= subtype identifier is subtype_indication;
subtype_indication ::= type_mark [constraint]
type_mark ::= type_name | subtype_name
constraint ::= floating_point_constraint
```

Floating Point Type and Subtype Declaration
Syntax Definition 11.1

This syntax can also be shown in graphic form as follows.

full_type_declaration ::=

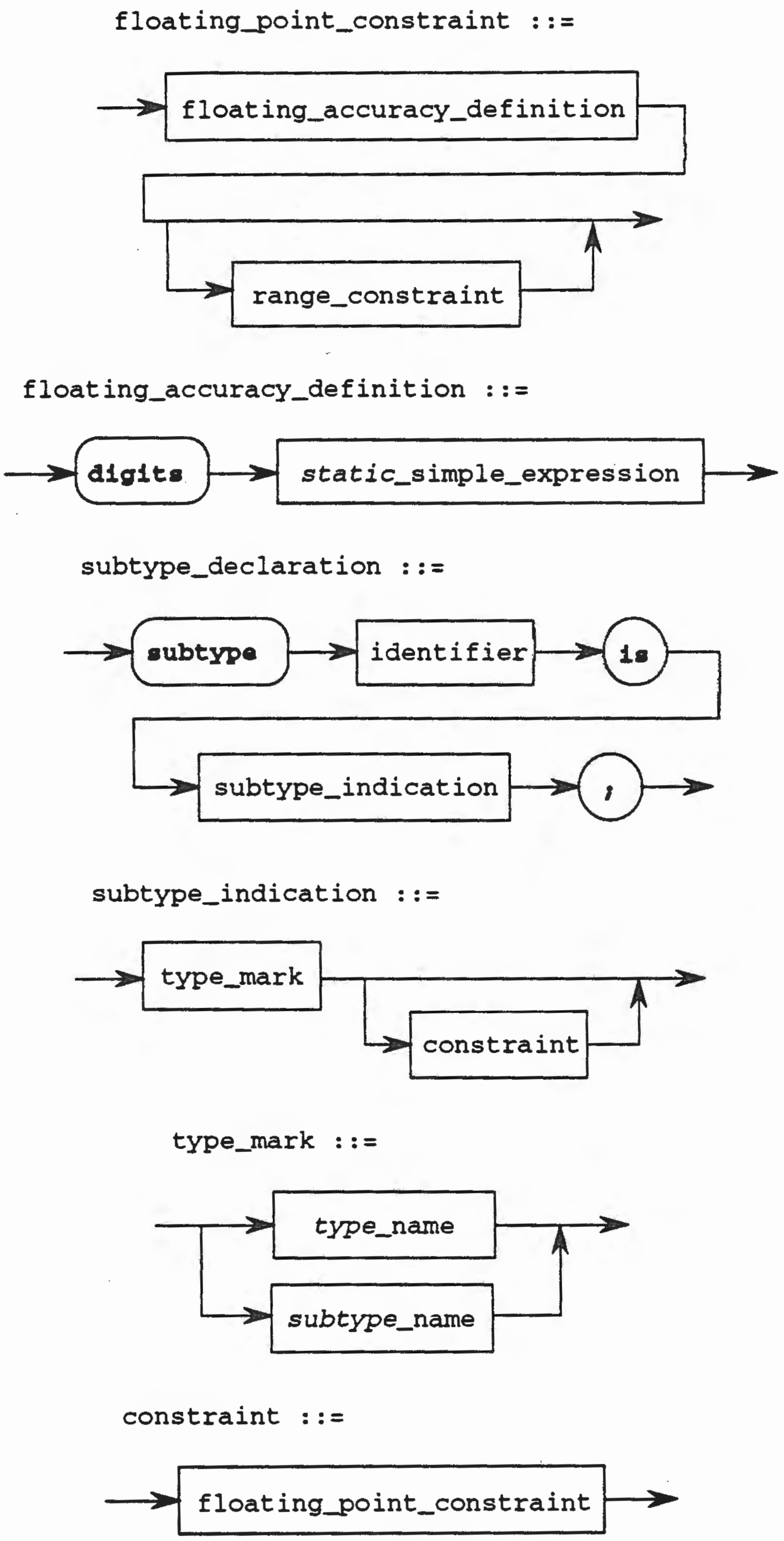


type_definition ::=

real_type_definition

real_type_definition ::=

floating_point_constraint



floating_point_constraint ::=

floating_accuracy_definition ::=

subtype_declaration ::=

subtype_indication ::=

type_mark ::=

constraint ::=

Floating Point Type and Subtype Declaration
Syntax Chart 11.1

Sample floating point type and subtype declarations are

```
type Ratio is digits 3 range 2.0 .. 5.3;
type Hand is digits 5;
type Length_Measures is digits 6 range -100.0 .. 100.0;
subtype Small_Hand is Hand digits 3 range 0.0 .. 1.2;
subtype New_Ratio is Ratio;
subtype My_Length_Measures is Length_Measures range -10.0 .. 10.0;
```

In these examples, the type `Ratio` is defined as a floating point type where objects of this type are represented internally with three digits of precision throughout the range from 2.0 to 5.3. The type `Hand` is represented with five digits of precision, and takes the default range of values for floating point numbers as provided by the implementor of the Ada compiler. The type `Length_Measures` defines a floating point type with six digits of precision throughout the range -100.0 to 100.0. Note that a given Ada implementation is allowed to limit the number of digits of precision that it supports, according to the underlying hardware. Thus, on one machine it may be the case that six digits of precision is the maximum that will be allowed, whereas on another machine you might be able to have up to nine digits of precision. The compiler will tell you at compile time if it is unable to support the requested level of precision.

The subtype `Small_Hand` represents a restricted subset of the values in the type `Hand`, reducing the required precision to only three digits and introducing a range constraint not present in the type `Hand`. Note that the precision specified in a subtype declaration must be no greater than the precision required in the type. Thus, you can lower the required precision, but you cannot increase it. The subtype `New_Ratio` does not restrict the type `Ratio` in any way, either in range or in precision. In effect, this is simply an alias for the type `Ratio` and achieves the same effect as a renaming of this type. The subtype `My_Length_Measures` represents a reduction in the possible range of values from the type `Length_Measures` without reducing the required precision.

In addition to user defined types, which are based on an analysis of the requirements of the problem to be solved, each implementation of Ada provides a predefined type named FLOAT. This type is used in the same manner as the predefined type INTEGER. The predefined type FLOAT has a degree of precision that is implementor defined and a range that is also implementor defined. The values of these limits can be found in the required Appendix F to the Reference Manual for the Ada Programming Language (LRM) that the vendor must supply with the compiler.

We could declare objects using these types in the same manner as we have seen previously, such as

```
My_Hand : Hand;
The_Distance : My_Length_Measures := 1.7;
Pi : FLOAT := 3.14159;
```

It might be useful to examine what is meant by the term *digits of precision*. What does it mean to specify that there will be five digits of precision? This term is used to refer to the number of significant digits that an object will require. If we have five significant digits, then the leftmost five digits in a canonical representation of the number are the only digits that will be meaningful. This is perhaps easier to see in the illustration given in Figure 11.1.

```
type NUMBERS is digits 3 range 0.0..20_000.0;

0.001, 0.002, 0.003, … 98.1, 98.2, … 997.0, 998.0, 999
1000.0, … 1010.0, … 10000.0, 10100.0
```

Relative Precision
Figure 11.1

Figure 11.1 also shows what is meant by the term *relative precision.* Note that numbers closer to zero give greater significance to the number of significant digits. Thus, near zero in this example, three digits of precision means that the difference between any two values in this type is only 0.001 or one one-thousandth. As we move further from zero, even with the same number of digits of precision, the significance of those digits lessens. Thus, when we approach 100, we see that with three significant digits we have 98.1, 98.2, *etc.*, where the distance between two successive values in the type is now every tenth instead of every one thousandth. As we pass 100 and approach 1000, we see that 997.0, 998.0, *etc.* are separated by 1.0 which is three orders of magnitude less significant than when we were close to zero and the distance between two successive values was one thousandth! Thus, we say this is *relative precision* because the number of significant digits does not change, but the meaningfulness of those digits increases as we approach zero and decreases as we get further away from zero. Similarly, the number of significant digits is merely the number of digits that have meaning in a number. Thus, the number 10_000_000.0 represented by a type that was declared to be `digits` 3 would mean that only the leftmost three digits (10000000.0) would have meaning. Adding one to this number would not change it because we would lose the significance of the one, *i.e.*, 10000000.0 + 1.0 = 10000000.0 because 10000001.0 is not representable due to the limitation of only three significant digits.

This phenomena can be a very large problem and may materially affect the results obtained in somewhat innocent looking computations. Thus, you must always be vigilant not to allow this type of a problem to creep into your algorithms. There is a whole field of study called *numerical analysis* which concentrates on solutions to problems of this nature. If you are interested in learning more about roundoff error, relative precision problems, truncation, and internal representation of numeric values you should consider taking a course in numerical analysis.

Attributes

For any given floating point type, say `F`, the attributes available to the user include the following

`FIRST` Yields the first value of type `F`, denoted `F'FIRST`
`LAST` Yields the last value of type `F`, denoted `F'LAST`
`DIGITS` Yields the number of decimal digits in the decimal mantissa of the model numbers for this subtype, denoted `F'DIGITS`
`MANTISSA` Yields the number of binary digits in the binary mantissa of the model numbers of this subtype, denoted `F'MANTISSA`
`EPSILON` Yields the absolute value of the difference between the model number 1.0 and the next model number above for this subtype, denoted `F'EPSILON`

Attributes can be used to query the system at runtime to determine values that may influence the executing algorithm. They are very useful for achieving portability.

As you can see, there is a whole new world of numeric values possible when we include the floating point types. Many computations that would not be possible with only the whole numbers are now available for our use. But there is another kind of real number that we still need to discuss, the fixed point number.

11.1.2 Fixed Point Types and Subtypes

Fixed point types are somewhat unique in programming languages. If you have ever used another programming language, then more than likely the real numbers that you have used were floating point numbers. In Ada, we have the choice between the floating point numbers as previously described, and fixed point numbers. You will recall that floating point numbers represented a *relative bound* on the error of a number's representation. Fixed point numbers, in contrast, represent an *absolute bound* on this error.

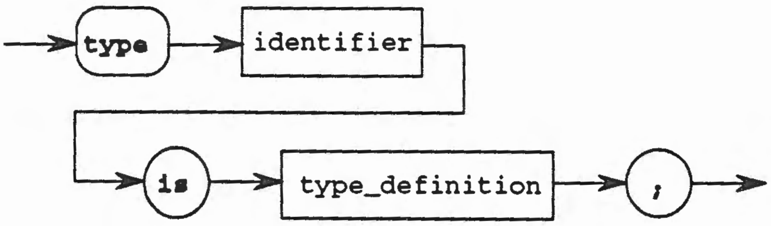
The formal syntax of a fixed point number is given next.

```
full_type_declaration ::= type identifier is type_definition;
type_definition ::= real_type_definition
real_type_definition ::= fixed_point_constraint
fixed_point_constraint ::= fixed_accuracy_definition
                           [range_constraint]
fixed_accuracy_definition ::= delta static_simple_expression
subtype_declaration ::= subtype identifier is subtype_indication;
subtype_indication ::= type_mark [constraint]
type_mark ::= type_name | subtype_name
constraint ::= fixed_point_constraint
```

Fixed Point Type and Subtype Declaration
Syntax Definition 11.2

This syntax can also be shown in graphic form as follows.

full_type_declaration ::=

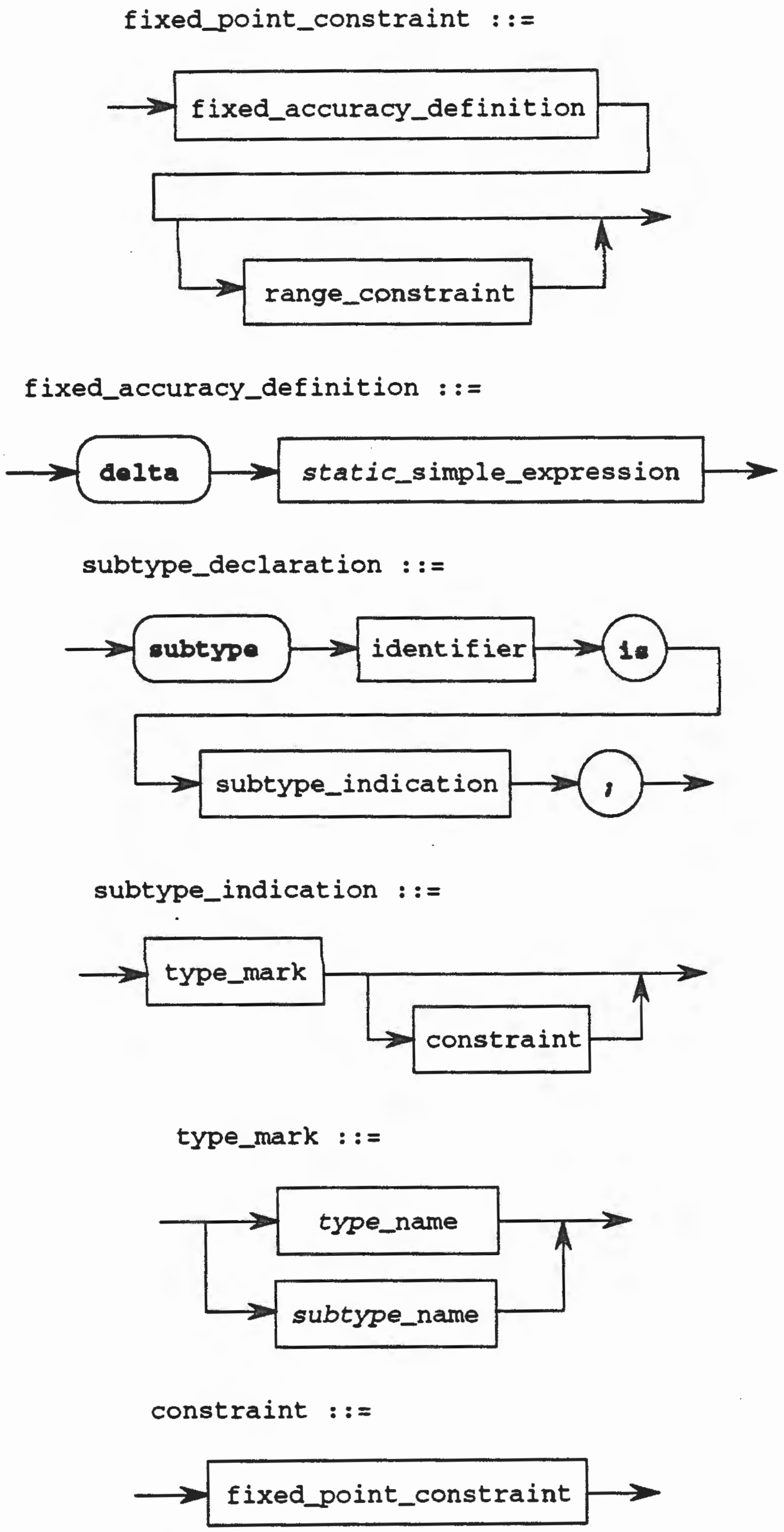


type_definition ::=

real_type_definition

real_type_definition ::=

fixed_point_constraint



Fixed Point Type and Subtype Declaration
Syntax Chart 11.2

Note that the range constraint is shown as optional. However, when you are declaring a new fixed point *type* the range constraint is required. On the other hand, when you declare a fixed point *subtype* the range is optional. Therefore, in the Syntax Definition and Syntax Chart the range must be shown as optional, even though we now know that for a type declaration it is required.

Sample fixed point type declarations are

```
type Dollar_Value is delta 0.01 range 0.0 .. 1.0;
type Very_Precise is delta 0.0006 range -10.185 .. 17.893;
subtype Not_So_Precise is Very_Precise delta 0.001;
subtype Even_Less_Precise is Very_Precise delta 0.01
  range -1.0 ..1.0;
subtype My_Dollar is Dollar_Value;
```

In these examples, the type `Dollar_Value` is defined as a fixed point type (note the use of the reserved word **delta** to mean fixed point types, while the reserved word **digits** is used for floating point types) where successive values of objects of this type are 0.01 apart throughout the entire range of 0.0 to 1.0. Remember that in floating point types, with relative precision, the closer we were to zero the more precise was our represented value. In this fixed point type, throughout the entire range of values, all values will be 0.01 apart[1]. This is termed an *absolute bound* on the error, contrasted with the relative bound provided by floating point types.

For the type `Very_Precise`, we see that objects of this type will be represented by values separated by 0.0006 throughout the range -10.185 to 17.893. Recall that for fixed point type definitions, the range *must* be provided. The subtype `Not_So_Precise` represents a restriction on the type `Very_Precise` where the delta or difference between successive values has been reduced to 0.001 from 0.0006. As in floating point types, we can reduce the required precision, but we may not increase it. Also, this subtype does not have, nor does it require, a range constraint. Since one is not provided, it will have the same range constraint as the type `Very_Precise`. The subtype `Even_Less_Precise` is also a restriction on the type `Very_Precise`, but in this case we have not only reduced the required precision (delta), we have also reduced the applicable range.

Finally, for the subtype `My_Dollar`, we have merely provided an alias or renaming of the type `Dollar_Value` without restricting either the accuracy requirements or the range. Thus, objects of this type will be identical to those of `Dollar_Value`.

All fixed point types in Ada must be declared by the user. In fact, there is only one predefined fixed point type, `Duration`, which is used to represent time intervals. In particular, there is no predefined type FIXED that can be equated to the predefined type FLOAT.

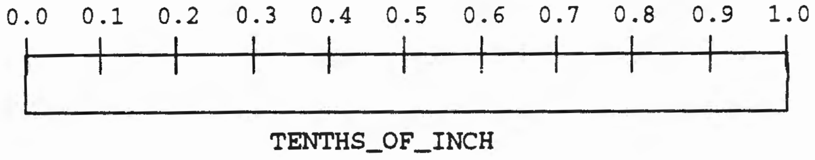
We could declare objects using these types in the same manner as we have seen previously, such as

```
The_Dollar : Dollar_Value;
The_Value : Very_Precise := -9.285;
```

In the last section, we explained what we meant by the term relative precision by showing a figure that illustrated our points. In this section we will do the same to illustrate the concept of absolute accuracy. Refer to Figure 11.2.

[1] This is not quite true. We have actually specified the largest power of two not greater than the delta value. However, this is not a course in numerical analysis, so for our purposes we will infer that the delta represents the actual bound on the error.

```
type TENTHS_OF_INCH is delta 0.1 range 0.0..1.0;
```



Absolute Error
Figure 11.2

In this figure we illustrate that throughout the entire range of permissible values for this type, the distance between any two representable values is always the same, 0.1. It does not matter whether we are close to zero or far away, the distance between any two values will always be the same. This is what is meant by an *absolute bound* on the error. Of course, in real life, we must deal with machines that use binary to represent these values. Therefore, the actual values will not be exactly as illustrated here. However, for our purposes, this explanation will suffice.

Attributes

For any given fixed point type, say F, the attributes available to the user include the following

FIRST Yields the first value of type F, denoted F'FIRST

LAST Yields the last value of type F, denoted F'LAST

DELTA Yields the value of the delta specified in the fixed accuracy definition for this subtype, denoted F'DELTA

MANTISSA Yields the number of binary digits in the binary mantissa of the model numbers of this subtype, denoted F'MANTISSA

SMALL Yields the smallest positive (non-zero) model number for this subtype, denoted F'SMALL

LARGE Yields the largest positive (non-zero) model number for this subtype, denoted F'LARGE

Thus, we see that fixed point numbers are conceptually very intriguing, but due to limitations in their implementations, they are not as useful as we might hope. For this reason, we will deal with floating point values for most of the rest of this course.

11.1.3 Exercises

<< To be added >>

11.2 Composite Types Revisited

We have already seen Ada's composite types, namely the record and the array. We have discussed some of their more basic features and shown their primitive operations. In this section, we will examine extensions and combinations of these data types to see what additional computation power is derived from their use. These basic types are very useful in combination to represent real-world problems internally in a form that mimics the problem domain. This will be shown in more detail in the following subsections.

11.2.1 Records with Discriminants

Recall that a record is a data structure that allows us to collect information about an entity into a single organized structure. We say that we have physically localized information that is logically related. This supports the software engineering concepts of localization and abstraction. Records allow us to group together heterogeneous data types that each represent some characteristic of an entity. Collectively they represent all of the information that we know about an entity.

However, one limitation of this composition of data is that all of the components of the record must be independent. By this we mean that each component represents data that is complete unto itself. There is no way to have one component depend upon the value of another component. For example, it would be very useful if we could create a record as follows

```
type Buffer is -- ILLEGAL!!!!
record
  Size : Positive;
  Item : String (1 .. Size);
end record;
```

where the number of characters in the string called `Item` is dependent on the value in `Size`. We can see immediately what the problem is here. The component `Size` does not have a value so how many characters should be allocated for the string `Item`? Thus, we are not allowed to have this type of relationship.

However, we know that record components can be given initial values. What if we corrected the record to appear as follows?

```
type Buffer is -- STILL ILLEGAL!!!!
record
  Size : Positive := 10;
  Item : String (1 .. Size);
end record;
```

While this might have allowed us to determine that the number of characters in `Item` is ten, it would not allow us much flexibility if we decided that we wanted another object of this type that had twenty characters in `Item`. Thus, we are left with a partial and unsatisfactory solution to our problem. Is there a better, more flexible means to accomplish what we need?

The answer is a record with a discriminant. A discriminant to a record acts as a parameter to allow us flexibility in declaring varying objects of the given type. It makes the record analogous to the unconstrained array we studied earlier, where different objects of the same array type could be of different sizes.

The discriminant part of the record specifies the discriminants for that record. In addition to acting as a parameter to the record, the discriminant is also considered a component of the record. The type of the discriminant must be discrete. The syntax for a record with discriminants is as shown below.

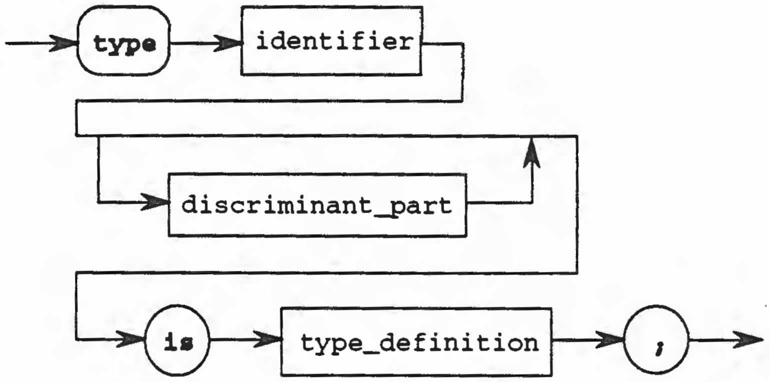
```
full_type_declaration ::= type identifier [discriminant_part] is
                          type_definition;
discriminant_part ::= (discriminant_specification
                      {; discriminant_specification})
discriminant_specification ::= identifier_list : type_mark
                               [:= expression]
type_definition ::= record_type_definition
```

```
record_type_definition ::= record
                                component_list
                            end record
component_list ::= component_declaration { component_declaration)
                   | null
component_declaration ::= identifier_list :
                          component_subtype_definition [:= expression];
component_subtype_definition ::= subtype_indication
subtype_indication ::= type_mark [constraint]
type_mark ::= type_name | subtype_name
constraint ::= range_constraint | floating_point_constraint
               fixed_point_constraint | index_constraint
```

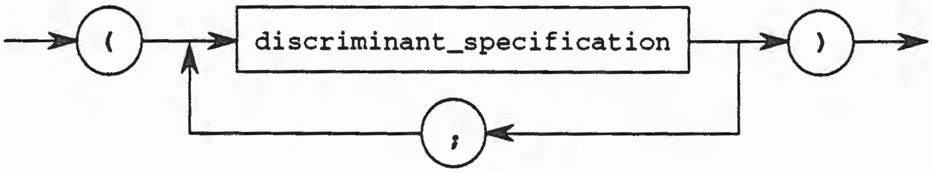
Record Type Declaration
Syntax Definition 11.3

This syntax can also be shown in graphic form as follows.

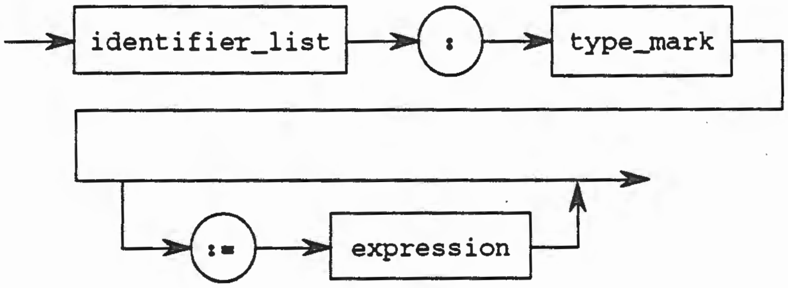
full_type_declaration ::=



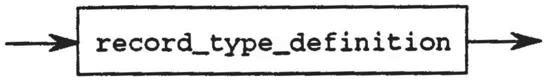
discriminant_part ::=



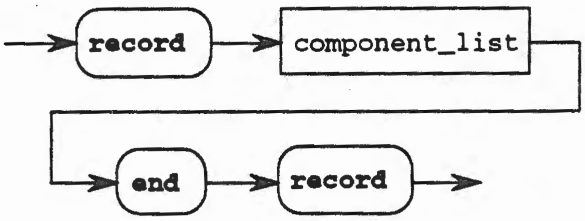
discriminant_specification ::=



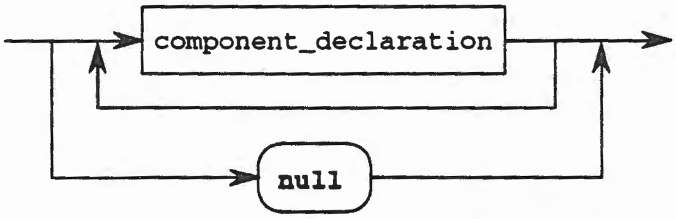
type_definition ::=



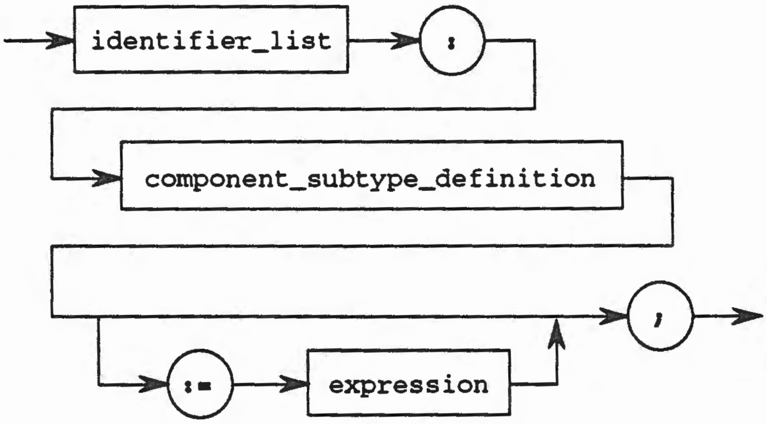
record_type_definition ::=



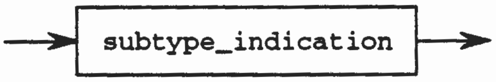
component_list ::=



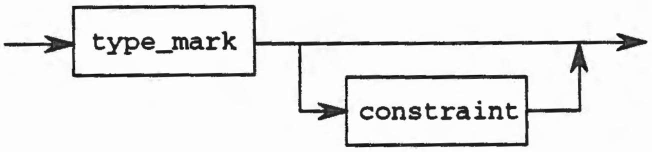
component_declaration ::=



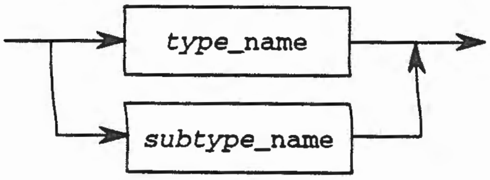
component_subtype_definition ::=



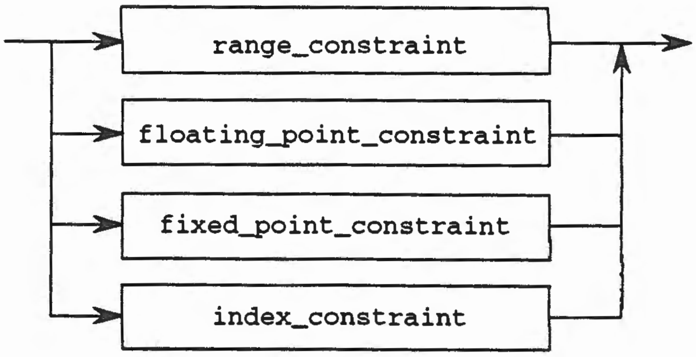
subtype_indication ::=



type_mark ::=



constraint ::=



Record Type Declaration
Syntax Chart 11.3

Returning to our previous example, we can use the discriminant to associate the size (length) of the string with the value of the record component `Size`. Thus, we could define the following record

```
type Buffer (Size : Positive) is
record
  Item : String (1 .. Size);
end record;
```

where `Size` is the discriminant for the record and is used to specify the upper bound to the string component `Item`. When we declare an object of this type we are required to specify the value for `Size`. Thus, we would declare

```
My_Buffer   : Buffer (10);
Your_Buffer : Buffer (20);
```

where `My_Buffer` would contain a value of 10 for `My_Buffer.Size` and `My_Buffer.Item` would be a ten character string. `Your_Buffer` would contain a value of 20 for `Your_Buffer.Size` and `Your_Buffer.Item` would be a twenty character string.

We see that we have two objects of the same type, but with different size components. This is the value of a record with discriminants. We can parameterize it to fit variable sized data and we can associate two or more components of a record to achieve a dependence and a correspondence between them.

Note that we have created a small problem. Suppose that we have a user that attempts to declare an object of type `Buffer` in the following manner

```
The_Buffer : Buffer;    --ILLEGAL!!!!
```

This would be illegal because the user did not provide a value for Size, the discriminant. Consequently, the compiler would not have any idea how long the string Item should be. In the absence of this data, the compiler must reject this declaration and the user would get a compilation error.

We can solve this problem by introducing the concept of a default value. A default value is one that we provide in the record declaration that will be used if the user does not specify a value for the discriminant. We would change our definition of Buffer to be

```
type Buffer (Size : Positive := 10) is
record
  Item : String (1 .. Size);
end record;
```

where we have added the default value of ten for Size. Now the user can declare an object of type Buffer and specify the value of Size which will be the one used to determine the upper bound on the string in Item. On the other hand, if the user does not specify such a value, the default value of ten will be used. Returning to our previous object declarations we see that

```
My_Buffer   : Buffer (10);
Your_Buffer : Buffer (20);
The_Buffer  : Buffer;  -- uses the default value of ten
```

are all now legal object declarations. The first object has a ten character string for My_Buffer.Item, the second has a twenty character string for Your_Buffer.Item, and the last object uses the default of ten to obtain a ten character string for The_Buffer.Item.

As was previously mentioned, the discriminant is a component of the record and can be selected in the same manner as any other component of the record using the dot notation. It is somewhat unique, however, in that it may not be changed directly. This should be obvious since there is a dependence between the discriminant and other components of the record. If we changed the discriminant, this relationship would be destroyed. For example, suppose we used our previous type declaration to declare My_Buffer as above. My_Buffer.Size has a value of 10 and My_Buffer.Item is ten characters long since the string is defined to be the length of My_Buffer.Size. Suppose that we now decided to change the value of My_Buffer.Size, say as follows

```
My_Buffer.Size := 20;
```

What would be the relationship now between My_Buffer.Item and My_Buffer.Size? How could we have a ten character string (Item) when Size has a value of twenty? In order to avoid this problem, Ada will not allow you to directly change the value of the discriminant.

However, we mentioned that one of the advantages of the discriminant is that it allows us to parameterize the record object. This would not be very useful if it was a one-time feature that could only be used when the object was initially declared. Thus, Ada provides a mechanism to change the discriminant, and consequently, the whole record. This can be done by using an aggregate assignment, where all of the new values for the record component are consistent, *i.e.*, the dependencies are preserved. For example, we could change My_Buffer to be a twenty character string by using the following aggregate assignment

```
   My_Buffer := (20, "This is new stuff!!!");
```

which simultaneously changes the value of the discriminant to twenty, and the length of My_Buffer.Item to be twenty characters long, preserving the relationship between these two components.

Thus, to change the value of a discriminant we must do an aggregate assignment. Also, in order to be able to do the aggregate assignment, there must be a default value provided for the discriminant. If no default value is provided when the type is declared, then this record cannot be mutated (changed), even with an aggregate assignment.

11.2.2 Arrays of Records

We have previously discussed the composite data structures of the array and the record. The array is used when all of the components are of the same type, *i.e.*, the components are homogeneous. The record is used when the components of the entity are of potentially different types, *i.e.*, the components are heterogeneous. In and of themselves, these composite data structures are powerful structuring tools for data, helping to model real-world entities inside of the computer.

As powerful as these data structures are alone, they are even more powerful in combination. It is sometimes very useful to define an array whose component values are records. This is called an array of records. Since the values stored in each component of the array are the same, *i.e.*, they are objects of the record type, this combination makes sense.

Consider the following data type declarations,

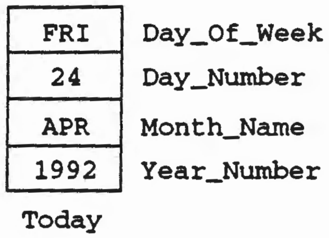
```
   type Months is (JAN, FEB, MAR, APR, MAY, JUN,
                   JUL, AUG, SEP, OCT, NOV, DEC);
   type Days is (SUN, MON, TUES, WEDS, THURS, FRI, SAT);
   type Years is range 1901 .. 2010;
   type Periods is range 1 .. 31;
   type Dates is
   record
     The_Day    : Days;
     The_Month  : Months;
     The_Period : Periods;
     The_Year   : Years;
   end record;
   Today : Dates;
```

Given these declarations, we can specify the components of a single day of the year by the aggregate assignment

```
   Today := (The_Day => FRI, The_Month => APR,
             The_Period => 24, The_Year => 1992);
```

As we have already seen, this will give us a record with the structure shown in Figure 11.3

```
  Today : DATE;
begin
  Today.Day_Of_Week := FRI;
  Today.Day_Number  := 24;
  Today.Month_Name  := APR;
  Today.Year_Number := 1992;
end;
```



Date Picture
Figure 11.3

in which we could access individual components using the dot notation.

We can now combine this data structure with the array structure to generate the following data declarations to define a collection of dates,

```
type Agendas is array (Periods) of Dates;
```

which results in a collection of thirty one records capable of holding one date each. This would allow us to define

```
January : Agendas;
```

to be an array of thirty one dates. However, when we tried to use this for `February`, we would discover that we would also have thirty one dates in `February`! We could choose to handle this by ignoring the dates above 28 (or 29 in a leap year). However, we already have seen a better way to define this type. Suppose we change the definition of `Agendas` to be

```
type Agendas is array (Periods range <>) of Dates;
```

This results in an unconstrained array that requires us to specify the number of days in the period when we declare an object of this type. Thus, we would declare the arrays for each month as

```
January  : Agendas (31);
February : Agendas (28);
March    : Agendas (31);
April    : Agendas (30);
```

in which case each month would have exactly the number of dates that is required. An assignment of a date to a particular place in the `April` array would be as follows

```
April(24) := (The_Day => FRI, The_Month => APR,
              The_Period => 24, The_Year => 1992);
```

Thus, we see how we can combine the record data structure with the unconstrained array to produce a powerful synergy.

Now if we examine what we have done so far, we see that there is a relationship between two of the components of the type `Dates`. Note that the number of elements in `The_Period` which represents the number of days in any given month, is directly related to the month. The current type declaration allows us to have a valid value for `The_Period` of thirty one even when `The_Month` is `FEB`. We can solve this problem by using a discriminant for the type `Dates`. Consider the following declaration

```
type Dates (The_Month : Months := JAN) is
record
  The_Day    : Days;
  The_Year   : Years;
  case The_Month is
    when JAN | MAR | MAY | JUL | AUG  | OCT | DEC =>
      Long_Month : Periods;
    when APR | JUN | SEP | NOV =>
      Normal_Month : Periods range 1 .. 30;
    when FEB => Short_Month : Periods range 1 .. 28;
  end case;
end record;
```

This is an example of a *variant record* and is presented here only to show you some possibilities. We will return to this kind of a record later and discuss it in more detail. Note that it allows us to vary the layout of the record based upon the value of the discriminant.

11.2.3 Records with Arrays

In the last section we examined the ways in which we could combine the record composite type and the array composite type to form arrays of records. In this section we will examine the other possibility, records that contain arrays. Actually we have already seen this idea in the discussion of records with discriminants. Recall that we had the following type declaration

```
type Buffer (Size : Positive := 10) is
record
  Item : String (1 .. Size);
end record;
```

in which the second component of the record type `Buffer` is an array of characters (string).

Consider a class of students that have grades from several examinations. We might choose to represent the examination scores as an array, indexed by the examination number in one dimension and the student number in the other dimension. Collectively this information represents all of the grade data for a given class. Thus, we might have

```
subtype Grades is Natural range 0 .. 100;
type Student_Number is range 0.. 99;
type Exam_Number is range 1 .. 10;
type Examination_Grades is
  array (Student_Number, Exam_Number) of Grades;
CS2003 : Examination_Grades;
```

This is a powerful way to encapsulate a significant amount of information. However, suppose we wanted to further capture all of the information about examination grades for any given professor. Suppose we have a professor that teaches CS2003, CS5160, and CS5161. We might define a record that contains all of her examination data as follows

```
type Grades is
record
  CS2003 : Examination_Grades;
  CS5160 : Examination_Grades;
  CS5161 : Examination_Grades;
end record;
The_Professor : Grades;
```

Now we can access and assign a perfect score to the fourth examination grade of the twenty-seventh student in CS5161 of `The_Professor` by the following

```
The_Professor.CS5161(27,4) := 100;
```

Thus, we can see once again the power of combining these two structuring tools into data structures that represent the way the data is viewed or understood in the problem domain.

11.2.4 Exercises

<< To be added >>

11.3 Private Types

One of the principles of software engineering that we have discussed previously is the concept of information hiding. This means that we desire to provide a type or a set of operations on a type to the user, but we do not want to allow access to, or information about, the way the data type is constructed. Indeed, we often do not even want the user to know how the data type is constructed.

On first thought, this seems somewhat extreme. Why not let the user know how something is implemented? The answer is that the user will either consciously or unconsciously take advantage of that knowledge. Often the user will consciously adjust their using code to reflect what they view as an optimal way of using the code provided. This may cause their code to *depend* upon the current implementation. For example, let us say that the exported type is a stack. Suppose the user needs to know the value of the item that is the third element down on the stack. If the user relies solely upon the abstraction, there is no way to examine the value of the third element in a stack without first "popping off" the two elements on top of it. This would require popping the first element and storing it somewhere, then popping the second element and storing it somewhere, and finally, popping the third element and examining it. Next, in order to restore the stack to its original condition, the user would need to push these three elements back onto the stack in their original order.

It should be obvious that this effort is time consuming and fraught with the danger of pushing the elements back on the stack out of order. In short, there is a lot of work involved and the potential for error is high, when all we need is the value of the third element in the stack. Therefore, if the user knows that the stack is actually implemented as an array, it is much easier to just index into the array at `Top-2` to see the value stored there, then it is to go through this popping and storing action. Naturally, the user is inclined to use the indexing "trick" to avoid the difficulty with using the abstraction as it is designed.

The problem with this is that now the user's code depends upon the fact that the stack is implemented as an array. Therefore, if we were to decide to change the implementation of our abstraction from a bounded stack using an array to an unbounded stack using a linked list, the user's code will fail. If the user had lived with our abstraction, then no change would have been necessary for the user's code. However, now we must change not only the way the stack is implemented, but every using program that took advantage of how it was implemented instead of following the abstraction. This is a very large problem with huge cost implications.

The solution to this problem is to prevent the user from gaining any knowledge of how the abstraction is implemented. In fact, we want to force the user to live with our abstraction and follow the accepted interface activities. In the case of a stack, perhaps that would be simply push and pop operations. The difficulty, of course, is how are we able to export a stack that is array based without also exporting this implementation fact to the user? The answer is to use a private type.

11.3.1 Introduction to Private Types

A private type is a type that allows us to export the data type and its associated operations to the user so that the user may declare objects of this type and manipulate them using the operations that are exported. However, the user cannot use any other operation on these types except for assignment, the test for equality, and the test for inequality. This limits the dependence of the user on the implementation details of an abstract data type and forces the user to follow the abstraction. However, the operations that are implicitly allowed in addition to those explicitly provided may sometimes cause difficulty. We shall see an example of how this works and how it helps in preserving the intended abstraction.

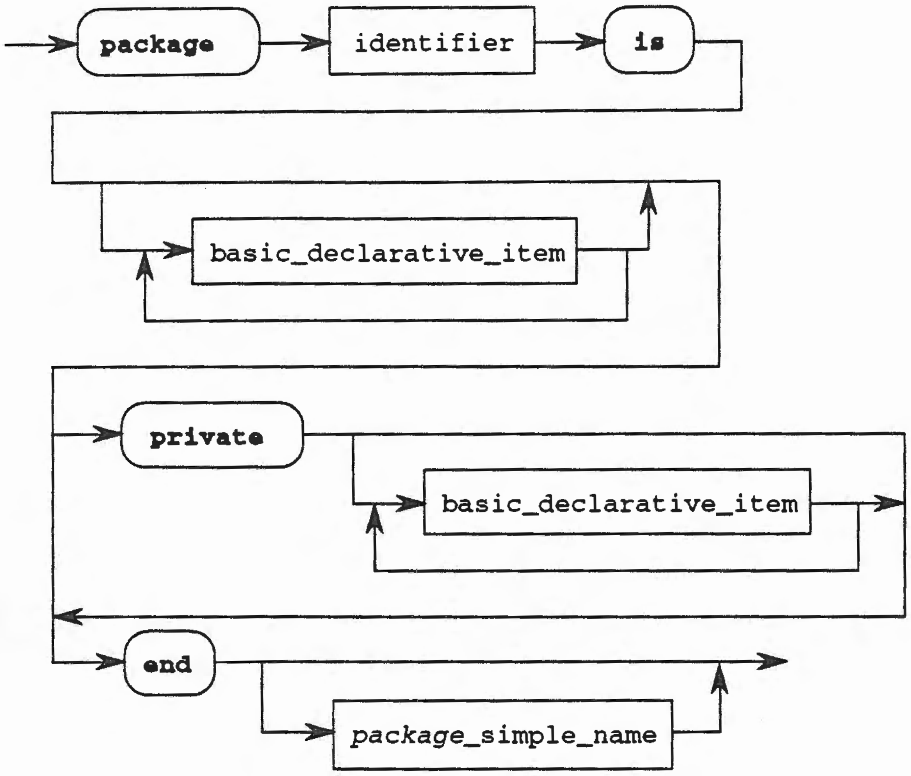
First, let us concentrate on the syntactic details of how a private type is declared. Private types may only be declared in a package specification. In fact, the introduction of the private type requires us to further define the package specification as having a visible part that is exported to the user and a private part that is required but which is not exported. Let us examine the syntax first and then return to our discussions and explanations.

```
package_specification ::= package identifier is
                          {basic_declarative_item}
                          [private
                          {basic_declarative_item}]
                          end [package_simple_name]
basic_declarative_item ::= basic_declaration
basic_declaration ::= type_declaration
                      | deferred_constant_declaration
type_declaration ::= private_type_declaration
private_type_declaration ::= type identifier
                             [discriminant_part] is
                             [limited] private;
deferred_constant_declaration ::= identifier_list :
                                  constant type_mark;
```

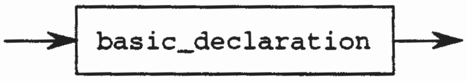
Private Type Declaration
Syntax Definition 11.4

This syntax can also be shown in graphic form as follows.

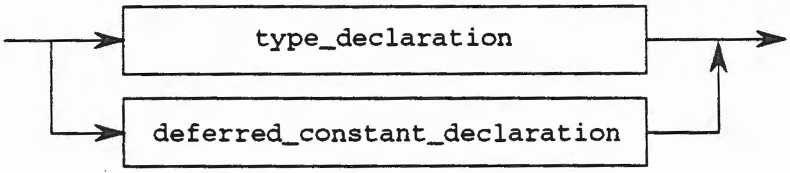
package_specifcation ::=



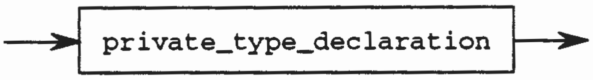
basic_declarative_item ::=



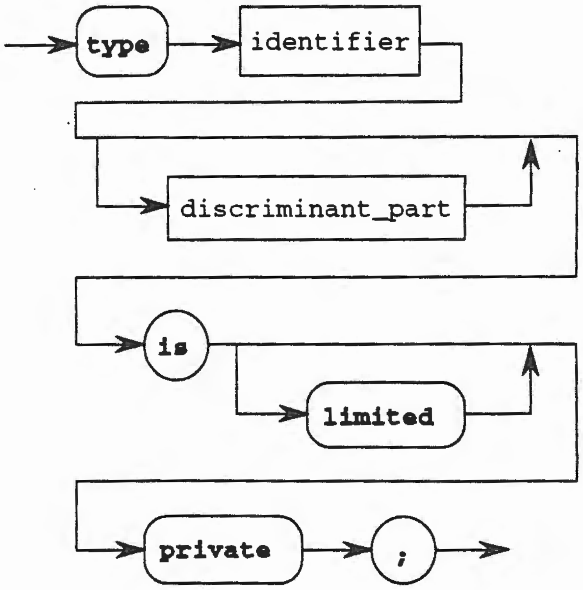
basic_declaration ::=



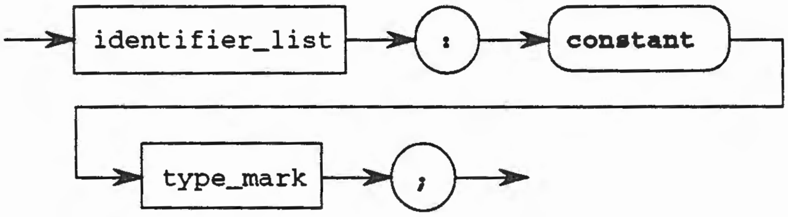
type_declaration ::=



private_type_declaration ::=



deferred_constant_declaration ::=



Private Type Declaration
Syntax Chart 11.4

Perhaps the best way to understand how to use a private type is to see an example of its use. Consider the local ice cream shop where each customer enters and takes a number from the Serv_a_Matic machine. Over the head of the server is usually a sign that says **Now Serving** and indicates a number. The server increments the Now Serving number every time he or she serves another customer. The customer waits in line until the number displayed in the Now Serving device is the same as the number that the customer obtained from the Serv_a_Matic. When that event occurs, the customer gets served ice cream.

Imagine how we might automate this process and model it in a computer. We could declare a package called `Ice_Cream_Shop` as follows

```
package Ice_Cream_Shop
is
  type Numbers is range 0 .. 99;
  procedure Take (A_Number : out Numbers);
  function Now_Serving return Numbers;
  procedure Serve (Number : in Numbers);
end Ice_Cream_Shop;
```

In this model we have defined a maximum of 100 customers (0 .. 99) that can be served. The procedure Take models the customer entering the shop and taking a number from the Serv_a_Matic. The function Now_Serving returns the value of the current number customer being served. The procedure Serve represents the actual preparation of your ice cream. We can imagine a simplified package body that might look like the following

```
package body Ice_Cream_Shop
is
  Serv_a_Matic : Numbers := 1;

  procedure Take (A_Number : out Numbers)
  is
  begin
    A_Number     := Serv_a_Matic;
    Serv_a_Matic := Serv_a_Matic + 1;
  end Take;

  function Now_Serving return Numbers
    is separate;
  procedure Serve (Number : in Numbers)
    is separate;
end Ice_Cream_Shop;
```

Given this abstraction, it seems obvious that a well behaved user of this package would design a procedure that takes a number and simply waits until it is time to be served. Then the procedure Ice_Cream_Shop.Serve is called to get the ice cream. Thus, we expect a user to create a procedure as follows

```
with Ice_Cream_Shop;
use Ice_Cream_Shop;
procedure Eat_Ice_Cream
is
  My_Number : Ice_Cream_Shop.Numbers;
begin
  Ice_Cream_Shop.Take(My_Number);
  Wait_My_Turn:
  loop
    if Ice_Cream_Shop.Now_Serving = My_Number
    then
      Ice_Cream_Shop.Serve;
      exit Wait_My_Turn;
    end if;
  end loop Wait_My_Turn;
end Eat_Ice_Cream;
```

This user has taken advantage of the infamous *use clause* to provide direct visibility to the equality operator needed to compare the function Ice_Cream_Shop.Now_Serving to the value contained in the object My_Number. We will ignore the reason for this right now, but suffice it to say that this is a valid use of the *use clause*. Note though, that the behavior of this user is the one we described when we created the ice cream shop model. The user gets a number from the Serv_a_Matic, then patiently waits until the Now_Serving function indicates that it is time to serve the number the customer holds. When the customer has been served, he or she exits from the waiting loop.

Unfortunately, not all users are so well behaved, nor will they all follow the chosen abstraction. Either intentionally, or unintentionally, the user of the `Ice_Cream_Shop` package may defeat our abstraction in order to optimize their own implementation. For example, consider a not so well behaved user who might write the following procedure

```
with Ice_Cream_Shop;
use Ice_Cream_Shop;
procedure Eat_Ice_Cream
is
  My_Number : Ice_Cream_Shop.Numbers;
begin
  Ice_Cream_Shop.Take(My_Number);
  Wait_My_Turn:
  loop
    if Ice_Cream_Shop.Now_Serving = My_Number
    then
      Ice_Cream_Shop.Serve;
      exit Wait_My_Turn;
    else
      My_Number := My_Number -1;
    end if;
  end loop Wait_My_Turn;
end Eat_Ice_Cream;
```

In this procedure the user has decided that if the value returned by the function `Now_Serving` is not the same as his or her number, then they will decrement the number. Thus, every time through this loop if they are not able to be served they will reduce their number until it gets to the value returned by `Now_Serving`. This is equivalent to checking the number, stepping around the person in front of you in line, checking the number again, *etc.* until you get to the head of the line. In the real world, if you were to try this, it would make many people very angry. It is also not part of our abstraction of the ice cream shop. Why, then, was the user able to violate our abstraction?

Upon examination we see that the problem occurs because we have exported (by virtue of listing it in the package specification) the type `Numbers`. The type `Numbers` is an integer type. This means that the user can declare an object of this type and manipulate it using the operations that we explicitly exported for it such as `Take`, `Now_Serving`, and `Serve`. However, the user may also use any of the operations appropriate for an integer type! Thus, addition, subtraction, *etc.* are all available to this user even though we did not want that to be the case in our abstraction. We are relying on the use of only the exported operations. We need a means to ensure that the user can only use our exported operations and not any that are exported as a consequence of the underlying type being used to represent a portion of our abstraction. We can do this with a private type.

Consider the following package specification that changes the type `Numbers` from an integer type to a private type.

```
package Ice_Cream_Shop
is
  type Numbers is private;
  procedure Take (A_Number : out Numbers);
  function Now_Serving return Numbers;
  procedure Serve (Number : in Numbers);
private
  type Numbers is range 0 .. 99;
end Ice_Cream_Shop;
```

In this new package specification, the *visible part* of the package is from the word package until the word private. Thus, we still are exporting all of the operations on the type `Numbers`, as well as the type `Numbers`. The difference is that we have specifically stated that the type `Numbers` is private. Consequently, the user may declare objects of this type and may manipulate them using *only* the operations that we explicitly export, as well as assignment, the test for equality, and the test for inequality, the three operations exported automatically with all private types.

The part between the word `private` and the end of the package is called the *private part*. Its sole purpose is to indicate to the compiler how much storage is need to represent objects of this type. Without the private part, we would lose the ability to separately compile specifications and bodies when private parts were used. Therefore, it was decided to use the private part to provide this information to the compiler, but to require that anything after the word private is not exported. Thus, the user may be able to actually see how the abstraction is implemented, but he or she cannot take advantage of it. This limitation is enforced by the compiler. To the user, it is as if the private part did not exist. For this reason, in many manuals and vendor's listings of their source code you will see an ellipsis after the word private because they do not want you to know how they implement things, even though you could not use the information anyway. This part of the package is a compromise between the needs of the compiler vendor and the requirements of good software engineering abstraction and information hiding.

In the private part is the complete implementation of the types declared in the visible part as being private. You may not leave the package specification without providing the complete declaration of any types declared as private. Any constants declared whose type is a private type must be given their values in the private part. Such a type cannot be given a value in the visible part for that would give away the implementation. Thus, we must declare the constant to be of the private type and hold off on providing its initial value until the private part. Such a constant is called a *deferred constant* because the value is deferred for the private part.

Now the user cannot use the procedure that he or she wrote last time. Specifically, the subtraction operation that was previously available when we exported `Numbers` as an integer type is no longer available. Thus, the user's code shown above would not compile with this new package specification.

We can see that private types are an extremely value asset in our implementation of the abstractions that require information hiding. However, sometimes, the limitations of private types are still not enough. We will examine this further in the next section.

11.3.2 Introduction to Limited Private Types

As we said in the previous section, the private type helps us to enforce our abstractions by providing the capability to hide information from the user. With a private type the user is only able to use the subprograms (operations) that we explicitly export to manipulate the private type, as well as the operations of assignment, the test for equality, and the test for inequality which are automatically available for all private types. Sometimes, however, even this is too much freedom. It allows the user to subvert our abstractions.

To see this, consider the following procedure that a user wrote to use our `Ice_Cream_Shop` package as modified.

```
with Ice_Cream_Shop;
use Ice_Cream_Shop;
procedure Eat_Ice_Cream
is
  My_Number : Ice_Cream_Shop.Numbers;
begin
  Ice_Cream_Shop.Take(My_Number);
  Wait_My_Turn:
  loop
    if Ice_Cream_Shop.Now_Serving = My_Number
    then
      Ice_Cream_Shop.Serve;
      exit Wait_My_Turn;
    else
      My_Number := Ice_Cream_Shop.Now_Serving;
    end if;
  end loop Wait_My_Turn;
end Eat_Ice_Cream;
```

Note that in this case, the user has once again violated our abstraction by not waiting his or her turn. In this case, the user checks to see if their number is the same as the value returned by the function Now_Serving. If it is, then the user gets the ice cream and exits the loop. If it is not the same, the user assigns the value returned by the function Now_Serving to their number. This is equivalent to going straight to the head of the line. In real life you cannot do this, so why was the user able to defeat our abstraction and avoid waiting his or her turn? The answer is that the private type gave the user operations (assignment, equality, and inequality) that we did not foresee the user having available. The user then used these operations to defeat our abstraction.

We can prevent this by using a limited private type. The limited private type is exactly the same as the private type except that no operations are exported implicitly. This means that the available operations are limited to only the operations (subprograms) that we explicitly export in the visible portion of the package specification where the limited private type is declared. This is very handy for preventing the user from defeating our abstraction. Consider the following change to the package specification to restrict the type Numbers to a limited private type.

```
package Ice_Cream_Shop
is
  type Numbers is limited private;
  procedure Take (A_Number : out Numbers);
  function Now_Serving return Numbers;
  procedure Serve (Number : in Numbers);
  function "=" (Left, Right : Numbers) return BOOLEAN;
private
  type Numbers is range 0 .. 99;
end Ice_Cream_Shop;
```

Note that in this new package specification we have changed Numbers to be a limited private type. A consequence of doing this is that we no longer export any operations other than those specifically declared in this package specification. Since we need to allow the user to compare two objects of type Number as part of the if statement, we will need to explicitly export the test for equality operation. You can see this in the package specification.

As a result of the foregoing, the user is now prevented from violating our abstraction. We have forced the user to use the abstraction the way it was designed. Does this defeat our intrepid user? Consider the following procedure.

```
with Ice_Cream_Shop;
use Ice_Cream_Shop;
procedure Eat_Ice_Cream
is
  My_Number : Ice_Cream_Shop.Numbers;
begin
  Ice_Cream_Shop.Take(My_Number);
  Wait_My_Turn:
  loop
    if Ice_Cream_Shop.Now_Serving = My_Number
    then
      Ice_Cream_Shop.Serve;
      exit Wait_My_Turn;
    else
      goto Cookie_Shop;
    end if;
  end loop Wait_My_Turn;
end Eat_Ice_Cream;
```

In this procedure (which is not quite correct Ada), if the user cannot defeat us, he or she will not stay at the ice cream shop, but will instead move on to the cookie shop!

Together, the addition of private and limited private types is a powerful new capability. Figure 11.4 shows a comparison of the capabilities of these two features.

Private	Limited Private
- Declared in Pkg Spec - Operations: Exported Subprograms Assignment Equality Inequality	- Declared in Pkg Spec - Operations: Exported Subprograms

Comparison of Capabilities
Table 11.1

11.3.3 Exercises

<< To be added >>

11.4 Text File Input and Output

We have seen throughout this textbook, that common operations in computing are input and output. So far we have been limited to reading from the keyboard and writing to the terminal screen. This is convenient and relatively easy in Ada, as long as you limit your I/O to characters and strings. However, it suffers from the drawback that it does not provide a means for non-temporary storage of the results. If we write to the terminal screen, those values disappear as they scroll off the screen when we write other things. Similarly, if we need to enter data we must laborious enter the data by hand. What we need is a mechanism whereby we can read from data that is previously captured and stored external to our program, and whereby we can write to entities that will exist after our program has terminated.

Such a mechanism is the file, and Ada provides for both input from and output to files. Basically, there are two classes of files, those that store the data as binary information encoded and interpreted as ASCII text and those that store the binary image as represented internally without any encoding. The first class of file is called a text file since it contains data that we can read and interpret as recognizable characters. The second class of file is called a binary file. We will not discuss binary files in this volume of the textbook. For now, we will concentrate on text files and the operations that we can perform on them.

11.4.1 Creating and Using Text Files in Ada

It is important to note from the outset that input and output (I/O) of text is not a part of the Ada language. Instead, it is provided by a predefined, standard package that is required to be provided by all implementations that have I/O capabilities. Thus, some compilers for embedded processors that do not have text I/O capabilities may not be required to provide this package. However, for the vast majority of the implementations that you are likely to see, you can expect that the predefined package will be available.

This predefined, standard I/O package for text is called TEXT_IO. The complete specification of this package, as well as an explanation of all of its provided functionality, is contained in Chapter 14 of the Reference Manual for the Ada Programming Language (LRM). It is strongly suggested that you have a copy of the LRM at hand as you read this section.

11.4.1.1 Creating a Text File

In the package TEXT_IO, there is a limited private type called `FILE_TYPE` that is the type of all text files. In order to create a text file we must first declare an object of this `FILE_TYPE` so that we have an internal object that we can operate upon. Next, we must associate the internal file name that we have just declared with an external entity, *i.e.*, one that is known to the operating system that will retain the file after the program has terminated. Once we have completed this task, we need only write information to the file, which can be accomplished by specifying the internal file name in the `Put` statement. Finally, in order to guarantee that the file will exist after the termination of our program, we must `Close` the file.

We can better understand these actions if we see an example. Consider the following simple program to write a string literal to a file.

```
with TEXT_IO;
procedure Demonstrate_File_IO
is
  The_File : TEXT_IO.FILE_TYPE; -- internal file name
begin
  TEXT_IO.Create (File => The_File,
                  Mode => TEXT_IO.Out_File,
                  Name => "demo.txt",
                  Form => "");
  TEXT_IO.Put(Item =>"This is written to the screen.");
  TEXT_IO.Put(File => The_File,
              Item => "This is written to the file.");
  TEXT_IO.Close (File => The_File);
end Demonstrate_File_IO;
```

In this procedure we have declared an object that is a `FILE_TYPE` and can therefore be used for representing our internal file name. The first statement in this procedure creates a file that will be known externally (and therefore after our program has terminated) as `demo.txt`. We have associated this external name with our internal name of `The_File` as a consequence of executing the Create statement. The Mode parameter in the Create statement specifies whether or not we want to read from this file or write to it. The mode must be In_File for files that we will read from or Out_File for files that we will write to. Accordingly, this is an Out_File since we will be creating it and writing to it. The Form parameter is used by an implementation to handle special cases that a given hardware may require. For the most part, and in all of our examples, you will not see this parameter used. It is given the empty string to signify that we have no actual parameter to send to it.

The next statement is a Put statement similar to the ones that we have always been using. Note that the only parameter to this Put statement is Item and that is what will appear, by default, on the terminal screen. The second Put statement is different from those that we have been using. it has a second parameter, namely File. This is where we specify the name of the file to which we want to write the Item. If we do not specify a file name using the File parameter, then by default it will be written to the terminal screen. If we specify a file, then the data will be written in the specified file, assuming that the file is open. In naming the file for this parameter, we always use the internal file name, in this case The_File.

Finally, we see a Close statement. This statement is used to tell the runtime system that we are finished writing to the file. The runtime system may have been keeping the data that we wrote in its own internal buffers and so the Close statement may be a cue to the runtime system to flush its buffers. In any case, this statement severs the association between our internal file name and the external file. We may no longer write to this file. However, when our program terminates, any data that we wrote to the file during the execution of our program will be preserved and kept around in the data file known to the operating system as demo.txt, the name we chose for the external file name when we created the file.

You should be aware that if you call the Create procedure and a file with the same external name already exists (demo.txt in our example), then the existing file will be deleted to make it possible to create the new file. Thus, you should always use care that your program chooses an external file name that is unique, unless you want to overwrite an existing file.

11.4.1.2 Opening a Text File

The example in the previous subsection demonstrated how we can create new files in Ada. The procedure to read files that already exist is very similar. Instead of the Create procedure from TEXT_IO, we use the Open procedure. The parameters and their meanings are exactly the same as the Create procedure. For example, a program to open the file that we created in the previous section might be as follows,

```
with TEXT_IO;
procedure Demonstrate_Open
is
  The_File : TEXT_IO.FILE_TYPE; -- internal file name
  The_Character : CHARACTER;
begin
  TEXT_IO.Open(File => The_File,
               Mode => TEXT_IO.In_File,
               Name => "demo.txt",
               Form => "");
  TEXT_IO.Get (File => The_File,
               Item => The_Character);
  TEXT_IO.Put ("The first character read is "
               & The_Character);
  TEXT_IO.Close(File => The_File);
end Demonstrate_Open;
```

In this example, we see that the first statement is the Open statement. Every parameter that existed in the Create statement is repeated here with the same meaning. Note that the actual parameter used for the Mode formal parameter is In_File for the Open statement because we will be using the file to read data, *i.e..*, the data will be coming in from the file. Recall that in the Create statement, the Mode parameter was Out_File because we were sending data out to the file.

In our example, the second statement is the Get statement. Notice that unlike our previous experience with the Get statement, this version has two parameters, one for the item to be read and another for the file from which to read it. The file is specified by using the internal file name we declared, namely The_File, which we associated with an external file name in the Open statement. The next statement in our example is the Put statement and all this does is write to the terminal screen the string literal "The first character read is " catenated with the character read from the file. Finally, as before, we closed the file by calling the procedure Close from TEXT_IO.

We have now seen all that we really need to see to use text files. The Create, Open, and Close procedures provide the bulk of the functionality that we will require in manipulating text files.

11.4.2 Input and Output Example

In this section we will demonstrate the use of file input and output simultaneously occurring in a program. Clearly, since we must specify the mode of a file as either In_File or Out_File, we introduce the concept of one way communication with a file. Consequently, it is not possible to both read and write to/from the same text file simultaneously. Therefore, the following example shows the case where the program will read data from one file, known to contain exactly ten scores, compute the average, and then write the scores and the average to another file.

```
with TEXT_IO;
procedure Demonstrate_File_IO
is
   Raw_Scores, Results : TEXT_IO.File_Type;
   subtype Scores is INTEGER range 0 .. 100;
   The_Score : Scores;
   The_Sum, The_Average : NATURAL := 0;
   Total_Scores : constant := 10;
   package Int_IO is new TEXT_IO.Integer_IO (Scores);
     -- provide I/O for values of type Scores
 begin
   TEXT_IO.Open   (File => Raw_Scores,
                   Mode => TEXT_IO.In_File,
                   Name => "scores.in",
                   Form => "");
   TEXT_IO.Create (File => Results,
                   Mode => TEXT_IO.Out_File,
                   Name => "scores.out",
                   Form => "");
   for Count in 1 .. Total_Scores
   loop                 -- the file contains 10 scores
     Int_IO.Get (File => Raw_Scores,
                 Item => The_Score);
     TEXT_IO.Put(File => Results,
                 Item => "Score number ");
     Int_IO.Put (File => Raw_Scores,
                 Item => Count,
                 Width => 0);
```

```
    TEXT_IO.Put(File => Results,
                Item => " is ");
    Int_IO.Put (File => Results,
                Item => The_Score,
                Width => 0);
    TEXT_IO.New_Line (File => Results);
    The_Sum := The_Sum + The_Score;
  end loop;
  The_Average := The_Sum / Total_Scores;
  TEXT_IO.New_Line (File => Results);
  TEXT_IO.Put(File => Results,
              Item => "The average is ");
  Int_IO.Put (File => Results,
              Item => The_Average,
              Width => 0);
  TEXT_IO.Close(File => Results);
  TEXT_IO.Close(File => Raw_Scores);
end Demonstrate_File_IO;
```

This example shows that we can have multiple files open at the same time, some for reading and some for writing. We can keep track of which file we are reading from or writing to by examining the file parameter. This parameter specifies which file we are performing the specified operation upon. In this example, we read the scores from the file Raw_Scores and wrote them with an appropriate message to a file called Results. As we read the files, we accumulated the total and then computed the average. This value, also with an identifying message, was then written to the output file. Finally, both files were closed.

As a final example, let's study a program where we will read character data from a file, echo it to the screen, and write it to another file. In effect, we will be duplicating the file and echoing the file on the screen. Take a minute to see if you can think about how to write this program before you read about our solution.

⊗ STOP–Think!!!

```
with TEXT_IO;
procedure Demonstrate_File_IO
is
  Data_File, Duplicate_File : TEXT_IO.File_Type;
  The_Character : CHARACTER;
begin
  TEXT_IO.Open   (File => Data_File,
                  Mode => TEXT_IO.In_File,
                  Name => "data.in",
                  Form => "");
  TEXT_IO.Create (File => Duplicate_File,
                  Mode => TEXT_IO.Out_File,
                  Name => "duplicate.out",
                  Form => "");
  Echo_Data:
  while not TEXT_IO.End_Of_File(File => Data_File)
  loop
    Echo_Line:
    while not TEXT_IO.End_Of_Line(File => Data_File)
      TEXT_IO.Get(File => Data_File,
                  Item => The_Character);
      TEXT_IO.Put(File => Duplicate_File,
                  Item => The_Character);
      TEXT_IO.Put(Item => The_Character);
    end loop Echo_Line;
    TEXT_IO.Skip_Line(File => Data_File);
    TEXT_IO.New_Line (File => Duplicate_File);
  end loop Echo_Data;
  TEXT_IO.Close(File => Duplicate_File);
  TEXT_IO.Close(File => Data_File);
end Demonstrate_File_IO;
```

This simple program works as described above. Note the use of two new features that will be discussed in the next section, namely the functions `End_Of_Line` and `End_Of_File`. If you did not include these in your solution, but got the rest correct, then you did well. Note that in this solution, the first `Put` is specified as going to the file `Duplicate_File` because of the parameter for `File`. The second `Put` omits this parameter and consequently, the output goes to the default output file, which is the terminal screen.

The rules for writing to a file are not hard to learn and are always available for reference in Chapter 14 of the LRM. You should get used to writing your output to a file, since it is more likely that you will write to files than to the terminal screen in real-world applications. You now have all of the information that you need to manipulate files, but we will expand this knowledge to provide you more powerful tools in the next section. For now, let's examine the use of TEXT_IO to read and write real values, *i.e.*, floating point and fixed point numbers.

11.4.3 Input/Output of Real Values (Fixed and Float)

Recall that when we wanted to read or write numeric values that had integer representations, we could not do this directly with TEXT_IO. We first had to create an instantiation of `Integer_IO`, a subpackage of TEXT_IO. Once this had been accomplished, we were able to manipulate numeric values of the type specified in the instantiation. This is also the same thing we did for enumeration values.

For real numbers we have to do the same actions as we did for integer types and enumeration types. In order to be able to do this, we are provided with two more subpackages of TEXT_IO. One is called `Float_IO` and is used to create packages to read and write floating point values. The other is called `Fixed_IO` and provides the same capabilities for fixed point values. You should look in Chapter 14 of the LRM and locate these two subpackages inside of TEXT_IO.

Without spending additional time explaining generic instantiations, let us proceed with showing the mechanics of how they are accomplished for fixed and floating point values. The syntax for an instantiation is nearly identical to that for integer input/output, as shown below.

```
type My_Float is digits  6 range 0.0 .. 100.0;
type My_Fixed is delta 0.1 range 0.0 .. 10.0;
package My_Float_IO is new TEXT_IO.Float_IO (My_Float);
package My_Fixed_IO is new TEXT_IO.Fixed_IO (My_Fixed);
```

The identifier chosen for this package may be any legal identifier. It is useful to choose a name that represents the type being manipulated. In this case, we have created two new packages as instances of generically defined packages. The first one allows us to perform I/O on the floating point type `My_Float` and the second allows us to do the same for the fixed point type `My_Fixed`.

Once these instantiations are provided in a program, all of the capabilities of the generic packages, as specified in their specifications found in TEXT_IO, are available. Primarily this is just `Get` and `Put` for these numeric types. In addition, there are conversion routines to allow us to convert a text string of digits to a floating point value and, conversely, to convert a floating point value to a text string of digits. You will not have need of these routines often so we will not discuss them further.

The Get procedure is provided in two forms. One takes a single parameter of the type specified in the instantiation and obtains a value for this object from the current input file, which is, by default, the keyboard. The other Put procedure takes two parameters. One is for the file from which to read the value, the other is for the value to be read. Just as we have previously seen for characters, we may specify where we want to read the data, from the keyboard or from any given open file.

The Put procedure in an instantiation of Float_IO or Fixed_IO also is provided in two forms. The Put statement in the first form does not include a file specification and so the value is written to the current output file which, by default, is the terminal screen. The second form of the Put is identical except that the first parameter is the file to which we desire to write the value.

An example program may help to understand the use of these subprograms when we are working with floating point values. Consider the following program.

```
with TEXT_IO;
procedure Demonstrate_Float_IO
is
  Data_File : TEXT_IO.File_Type;
  type My_Float is digits 6 range -10.0 .. 125.0;
  The_Number : My_Float;
  package My_Float_IO is new TEXT_IO.Float_IO (My_Float);
begin
  TEXT_IO.Open(File => Data_File,
               Mode => TEXT_IO.In_File,
               Name => "float.dat",
               Form => "");
  My_Float_IO.Get(File => Data_File,
                  Item => The_Number);
  My_Float_IO.Put(Item => The_Number,
                  Fore => 2,
                  Aft  => 4,
                  Exp  => 0);
  TEXT_IO.Close(File => Data_File);
end Demonstrate_Float_IO;
```

This simple program opens a data file that contains floating point values of type My_Float. It reads a single value using the Get procedure. It then writes this value to the terminal screen and closes the file.

Note the parameters to the Put statement. The first is the Item to be written. This parameter will contain the actual value to be written. Next comes the Fore parameter. This parameter specifies how many digits there are to be printed to the left of the radix point (in base ten this is called the decimal point). If the number of digits to be written to the left of the radix point is less than the number specified for this parameter, then leading blank spaces will be written. If the number of digits to be written is more than the specified value, then all of the digits will be written, overriding the number of printing positions left of the radix point that this parameter specifies. Do not forget that the minus sign in a negative number must be counted as a printing position.

The next parameter is the `Aft` parameter. This specifies the number of digits (printing positions) to be written to the right of the radix point. If the number of digits specified is more than the value contained in `Item` for digits right of the radix point, then trailing zeros will be printed. If this value is less than the number of digits in `Item`, the digits in `Item` will be printed up to the last one specified and this one will be printed with rounding. For example, if we have 32.2394 as the value in `Item` and the value for `Aft` is two, then we would print 32.24 since the second digit is rounded up because the value of the third digit is greater than five. The rules for rounding are if the next digit is less than five, then do nothing to the last digit except print it. If the next digit is greater than five, then round up and print this rounded digit. If the next digit is *exactly* five, then it is implementation dependent as to whether it is rounded up or left alone.

The final parameter is the `Exp` parameter. This specifies the number of digits to be printed in the exponent of the number. If this value is zero, then no exponent position is printed and the number appears in "normal" notation without an exponent, *i.e.*, a sting of digits with an embedded radix point. If the value in `Exp` is anything other than zero, then scientific notation is used to write the number and the number of digits in the exponent field will be as specified in `Exp`. Note than when counting the position for the exponent, the E (or e) is not counted, but the sign position is counted. If the number of digits specified is more than what is needed to write the exponent, then leading zeros are printed. If the number of digits that needs to be printed is greater than what is specified, then the requested field width is overridden and the actual number of digits needed will be used.

Note that there is an additional parameter that can be added to specify the file to which the value in `Item` is to be written. We have not shown any examples with this parameter because it is analogous to what has already been demonstrated for TEXT_IO.

These rules may seem difficult to understand without seeing some examples. The following table lists the `Put` statement and the result that is printed.

Put Statement	Result
`Put(12.3456, 4, 3, 0)`	`ΔΔ12.346`
`Put(12.3456, 4, 1, 0)`	`ΔΔ12.3`
`Put(12.3456, 0, 3, 0)`	`12.346`
`Put(12.3456, 3, 4, 2)`	`ΔΔ1.2346E+1`
`Put(-12.3456E99, 0, 3, 1)`	`-1.235E+100`

`Δ denotes a significant space`

Real Number Output
Table 11.2

11.4.4 Exercises

<< To be added >>

11.5 Operations on Text Files

Chapter 14 of the LRM lists the specification of the package TEXT_IO. In that package there are several useful subprograms that we may use to make our programming jobs easier. In this section we will examine a few of them. For a complete list of the subprograms available, refer to the package specification for TEXT_IO in Chapter 14 of the LRM.

There are subprograms available to interrogate the system about the values of the parameters that were used with the Open or Create statements. There are subprograms to redirect the current output to a file by default, rather than send it to the standard output file which is usually the terminal screen. Similar, we can change the default input file from the keyboard to another file. There are subprograms to set the line and page length. There are subprograms to skip a line in the input or output file. We can even specify a specific column or line in which to start our output. As we have already seen, there are also functions to tell us when we are at the end of a line, or a page, or a file. Finally, as we also have already seen, there are subpackages that allow us to perform input and output on integer, real (float and fixed), and enumeration types.

11.5.1 Reading Strings (Get versus Get_Line)

Many times programs call for the input and output of string values. In Ada, this is somewhat frustrating because of the strong typing rules. There are no dynamic or variable length strings provided in Ada, although it is possible to write your own using the mechanisms in the language. However, using the predefined type STRING we must insure that if we declare the string object to be ten characters long (by constraining it at declaration), then we MUST provide a ten character sting for input. Sometimes, this does not give us the flexibility that we desire.

For example, consider the following object declaration of the predefined type STRING.

```
   The_Name : STRING(1..10);
```

This declaration creates an object capable of holding strings of characters that are exactly ten characters long. Therefore, if we tried to execute the following code segment

```
   TEXT_IO.Get(The_Name);
```

we would be required to provide a ten character string. This seems simple enough. We are using the ability of TEXT_IO to read and write characters and strings to get the value for this string. However, if we typed a name such as "Chuck" followed by a return (also called enter) nothing would happen! The cursor on the screen would continue to flash at us without responding.

The reason for this seemingly bizarre behavior is the strong typing in Ada. In a sense, we promised the runtime that when we provided a value for the sting object The_Name, it would have ten characters. When we provided only five characters (the length of the name Chuck), the runtime system continued to await the other five characters that we told it to expect. Until it has been able to read ten characters it will not go to the next line of the program. This problem can be very annoying for most beginners.

The actual root cause of the problem is lack of knowledge of the procedures provided by TEXT_IO. The Get procedure is provided for those situations where you know exactly the number of characters that a string object will contain. Trying to use it for interactive input/output where these guarantees do not occur is using the wrong tool. The procedure that is probably desired is Get_Line.

The Get_Line procedure actually has two parameters. The first is the Item that will contain the string that we are reading, and the second is the count of the number of characters actually read. Optionally, there can be a third parameter specifying the file from which we want to read the string as discussed previously.

Consider the following statement that will read the string object The_Name as we tried to do previously.

```
TEXT_IO.Get_Line(Item => The_Name,
                 Last => Length);
```

This statement will read from the input file all of the characters provided into the string object The_Name. There are three possibilities when reading these characters. Either there will be less characters than the number of character positions in the string object; there will be exactly the same number of characters; or there will be more characters input than there are character positions in the string object. Let us examine the behavior of this statement in each of these cases.

In the first case, where there are less characters input than there are character positions in the sting object, the Get_Line procedure will read the characters input into the string from left to right (lowest index position to highest) until all of the characters have been read. The remaining character positions in the string object will contain whatever values they held previously, usually garbage. The parameter Last will then contain a count of the number of characters read by Get_Line. For example,

```
  Length : Natural;
  The_Name : String(1.. 80);
begin
  TEXT_IO.Get_Line(Item => The_Name,
                   Last => Length);
```

This code segment shows that the parameter Last is declared to be a Natural object. Suppose that the user entered the string "Luwana" followed by a return in response to this Get_Line procedure. The length of the string object is 80 characters (see the declaration). However, the string entered is only six characters long. Therefore, in positions 1 through six of the string object The_Name would be the characters Luwana. The remaining character positions for the string object, namely 7 through 80 inclusive, would contain garbage. The parameter Length would contain the value 6, the number of characters actually read. Suppose that we then wanted to write this same string out to the terminal screen. If we simply called the Put procedure and passed it the string The_Name, then all of the string would be written out, *i.e..*, the meaningful first six characters as well as the remaining 74 garbage characters. The problem is, of course, the undesirability of writing the garbage. Is there a way to only write the meaningful characters? Of course we can, using the concept of string slices. Thus, to write out the meaningful portion of The_Name we would call the Put procedure as follows:

```
TEXT_IO.Put(Item => The_Name(1..Length));
```

Note that we have actually passed only a slice of the string The_Name. Using the Length value obtained from the call to Get_Line, we pass only the slice of The_Name that is meaningful. This technique is very powerful and is used often in situations where the input string may be of varying length, subject only to the maximum of 80 characters as defined in the constraint on The_Name.

In the second case, where the number of characters read from the current input is exactly the same as the declared length (number of character positions) of the string object, the Get_Line procedure will read the characters into the string object filling all of the character positions from lowest index position to highest. It will place the number of characters actually read into the object Last, all as was done in the first case. However, in this case, the Skip_Line procedure will not be called and the input cursor will remain on the same line as the original text. This means that if you want to get the next data from the next input line, then you must manually and explicitly call the Skip_Line procedure.

In the third case, where the number of character positions in the string object is less than the number of characters on a single line of the current input, Get_Line will read the input characters in order into the string object from lowest to highest index. It will place in the object Last the number of characters read which will be the same as the length of the string object. Finally, it will not call Skip_Line, but will instead leave the input cursor on the same line as the one from which it last read, in preparation to read more of the data.

The problem with this treatment of the Get_Line procedure is that the name of the procedure implies that you will read all of the data on a line and then go to the next line. Unfortunately, this is only true in the first case described here. In the other two cases, the input cursor will be left on the same line as the original text and an attempt to get another character, or even another call to Get_Line, could produce unexpected results. There are at least two easy solutions to this potential problem. The first is to always declare the length of the string object to be the size of the input line length plus one (which always works for terminal input, but may not for file input). This guarantees that there will always be more character positions in the string object then there can possibly be in the input line, forcing the situation to always be as described in the first case above.

The second solution is to check the value of the object Last after the call to Get_Line. If the current value of Last is the same as the declared length of the string object, then all of its character positions have been filled, but, as we saw from the second and third cases above, the Skip_Line procedure was not called. Thus, we could simply explicitly call the Skip_Line procedure if the value in Last is the same as the declared length of the string object, and not do so otherwise.

There are also corresponding procedures for writing string and character values. The Put procedure writes out the value of Item. The Put_Line procedure writes out the value of Item and sends a carriage return, *i.e.*, it performs a New_Line. Thus, the only difference in the two statements is the final position of the output cursor, or the place where the next character will be written. Therefore, two consecutive Put statements would write the output on the same line of the output file, whereas two Put_Line statements would cause Item for each call to the procedure to be written on two lines.

For example, the following sequence of statements,

```
   TEXT_IO.Put (Item => "Hi ");
   TEXT_IO.Put (Item => "there. ");
```

would cause the output file to contain a single line of characters, namely "Hi there. " If, on the other hand, we had the following sequence of statements,

```
   TEXT_IO.Put_Line (Item => "Hi ");
   TEXT_IO.Put_Line (Item => "there. ");
```

then we would use two lines in the output file, the first line containing the single word "Hi " and the second containing the single word "there. " Thus, these two output alternatives give you some control of the layout in your output.

Another handy layout control procedure is New_Line. This procedure in TEXT_IO causes the output file to place the next character received on a new line in the output file. This procedure is much like sending a carriage return and line feed to the output file, although the actual mechanism used to cause the output to start appearing on the next line is not specified by the language. There is a corresponding procedure called Skip_Line that causes the input file to skip over any other data that may remain on the current input line and reposition itself immediately after the next New_Line in the input file. In effect, this causes the input file to get its next data from the next line of the input. These two procedures are overloaded so as to take a file name or to not require one. Thus,

```
TEXT_IO.New_Line (File => Result_File);
TEXT_IO.New_Line;
```

causes the new line to occur in the file Result_File (the first example) or the current output file, which by default is the terminal screen, in the second example. Additionally, we may include an optional parameter to specify more than one New_Line is to be written. Thus,

```
TEXT_IO.New_Line;
TEXT_IO.New_Line;
TEXT_IO.New_Line;
TEXT_IO.New_Line;
TEXT_IO.New_Line;
```

is identical to writing

```
TEXT_IO.New_Line (Spacing => 5);
```

where the Spacing parameter is optional. By default, the value of Spacing is 1, but we may change it to be any positive value. We have done this in our example, causing 5 New_Line's to be placed in the output file.

Skip_Line, which is the inverse of New_Line, in the sense that it applies to the input file, not the output file, has the same properties, including the optional spacing parameter. The only difference between these two procedures is the file to which they apply, either the input file for Skip_Line or the output file for New_Line.

11.5.2 Dynamic File Interrogation

It is often necessary to obtain information about a file while the program is in execution. In order to be able to do this, TEXT_IO provides a number of subprograms designed to allow for dynamic file interrogation.

The first among these is a function that we already saw before, briefly, in an example. This function allows us to determine if we have reached the end of the line in an input file. It is a boolean function, meaning that it returns either the value TRUE or FALSE of the predefined type BOOLEAN. The function is often used in a while loop to cause reading of characters to stop when the end of the line is reached. It is implementation dependent as to the effect of trying to read the end of line indicator, so we generally try to avoid this. Thus, to read a line from an input file we usually have a loop like the following,

```
Get_A_Line:
while not TEXT_IO.End_Of_Line (File => Data_File)
loop
  TEXT_IO.Get(File => Data_File,
              Item => The_Character);
  -- sequence of additional statements
end loop Get_A_Line;
TEXT_IO.Skip_Line(File => Data_File);
```

The Skip_Line at the end of this code segment is important. It causes the next character to be read to come from after the new line indicator used by this implementation. Without this procedure call, any subsequent Get operation would attempt to read the end of line marking character and the effect of that is implementation dependent.

Usually an input file consists of more than one line. Therefore, we often need to read multiple lines which is easy to do by simply iterating over the code segment given to read one line. However, we have to be careful not to attempt to Get a character when there are no more characters left. Thus, we want to iterate over this code segment getting a line of text at a time until there are no lines left in the file. How can we detect when there are no more lines left in the file? TEXT_IO provides us another BOOLEAN function that detects the end of a file. It is often used in conjunction with the end of line function to prevent attempting to read beyond the end of a file. Thus, the previous code segment usually appears as follows,

```
Get_The_File:
while not TEXT_IO.End_Of_File (File => Data_File)
  Get_A_Line:
  while not TEXT_IO.End_Of_Line (File => Data_File)
  loop
    TEXT_IO.Get(File => Data_File,
                Item => The_Character);
    -- sequence of additional statements
  end loop Get_A_Line;
  TEXT_IO.Skip_Line(File => Data_File);
end loop Get_The_File;
```

These functions are not as critical when reading numeric data since the language defines that the Get will skip over any blank characters or end of line markers to read a numeric value. However, for characters and strings, this is not the case. We must check for the end of line marker and manually skip over it as demonstrated above.

There are also functions that can determine information about a given file. The functions TEXT_IO.Mode, TEXT_IO.Name, and TEXT_IO.Form return as their values the value provided when the file was created or opened. Recall that these are the parameters that need to be specified in the Create and Open procedures. The functions listed here merely provide a means to check that the file was opened or created using the proper values. There is an additional function called Is_Open that returns a BOOLEAN value as to whether or not the file is open.

There are many other procedures that can be called from TEXT_IO that provide a means for gaining more information about files that are open or for changing the default characteristics of files. We will not discuss them in this textbook, but you can obtain more information on these subprograms by reading the specification for TEXT_IO in Chapter 14.

11.5.3 Exceptions in Text Files

We have so far discussed only the situations where everything works as it is supposed to work. Unfortunately for us, this is not always the case. Consequently, there are a number of potential exceptions that are associated with input and output of text files. Collectively, they are contained in a package that has only a specification. This package is called IO_Exceptions and is reproduced in its entirety here.

```
package IO_Exceptions

   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;

   end IO_Exceptions;
```

These exceptions represent all of the predefined exceptions associated with input and output. They are declared in their own package so that they can be renamed inside all of the various kinds of I/O packages, both textual and non-textual I/O. This allows there to be a single exception for, say `Use_Error`, and not multiple possible exceptions that are really the same problem but associated with different I/O packages.

We will examine the meaning of each of the I/O errors in this package in light of how they apply to TEXT_IO. Some of these errors you are likely to see many times in your career; some of these errors you are not likely to ever see. However, it is important to know where to go to find out information about any given exception. These exceptions are completely explained in Chapter 14 of the LRM. We will summarize here its discussion of these errors.

`Status_Error` is the exception that is raised if you attempt to operate upon a file that is not open. For example, if you were to attempt to read from a file that you had not yet opened, then you would raise `Status_Error`. Similarly, you would raise `Status_Error` if you attempted to open a file that was already open. Of course, you could always include a call to the BOOLEAN function `Is_Open` first to insure that you never attempted to open a file that was already open, but, in practice, this is rarely done.

`Mode_Error` is raised if you violate the mode for a given file. For example, if you create a file you will probably specify that the mode is `Out_File` since you will be writing data to the file. If, however, you attempt to read from this file, while it is still in `Out_File` mode then you will raise `Mode_Error`. Also, if you attempt to test for `End_Of_File` for a file that is in `Out_File` mode, then you will raise `Mode_Error` since the end of file function only makes sense when you are reading data from a file. Further, if you attempt to write to a file that you opened for input, then `Mode_Error` will be raised. Lastly, many of the functions provided by TEXT_IO such as `Skip_Line`, `Set_Input`, *etc.*, only make sense in one mode. Attempting to call these subprograms when the file is in the other mode will raise `Mode_Error`, such as calling `Skip_Line` for a file that is opened in `Out_File` mode.

The exception **`Name_Error`** is raised when you call `Create` or `Open` and the parameter that you provide for `Name` is not allowable as a name for a file in the operating system under which you are executing. Thus, specifying a name for a file that has a format unacceptable to your operating system will raise `Name_Error`. Also, if you specify a name for a file in an `Open` statement and that file does not exist, then you will raise `Name_Error`. The details of this exception are provided in more detail in Appendix F of your implementation's LRM.

We can raise **Use_Error** by attempting to perform an operation that is not possible because of the characteristics of the external file. For example, if the Form parameter in Create contains invalid access rights for the file, then Use_Error will be raised. Similarly, we will raise this exception if we attempt to Create a file and specify in the Form parameter a device that is input only. The details of this exception are provided in more detail in Appendix F of your implementation's LRM.

Device_Error is raised by the runtime system automatically if an input-output operation cannot be completed because the underlying operating system has a malfunction. This is normally associated with hardware defects of some type or a broken connection. The details of this exception are provided in more detail in Appendix F of your implementation's LRM.

The **End_Error** exception is raised if your program attempts to read or skip past an end of file marker. For example, trying to read data using a loop after the end of file is reached will cause End_Error to be raised.

Data_Error is raised when the input sequence of characters does not represent a permissible format. For example, trying to read a floating point value when the input contains characters other than digits would raise Data_Error. Also, if you attempt to get a value that is outside of the permissible range of values then you will raise Data_Error. This means that if we have a subtype of INTEGER with a range constraint of 1 .. 100 and attempt to read a value for an object of this type, we will raise Data_Error if the input value is 102, or any value outside of the given range. Likewise, if we have an enumeration type and it lists four values, then an attempt to read a value for an object of this type that was not one of the values defined in the enumeration declaration will raise Data_Error.

Finally, **Layout_Error** can be raised when you attempt to specify a value for the column, line, or page that is beyond the established limits. We did not discuss these subprograms and so it is unlikely that you will raise this exception unless you are experimenting with the other TEXT_IO procedures that we did not discuss.

All of these exceptions provide us with a good indication of the type of problem that was encountered during the execution of our program. They also allow us to take corrective action dynamically, because we can specify a different reaction for each exception. You should always include likely exceptions in the exception handlers for all of your programs. This means that whenever I/O is a part of a program unit, the exceptions for potential problem areas in input-output must be taken into account.

11.5.4 Exercises

<< To be added >>

Syntax Chapter 11

```
basic_declaration ::= type_declaration
                      | deferred_constant_declaration

basic_declarative_item ::= basic_declaration

component_declaration ::= identifier_list :
                        component_subtype_definition [:= expression];

component_list ::= component_declaration { component_declaration}
                   | null

component_subtype_definition ::= subtype_indication

constraint ::= floating_point_constraint
           ::= fixed_point_constraint
           ::= range_constraint | floating_point_constraint
               fixed_point_constraint | index_constraint

deferred_constant_declaration ::= identifier_list :
                                  constant type_mark;

discriminant_part ::= (discriminant_specification
                      {; discriminant_specification})

discriminant_specification ::= identifier_list : type_mark
                               [:= expression]

full_type_declaration ::= type identifier is type_definition;
                      ::= type identifier [discriminant_part] is
                          type_definition;

fixed_point_constraint ::= fixed_accuracy_definition
                           [range_constraint]

fixed_accuracy_definition ::= delta static_simple_expression

floating_point_constraint ::= floating_accuracy_definition
                              [range_constraint]

floating_accuracy_definition ::= digits static_simple_expression

package_specification ::= package identifier is
                          {basic_declarative_item}
                          [private
                          {basic_declarative_item}
                          end [package_simple_name]

private_type_declaration ::= type identifier
                             [discriminant_part] is
                             [limited] private;

real_type_definition ::= floating_point_constraint
                     ::= fixed_point_constraint
```

```
record_type_definition ::= record
                              component_list
                           end record

subtype_declaration ::= subtype identifier is subtype_indication;

subtype_indication ::= type_mark [constraint]

type_declaration ::= private_type_declaration

type_definition ::= real_type_definition
                ::= record_type_definition

type_mark ::= type_name | subtype_name
```

Chapter 12

Software Design for Interactive Use

Our next step is to extend issues of program design and testing to interactive computer use. Strictly speaking, we have been in interactive computer use from the very start with Ada. The framework of a computer starting with a message requesting input from the user, the user responding with the input, then the computer responding with an answer to that input with output and terminating is interactive of a simple form. But such a framework can be easily extended by repeating the user input, computer output iteration many times, not simply once. Your experience with word processors demonstrates such a framework. You may think of the user's work in large chunks, but the computer treats the work in terms of single inputs, say defined by "enter" keys. In a single session, you may enter hundreds or thousands of inputs, getting an output back each time. Some of the outputs will be on the screen, some on printed paper. So how you use computers right now is a good model of interactive software.

Theoretically, one can regard each single response as generated by an independent program part, all such program parts working from a common data base. Aside from using a common database, the program parts are quite independent. When the computer is to give an output, the current program part will lose control, and the next program part, if any, to respond to the user input can be identified. However, such a set of independent program parts can become very large in quantity and repetition. So the common reuse of various subparts of such program parts is a practical necessity. But the reuse of subparts must work exactly as a set of independent programs would. That reuse isn't impossible, but there are enough opportunities for errors in putting such subparts together that real discipline is called for.

In this Chapter 12, we first review the new ideas in Ada introduced in Chapter 11. Then, we introduce how to move into interactive use of software in a logical and disciplined way. It begins with specifications of interactive software, then designs, verifications, and certifications to meet those specifications. As noted above, the first concern is in correctness of designs to their specifications, then demonstrating that correctness by the joint verification and certification of the software.

12.1 Ada Capabilities from Chapter 11

In this section, we examine the new Ada capabilities from Chapter 11 in real number types, parameterized record types, combinations of records and arrays to create more complex data structures, the private type for information hiding, and TEXT_IO files. Each of these extensions to Ada give more power for design and permit simpler solutions to data processing problems.

12.1.1 REAL Number Types

As noted above, REAL number types allow new capabilities in FLOAT and FIXED point types and subtypes. Recall that REAL numbers are always shown with a decimal point, whether a floating point number or a fixed point number. Also recall that while REAL numbers always contain a decimal point, whether FLOAT or FIXED, INTEGER numbers never contain a decimal point. All three number types must be kept separate and straight for their use.

INTEGER arithmetic among numbers will be exactly right unless overflow or underflow occurs. FLOAT and FIXED arithmetic among numbers will face round off errors in any case as well as the possibilities of overflow or underflow. In every case, there is a finite set of sorted values in each type moving from FIRST to LAST. But in each case arithmetic operations between members of a type can well define a real number outside the type and the program will terminate abnormally. For addition, about half of randomly chosen members will define a number outside the type. But for multiplication, most randomly chosen members will define a number outside the type.

There is one additional question for FLOAT and FIXED numbers in the accuracy of results. FLOAT and FIXED are approximate in decimal numbers, but their internal representation and arithmetic operations are actually in binary numbers. We will not go any further into this issue here, but treatment of roundoff in arithmetic operations needs to recognize it.

12.1.1.1 Floating Point Behavior

As noted, the floating point behavior will be approximate, first depending on the hardware numbers precision, and second on declared numbers of precision no more than the hardware precision. Notice also that the successive differences between adjacent floating point numbers grow larger as their absolute values become larger. The relative differences between adjacent fixed point numbers are approximately equal as their absolute values become larger.

As a result, verification of floating point behavior must deal with approximation, unlike verification of INTEGER behavior which is exact. That is, the result of floating point arithmetic will approximate real floating point arithmetic which is exact but cannot be carried out in finite computers. In a given computer the result will always be the same, but the results can be different between different computers because of different word sizes. As noted, there will be operations in real floating point arithmetic, with no approximation. The departure of arithmetic in a given approximate floating point arithmetic from real floating point arithmetic provides a basis for analysis and possibly redesign to better meet objectives.

In illustration, consider floating point declarations

```
   type Length_Measures is digits 6 range 0.0 .. 100.0;
   Base, Altitude, Perimeter : Length_Measures;
```

and statement

```
   Perimeter := 2.0 * (Base + Altitude);
```

Now, what accuracy can be expected for the value of `Perimeter` computed above?

First, the accuracy of `Base` and `Altitude` from initial input will be relative to 6 digits. Now for `Perimeter` determine the accuracy of (`Base` + `Altitude`). The true value of the sum can be anywhere in the set

```
   (Base * (1 +/- 10**-6)) + (Altitude * (1 +/- 10**-6))
```

So the accuracy error will be no greater than the maximum of `Base` and `Altitude` less 1 digit. And the accuracy of `2.0 * (Base + Altitude)` will continue to be 5 digits.

12.1.1.2 Fixed Point Behavior

As noted, the fixed point behavior will be approximate, first depending on the hardware numbers precision, and second on declared numbers of precision no more than the hardware precision. Notice also that the successive differences between adjacent fixed point numbers are approximately equal as their absolute values become larger. The relative differences between adjacent fixed point numbers grow smaller as their absolute values become larger.

As a result, verification of fixed point behavior must deal with approximation, unlike verification of INTEGER behavior which is exact. That is, the result of fixed point arithmetic will approximate real fixed point arithmetic but cannot be carried out in finite computers. In a given computer the result will always be the same, but the results can be different between different computers because of word sizes. As noted, there will be operations in real fixed point arithmetic, with no approximation. The departure of arithmetic in a given approximate fixed point arithmetic from real fixed point arithmetic provides a basis for analysis and possibly redesign to better meet objectives.

In illustration, consider fixed point declarations

```
   type Length_Measures is delta 0.001 range 0.0 .. 100.0;
   Base, Altitude, Perimeter : Length_Measures;
```

and statements

```
   Perimeter := 2.0 * (Base + Altitude);
```

Now, what accuracy can be expected for the values of `Perimeter` and `Area` computed above?

First, the accuracy of `Base` and `Altitude` from initial input will be fixed to .001. Now for `Perimeter` determine the accuracy of `(Base + Altitude)`. The true value of the sum can be anywhere in the set

```
   Base (+/- .001) + Altitude (+/- .001)
```

So the accuracy error will be no greater than

```
   (+/- .001) + (+/- .001) = (+/- .002).
```

And the accuracy of `2.0 * (Base + Altitude)` will be `(+/- .004)`.

12.1.2 Composite Types of Advanced Designs

In Chapter 11, records with discriminants were introduced to provide more flexibility in software design. Then it was shown how to use records as components of arrays, and how to use arrays as components of records. Of course, arrays can use other arrays and records use other records as well. In this way there is a complete capability to form compound types in hierarchical structures of arrays and records with or without discriminants.

12.1.2.1 Records with Discriminant

The record with discriminant of `Buffer` as follows

```
type Buffer (Size : Positive := 10) is
record
  Item : String (1 .. Size);
end record;
```

in Chapter 11, which defines sample object declarations of

```
My_Buffer   : Buffer (10);
Your_Buffer : Buffer (20);
The_Buffer  : Buffer; -- uses the default value of ten
```

Later, `My_Buffer` can be changed in size from 10 to 20 in an aggregate assignment

```
My_Buffer := (20, "This is new stuff!!!");
```

In this way, designers can define records with discriminants in declarations or modified in statements as shown just above. The structure of such records are an expansion of simple records. The verification of programs with discriminants follows their structure directly.

Discriminants add a dimension to give much more flexibility and power to records. As any programming language, Ada must be designed explicitly, so its capabilities are not what its users hope they are from appearances or hopes, but what they have been defined to be. It takes some time and effort to understand such capabilities, but provides significant new ability to define useful data.

12.1.2.2 Arrays of Records

As noted in Chapter 11, arrays may have records as components. Since arrays must have identical types of objects in every location, records may satisfy that condition of the record components. These records may be defined with discriminants. As noted above, declarations can be used as below

```
type Months is (JAN, FEB, MAR, APR, MAY, JUN,
                JUL, AUG, SEP, OCT, NOV, DEC);
type Days is (SUN, MON, TUE, WED, THU, FRI, SAT);
type Years is range 1901 .. 2010;
type Periods is range 1 .. 31;
type Dates is
record
  The_Day      : Days;
  The_Month    : Months;
  The_Period   : Periods;
  The_Year     : Years;
end record;
type Agendas is array (Periods range <>) of Dates;
```

with declares for months as follows

```
January   : Agendas (31);
February  : Agendas (29);
March     : Agendas (31);
April     : Agendas (30);
```

and so on. However, a variant record could reference both `The_Year` and `The_Month` as follows.

```
type Dates (The_Year : Years, The_Month : Months)
is
record
  The_Days  : Days;
  case The_Month is
    when JAN | MAR | MAY | JUL | AUG | OCT | DEC =>
      Long_Month : Periods;
    when APR | JUN | SEP | NOV =>
      Normal_Month : Periods range 1 .. 30;
    when FEB =>
      if The_Year rem 4 = 0
      then
        Short_Month : Periods range 1 .. 29;
      else
        Shortest_Month : Periods range 1 .. 28;
      end if;
  end case;
end record;
```

12.1.2.3 Records with Arrays

In the reverse direction, records can hold arrays. As described in Chapter 11, a set of university grades can be a record with arrays in them. As shown,

```
subtype Grades is Natural range 0 .. 100;
type Student_Number is range 0 .. 99;
type Exam_Number is range 1 .. 10;
type Examination_Grades is
  array (Student_Number, Exam_Number) of Grades;
CS2003 : Examination_Grades;
```

This general use of records in arrays and arrays in records in this and the previous section completes the Ada capability in organizing data. Records and arrays have quite different properties, of course, so data structures that organize deep hierarchy of access into data parts permit powerful designs to meet natural issues in applications. For example, in type `Examination_Grades` above, type `Student_Number` might be generalized from the range 0 .. 99 to a record `Student_Record` that contains `Student_Number` as one part.

12.1.3 Private Types

As noted in Chapter 11, private types provide the capability for information hiding about details of operations on data in package design. If users know detailed design about data and operations in the subprograms of a package, or in other packages or subprograms called, they may make assumptions on operations that may not be good. As also noted in Chapter 11, going back to linked list design, it is not needed to know whether data is stored and addressed in arrays or in access types (or some other way) in order to use it. So private types provide a basis for separating implementation from specification of low level packages.

12.1.3.1 Limited Private Types

The set of package designs for `Ice_Cream_Shop` in Chapter 11 illustrates how the customers can be defined as values for object `My_Number` of type `Numbers`. They also show how customers could cheat with moving `My_Number` down if not served. So the package designer is motivated to make `Numbers` private to prevent cheating with `My_Number`.

However, making `Numbers` private in the `Ice_Cream_Shop` still allows more clever customers to cheat, so the designer provides a stronger control in limited private, to really prevent cheating.

Software design must take into account user needs, but also user behavior to get around the software for one reason or another. As illustrated above, if users can reach unintended software benefits by changing software parts without permission, active participation in software systems can be difficult to control. Limited private types address this issue.

There is another need for strong control of system information in software design. The computer always understands exactly what is to be done next from the software provided it. But in large software systems the designers don't always understand or remember the totality of system behavior. So private and limited private types allow the technical management of the design to maintain intellectual control part by part.

12.1.4 Text Files in Ada

As noted, we have only used a small part of the package TEXT_IO of Ada in this book, namely the treatment of the Input and Output files that are provided by shorthand notation. Additional files for input and/or output are available with more information about their properties. Such files have two identifiers, one internal in Ada and one external outside Ada. Other than additional references to their outside properties, the operations on these files use the same subprograms, procedures, and functions.

In TEXT_IO, all text files are of limited private type called FILE_TYPE. So this is another general use of limited private for better control of software parts. In Chapter 11, The_File was created with

```
TEXT_IO.Create (File =>  The_File,
               Mode => TEXT_IO.Out_File,
               Name => "demo.txt",
               Form => "");
```

so demo.txt is the external name of the internal file The_File.

12.1.5 Binary Files in Ada

Binary file input and output is the most efficient way to use files in Ada. But more design effort is required by the user than for TEXT_IO. The description is in the Ada Reference Manual, Chapter 14. We do no more than introduce them here. For example, whereas text files take some 20 pages to describe, binary files take 7 pages to describe both sequential and direct forms. In place of the fifty subprograms used to access and modify existing TEXT_IO files, sequential files are accessed and modified by just three subprograms, namely

```
procedure READ   (FILE : in FILE_TYPE; ITEM : out
ELEMENT_TYPE);

procedure WRITE  (FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
```

and direct files are accessed and modified by eight subprograms, namely

```
procedure READ  (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE;
                 FROM : POSITIVE_COUNT);
procedure READ  (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE;

procedure WRITE (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE;
                 TO : POSITIVE_COUNT);
procedure WRITE (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE;

procedure SET_INDEX (FILE : in FILE_TYPE; TO : in
POSITIVE_COUNT);

function  INDEX (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function  SIZE  (FILE : in FILE_TYPE) return COUNT;

function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
```

Note, among other changes from TEXT_IO that Get and Put procedures are changed to READ and WRITE procedures.

12.1.6 Exercises

1. For the floating point declarations in 12.1.1.1, consider the statement

   ```
   if Base > Altitude
   then
     Maximum := Base;
   else
     Maximum := Altitude;
   end if;
   ```

   and determine what accuracy can be expected for the value of `Maximum`.

2. For the fixed point declarations in 12.1.1.2, consider the statements

   ```
   Base := (Perimeter / Altitude) / 2.0;
   Base := Area / Altitude;
   ```

   and determine what accuracy can be expected for the values of `Base`.

3. Provide an example of a possible array of records in student data.

4. Provide an example of a possible record with arrays in calendar data.

5. What is the reason to define text files of limited private type? What is being protected in this case?

12.2 Interactive Software Segments

Interactive software will make a particular point of using and exploiting Ada capabilities. Early computers and their software were strictly batch operating, from possibly very complex stimuli prepared beforehand to responses. With more modern computers and better input, output facilities, the idea of interactive software became possible. Ada recognizes that interactive need and provides for it from the outset.

12.2.1 Interactive Software Is Real Time

Any computer program operates in real time in any computer. There is no other time to operate in. Many programs operate in the same way in any computer used, so operating in real time is not a problem. However, many important programs need to operate in real time in specific ways to meet external requirements in both input and output. For example, an item of input may be available only for a limited duration in real time, so reading it too early or too late will give incorrect data. This real time behavior will depend on both the software design and the hardware speed. The hardware behavior must be known, and possibly different hardware used. The software design must not only be right as a sequential or parallel algorithm, but be right in time performance as well. On the other side, an item of output may be needed in a limited duration in real time, so sending early or late gives irrelevant data to the user, either human or machine. Again, both hardware and software must meet the need at hand.

Interactive software is certainly real time, receiving input from one or more users (possibly thousands concurrently) over and over during execution. For any one user, as already seen, control alternates between user and computer. When execution halts and waits for inputs, the users are in control. Typically, users view the interaction on a macro basis to complete steps dealing with the application. In this case, each macro step may require several micro steps of input entry. Occasionally, users will not provide inputs as expected, or computer or communication breakdowns prevent inputs. Interactive systems must handle such breakdowns as well.

So interactive software poses no new problems beyond timely relations of input and output with users. Program micro steps will begin at each `TEXT_IO.Get` statement and continue to or beyond the next `TEXT_IO.Get` statement that is reached, which may be one of several that is possible. The entire software is mapped into a set of partial programs, one for each `TEXT_IO.Get` statement terminating in a set of next `TEXT_IO.Get` statements or program termination. These partial programs, of course, assemble into the complete program and may share program subparts, but it is useful to disentangle them into separate partial programs for analysis and verification of real time behavior, even though common subparts are used. Note that `TEXT_IO.Put` statements play no role in this decomposition of execution, only `TEXT_IO.Get` statements. `TEXT_IO.Put` statements provide information for users, but program execution is not affected by them otherwise.

12.2.2 Interactive Software Segments Revisited

In illustration of interactive software segments, suppose a loop contains several `TEXT_IO.Get` statements. Then each of these `Get` statements defines a partial program. Starting from any one of these `Get` statements, the execution may reach any of these as the next `Get` statement or leave the loop and reach outside `Get` statements or program termination. Which `Get` statement or termination is reached may depend on data, so for each `Get` statement there is a specific control chart leading to the internal `Get` statements, to external `Get` statements or terminating execution. The control chart may involve several loops in reaching the next internal `Get` statement. These control charts are independent partial programs even though common parts of the loop are used. Looking backwards from the loop, there is likely a previous `Get` statement, possibly several, that gets into the loop and its `Get` statements as well as reaching other `Get` statements outside the loop.

More concretely, consider procedure `Create_List`, introduced as part of package `List_Processing` in Chapter 9. It has a single `Get` statement in a while loop, but it also calls procedure `Insert_In_List` within this loop, as well, which contains another `Get` statement. Relevant parts of these procedures are given next.

```
with TEXT_IO;
procedure Create_List
is
  More_Names : CHARACTER := 'y';
begin
  ...
  Get_Names:
```

```
 while (More_Names = 'Y') or (More_Names = 'y')
 loop
   Insert_In_List;
   TEXT_IO.Put (Item => "More names - y or n? ");
   TEXT_IO.Get (Item => More_Names);
   TEXT_IO.New_Line;
 end loop Get_Names;
end Create_List;

procedure Insert_In_List
is
  New_Name : Names;
  Count : Index_Type := 1;
begin
  TEXT_IO.Put (Item => "Enter value to be inserted => ");
  TEXT_IO.Get (Item => New_Name);
  TEXT_IO.New_Line;
  ...
end Insert_In_List;
```

Note that `TEXT_IO.Put` statements play a key role in alerting users to enter `TEXT_IO.Get` statements, but play no part in decomposing the execution between user and machine.

Now there are three executable program parts with these initial statements

```
P1: procedure Create_List

P2: TEXT_IO.Get (Item => More_Names);

P3: TEXT_IO.Get (Item => New_Name);
```

and with the following connections,

```
P1 to P3

P2 to P3 or Exit

P3 to P2
```

This is a straightforward set of program parts by careful design. But other designs may not have such straight forward program parts. In any case such a set of program parts defines a collection of program paths, starting with `P1` and ending with `Exit`. Any given program execution will follow one of these program paths, based on input data.

As noted, interactive software segments in Ada execute between user inputs. Each segment starts execution with a user input and continues until another user input is required. In the meantime it calculates, creates outputs in both visible screens and print forms, and possibly other forms. Once the output is created it awaits the next input.

Now, to examine the behavior of software segments in more detail, we put down the relevant code of `Create_List` with the program parts identified therein.

```
P1: with TEXT_IO;
    procedure Create_List
    is
      More_Names :CHARACTER := 'y';
    begin
      ...
      Get_Names:
      while (More_Names = 'Y') or (More_Names = 'y')
      loop
        Insert_In_List;
        -- procedure Insert_In_List
        -- is
        --   New_Name : Names;
        --   Count : Index_Type := 1;
        -- begin
        --   TEXT_IO.Put (Item =>
        --     "Enter value to be inserted => ");
P3:    --   TEXT_IO.Get (Item => New_Name);
        --   TEXT_IO.New_Line;
        --   ...
        -- end Insert_In_List;
        TEXT_IO.Put (Item => "More names - y or n? ");
P2:    TEXT_IO.Get (Item => More_Names);
        TEXT_IO.New_Line;
      end loop Get_Names;
    end Create_List;
Exit:
```

The program parts noted before are fully visible here, going from P1 to P3, then from P3 to P2, and finally from P2 to either P3 or Exit. The **while loop** from P3 to P2 can continue indefinitely, and Exit is the only way out of the loop. In this simple example we have inserted the text of Insert_In_List as Ada commentary just under the call of the procedure. In more complex examples such text may have to be treated separately and the reader move from one piece of text to another in a systematic way. But identifying all Get statements, along with the entry and exits brings the entire execution behavior into view.

12.2.3 Performance Requirements for Interactive Software

Since humans are frequent users of interactive software, it not only should be correct functionally, it should be timely as well. In illustration, sort programs can be of radically different performance in time for given data. If they are being used in interative software for human use, it will be important to complete their actions in a few seconds at most. This means the sort algorithm and the sort problems must be compatible for the interactive requirements.

As a practical experience, the first assembler programs for the IBM 360 computers were designed very nicely from a theoretical viewpoint. But they ran so slowly they had to be redesigned in much more complex form to run faster–not a little faster, but a hundred times faster. Intuition about performance can be very misleading. Computers do small steps so fast, it is hard to imagine just how all those small steps accumulate into a total job. One example comes in computing trigonometric functions, such as sine, cosine, tangent. People look up trigonometric values in tables, and such tables could be stored in computers, as well. But it turns out that it is much more efficient for computers to calculate such trigonometric values from scratch, every time needed, rather than keeping them all stored in tables.

So realistic performance requirements, and how to achieve them with good design, need deep analyses, in parallel with function design.

12.2.4 Correctness Requirements for Interactive Software

Correctness of interactive software is especially important. Failures during interaction can be at different levels. One level is to halt the interaction altogether, terminating execution. Another is to alter the responses entirely, making the interaction beyond user understanding. Still another is to return incorrect data, which may be possibly important data. And still another is to return small errors in syntax.

In this first human generation of creating computer software, intuitive methods of program development have created interactive software systems with many failures. Well used systems, such as word processors, have removed most critical failures, but many failures still remain to work around. The idea of completely correct interactive software is still imagined as an impossible dream by many involved in building them. To go to absolutely correct software will require much work in large systems, but is not impossible.

As noted above, Ada programs execute in real time, because there is no other time to execute in. Since each step, in declarations or statements, takes some time, some steps can execute incorrectly in real time in certain circumstances. While those circumstances are rare, systems with exceptional performance requirements may require analyses at the performance level as well as the function level to meet their specifications.

12.2.5 Exercises

1. Partial programs begin with the start of a program or with each `TEXT_IO.Get` of the program, with all subprogram calls replaced by the subprograms themselves. Partial programs stop at each `TEXT_IO.Get` reached or at the end of the program. That is, partial programs have unique entries, but possibly non-unique exits. Under what conditions are exits unique?

2. What are the partial programs for `Insert_In_List`?

3. What are the partial programs for `Delete_From_List`?

4. What are the partial programs for `Show_List`?

5. Humans can do all the elementary steps done by computers, only much slower. Estimate relative speeds for compare, addition, multiplication of 10 place numbers.

6. Failures in interactive software are not self descriptive. What methods might be used in the software itself to warn users of possible failures?

12.3 Singly Linked Lists Again

In Chapter 9 you were introduced to three methods of dynamic implementation of lists - singly linked lists, circular linked lists, and doubly linked lists. We will examine singly linked lists in more depth for defining their specification, design, and verification.

12.3.1 Black Box Specification of Singly Linked List

First, we reexamine the specification of singly linked lists as black boxes. The stimuli and responses for the black box are of four types:

`Create_List`: Start a new stimulus history for the list; this will not erase any previous history for the black box, but it makes such history irrelevant for the future;

`Insert_In_List`: Add any value to the list history; duplicate values may be added;

`Delete_From_List`: Add any value negation to the list history; duplicate value negations may be added;

`Show_List`: Sort all values added but not later negated since the last `Create_List` and print;

As a black box, only the history of stimuli are known. The specification is quite straightforward. No data storage is permitted. Singly linked lists are not present in this black box specification. They will show up in going to the state box where internal data is introduced. `Show_List` is the most substantial black box operation, examining the history from the last `Create_List` forward, removing any value and value negation pair where the value in `Insert_In_List` appears subsequently as a matching value negation in `Delete_From_List`, removing all remaining value negations, then sorting and printing the remaining values. While straightforward, it is not very efficient.

12.3.2 State Box Design and Verification of Singly Linked List

Next, the state box will require data storage design in place of stimuli history. It will also require a new subspecification of the process needed to deal with this internal state. Various alternatives are possible, including the two introduced in Chapter 9, using both arrays and access types. In this case we will pursue the use of access types, having verified the array solution in Chapter 10.

The first design decision in moving from the black box to the state box is the state design. In the black box, data appears in historical form, first in `Create_List` which identifies the start of the next relevant stimulus history, then `Insert_In_List` and `Delete_From_List` steps that build and modify the history, and finally in `Show_List` which produces sorted printouts of what is currently active in the history since the last `Create_List`. In the state box it seems sensible to store what is currently active in the relevant history in already sorted order. This will add more complexity to `Insert_In_List` and `Delete_From_List`, and make `Show_List` simpler. We will also introduce names for both the list and the elements in the list, `The_List` and `The_Element`.

As you recall from Chapter 9, each node in a singly linked list was designed to contain a field that provides the access value of the next node in the list. The last node in the list has the value `null` in the field giving access to the next node. Also recall that such a list commonly has a special access variable, the external pointer, that points to the start of the list. As we discuss insertion and deletion in a linked list, we will treat this external pointer as a special case. First, let us define the Ada types that we use to support our singly linked list in the state box.

```
subtype Element_Type is STRING (1 .. 5);
type Node;
type Node_Access is access Node;
type Node is
  record
    Data : Element_Type;
    Link : Node_Access;
  end record;
```

These types define the state of the state box being designed here. Now let us consider each of our desired operations in the state box, in contrast to the black box. We have decided to retain the active elements of a singly linked list in a sorted list. That is any successor element will be equal or greater in size to the current element. Specifically, what do we want each operation to do? As noted in Chapter 9:

Create_List should initialize a list to empty.

Insert_In_List should add a given element to a given list at the appropriate place. This means that both the name of the list into which we are to insert and the value of the element to be inserted should be provided by the calling routine. Thus this operation will have two parameters - the list name and the element value. The place in the list at which the element is to be inserted will be determined by the choice of ordering that we impose on the list. For our example, we keep our list in increasing order.

Delete_From_List should delete a given element (if present) from a given list. Therefore this operation will also have two parameters -the list name and the element value. If the element is present it should be removed from the list without disturbing the remaining elements. If the requested element is not in the list we display a message to that effect.

Show_List will display every node in the list in order.

As found in Chapter 9, let us summarize our set of state box operations in the following chart.

```
Create_List (The_List):
  Purpose: Initialize The_List to null
  Input:  The_List
  Output: The_List with value null

Insert_In_List (The_List, The_Element):
  Purpose: Adds The_Element to The_List
  Input:  The_List and The_Element
  Output: The_List with The_Element inserted

Delete_From_List (The_List, The_Element):
  Purpose: Deletes The_Element from The_List if present
  Input:  The_List and The_Element
  Output: The_List with The_Element removed or
          The_List and an output message

Show_List (The_List):
  Purpose: Prints all the elements in The_List
  Input:  The_List
  Output: The elements of The_List in order
```

The state box type definitions and operations described above are all logically related. That is, they all deal with our linked list. Therefore let us physically relate them by defining them in a package. The package specification is given below.

```
package Singly_Linked_List
is
  subtype Element_Type is STRING (1 .. 5);
  type Node;
  type Node_Access is access Node;
  type Node is
    record
      Data : Element_Type;
      Link : Node_Access;
    end record;
  List_Underflow : exception;
  List_Overflow : exception;
  procedure Create_List (The_List : in out Node_Access);
  procedure Insert_In_List (The_List : in out Node_Access;
                            The_Element : in Element_Type);
  procedure Delete_From_List (The_List : in out Node_Access;
                              The_Element : in Element_Type);
  procedure Show_List (The_List : in Node_Access);
end Singly_Linked_List;
```

You will see that we have encapsulated our types and operations in the package `Singly_Linked_List`. Additionally we have defined two user defined exception names, `List_Underflow` and `List_Overflow`. `List_Underflow` is provided to alert a using program of an attempt to remove an element from an empty list. `List_Overflow` is provided to alert a using program that the available storage space has been exceeded.

Now that we have our package specification, we can compile it and enter the compilation information into our Ada library.

At this point we have seen a black box and a state box definition for the Singly Linked List. The black box defines a specification for Singly Linked List and the state box defines a design to meet that specification. It needs to be seen if they are compatible, that is to verify that the state box does meet the requirements of the black box. The state box has defined state and operations to meet the black box requirements. Both state box and black box identify four operations with identical names. The state box operations have formal arguments, and The_Element. is, indeed, an internal state with no correspondence in the black box. In this design is stored by the user, not by the state box. But is a stimulus common to both black box and state box. It is called "value" in the black box, and might be called there as well.

So a verification that the state box is equivalent with the black box can be done at this point. It can be done informally as follows. Consider any history of stimuli to the black box and clear box.

`Create_List`: In the black box, each `Create_List` will start a new relevant history, with all previous stimuli irrelevant from then on. In the state box, each `Create_List (The_List)` will initialize `The_List` to null. The effect is equivalent.

Insert_In_List: In the black box, each Insert_In_List will continue the stimuli history, recognizing the value given. In the state box, each Insert_In_List (The_List, The_Element) will enter The_Element into The_List in sorted order. In each case the value of The_Element is retained.

Delete_From_List: In the black box, each Delete_From_List will continue the stimuli history, recognizing the value given negated. In the state box, each Delete_From_List (The_List, The_Element) will seek to remove The_Element from The_List if possible. In each case the value of The_Element is recognized for negation.

Show_List: In the black box, with Show_List the history to the preceding Create_List is examined and values followed by a negated identical value removed, then the remainder of values sorted and printed. In the state box, Show_List (The_List) prints The_List.

Thus, although the data and internal operations are quite different, the final result in the Show_List operation is identical, and that is the only operation visible to the users.

12.3.3 Clear Box Design and Verification of Singly Linked List

We now need to construct the package body thereby providing the implementation of each of our procedures whose specifications were given above. Let us begin our implementation analysis by developing suitable algorithms for our procedures. As carried out in Chapter 9, these algorithms were defined and analyzed informally before carrying them over to Ada. They are repeated here with small elaborations following their specifications for continued analysis and verification. The specification from the state box is listed and compared with the design from the clear box.

Specification

```
Create_List (The_List):
  Purpose: Initialize The_List to null
  Input:   The_List
  Output: The_List with value null
```

Design

```
Create_List (The_List):
  set The_List to null
```

Verification

The verification is obvious.

Specification

```
Insert_In_List (The_List, The_Element):
  Purpose: Adds The_Element to The_List
  Input:   The_List and The_Element
  Output: The_List with The_Element inserted
```

Design

```
Insert_In_List (The_List, The_Element):
  if The_List is empty
  then
    make a node containing The_Element
    set The_List to point to the new node
  else
    if The_Element <= the value in the first list node
    then
      make a node containing The_Element
      change the pointers to make the new node the
        first node in the list
    else
      let Current_Ptr point to the first element
      let Previous_Ptr be null
      while Current_Ptr.Data < The_Element and
        Current_Ptr.Link /= null
      loop
        Previous_Ptr is set equal to Current_Ptr
        advance Current_Ptr
      end loop
    make a node containing The_Element
    change the pointers to insert the new node
    end if
  end if
```

Verification

As discussed in Chapter 9 , the insertion algorithm covers four cases–an empty list, insertion at the first node, insertion later in the list, and insertion at the end of the list. Before and after diagrams are shown in Chapter 9. The four cases are determined by a nested pair of **if** statements with the nested **else** part containing a **while** statement.

The first case is defined by the outside **if** condition TRUE, namely

```
The_List is empty
```

and is very simple. We just make The_List point to our new node holding The_Element as data.

The second case is defined by the outside **if** condition FALSE and the inside **if** condition TRUE, namely

```
The_List is not empty and
The_Element <= the value of the first list node
```

and is only slightly more complicated. If The_Element precedes the value in the first node we must make The_List point to a new node with value The_Element and make the new node point to the rest of the list.

The third case requires that we must traverse The_List until we find the proper point within The_List at which to insert. We use Current_Ptr to point to the node against which we are current testing for position and Previous_Ptr to point to the node we just visited. The intention is that the new node will be inserted between Current_Ptr and Previous_Ptr. Its condition is

```
The_List is not empty and
The_Element > the value of the first list node and
Current_Ptr.Data >= The_Element
```

and in this case a new node is required for The_Element to be placed between Previous_Ptr and Current_Ptr, as specified following the **loop** statement.

The fourth case arises if the proper position for The_Element is in a new last node. This situation will arise when Current_Ptr points to the last node and Current_Ptr.Data < The_Element. Its full condition is

```
The_List is not empty and
The_Element > the value of the first list node and
Current_Ptr.Data < The_Element and Current_Ptr.Link = null
```

When this occurs, we must make Current_Ptr.Link point to the new node as specified following the **loop** statement.

These four cases exhaust the possibilities and complete the proof more explicitly than in Chapter 9.

Specification

```
Delete_From_List (The_List, The_Element):
  Purpose: Deletes The_Element from The_List if present
  Input:  The_List and The_Element
  Output: The_List with The_Element removed or
           The_List and an output message
```

Design

```
Delete_From_List (The_List, The_Element):
  if The_List is empty
  then
    signal List_Underflow
  else
    if The_Element is in the first node
    then
      change The_List to point to the second node
    else
      let Current_Ptr point to the first element
      while Current_Ptr.Data /= The_Element and
        Current_Ptr.Link /= null
      loop
        advance both Current_Ptr and Previous_Ptr
      end loop
      if The_Element has been found
      then
        change the pointers to delete the node
      else
        signal node not found
      end if
    end if
  end if
```

Verification

As discussed in Chapter 9 , the deletion algorithm covers four cases–an empty list, deleting the first node, deleting later in the list, and The_Element not present in the list. Before and after diagrams are shown in Chapter 9. The four cases are discovered by a nested pair of **if** statements with the **else** part of the inner **if** statement containing a **while** statement and an **if** statement.

The first case is defined by the outside **if** condition TRUE, namely

```
The_List is empty
```

and is very simple. We just signal List_Underflow to the user.

The second case is defined by the outside **if** condition FALSE and the inside **if** condition TRUE, namely

```
The_List is not empty and
The_Element is in the first node
```

and is only slightly more complicated. Deleting the first element in the list requires that we change The_List to point to the second element in the list if there is one.

The third and fourth cases are defined by the outside **if** condition FALSE and the inside **if** condition FALSE, namely

```
The_List is not empty and
The_Element is not in the first node
```

Deleting from later in the list requires that we must traverse the list until we find the desired node. We use Current_Ptr and Previous_Ptr as before with the intention that Current_Ptr will stop on the node that we are to delete. We then change pointers to remove the node indicated by Current_Ptr. The loop will terminate whether or not The_Element is found. So the third **if** statement following determines that for cases three and four. If The_Element has been found, it will be deleted in case three. However, it is possible that the value that we wish to delete is not in the list. In this scenario, Current_Ptr will stop at the last node and Current_Ptr.Data /= The_Element. If this is the case, we issue a message stating that the value was not found, to complete case four.

These four cases exhaust the possibilities and complete the proof more explicitly than in Chapter 9.

Specification

```
Show_List (The_List):
  Purpose: Prints all the elements in The_List
  Input:   The_List
  Output: The elements of The_List
```

Design

```
Show_List (The_List):
  set Current_Ptr to The_List
  while Current_Ptr /= null
  loop
    write out Current_Ptr.Data
    advance Current_Ptr
  end loop
```

Verification

The verification of `Show_List (The_List)` is by direct inspection. The task is to print out the contents of `The_List` which is straightforward.

12.3.4 Ada Design and Verification of Singly Linked List

Now we are ready to proceed with our implementation by presenting the package body. The verification needs only address correct implementation of the design above, and needs not reach back to the original specification which was addressed by the design. The implementation developed in Chapter 9 is as follows, preceded in each procedure by its design as commentary.

```
with TEXT_IO; -- needed for the list traversal
package body Singly_Linked_List
is

  procedure Create_List (The_List : in out Node_Access)
  --  set The_List to null
  is
  begin
    The_List := null;
  end Create_List;

  procedure Insert_In_List (The_List : in out Node_Access;
                            The_Element : in Element_Type)
  --  if The_List is empty
  --  then
  --    make a node containing The_Element
  --    set The_List to point to the new node
  --  else
  --    if The_Element <= the value in the first list node
    --    then
  --      make a node containing The_Element
  --      change the pointers to make the new node the
   --       first node in the list
  --    else
  --      let Current_Ptr point to the first element
  --      let Previous_Ptr be null
  --      while Current_Ptr.Data < The_Element and
  --        Current_Ptr.Link /= null
  --      loop
  --        advance both Current_Ptr and Previous_Ptr
  --      end loop
  --      make a node containing The_Element
  --      change the pointers to insert the new node
  --    end if
  --  end if
```

```
is
  Previous_Ptr : Node_Access;
  New_Node, Current_Ptr : Node_Access;
begin
    -- test for an empty list
    if The_List = null
    then
      -- build the first node
      The_List := new Node;
      The_List.Data := The_Element;
      The_List.Link := null;
    -- test against the first element
    elsif The_Element <= The_List.Data
    then
      -- insert the value as the first node
      Current_Ptr := new Node;
      Current_Ptr.Data := The_Element;
      Current_Ptr.Link := The_List;
      The_List := Current_Ptr;
    else
      -- initialize two pointers
      Current_Ptr := The_List;
      Previous_Ptr := null;
      -- search for the proper place
      Find_Place:
      while (Current_Ptr.Data < The_Element) and
            (Current_Ptr.Link /= null)
      loop
        Previous_Ptr := Current_Ptr;
        Current_Ptr := Current_Ptr.Link;
      end loop Find_Place;
      -- insert the new value
      New_Node := new Node;
      New_Node.Data := The_Element;
      if Current_Ptr.Data >= The_Element
      then
        New_Node.Link := Current_Ptr;
        Previous_Ptr.Link := New_Node;
      else
        Current_Ptr.Link := New_Node;
        New_Node.Link := null;
      end if;
    end if;
    exception
      when STORAGE_ERROR => raise List_Overflow;
    end Insert_In_List;

procedure Delete_From_List (The_List : in out Node_Access;
                            The_Element : in Element_Type)
--  if The_List is empty
--  then
--    signal List_Underflow
--  else
--    if The_Element is in the first node
--    then
--      change The_List to point to the second node
--    else
--      let Current_Ptr point to the first element
--      while Current_Ptr.Data /= The_Element and
--        Current_Ptr.Link /= null
--      loop
--        advance both Current_Ptr and Previous_Ptr
--      end loop
--      if The_Element has been found
--      then
--        change the pointers to delete the node
--      else
--        signal node not found
--      end if
--    end if
--  end if
is
  Previous_Ptr : Node_Access;
  Current_Ptr : Node_Access := The_List;
begin
    -- test for an empty list
    if The_List = null
    then
      raise List_Underflow;
    end if;
    -- test against the first element
    if The_List.Data = The_Element
    then
      The_List := The_List.Link;
    else -- search the list
      Find_Element:
      while (Current_Ptr.Data /= The_Element) and
            (Current_Ptr.Link /= null)
      loop
        Previous_Ptr := Current_Ptr;
        Current_Ptr := Current_Ptr.Link;
      end loop Find_Element;
      -- remove the value if it was found
      if Current_Ptr.Data = The_Element
      then
        Previous_Ptr.Link := Current_Ptr.Link;
      else
        TEXT_IO.Put (Item => "Element not found.");
        TEXT_IO.New_Line;
      end if;
    end if;
  end Delete_From_List;

procedure Show_List (The_List : in Node_Access)
--  set Current_Ptr to The_List
--  while Current_Ptr /= null
--  loop
--    write out Current_Ptr.Data
--    advance Current_Ptr
--  end loop
is
  Current_Ptr : Node_Access := The_List;
  begin
    TEXT_IO.Put (Item => "Here is your list ");
    TEXT_IO.New_Line;
    Travel_List:
    while Current_Ptr /= null
    loop
      TEXT_IO.Put (Item => Current_Ptr.Data);
      TEXT_IO.New_Line;
      Current_Ptr := Current_Ptr.Link;
    end loop Travel_List;
  end Show_List;

end Singly_Linked_List;
```

The verification of the Ada code against the design is carried out for each procedure as follows. It is less formal than previous proofs, which is needed to deal with larger programs. Where necessary, more formality and detail is possible.

`Create_List`: The verification is by direct inspection. The single statement of the procedure carries out the objective of the design.

`Insert_In_List`: The design and the code have different control structures of if statements. The design has a pair of nested if statements, while the code has a single if statement with an additional **elsif** part. However, on close examination, the design and the code have identical control, the **then** part of the inner if statement in design becoming an **elsif** part and the **else** part of the inner if statement in design becoming the **else** part of the outer if statement in code. Now these parts can be compared more closely for detailed behavior.

The **then** parts of the design and code deal with the case `The_List = null`, and the code carries out the design in specific Ada terms, defining a first node for `The_List` with `The_Element` as data, just as required.

The **elsif** part of the code carries out the internal **then** part of the design in specific Ada terms, making `The_Element` the new first item in `The_List`, just as required in the design.

The **else** part of the code clearly corresponds to the nested **else** part of the design. The code then carries out the initialized **while loop** in searching for the place to put `The_Element`, either inside `The_List` or at its end in the final if statement. The verification can be carried out in more detail if any concern arises.

Finally, the Ada code `Insert_In_List` deals with the exception of `STORAGE_ERROR` not discussed in the design. Actual, finite data conditions expand on the design in a satisfactory way.

`Delete_From_List`: The design and the code have different control structures of if statements. The design has a pair of nested if statements, while the code has a sequence of if statements, the second with a while statement followed by an if statement. However, on close examination, the design and the code have identical control, the inner if statement in design becoming an if statement and the **else** part of the inner if statement in design becoming the **else** part of the second if statement in code. Now these parts can be compared more closely for detailed behavior.

The first **then** parts of the design and code deal with the case `The_List = null`, and the code carries out the design in specific Ada terms, to raise `List_Underflow` when deletion is proposed for an empty list. This is exactly what is called for in the design.

The **then** part of the second **if** statement in code checks if the first element of `The_List` is to be removed, which is the same case as the nested **if then** statement in the design. They are equivalent.

The **else** part of the second **if** statement in code first searches for `The_Element` in the remainder of the list in the **while loop**, then checks in the final **if** statement whether `The_Element` was found. It then removes `The_Element` if found or sends a message "`Element not found.`" to the user. This is precisely what is called for in the design.

Show_List: The code carries out what is called for in the design with an opening line "`Here is your list `", putting each item on a new line.

This completes a verification at a reasonable level. If more importance is assigned the package, it can be verified in greater detail.

12.3.5 Certifying Singly_Linked_List

As noted in Chapter 9 we can now compile the package body and place the compilation information in our Ada library. At that point, in order to test our new package, we constructed a sample test program called `Build_List` for interactive testing, step by step, with the user selecting one of the four procedures of the package, and names when necessary. This testing gives us a sense of correctness of the package for the cases we provide it. Of course, any failures will point to errors in the package as well. However, this sense of correctness is just that. The evidence only addresses the cases we have provided.

In order to certify the package `Singly_Linked_List`, we need to define a statistical basis for testing. At each step we need to define probabilities for each of the four possibilities, and if names are needed, further probabilities for them. At first glance this may seem impossible -how will one know what the probabilities should be? But at second glance a statistical basis for testing is clearly important and a reasonable hypothesis for the probabilities is really useful. The statistical basis for testing will allow the package to go into the field and the real statistics will emerge there. We will need probabilities for each of the four possibilities and also a probability of termination of input data.

The simplest form of probabilities is constant, say five nonnegative fractions P1, P2, P3, P4, P5 that sum to one, and uniform random names if needed. If necessary, these probabilities can vary with the situation, too. For example the probability P1, for `Create_List` might begin quite small after `Create_List` appears and grow over time, or grow with the size of the list, *etc.* The probabilities for `Insert_In_List` P2 and for `Delete_From_List` P3 may be in fixed proportion but variable, or may not, *etc.* The names may likewise be selected in various ways. One way is to use a telephone book, ignoring names under 5 characters and deleting characters beyond 5 in the rest. For example, "smith" would be a popular name, as would "mille" (from "miller"). Finally, termination probability might vary based on the application, being zero then jumping to one in cases when applications call for it. We illustrate the simple case of fixed probabilities below, using two random packages, Random_Choice and Random_Name, to create inputs until termination.

```
with TEXT_IO, Singly_Linked_List, Random_Choice, Random_Name;
procedure Build_Random_List
is
  Name : STRING (1 .. 5);
  Name_List : Singly_Linked_List.Node_Access;
  Choice : CHARACTER := ' ';
begin
  TEXT_IO.Put (Item =>
    "This program tests singly linked list ");
  TEXT_IO.Put (Item =>
    "package with random data.");
  TEXT_IO.New_Line;
  -- first create the random list
  Singly_Linked_List.Create_List (Name_List);
  -- now exercise the options
  Test_List:
  while Choice /= 'E'
  loop
    Choice := Random_Choice;
    TEXT_IO.Put (Item => Choice);
    TEXT_IO.New_Line;
    case Choice
    is
      when 'C' | 'c' =>
        Singly_Linked_List.Create_List (Name_List);
      when 'A' | 'a' =>
        Name := Random_Name;
        TEXT_IO.Put (Item => Name);
        TEXT_IO.New_Line;
        Singly_Linked_List.Insert_In_List
          (Name_List, Name);
      when 'D' | 'd' =>
        Name := Random_Name;
        TEXT_IO.Put (Item => Name);
        TEXT_IO.New_Line;
        Singly_Linked_List.Delete_From_List
          (Name_List, Name);
      when 'S' | 's' =>
          Singly_Linked_List.Show_List (Name_List);
      when 'E' | 'e' =>
        Choice := 'E';
      end case;
   end loop Test_List;
 exception
   when Singly_Linked_List.List_Overflow =>
     TEXT_IO.Put (Item => "Not enough memory space.");
     TEXT_IO.New_Line;
   when Singly_Linked_List.List_Underflow =>
     TEXT_IO.Put (Item => "Your list is empty.");
     TEXT_IO.New_Line;
   end Build_List;
```

The procedure shown above is designed to allow us to statistically test our new package by creating an access variable for a list and then allowing additions or deletions of as many elements as determined statistically. We establish visibility to our package and two random data generators through the with clause.

```
    with TEXT_IO, Singly_Linked_List, Random_Choice, Random_Name;
```

As noted in Chapter 9, we then proceed to call the appropriate operations from our package by referring to the `package name.operation name`. For example,

```
    Singly_Linked_List.Insert_In_List (Name_List, Name);
```

calls the `Insert_In_List` operation from the package `Singly_Linked_List`. The random data generators, `Random_Choice`, `Random_Name` provide constant probabilities. If more tailored data generators are required that is directly possible.

This statistical testing permits the certification of `Singly_Linked_List` for its correctness.

12.3.6 Exercises

1. Carry out a similar verification and certification for `Circular_Linked_List`.

2. Carry out a similar verification and certification for `Doubly_Linked_List`.

3. Identify a more sophisticated test basis for `Singly_Linked_List` which recognizes a time shift in the relative frequencies of `Insert_In_List` and `Delete_From_List`, possibly cyclic.

4. Expand the verification of `Singly_Linked_List` to another level of detail. If this package is to be widely used in medical systems with human life at stake from failures, what verification level is appropriate?

12.4 System Level Software Development

We discuss the issues in real software development, which usually involves the reuse of as much component software as possible. A substantial library of Ada components is developing. Ada provides programming language capabilities of such breadth and depth that software development in Ada is reaching new levels of engineering discipline, not only in developing new Ada programs, but in reusing Ada packages and subprograms.

In this section, we carry through a small example and examine system development using both existing components and creating other components. This example and explanations were carried out in an actual development, and some duplication of previously discussed ideas are given to keep them in a single package.

12.4.1 The Problem: Develop and Certify a Correct System

Suppose that the system represented in Figure 12.1 is to be developed. The flow chart of Figure 12.1 is presented in pseudo Ada as shown in Figure 12.2. Let us further suppose that modules A and D are to be newly written, that B and C will be used from the repository, and that E will be slightly modified from the repository. We can imagine that A is new because it is the user front end which, along with the predicate P, will control whether we use module B or C. Predicate Q will control the while loop using module C. Imagine further that module D is newly written because it takes the results from B and C, performs something unique to this system and then passes the data on to E. Imagine that E presents data in some standard reporting format and we only make a small modification to E to adapt it for the proposed system. To summarize, our system requires A, D, P and Q to be written from scratch, modules B and C to be reused without change, and module E to be reused after a small modification. The repository information concerning the reuse modules gives us the data presented in Table 12.1.

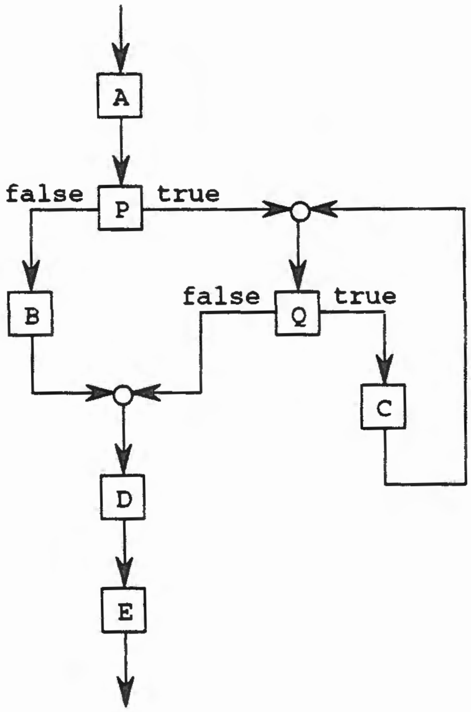


Figure 12.1
Schematic of an Example System

Let us assume that the statement of work requires the system to exhibit an operational mean time to failure of 1,000 hours. A correct system will meet this objective, of course. How do we build the system in order to make the best use of the modules available and achieve the correctness required by the statement of work?

```
procedure EXAMPLE
is
begin
  A;
  if P
  then
    while Q
    loop
      C;
    end loop;
  else
    B;
  end if;
  D;
  E;
end EXAMPLE;
```

Figure 12.2
Example System

The first step in addressing correctness of a system built from new and reused components is to assess the probabilities of making a transition from one module to another. Looking at Figure 12.1, it is clear that the probability of going A to P is 1; the system always goes A to P and never goes from A to anything else. Likewise, the probability of a direct transition A to E is 0 because execution of the proposed system must go through other modules. The more interesting direct transitions are, for example, P to Q versus P to B and Q to C versus Q to D.

Let us suppose that the architect of the new system determines the probabilities as follows:

From	To	Probability
A	P	1.0
P	Q	0.6
P	B	0.4
Q	C	0.9
Q	D	0.1
C	Q	1.0
B	D	1.0
D	E	1.0

Table 12.1
System Transfer Probabilities #1

with all other module to module transitions having probability zero.

Module	Repository Information
A	(none)
B	This module has not been verified, nor has it been statistically tested. No usage record of any kind exists.
C	This module has been statistically tested and has reliability r=0.999 with 98% confidence. The usage profile of the test conforms to the expected use in this system.
D	(none)
E	This module has been certified with MTTF=8,000 hours.
P	(none)
Q	(none)

Table 12.2
Summary of Repository Information

By now objections to this entire effort may be mounting. Objections may be that these probabilities are not known, that data attributed to modules B, C and E are not available, we don't know this and we can't do that. The simple truth is that if we are going to take software development and certification seriously, we must begin to know such facts about software. What we don't know can be hypothesized. The less we know and the more we hypothesize the weaker the analysis. The more we know and the less we hypothesize the stronger the analysis. Even though we may begin with a weak base of facts, we will learn what is important and why by making conjectures, performing analyses and collecting field data. Over time the base of facts will improve and the quality of the analysis will improve. Every engineering discipline that has emerged over the centuries has gone through this rite of passage and so will software engineering.

Let us assume that the newly written modules A and D will be developed to reliabilities of 0.95 and that the predicates P and Q will be given reliabilities of 1.0. The justification might be that our organization has a record of performance to justify the 0.95 on projects of this type and P and Q are small and will be formally verified. Finally, let's suppose that B is as good as A and D.

The statistical analysis shows that under these conditions, the system reliability lower bound will be 0.88, not good enough. It also shows the sensitivities (*i.e.*, "weights" of the reliability failure based on the expected number of uses of the module) as follows:

Module	Sensitivity
A	0.93
P	0.89
Q	5.32
B	0.36
C	4.79
D	0.93
E	0.88

Table 12.3
Module Sensitivities

The reliability goal of MTTF = 1,000 is equivalent under our methods to a reliability of 0.999. This target will be allocated to the modules in the proportions of the sensitivities. Modules Q and C carry by far the greatest reliability burdens of any parts of the system.

Module B is troublesome because no information about the correctness of B is available; on the other hand, B carries the least burden for total system correctness. It would be prudent to conduct an experiment with B to empirically estimate its correctness.

One key idea is to schedule the building of the system in a pipeline of executable product increments. The system proposed in Figure 12.1 might be scheduled as follows:

Increment 1: A
Increment 2: A, P, B
Increment 3: A, P, B, D
Increment 4: A, P, B, D, E
Increment 5: A, P, B, D, E, Q, C

Each increment is executable and will be subjected to statistical certification testing. This is not the only schedule that would build the system. Further study of the system flow chart, the transition probabilities and sensitivities shows that the Q C loop will be the most heavily used. Also E has a stronger record than C and represents less risk. Consider this alternative:

Increment 1: A
Increment 2: A, P, Q, C
Increment 3: A, P, Q, C, B
Increment 4: A, P, Q, C, B, D
Increment 5: A, P, Q, C, B, D, E

The architect's work of deciding the schedule is different from the work of doing the schedule. The schedule will reflect the strategy for keeping the development process under control and meeting the correctness goal.

Based on the statistical analysis we conclude: (1) we should formally prove the correctness of predicates P and Q and schedule the Q C work as early as possible, because these parts of the system need the more extensive testing that accrues to earlier increments, (2) A P Q C can be built and tested in the first two increments so they get the most extensive testing, (3) changes to E should be verified because it will get relatively less testing but it already has a strong performance record.

With this development plan, we must think ahead to certifying the final product and our expectations for each increment. Specifications will be given to the certification team who must construct a statistical profile of the intended usage environment. Test scenarios will be randomly generated based on this profile. When an increment is completed by the developers it will be submitted for certification testing. When a failure is observed by the testers, the developers will make an "engineering change," creating a new "version" of the increment which corrects the error. Testing continues in this way until no further failures are found before management decides to stop testing.

Management must have some expectations in mind for the testing results. For example, the record of the development group may show about three failures per KLOC in each increment before deciding the increment is correct. Assuming that one team develops the entire system, the testing experience for the first increment, module A, will be a good indicator of what to expect on subsequent increments. We can monitor progress relative to these expectations.

12.4.2 Using Cleanroom in System Level Development

Cleanroom software engineering is a team process for software development under statistical quality control, with no program debugging or testing prior to independent statistical correctness certification testing. The name of the methodology was taken from the semiconductor industry, where great care is exercised to prevent defects in design and impurities in process. Comparably exacting standards in software development will also lead to greater quality and productivity. We will use the term "Cleanroom" to reference the entire software development process, from specification to delivered code.

The Cleanroom methodology strives for zero defect software. Each step of the software development process is addressed in Cleanroom, beginning with the contractual statement of what is to be done through delivery of the finished product with certification. Software development is brought under statistical quality control by this methodology. Increased productivity is achieved as a by product of quality concerns. Experience suggests that for skilled practitioners of the Cleanroom methodology, as compared with other competent professionals, the goals of producing a finished job in half the time, with one tenth the number of errors delivered, and one third as many lines of code, are achievable.

Cleanroom characteristics are (1) intellectual aids that scale up from small to large, (2) a management style that maintains intellectual control of the project, and (3) an intense team effort to create a correct and excellent design from which to produce zero defect code.

The steps in Cleanroom development of software are as follows:

1. Establish meaningful management control over the development process.

2. Use the Cleanroom life cycle of executable product increments for specification, design, development and certification.

3. Use the box structure technique for system analysis and design.

4. Design statistical tests based on the intended usage environment. Allow insight gained in defining and constructing the tests to question the specification and influence the design.

5. Use functional verification to determine that the design is ready to code and certify.

6. Conduct the statistical tests to establish correctness and to demonstrate that the software meets the contract requirements.

Each of these concepts is briefly discussed next.

Meaningful Management Cleanroom developments are done by "chief programmer" Cleanroom teams and the manager is a fully participating member of the team. Cleanroom teams will vary in size typically from four to eight persons and will sometimes have specialists in key activities. The manager's role includes interacting with the customer, comprehensive knowledge of the Cleanroom process, strict adherence to the process, building teamwork and achieving team consensus. Active technical management, rather than administrative management, is essential to Cleanroom, and one would expect the manager to be a senior person with respect to knowledge and experience in the formal methods employed by Cleanroom.

Life Cycle of Executable Product Increments Cleanroom projects are organized into sequential segments for development, delivery and field use. Thus, the project feeds field experience back into the development process from the beginning. Specification of each segment, its usage environment and the degree of correctness that must be warranted are required. Stable specifications are the basis for statistical quality control in software, and the incremental approach permits early stable specifications without forcing customers to specify beyond their knowledge or understanding.

Box Structure Analysis and Design Box Structure analysis and design provides a hierarchy of black box, state box and clear box descriptions plus a means of moving up and down the hierarchy for the purpose of clarifying key information for the final system. The black box focuses on the highest level stimulus, response and transition rule description of a process and gives a state free, procedure free description. The state box description introduces stored information in the form of state data and abstracts from procedure. Clear boxes introduce procedures that are to be invented to complete state boxes. In the Box Structure process, analysis and design, discovery and invention, are kept separate and both are enhanced.

Statistical Testing While the systems analysis and design activity is underway, the environment in which the system will operate must be analyzed. Based upon an understanding of the usage environment, input sequences are drawn randomly from a probability distribution of user inputs. These samples must be drawn in a way and in such quantity as to be appropriate for the certification and warranty that has been agreed upon between the developer and customer. It is furthermore necessary to establish (either manually or by an automated means) the outputs from the software under development which will be considered correct for the input sample. Appropriate files should be constructed to expedite the actual test when the code goes to the machine.

Functional Verification The box structure process produces PDL and Ada for the system under development. The process of functional verification is a mathematical thought and argument process which facilitates a detailed examination by the team in a way that will either convince the team members that the code is correct, or will expose mistakes. Mistakes are corrected and the code is again verified. Cleanroom developers do not execute or debug code before certification.

Certification Model Typically, the persons developing the code will not be present or involved in putting the code on the machine for testing and certification. If the development team was successful, performance measures will be within the contract specification and the code will be delivered. If the development team failed, the code will be pulled off the machine and the team will return to the analysis and design phase. The Certification Model prescribes a calculation based upon the results of statistical testing which yields a correctness measure. If these calculations show that the performance of the software equals or exceeds the contractual requirement, the software is so certified. Each software increment, after the first, will contain software previously certified plus newly written software, and the composite will be tested and certified. Certifications subsequent to the first will have increasingly broader usage profiles which will make the certifications themselves more accurate.

12.4.2.1 Cleanroom Software Development Life Cycle

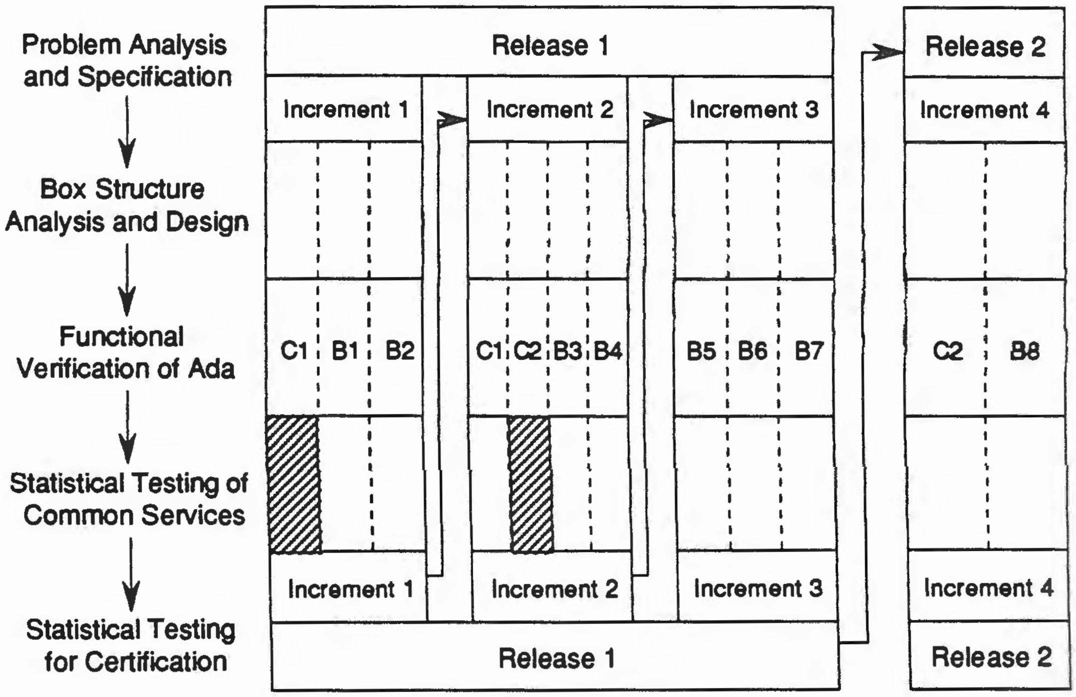

Problem Analysis and Specification
Box Structure Analysis and Design
Functional Verification of Ada
Statistical Testing of Common Services
Statistical Testing for Certification
Release 1
Release 2
Increment 1
Increment 2
Increment 3
Increment 4
C1 B1 B2
C1 C2 B3 B4
B5 B6 B7
C2 B8
Increment 1
Increment 2
Increment 3
Increment 4
Release 1
Release 2

Figure 12.3
Cleanroom Software Development Life Cycle

Figure 12.3 depicts the Cleanroom software development life cycle. Problem analysis and specification should lead to a structured architecture that defines not only what a software system is to be when finished, but also a constructive plan to design and certify the software in a pipeline of increments. The product development plan is cast as a series of releases to customers, each of which is comprised of cumulatively certifiable executable product increments. Field experience from release R(i) is feedback into the management plan to influence R(i+1),...,R(n). Certification tests are performed on each increment so that experience from increment I(i) is feedback into the management plan for increments I(i+1)...I(m). Whenever an increment fails to meet certification standards, the team manager can investigate the root causes, correct the software development process, redesign the increment and again attempt to certify the software through that increment.

An experienced Cleanroom team might deal with increments of 10,000 lines of code, which might result from more than 100 modules created in the top down usage hierarchy for the increment. Figure 12.3 illustrates the relationship of modules to increments. B1, B2, *etc.*, represent boxes in the box structure design. C1 and C2 are also boxes, but denote common services, modules that are reused. Box structure analysis and design creates a nested hierarchy of boxes, only one level of which is shown in Figure 12.3.

When the specification is set, or is taken as a working specification, the process of giving statistical expression to the intended usage environment can begin. Usage statistics are effectively an addition to the function and performance specifications. One must decide what constitutes a test case, and then generate a statistically correct series of test cases reflecting the intended usage environment and adequate to the correctness and confidence levels sought. Test results are logged for I(1), I(1,2), ..., I(1,2,...,m).

Cleanroom protocol for software certification testing proves to be a powerful component of Cleanroom software development. Early in the process of characterizing the usage environment and constructing a sample of test cases, one confronts specification and design issues that otherwise might not arise until too late. One doesn't want to produce systems that cannot be tested or that are difficult to test, so when aspects are found to be difficult to test, one questions the specifications and the design. Typically, both can be improved upon in a way that is straightforward to test.

12.4.2.2 Formal Methods In Cleanroom

The formal methods in Cleanroom are box structure analysis and design, functional verification and statistical testing. Success depends critically on the team's ability to apply formal methods in specification, design, verification and certification.

Box structures have several formal aspects. The top down process separates function, data structure and procedure at each step in an effort to achieve a sharp focus of attention on the three critical aspects of each design step. Furthermore, a verification process is prescribed to confirm that the expansion from black box to state box is correct and again that the expansion from state box to clear box is correct. The process is one of taking small creative, inventive steps and then confirming that each step is correct before going to the next step. Experienced designers may use less formal verification of some state boxes and clear boxes; fine grain verification may be more or less enforced depending upon the nature of the work and the prowess of the team.

Box structure analysis includes four principles which assist the team in deriving a good design from the specification. *Transaction closure* invites the assurance that all the information that is needed is available. *State migration* invites the assurance that data is available or hidden at the optimal level. *Common services* are emphasized and should be identified to support reuse of modules. *Referential transparency* is sought to establish independence and limit complexity of modules. Fully structured descriptions of systems are thus drawn directly from the specifications. Box structures do not automatically guarantee a good design adhering to these four principles, but the methodology leads one through a sequence of steps which if performed correctly with these principles borne in mind will produce a structured design of high quality that supports the Cleanroom life cycle.

Box structures are used to state or restate the specification, depending upon how the problem reaches the Cleanroom team. The analysis of new systems begins in the black box and is followed by invention of state box and clear box through an expansion process. The analysis of old systems begins in the existing clear box and is followed by discovery of state box and black box through a derivation process. The result is a structured specification, each part of which (substructure) is amenable to further work with box structure methods to derive a design. Each part of the design is amenable to further work with box structures to derive a procedure. Procedure parts are further developed until the parts are so direct and simple that transliteration into a target language is possible. A unified formal method is used to produce a hierarchical continuum of (black box, state box, clear box) nodes from the highest level specification to code, without discontinuities in intellectual control. Each step of the hierarchy is verified in a hierarchy of functional composition extending from the problem specification down to the base of primitives at implementation.

Functional verification is a formal method more akin to "Naive Set Theory" than to "Metamathematics". Indeed, the description of functional verification admonishes the designers to find "the right balance in mathematical proof between formal procedure and economy of effort." Functional verification is a process whereby team members argue the correctness of designs. The process is not unlike that of a group of mathematicians arguing the correctness of theorems. The argument will be less formal on segments the team finds familiar and "easy". Arguments will be more formal for a segment which some member of the team is not convinced is correctly designed or verified. The quality of the verification depends upon the intellectual toughness and integrity of the team.

Functional verification provides a basis for reducing the problem of program correctness to that of determining the correctness of constituent program structure primes. Since the design process leads to the use of only sequence, if, while and case statements, the techniques needed are few.

However, the emphasis in functional verification is on practical application. Techniques for conducting practical verifications are presented in a way that focuses on correct reasoning about programs. Trace tables are prescribed for reasoning about sequences. Disjoint rules are introduced to restate conditional rules in a more convenient format. Case structure trace tables deal with component rules resulting from conditionals. Finally, direct assertion is encouraged for segments that are well understood without detailed verification. The emphasis of functional verification is the productive employment of a sound theory.

Statistical certification testing requires the analytical mathematics of correctness theory and applied statistics, also formal methods. For each increment and for the evolving product one must deal with the correct statistical characterization of the usage environment, the construction of a statistically correct sample of test cases and the number of test cases required to state correctness with a certain confidence limit. Furthermore, as each executable product increment is certification tested, the time dimension comes into consideration producing MTTF estimates, along with the confidence in the current failure free system.

Certification is a sobering process for all parties to the software phenomenon. First, there is the realization that one may have any confidence levels in correctness, but that the greater levels entail higher costs and longer certification testing periods. Second, there is the realization that other forms of testing, however carefully reasoned, do not contribute to making a scientific statement about correctness of the software. The Cleanroom thesis is that functional verification in anticipation of independent statistical testing will engender intellectual control and greater correctness at less expense than through other testing and debugging processes.

The life cycle of executable product increments coupled with certification testing creates the basis for statistical control of the process of software development. Whenever an increment is judged by certifiers to have a failure, the process is out of statistical process control. The increment must be returned to the development team, and get the process back into statistical control.

12.4.2.3 Cleanroom Experience

One large Cleanroom project with rather complete data is the IBM product known as COBOL/SF, COBOL Structuring Facility. COBOL/SF is comparable in function and complexity to a modern high level language compiler. By use of proprietary graph theoretic and function theoretic algorithms, it transforms working old COBOL programs into hierarchies of structured procedures in modern COBOL. The current product of some 80,000 lines of code was produced in increments, the last five of which are reported below. In accordance with the Cleanroom methodology, the code was taken to the machine at each increment with no testing or debugging, with the following results.

Increment	Lines of Code	Failures/KLOC
1	4150	1.4
2	11125	2.2
3	10080	2.3
4	19543	5.7
5	7117	2.1
	52015(total)	3.4(avg.)

Table 12.4
Test Failures in Cleanroom Project

The cumulative results over the accumulated increments were I(1, 2) = 1.9 failures/KLOC (Increment 1 already tested with 1.4 Failures/KLOC already fixed, Increment 2 never executed before), I(1, 2, 3) = 2.1, and I(1, 2, 3, 4) = 3.6. With these failures removed, failures in field tests were below 0.1 Failures/KLOC and readily removed to zero.

The HH60 helicopter flight control program was developed by the IBM Federal Systems Division. Cleanroom was used to develop the system in three increments of more than 10,000 lines of code each. The code was taken directly from functional verification to statistical testing with failure rates of I(1) = 2.7, I(1, 2) = 1.0, and I(1, 2, 3) = 0.9 before testing of latest code, going to near zero then.

A Cleanroom project at the University of Tennessee is to design and produce a Cleanroom tool called the Box Generator. Cleanroom teams typically produce a draft design in BDL, meet in review and mark up sessions, produce clean copy, and review again. The Box Generator is a single user, keyboard, screen interaction system. The first increment of the tool is to ease the paperwork burden of Cleanroom and has the character of a structure editor. Subsequent increments will generate Ada outer syntax, impose the box structures methodology, adding the flavor of computer assisted instruction, and install instrumentation to record various metrics.

An early controlled experiment at the University of Maryland compared Cleanroom with traditional software development on a task that resulted in 800 to 2300 lines of code. Among the conclusions supported by both data analysis and testimony was that the Cleanroom teams met requirements more completely, had a higher percentage of successful tests, had less dense control flow complexity, and had a better record of meeting schedules. The faculty also found interesting anxieties resulting from developers being denied execution of their code and dependence on statistically derived operational tests.

12.4.3 What Does It Mean To Say Software Is Correct?

It is difficult to get agreement across a broad community of software engineers on colloquial definitions of correct software. In an intuitive sense, software is correct if it behaves correctly sufficiently often and at a certain special moments. To behave correctly means to produce output when output is called for, to produce the correct output and not to produce unwanted side effects.

This definition begs difficult questions, *e.g.*, who judges whether output is correct? The standard answer is that the specification and the customer's interpretation of the specification judge the performance of the software. We know that specifications can be wrong and that correcting them can be time consuming and expensive. Assuming a correct specification, how often must the output be correct for the software to be labeled "correct"? Can the judgment be quantified and standardized?

Correctness is a metric that can be useful both in guiding the software development process and in assessing a programs fitness for use by conducting experiments to establish empirical evidence of quality. As a matter of convenience, these dual uses of correctness are given slightly different definitions. Correctness as a function of time, perhaps the more traditional definition, addresses the design of software that will operate according to specification for a period of time. But we can also use a simpler definition where correctness is the probability that a randomly chosen input will be processed correctly.

In most instances, we use the definition of correctness as the probability that randomly chosen inputs will be processed successfully. This definition is well suited to the idea of conducting experiments to establish empirical evidence of quality. This proves to be a very conservative notion of correctness, well suited to dealing with reuse of software for which little may be known about the process of its development but much may be known about its operational history.

In classical programming the underlying assumption is that the design and development process was not perfect and that some errors persist and are reflected as faults in the code. The choice of time as the random variable is based upon the idea that randomly selected inputs (according to a usage distribution) will cause paths through the program to be executed randomly and, consequently, as operating time increases the probability of encountering a fault in the code increases.

12.4.3.1 Verification of Software

There are methods that "guarantee" that a program will work properly on all inputs all of the time. After all, there simply must be at least theoretical ways to establish that a program is completely correct. These include:

1. Verification, by a proof that is done to your complete satisfaction.

2. Exhaustive testing, *i.e.*, test all inputs (though the number may be astronomical) and confirm that each yields the correct answer (though a wizard may be needed.)

3. Combinations of the above two methods. For example: Prove that the inputs divide into k classes and prove that it suffices to test one member of each class, then test one member of each class.

The strength of such methods is that they show the program to be completely correct. If you can afford to do any of these 100% methods, do so. Alternatively, if you cannot afford not to have 100% reliability, do so. Such methods are expensive, time consuming and hard to believe in spite of everything.

Many argue that formal verification cannot work because the proofs are longer than the programs and theorem proving is harder and more failure prone than programming. A proof is just an effort to convince a listener; even precise, mathematical proofs that have undergone peer review are, occasionally, later discovered to be wrong. Another example of a possible source of failure is that the verifier may fail to identify "all" inputs, or incorrectly judge that the program succeeded on a test. All formal methods are subject to these problems.

Exhaustive testing requires first being able to correctly identify all possible inputs, then being able to judge the performance of the software on all inputs, and finally being able to plan and conduct the tests correctly. When and if this is done, then it is surely a "proof by cases."

12.4.3.2 Testing Software

The classical alternative to proof of correctness, although seldom thought of in such terms, is testing. Testing is an enormous industry and the ingenuity of software testers is exceeded only by software writers. There are many testing strategies ranging from commonplace unit testing followed by integration testing to less practiced techniques such as mutation testing. There is coverage testing of various types, *e.g.*, path coverage, decision coverage, statement coverage, and feature coverage. However, the numbers defeat all methods.

Consider path coverage. Each branch and merge in a program doubles the number of paths. A branch and merge occurs each time an if statement is used. So, in a program or a system of 5 million lines of code, how many branch and merge events will occur? Pick up most any program of a few hundred lines of code and count the if statements. Raise 2 to the power corresponding to the number of if statements and you will quickly see that path testing is impossible for all but the simplest, most carefully written program. Most readers will recall that 2^{32} exceeds all 10 digit decimal numbers, but 32 if statements does not seem like so many. Add loops to the above analysis and the number of paths confronting path testing becomes even larger.

Any compromise version of coverage testing seems weak in comparison to path testing and weaker still when input histories are considered.

Most programs react to a history of inputs rather than to a single input. Histories typically span the Cartesian product of some reasonable size set of individual inputs. Suppose the input set contains just 2 individuals, but that the unique histories extend to sequences of 32, and you have the old friend 2^{32} again. In practice, the event set will be larger and the sequences longer.

Our conclusion is that testing is a bottomless pit. Methodological testing alone can never establish the degree of confidence that is needed in software being used in critical applications. Our conclusion is that formal methods are the only solution in the long term.

Cleanroom advocates the use of semi-formal methods, *i.e.*, fully mathematical methods that have been relaxed a bit to make them easier to practice. One doesn't have to be listing axioms and rules of inference in an arcane notation in order to be doing formal methods. Most forms of engineering design and analysis constitute formal methods, including, for example, control theory, circuit analysis and strength of materials. Formal proof will never be enough for society. Society will demand empirical evidence for a very long time to come because of the poor track record of software, the lack of standards for practitioners and the *intangibility* of software. This empirical evidence will come in the form of testing, but it is testing with a very big difference. Apart from special, critical case testing, the only form of testing that complements formal verification is statistical testing.

12.4.3.3 Statistical Testing, Without A Model

Another approach to demonstrating the correctness of software is statistical sampling, without a model. The basic idea is as follows. First sample inputs according to the actual distribution of inputs the user will encounter. The user must supply this usage distribution as part of the specification for the program's correct behavior. Then test the program on the sampled cases, and see how many work properly. This may require a user supplied "oracle" to evaluate output. Finally, make statistical claims based on the results.

For example, suppose there are 1 million possible inputs, and the user states that each is equally likely. Suppose the tester selects 2302 inputs at random, and discovers that the program works correctly on all of them. Then the tester will claim that the program will work correctly on at least 99.9% of the inputs. That is, the tester will claim that the program will work correctly on at least 999,000 of the 1 million inputs. The claim will be made with 90% confidence.

There are two good points about statistical usage testing. First, one can make a quantitative claim about software correctness. The next section explains how correctness is quantified under Cleanroom methods.

Second, the errors that would be most likely to appear after the software is placed in service are precisely the errors most likely to show up in statistical usage testing. Test resources are devoted to precisely the cases that are most likely to come up in practice.

The drawbacks to statistical testing without a model are as follows. Characterizing the usage distribution may be very hard. The testing does not allow for semantics of the program. For example, many different inputs may all yield the same result, in such a way that testing one successfully guarantees that all the others will work. Statistical testing without a model might be wasteful in such cases. Finally, a large number of test cases will be needed and the claim one can make will be pessimistic.

12.4.3.4 Statistical Testing with Models

In the Cleanroom Certification Model, inputs are selected according to the actual usage distribution. Coverage testing ignores the usage distribution. Thus we see that model based testing methods can, but need not, make claims based in part on statistical sampling of inputs. We argue in the next section that statistical sampling of inputs should be a part of any model based method.

In a method for software systems, we measure the reliability of components of the main program, and estimate the reliability of the program from those measurements, based on simplifying assumptions about the interactions of components. The underlying model specifies those assumptions. In this method, inputs are selected according to the actual usage distribution.

Model based testing methods have two strengths. First, for models that are sufficiently explicated, the tester can make a quantitative claim about the correctness of the software. However, your belief in the results obtained under such models must be tempered by the degree to which you believe the model is a reasonable approximation to reality. Second, the claims produced by model based testing can be, in general, much stronger than those of statistical sampling without a model. This is not unexpected. If one includes knowledge in the testing method, one expects to be able to make a stronger claim.

The weakness of model based testing is that the model is never exactly the same as reality; thus one is dealing with approximations. In general, it is hard to show the model is a "good enough" approximation. That is, it is hard to make mathematical claims about the quality of the final correctness estimate provided by the method. Note, however, that even if the model is poor, the method may work, because of "averaging" principles. For example, the assumptions of the system method are clearly far from reality, but the conclusions of his method may well be quite reasonable. This is because the errors introduced by his assumptions may cancel each other out.

The bottom line is that you want (1) to create correct software, and (2) to "validate" that the software is correct. That is, you want to have confidence that your methods have indeed performed as they should.

Cleanroom methods create correct software by the techniques described above. These techniques include the use of box structure analysis and design, stepwise refinement, and functional verification.

Cleanroom methods also validate that the software is correct. Validation serves three purposes:

1. It shows whether the software development process is in or out of control.

2. It provides a quantitative estimate of correctness. This estimate can be used as part of a contractual agreement with the user.

3. It permits improvements in the software, as failures are found and corrected. For most software development methods, this is the primary purpose of testing. For Cleanroom, it is only a by product of testing.

Cleanroom techniques for validation include the following:

1. The released product undergoes statistical testing. This generates a lower bound on the correctness of the software. This bound will usually be conservative, but the assumptions on which it is based are relatively believable.
2. The Cleanroom Certification Model provides a further estimate of the correctness of the released product. This estimate will usually be more realistic than straight statistical testing, but subject to more sources of error.

There is a growing consensus that neither testing alone nor correctness proving alone can assure the correctness of programs.

If you have the resources to do verification, exhaustive testing, or a combination of the two, do so. If you lack such resources, you can do statistical testing without a model. The results of such testing are based on only a few, relatively believable, assumptions. Unfortunately, such results are likely to be weak. To get stronger results (that is, to make claims that the software is very correct), one can adopt a model of some sort.

12.4.4 Principles of Statistical Testing

Statistical testing of software means that the test cases are (1) selected according to a statistical profile of the intended usage and (2) generated in a quantity adequate to support a scientific statement about the correctness of the software or its mean time to failure.

Our interest in statistical testing is for the purpose of certifying the correctness of the software, *i.e.*, giving empirical evidence that the software is in fact correct. We are not interested in statistical testing as a means of discovering faults in the code. Fault discovery is, of course, the traditional purpose of methodical testing. However, path testing does not give better assurance of program correctness than statistical or random testing. Indeed, random testing can be much more cost effective than path (partition) testing. Although we are not interested in debugging, we are vitally interested in fault removal in a way that extends the mean time to failure as dramatically as possible.

The key ideas in statistical testing are (1) sampling (*i.e.*, developing a set of test cases) according to the intended usage of the software, (2) judging the correctness of each test case, and (3) evaluating the experiment.

The best basis for statistical testing is extensive field data that shows in detail the statistical profile of the usage environment. Extensive field data can take various forms. A simulator is perhaps the ultimate statistical test. A simulator is capable of generating test data to suit the simulated conditions, and it isn't really necessary to know the distribution if it is clear the data are truly representative. Furthermore, simulators are capable of generating enormous samples and complex histories. Most of the objections to statistical testing are swept away by simulators.

In many applications data can be recorded and later played into the testing scenario with effects similar to that of a simulator.

Of course, there are the difficult situations in which no usage data are readily available. One can take the position that an important part of writing the specification for any system is to know how that system will be used, or at the very least to have working hypotheses for how the system will be used. Statistical tests can be designed from the same specification used by the system designers.

In the absence of data, one can make theoretical analyses and stipulations. In the absence of both data and knowledge (why is this software being written?) one can say all inputs are equally likely, use the uniform distribution, and in effect reduce the statistical test to a partial coverage test with uniform random sampling.

Usually, one would expect to do much better than uniform sampling. Many classical distributions are known to fit certain typical situations. One can establish theoretical grounds that certain things are normally distributed, others exhibit Poisson arrivals, behave logarithmically, *etc*. These classical distributions are extensively studied and a great deal is known about them and their relationships to each other. So much so that even when we have extensive data we are happy to notice that the data fits a familiar, classical distribution. One determines the parameters of the distribution and generalizes from the real data.

A system might have several usage environments. Customers might be classified by the way they use the product. Users might be classified from novice to expert. Each can be given a statistical profile and the software can be certified environment by environment.

For example, consider a word processing software package. What constitutes a test case? Intuitively, one would not consider a single key stroke to be a test case. Nor would one have to type a whole book such as *The Count of Monte Cristo* in order to have a test case. Experience with single user keyboard screen interaction software has led to definitions like the following: A test case is a sequence of major events ending with a "print" or a "save" event. A major event would be things like keying several paragraphs of text, moving a paragraph or spell checking the page. In this example, the sample space becomes a Cartesian power of the event space so that each sample is a history of events. The length of this history is important in the performance of the software and in the definition of a test case. One might establish that the histories are normally distributed with a certain mean and standard distribution.

One important strategy in developing correct software is to limit as much as possible the buildup of history. By initializing a system strategically one can convert unlimited histories to very short ones and in so doing convert untestable software into easily tested software. Statistical usage testing tends to focus one's attention on such matters to the great improvement of the correctness of the software.

It is important in conducting a statistical test that the correct response for each test case be determined in advance, based on the specification. To be sure, one can made a mistake in this determination, but the point is to take all possible steps to assure comparing two independent interpretations of the specification. We want each test (possibly a lengthy history of inputs) to be a simple binomial trial: the test either succeeds or fails.

Automated oracles are most desirable. If a new system is replacing an old system, the old can be the oracle for the new to the extent they overlap in function. If a system is being ported from one computer to another, the original version can be the oracle for the new one. Some systems show pictures, make noises, turn valves and do other things that are easily recognized as correct or incorrect by simple observation.

Sometimes it is necessary to painstakingly work out the details of what should be the correct response to an input or a history of inputs. There are too many different types of software to attempt to mention them all. Work is underway on single user keyboard screen interaction systems to develop guidance on constructing statistical tests. Similar work has begun to fashion statistical tests in simulators for various usage scenarios.

In looking directly at the program to be certified, we examine the process as versions 0, 1, 2, ..., m of the program are created. We test the preliminary versions and predict the correctness of the final version from the statistically estimated reliabilities of the preliminary versions. To make this prediction we need a model of program development through successive versions.

A distinction must be made between software increments and versions. There may be several versions within an increment.

12.4.4.1 The Basics of the Certification Model

In the Cleanroom methodology, the Cleanroom team performs a semi-formal verification of the software. After this, the software undergoes statistical testing. The testing proceeds as follows.

1. Input is sampled according to the usage distribution. The software is tested on the sample inputs, until a failure or testing is completed. Suppose 500 test cases are planned. That is, testing continues on inputs chosen randomly (but according to the usage distribution), until an input is found on which the program does not provide the correct behavior or test time has been exhausted. Let T_0 denote the number of test cases tested so far. For example, if the program worked properly on the first 3 inputs, but failed on the 4th, then T_0 is 4. Note that T_0 is an estimate of the MTTF of the software, measured in number of test cases.

2. The team then examines the software to see why their verification failed. In light of the observed failure, they change the software. Let us call this an "engineering change" (EC), to emphasize that the team reexamines the software design, rather than just doing a "bug fix". However, as part of the EC, the team also corrects the fault that caused the failure. Note that T_0 is no longer a reasonable estimate of the MTTF of the software; the new software is different (because of the EC), and so should have a new (presumably higher) MTTF.

3. Testing is resumed on a new collection of random inputs, continuing until a failure, if ever. Let T_1 denote the number of new inputs. Continuing the previous example, if the program worked properly on the 5th through 85th input, but failed on the 86th, then T_1 is 81. Note that we do not include in T_1 the 4 inputs on which the program was originally tested. T_1 is an estimate of the MTTF of version 1 of the software, where the original version was called version 0.

4. The team examines the software to find this second fault, and to see how it escaped their verification process. They perform a second EC, and in doing so, they also fix this second fault.

5. The team repeats steps 1 and 2 as needed. Let EC_k denote the k th EC, where EC_0 is the change made to the original version of the software. Let T_k denote the number of tests of version k of the software, that is, the number of tests between EC_{k-1} and EC_k. T_k is an estimate of the MTTF of version k of the software.

6. Eventually the team stops testing, after (say) m EC's and no failure in the last version. The current version is version m. Now we want to say one of two things.

   a. We found too many failures. Redesign the software.

   b. We believe no failures remain, and certify the software correct.

Scientific progress generally begins with the explication of a descriptive model of some real world phenomenon that we wish to understand better. At this stage justification for the model is based upon theoretical arguments keyed to observation and knowledge of related or similar phenomena. To make a model more useful we try to make it predictive as well as descriptive and then attempt to conduct experiments or otherwise acquire data that will either confirm or refute the model. Adjustments are made to the model in the face of compelling data and deeper understanding of the phenomenon. In any case, it is the strong structural aspects of the model that are of importance, not matters of detail. Progress in making the model genuinely useful depends upon the ability to recognize the difference between important structure and minutia and being able to characterize the structure correctly. Without belaboring the point, one might reflect on such models as epidemic or fluid dynamics for illustrations of this general principle.

At this juncture the strongest argument for the Certification Model is that it was derived to work as a matched set with the Cleanroom development process. Confidence in the model should flow directly from an examination of the assumptions and from arguments that the assumptions are plausible. Field data will ultimately confirm, modify or refute the model.

12.4.4.2 The Reliability Allocation Model

For example, consider the program depicted by the flow chart in Figure 12.1. Using the information in Figure 12.1, Figure 12.2, and Table 12.2, we can summarize the data as follows:

Module	Reliability
A	?
P	1
Q	1
B	?
C	.999
D	?
E	.999

Table 12.5
System Component Reliabilities

The question marks must be replaced by estimates or stipulations as will be explained below. The values of 1 for P and Q are there because these are small amounts of code that will be formally verified.

The following table gives the probability of transfer from one module to another. All other from to pairs are impossible and therefore have probability zero.

From	To	Probability
A	P	1.0
P	Q	0.6
P	B	0.4
Q	C	0.9
Q	D	0.1
C	Q	1.0
B	D	1.0
D	E	1.0

Table 12.6
System Transfer Probabilities #2

The assumptions made in the Reliability Allocation model are as follows.

1. For each module X, there is a constant R_X (which you must determine) such that each time X is executed during program execution, module X behaves properly with probability R_X.

2. For each pair of modules X and Y, there is a fixed probability C_{XY} that control passes to module Y, given that control is currently in module X.

3. If any failure occurs at any time in any module, the main program (or system) fails.

To use the Reliability Allocation model, you must determine the following.

1. The reliabilities of the modules. That is, you must determine, for each module X, the probability R_X that the module works correctly on a randomly chosen input. Here "random" means according to whatever usage distribution the module expects to see when the main program is used with its given usage distribution. It is possible that the correct usage distribution of the main program does not use the module in a way that corresponds to the usage distribution under which the module was certified. This may or may not be a serious concern.

2. The table of transition probabilities. These should be the probabilities as they would be if the program is assumed to have no errors.

Is it practical to estimate the transition probabilities accurately? Is it practical to know the usage distribution of the modules? Is it practical to estimate the reliabilities of the modules accurately? Today's answers may be negative, but as reliability measurement becomes a standard aspect of software, the answers will increasingly be in the affirmative. However, these estimates need not be very precise to serve our needs. Remember, we are not trying here to provide good estimates of the quality of the released product. We are only trying to acquire some guidance on issues that must be faced during the development process.

We use the Reliability Allocation model to judge (1) when it is safe to reuse software, and (2) when to accept an increment. The next two sections describe these two uses.

12.4.4.3 Judging Whether It Is Safe To Reuse Software

To see whether or not it is safe to reuse a particular set of software modules, proceed as follows. First, estimate the transition probabilities among modules in the system. This might be done through experiments with prototypes or by examining the modules and how they interact. In the example system above, we have only to determine the transitions from predicate P and the loop control predicate Q.

Second, acquire or estimate the reliabilities of the modules whose reuse is under consideration. Such information might be available from a history of use of a single copy or a history of use in a fleet. Such histories can be converted to approximate reliabilities in the sense of zero failures sampling. Reliability information might be available from previous testing under a usage distribution that is still applicable. We might use sampling without a model to conduct an experiment (an inexpensive one to start), the outcome of which might say a module should not be reused under any circumstance. In the example system, modules B, C and E are candidates for reuse. Let's assume that 135 randomly selected test cases all ran without failure giving us R_B=.95 with a high degree of confidence. Module C might be acceptable without further ado, and E has good standing, prior to modification.

Third, for reliabilities of modules other than those that might be reused, plug into the model whatever estimates you have for those yet to be developed. Such expectations will be based on previous experience with the product being built and previous products built by the development group. They may also be influenced by reliability goals set through the Reliability Allocation model, as described below. Again, referring to the illustration above, we are stipulating that because of formal verification of small codes, $R_P = R_Q = 1$. Cleanroom teams keep score and know the quality of their work and the team's ability, so if the newly developed modules are to be done by an experienced Cleanroom team, data will be available to guide those estimates. Let's assume that to be the case and stipulate that $R_A = R_D = .95$.

Note that you will make a binary decision in each case: reuse module X or not. Even a rough estimate for R_X may be sufficient to lead you to the correct decision. We can now restate the table of reliabilities as follows:

Module	Reliability
A	.95
P	1
Q	1
B	.95
C	.999
D	.95
E	.999

Table 12.7
System Transfer Probabilities #3

With this data, one can compute an estimate of the reliability of the system. This computation involves little more than inverting a matrix. In this example the system reliability is 0.88.

If the computed reliability is sufficiently large, then all is well. If not, you have three choices.

1. You can theoretically improve the reliability of any module by redesigning and reprogramming it.

2. You can improve the reliability estimate of any module by testing it further. Perhaps the module is adequate, but your estimate of its reliability is conservative. In this situation, it may be more economical to test it further than to reprogram the module. (One hopes further testing will increase the estimate of its reliability, however, it is possible to undermine the original estimate.)

3. If a certain module has low reliability but cannot be rewritten, it might be practical to gain overall system reliability by improving the other modules. That is, you can improve previously written modules, or raise your requirements for modules yet to be developed.

In the case when the predicted reliability for the program is not large enough, the Reliability Allocation model also provides guidance for choosing which module to improve. A reliability target for the system can be set, and an "allocation" can be computed for each module. The model shows where to focus attention and can show the potential consequences of reusing certain modules in certain ways.

You can use conservative (or optimistic) estimates for the module reliabilities to acquire a conservative (or optimistic) estimate for the program reliability. Thus, one can acquire a range, not merely a point estimate for the reliability. Of course, even the range estimate is still based on the modeling assumptions and the accuracy of estimates for the transition probabilities.

12.4.4.4 Judging When to Accept an Increment

Judging when to accept an increment as fit for use is strictly a management issue. We do not suggest that the model or the mathematics can usurp that responsibility, but the model can provide helpful information. Suppose that testing of increments will provide estimates for the reliabilities of component modules that comprise the final product.

Using the transition probabilities and the reliabilities of the continuing example, we have the following allocations for a system reliability goal of 0.999:

Module	Allocation
A	0.9990
P	0.9990
Q	0.9998
B	0.9975
C	0.9998
D	0.9990
E	0.9990

Table 12.8
System Transfer Probabilities #4

Any module which we can establish meets its allocation, or any increment which meets its allocation should be accepted because the Reliability Allocation model is conservative in the sense of sampling without a model.

12.4.5 Construction Plan

The first two issues to coordinate are reuse and incremental development. For most systems under consideration there will be several competing designs. These competing designs will entail different degrees of reuse without modification, reuse with modification and newly written software. In each case, the chief programmer will have more than one (but only a few) reasonable construction plans for each design. Many factors will enter into the management decision as to which design and which construction plan to ultimately select, but demonstrable reliability will be a major criterion. One should expect large differences in the plans with respect to the reliability of the final system and the ability to estimate and predict the reliability of the final system.

The steps in assessing each design and plan (with quick winnowing of the really bad ones) are as follows.

1. State the reliability certification goal or target for the system. This will be constant for all designs and all construction plans.

2. Draw the control flow diagram or write the Ada for the top level(s) of the system showing the transition relationships among the predicates and modules.

3. Determine to the best degree possible the probabilities of passing from one module to another. Although these can be changed in light of new information, they are of critical importance to the planning.

4. Enter the target system reliability and calculate a reliability allocation to each module.

5. Assemble the factual information on the reliability of the proposed reuse modules.

6. For predicates or modules that are to be written, one can stipulate a reliability of 1 based on intensive verification of a very small module. For larger modules that are to be written one can stipulate whatever the track record of the team will support.

7. Compare the facts and estimates with the allocations. For each module is it clear that the facts meet the allocation? Where the estimate of the reliability of an existing module is too low, consider the sampling without a model experiment that would be necessary to increase the estimated reliability (or demonstrate that the software is of too low reliability.) Does it appear plausible that the design can yield the target reliability? If not, reject this option. If so, then accept the design as a working hypothesis.

8. Work with the data to decide the experiments that are needed to demonstrate the reliabilities of reuse components. The sensitivity data will tell you where to work. It might be necessary to change the design if it makes too great use of a module for which adequate reliability cannot honestly be established. Conduct the experiments, evaluate the results (putting them in the repository for future use) and make the final decision on whether the design has a chance of meeting the target reliability. If not, reject the design and evaluate the next. If so, then keep this design among the competing alternatives.

9. Evaluate the competing alternatives to select from among all that hold promise of meeting the reliability target, the ones that are favored on other grounds.

Proceed to make a construction plan.

10. The construction plan will be the identification of the increments in which the system will be developed. One increment might be newly written, another might be a modification of an existing module, and still another might be the incorporation of a module without change. For the chosen design there will be a few competing construction plans. At issue with each plan is the time schedule from increment to increment. Taking the greater risks earlier so that bad news surfaces early in the entire schedule is a good strategy. Getting more extensive certification of the components with lower reliability estimates is a possibility. An orderly and sensible accumulation of the increments into an ever more complete system is a consideration. The construction plan is mostly a matter of management judgment, with only a little help from the mathematics.

11. Use box structures to design the total system and all new modules. Use stepwise refinement and verification. Deliver increments for certification.

12.4.5.1 Certification Management

Application of the Certification Model is far more than simply predicting the mean time to failure of the final system. This is where the system development process is brought under statistical quality control. As management reviews the testing data reported for each increment, decisions must be made as to whether the results are good enough. If the data from an increment is not satisfactory, then the work goes back to design. But that isn't all that must happen. If work is sent back to design, then the original schedule may not be possible with the existing plan. It is time for a new schedule if the original plan is to be kept, or time for a new plan if the original schedule must be kept.

If an increment is rejected, management must discern the reason and fix the problem. The result might be a new plan with more increments that are each smaller. The result might be intensive training on team design and review with the increments unchanged. Deep problems of team training, experience, poor specification, *etc.*, will surface early while there is still time to effect a solution. Management doesn't have to wait until all the money and time are spent to learn that there is trouble. If a team produces several increments on schedule that produce strong test results, one would expect the repercussions of a disappointing increment in the middle or latter part of the project to be less serious.

A construction plan that is executed without a disappointing increment can be depicted as follows for a system of four increments:

I_1
$I_1, I_{1,2}$
$I_1, I_{1,2}, I_{1,2,3}$
$I_1, I_{1,2}, I_{1,2,3}, I_{1,2,3,4}$

But, if trouble is detected along the way the plan might look as follows, where I is the initial plan, J is a second plan stemming from a disappointing increment $I_{1,2}$ and K is yet a third plan.

I_1

$I_1, I_{1,2}$ (not acceptable)

I_1, I_1J_1

$I_1, I_1J_1, I_1J_{1,2}$

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2,3}$ (not acceptable)

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2}K_1$

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2}K_{1,2}$

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2}K_{1,2,3}$

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2}K_{1,2,3,4}$

$I_1, I_1J_1, I_1J_{1,2}, I_1J_{1,2}K_{1,2,3,4,5}$

None of the reliability data and processes used in the planning phase have anything to do with the reliability of the final system as it will be certified in the process that follows. This is the primary reason the above sequence of steps in the construction plan can satisfactorily use estimates and a model whose assumptions are difficult to meet. Certification of the final system will be based solely on (1) the statistical testing of the increments as they accumulate into the final product and (2) the underlying Certification Model.

Certification management steps are as follows:

1. When the system specification is available, the independent test team should determine the usage distribution. If the specification changes, the usage distribution should be reviewed for possible change.

2. Determine the definition of a test case if that is relevant to the way time is being recorded. If time is a simple measure of the cpu clock, clock on the wall, or calendar time then the definition of a test case might not be necessary.

3. Establish the oracle. Determine in advance what will be considered a successful test (anything else is a failure) of the software for each test case or each test stream.

4. Submit each increment to testing:

   a. When a failure is reported from testing, note the time (or test number) since last failure, examine the design, fix the fault and let testing continue on the new version.

   b. Monitor the trend of mean time to failure. Expect to see an exponential growth.

   c. Monitor the number of failures reported for the increment relative to expectations. If the data are satisfactory, work on new increments can continue or begin. Testing can continue for the increment in hand.

5. If the data from testing are not good enough, stop testing, make a new plan and re-deploy the team(s) to the new plan.

6. When the last increment is in certification testing, engineering changes will be made creating new versions of the program in the usual way. Management will be receiving estimates of the mean time to failure of the latest version based on performance of all previous versions in the increment. When the predicted mean time to failure has reached the target reliability, testing can stop and the system can be released to customer use.

12.4.5.2 Management Observations

Everything about Cleanroom encourages reuse, discourages modification, and encourages simple designs in new modules. Early in the use of these concepts, there will be few components in the repository that are either formally verified or empirically certified. Chief programmers will be confronted with the not inconsiderable chore of estimating the reliability of existing modules by conducting experiments or excavating usage histories that are poorly recorded. This will be a time consuming and expensive, but necessary, part of building a new system. A formally verified or carefully certified module will be of such value (time and budget) in building a new system that there will be every incentive to use it.

There is every incentive to use a certified module without modification, because to modify is to breach the warranty, so to speak. It will be better to write small pre and post processors, which might be verified and assigned r = 1 in the planning process. However, even modification and possible re estimation of reliability will be more economical than writing from scratch, because of the investment in usage profile, test definition and related testing paraphernalia that will represent a significant head start over working from scratch.

Because of the nature of the reliability allocation, the weak initial base of the repository, and the sensitivity analysis, newly written software will need to meet very high standards indeed to carry the burden of the total system reliability. This will reinforce the need for excellent work in specification, design and verification. These are circumstances under which Ada should be at its zenith relative to other languages.

Finally, a system so developed and certified will enter the repository as a strong candidate for future reuse. Records will be kept of the use of such a system and all the modules it reused in a continuing effort to document the correctness of the system.

12.4.6 Exercises

1. Given a module for potential reuse, but which has known failures, how would you examine its specification, design, and code to better ensure those failures were fixed without creating new ones?

2. Given a module for potential reuse, which has no known failures, how would you define continued testing to certify its correctness?

3. How does one "find the right balance in mathematical proof between formal procedure and economy of effort" in functional verification? How does program usage criticality enter the question and how might the same program be treated differently in different uses?

4. How does one "find the right test size" in certification? How does program usage criticality enter the question and how might the same program be treated differently in different uses?

5. How are construction plans developed? What partial functions are defined for feasible increment development? How do construction plans depend on the experience and abilities of the development teams?

6. Why are unmodified certified modules better to use in assembling programs that modified uncertified modules? How can small changes in modules lead to large changes in behavior?

12.5 Cleanroom Engineering In Retrospect

12.5.1 Reviewing Cleanroom Engineering

In retrospect, it is apparent that Cleanroom engineering provides a set of rigorous methods for *software development under statistical quality control*, based on sound mathematical and statistical principles. While millions of people are involved in software today, most of them regard software development as an intuitive, heuristic activity. They do not imagine software engineering as a mathematics based subject with complete rigor possible. But software engineering should be and can be such a mathematics based activity and when mathematical rigor is applied both software quality and engineering productivity increase. Neither do they imagine software engineering based on statistics since computers are completely deterministic in behavior. And yet the usage of software is statistical in nature.

Software is a human generation old, while mathematics is many human generations old. Although not understood early or widely, software has direct mathematics foundations because of the very deterministic behavior of computers. A computer program is a rule for a mathematical function, mapping all possible initial states into final states. Such functions are very complex compared to functions in physical science and engineering, and traditional mathematical notation is very insufficient. But sufficient mathematical notation is emerging in computer science and software engineering for dealing with the syntax and semantics of programs.

Ordinary arithmetic provides an example of deep and useful mathematics. Place notation and long division moved arithmetic from intuition and heuristics to rigorous methods a thousand years ago in the western world. As a result, school children today can out perform Archimedes and Euclid in arithmetic. Similar movement is possible in software today, in replacing intuition and heuristics by rigorous mathematical methods for software engineering. It will replace the ordinary programming of this first human generation by real software engineering in the next generation. It will replace today's expectation of software with a few failures that can never be completely removed to software with zero failures in user practice.

Statistics is another subject of longer professional development than software. But only a hundred years ago, statistics was intuitive and heuristic, even though rigorous arithmetic was used in creating sums and averages. Yet in this time, statistics has become a rigorous, mathematics based subject, often finding counter intuitive facts about statistics in specific topics. The application of statistics makes it possible for software engineers to predict with confidence the quality level of the software when it is fully developed quite early in the development life cycle.

Cleanroom engineering not only puts software development under statistical quality control, but takes debugging out of development before independent testing and certification, using mathematical reasoning instead. Just as typists looked at the keys when typewriters first came out, programmers have felt the need to debug programs in this first human generation of programming. But while counter intuitive at the time, typists went to touch typing with both higher productivity and fewer errors. In the same way, well educated software engineers can create software with no execution or debugging before its testing by independent test and certification engineers. Such engineered software will ordinarily have a few failures in early testing, which are returned for correction by the developers and re-testing. But the entire joint result of developers and certifiers is software with more productivity and much greater quality.

12.5.1.1 Cleanroom Engineering Processes

Cleanroom Engineering is a set of processes that help software engineers create low or zero defect software products (*i.e.*, users typically do not experience any failures). Features of Cleanroom Engineering include:

1. **Theoretically Sound Engineering Practices** - Achievement of intellectual control by applying rigorous, mathematically based engineering practices. These practices include:

    a. Development of a rigorous specification, even if it is preliminary,

    b. Construction of the software in a pipeline of user executable increments that accumulate into the system,

    c. Design of the software for each increment in three views (a black box for an implementation free specification, a state box for a data specification and a clear box for a process specification) with verification of the accuracy of each design expansion to ensure correctness,

    d. Implementation of the software in each increment by the stepwise refinement of clear boxes into executable code,

    e. Verification that the code performs according to its specification using proof conversations with functional verification arguments,

    f. Certification of each accumulated set of increments using usage scenarios generated at random from a distribution that represents the expected usage profile of the software,

2. **Professional Excellence** - Establishment of a "failures are unacceptable" attitude and a team responsibility for quality.

3. **Effective Use of People** - Delegation of specification, development and certification responsibilities to separate teams, which allows:

    a. precise specification of a pipeline of software increments that accumulate into the final software product, which includes the statistics of its use as well as its function and performance requirements;

    b. software development of each increment using box structured design and functional verification, delivery for certification with no debugging beforehand, and subsequent correction of any failures that may be uncovered during certification;

c. statistical certification of the software reliability for the usage specification, notification to designers of any failures discovered during certification, and subsequent recertification as failures are corrected.

The goal of Cleanroom Engineering is development of a higher quality software product. Specifically, this entails the prevention of defects in the software product, rather than the appearance and subsequent removal of defects in the software. The pleasant surprise with Cleanroom is that productivity also increases. This leads to a win-win situation - *improved quality and improved productivity.*

12.5.2 Specifications and Construction Planning

Precision in specifications requires formal languages, just as programming does. A formal specification defines not only legal system inputs, but legal input histories, and for each legal input history, a set of one or more legal outputs. Any such formal specification, in any language, is a mathematical relation–a set of ordered pairs whose first members are input histories and second members are outputs. Then, there is a very direct and simple mathematical definition for a program meeting a specification. It is that the program determines a correct output for every input history in the domain of the specification relation.

In Cleanroom engineering, specifications are extended in two separate ways to create a structured software development. First, the formal specifications are structured into a set of nested subspecifications, each a strict subset of the preceding subspecification. Then, beginning with the smallest subspecification, a pipeline of software increments is defined with each step going to the next larger subspecification. Second, the usage of the system specified is defined as a statistical distribution over all possible input histories. The structured Cleanroom process makes statistical quality control possible in subsequent incremental software development to the specifications. The usage statistics provides a statistical basis for testing and certification of the correctness of the software in meeting its specifications.

A structured software development defines not only what a software system is to be when finished, but also a construction plan to design and test the software in a pipeline of subsystems, step by step. The pipeline must define step sizes that the development team can complete without debugging prior to delivery to the certification team. Well educated and disciplined development teams may handle step sizes up to ten KLOC or more. But the structured design must also determine a satisfactory set of user executable increments for the pipeline of overlapping development and test operations. Early increments need not provide complete services for users, but partial services that can be tested before going on.

The first task of a Cleanroom project is to prepare and publish a specification for the software. The specification should be in three parts

The External Specification

The Internal Specification

The Expected Usage Profile

The *External Specification* is a user's reference manual. It defines all the interfaces with the software and its behavior. The *Internal Specification* is more mathematical in nature. We know that a program or a software system is a rule for a mathematical function or a state machine transition function to meet the External Specification. The *Expected Usage Profile* defines how it is anticipated the software will be used. The primary use of this document is to guide the preparation of usage tests.

12.5.2.1 Construction and Certification

In the specification phase the sequence in which the software will be developed and certified is determined. This is done by decomposing the specification into executable increments that will accumulate into the entire software which is to be delivered. An executable increment is software that can be tested by invoking user commands. The reason increments need to be defined so they are executable by user commands is to permit usage testing on a pipeline of increments as they accumulate into the system. Testing has two objectives: (1) to find the existence of any failures that would be observed by a user of the software and (2) to verify the correctness of the software with any failures found now corrected.

The goal of the software developer is to develop and release software that is correct when the software is used by its users. With the decomposition of the specification into software increments, implementation and testing can begin. Each increment is built and tested in turn. A development team is responsible for building the increments and a certification team is responsible for testing accumulations of increments. These two phases proceed in parallel.

As noted before, in describing the activities of the Software Construction phase, no mention was made of testing or even of compilation. The Cleanroom development team does not test or even compile. They use mathematical proof (functional verification) to demonstrate the correctness of program units that make up increments. Testing and measuring failures by program execution is the responsibility of the certification team.

Also, as noted before, in parallel with the Cleanroom development team, the Cleanroom certification team is preparing to certify the software up to and including the increment being developed by the development team. The certification team uses the usage profile and the portion of the External Specification that is applicable to the increments to be verified to generate random test cases including solutions. When the development team is finished, the certification team compiles the increment, adds the increment to previous increments, and certifies the software accumulated.

12.5.2.2 The Basis for Box Structured Design

Box structured design is based on a usage hierarchy of packages and subprograms. Such packages and subprograms, also known as data abstractions or objects, are described by a set of operations that may define and access internally stored data. In Ada, operations are defined by the calls of the procedures and functions of the packages, and internal data declared in the package.

Stacks, queues, and sequential or random access files provide simple examples of such packages or subprograms. Part of their discipline is that internally stored data cannot *be accessed or altered in any way except through the explicit* operations of the package or subprogram. It is critical in box structured design to recognize that packages may exist at every level from complete systems to individual program variables. It is also critical to recognize that a verifiable design must deal with a usage hierarchy rather than a parts hierarchy in its structure. A subprogram that stores no data between invocations can be described in terms of a parts hierarchy of its smaller and smaller parts, because any use depends only on data supplied it on its call with no dependence on previous calls. But a specific realization of a package, say a queue, can depend not only on the present call and data supplied it, but also on previous calls and data supplied then.

The parts hierarchy of a structured program identifies every sequence, alternation, and iteration at every level. It turns out that the usage hierarchy of a system of packages (say an object oriented design with all objects identified) also identifies every call (use) of every operation of every package. The semantics of the structured program is defined by a mathematical function for each sequence, alternation, and iteration in the parts hierarchy. That doesn't quite work for the operations of packages because of usage history dependencies. But there is a simple extension for packages that does work. It is to model the behavior of a package as a state machine, with its calls of its several operations as inputs to the common state machine. Then the semantics of such a package is defined by the transition function of its state machine (with an initial state). When the operations are defined by structured programs, the semantics of packages becomes a simple extension of the semantics of structured programs.

While theoretically straightforward, the practical design of systems (object oriented systems) in usage hierarchies can seem quite complex on first exposure. It seems much simpler to outline such designs in parts hierarchies and structures, for example in data flow diagrams, without distinguishing between separate usages of the same module. While that may seem simpler at the moment, such design outlines are incomplete and often lead to faulty completions at the detailed programming levels. In spite of their common use in this first human generation of system design, data flow diagrams should only be used within rigorous design methods rather than leaving critical requirements to details with incomplete specifications.

In order to create and control such designs based on usage hierarchies in more practical ways, their box structures provide standard, finer grained subdescriptions for any package of three forms, namely as black boxes, as state boxes, and as clear boxes, defined as follows.

> **Black Box**: External view of a package, describing its behavior as a mathematical function from historical sequences of stimuli to its next response.
>
> **State Box**: Intermediate view of a package, describing its behavior by use of an internal state and internal black box with a mathematical function from historical sequences of stimuli and states to its next response and state, and an initial internal state.
>
> **Clear Box**: Internal view of a package, describing the internal black box of its state box in a usage control structure of other packages; such a control structure may define sequential or concurrent use of the other packages.

As noted before, box structures enforce completeness and precision in design of software systems as usage hierarchies of packages. Such completeness and precision lead to pleasant surprises in human capabilities in software engineering and development. The surprises are in capabilities to move from system specifications to design in programs without the need for unit/package testing and debugging before delivery to system usage testing. In this first generation of software development, it has been widely assumed that trial and error programming, unit testing and debugging were necessary.

But well educated, well motivated software professionals are, indeed, capable of developing software systems of arbitrary size and complexity without program debugging before system usage testing. A few failures may be found in testing, but they can be readily fixed (by the developers) and the systems operate failure free from then on.

12.5.2.3 Functional Verification of Software

Once the design is complete, the clear box at each level is expanded to code to fully implement the defined rule for the black box function at that level. Following each expansion functional verification is used to help structure a proof that the expansion correctly implements the specification. The nature of the proof revolves around the fact that a program is a rule for a function and the specification for the program is a relation or function. What must be shown in the proof is that the rule (the program) correctly implements the relation or function (the specification) for the full range of the specification and no more. The proof strategy is subdivided into small parts which easily accumulate into a proof for a large program. Experience indicates that people are able to master these ideas and construct proof arguments for very large software systems.

The development team expands each clear box in the usage hierarchy into the selected target code using stepwise refinement and functional verification. As the development team designs and implements the software, it is held collectively responsible for the quality of the software.

12.5.3 Statistical Certification

Cleanroom statistical certification of software involves, first, the specification of usage statistics in addition to function and performance specifications. Such usage statistics provide a basis for assessing the reliability of the software being tested under expected use.

As each specified increment is completed by the developers, it is delivered to the certifiers, who combine it with preceding increments, for testing based on usage statistics. As noted, the Cleanroom structured specification must define a sequence of nested increments which are to be executed exclusively by user commands as they accumulate into the entire system required. Each subsequence represents a subsystem complete in itself, even though not all the user function may be provided in it. For each subsystem, a certification of correctness is defined from the usage testing and failures fixed, if any.

As noted before, it is characteristic that each increment goes through a maturation during the testing, becoming more reliable from corrections required for failures found, serving thereby as a stable base as later increments are delivered and added to the developing system. For example, the Cleanroom developed HH60 flight control program had three increments of about 11 KLOC each. Increment 1 required 27 corrections for failures discovered in its first appearance in increment 1 testing, but then only 1 correction during increment 1/2 testing, and 2 corrections during increment 1/2/3 testing. Increment 2 required 20 corrections during its first appearance in increment 1/2 testing, and 5 corrections during increment 1/2/3 testing. Increment 3 required 21 corrections on its first appearance in increment 1/2/3 testing. In this case 76 corrections were required in a system of some 33 KLOC, namely 2.5 corrections per KLOC for verified and inspected code, with no previous execution.

In the certification process, it is not only important to observe failures in execution, but also the times between such failures in execution of usage representative statistically generated inputs. Such test data must be developed to represent the sequential usage of the software by users, which, of course, will account for previous outputs seen by the users and what needs the users will have in various circumstances. The state of mind of a user and the current need can be represented by a stochastic process determined by a state machine whose present state is defined by previous inputs/outputs and a statistical model that provides the next input based on that present state.

As noted before, when the development team has completed an increment, the certification team creates an *accumulated* system up through this increment. The certification team compiles the increment, combines it with previous increments, and certifies the accumulated system through this version. If failures are encountered in the certification of a version, they are returned to the development team for analysis and engineering changes to whatever increments are causing the failures. While failures are likely to be caused by the latest increment added, previous increments may be at fault and changed as well, as noted in HH60 above. Each redelivery of changed increments defines a new version. If no failures are encountered in the certification of a version, no additional versions are required.

Within each version of the accumulating system, tests are constructed at random in accordance with the specified usage statistics profile and then exercised. Test results are compared to a standard and either a failure occurs or the result was correct. The execution time between successive failures, if any, is a sample of the MTTF for that version.

When common software development methods and skills are used across increments, the certification model can be used across versions with growing increments. With significant reuse of software packages and subprograms and independent estimates of correctness, specific statistical analyses should be employed. What follows assumes common methods and skills in the software development in illustration. For certain systems, other measurements than long term MTTF may be desired. For example, systems with independent start ups for specific missions, may be better estimated on the basis of fraction of total missions failure free. Such statistics can be developed for what is needed.

There is a paradox in statistical testing. If software has a large number of failures, the statistics can become more precise; if software failures are rare, or never occur, the statistics is very imprecise. The paradox is that the best software has the worst statistics. The response to this paradox is that the continued usage of the software is sampling itself and adds to the precision of the statistics. Even if software is entirely failure free, that fact cannot be established by statistical testing. But failure free usage of the software becomes part of the statistics.

It is characteristic that each increment matures during the testing, becoming more reliable from corrections required for failures found, serving thereby as a stable base as later increments are delivered and added to the developing system.

12.5.4 History of Cleanroom Engineering

The term Cleanroom is taken from the hardware industry to mean an emphasis on preventing failures to begin with, rather than removing them later (of course any failures introduced should be removed). Cleanroom engineering involves rigorous methods that enable greater control over both product and process. The Cleanroom process can not only produce correct software of high performance, but can do so with high productivity and schedule integrity. The intellectual control provided by the rigorous Cleanroom process allows both technical and management control.

As stated, Cleanroom engineering is a process for software development under statistical quality control. It recognizes software development as an engineering process with mathematical foundations rather than as a trial and error programming process. But it also defines a new basis for statistical quality control in a design process, rather than the well known idea of statistical quality control in manufacturing to accepted product designs.

In this first human generation of software development, such mathematical and statistical foundations have been little understood and used, particularly on large projects in which the very management of all the people required seemed of foremost difficulty. One generation is a very short time for human society to master a subject as complex and novel as software development. For example, after a single generation of civil engineering, the right angle was yet to be discovered. Although many more people are working in software engineering in its first generation, fundamentals and disciplines still take time for discovery, confirmation, and general use.

As noted above, in Cleanroom engineering a major discovery is the ability of well educated and motivated people to create nearly defect free software before any execution or debugging, less than five defects per thousand lines of code. Such code is ready for usage testing and certification with no unit debugging by the designers. In this first human generation of software development it has been counter intuitive to expect software with so few defects at the outset. Typical heuristic programming creates fifty defects per thousand lines of code, then reduces that number to five or less by debugging. But program debugging leaves deeper defects behind while doing so at least 15% of the time. A second major discovery is that defects due to mathematical fallibility in the Cleanroom process are much easier to find and fix than defects due to debugging fallibility, with the time and effort required literally reduced by a factor of five and deeper defects created much less often.

As noted above, the mathematical foundations for Cleanroom engineering come from the deterministic nature of computers themselves. A computer program is no more and no less than a rule for a mathematical function. Such a function need not be numerical, of course, and most programs do not define numerical functions. But for every legal input a program directs the computer to produce a unique output, whether correct as specified or not. And the set of all such input, output pairs is a mathematical function. A more usual way to view a program in this first generation is as a set of instructions for specific executions with specific input data. While correct, this view misses a point of reusing well known and tested mathematical ideas, regarding computer programming as new and private art rather than more mature and public engineering.

With these mathematical foundations, software development becomes a process of constructing rules for functions that meet required specifications, which needs not be a trial and error programming process. The functional semantics of a structured programming language such as Ada can be expressed in an algebra of functions with function operations corresponding to program sequence, alternation, and iteration. The systematic top down development of programs is mirrored in describing function rules in terms of algebraic operations among simpler functions, and their rules in terms of still simpler functions until the rules of the programming language are reached. It is a new mental base for most practitioners to consider the complete functions needed, top down, rather than computer executions for specific data.

The statistical foundations for Cleanroom engineering come from adding usage statistics to software specifications, along with function and performance requirements. Such usage statistics provide a basis for measuring the reliability of the software during its development, and thereby measuring the quality of the design in meeting functional and performance requirements. A more usual way to view development in this first generation is as a difficult to predict art form. Software with no known errors from coverage testing at delivery frequently experiences many failures in actual usage.

12.5.4.1 Two Sacred Cows of the First Human Generation in Software

Software engineering and computer science are new subjects, only a human generation old. In this first generation, two major sacred cows have emerged from the heuristic, error prone software development of this entirely new human activity - namely program debugging and coverage testing. As noted above, program debugging before independent usage testing is unnecessary and creates deeper failures in software than are found and fixed. It is also a surprise to discover that coverage testing is very inefficient for getting correct software and provides no capability for scientific certification of correctness in use.

As a first generation effort, it has only seemed natural to debug programs as they are written, and even to establish technical and management standards for such debugging. For example, in the first generation in typing, it only seemed natural to look at the keys. Touch typing without looking at the keys must have looked very strange to the first generation of hunt and peck typists. Similarly, software development without debugging before independent, certification testing of user function looks very strange to the first generation of trial and error programmers. It is quite usual for human performance to be surprising in new areas, and software development will prove to be no exception.

Just as debugging programs has seemed natural, coverage testing has also seemed to be a natural and powerful process. Although 100% coverage testing is known to still possibly leave failures behind, coverage testing seems to provide a systematic process for developing tests and recording results in well managed development. So it comes as a major surprise to discover that statistical usage testing is more than an order of magnitude more effective than coverage testing in increasing the time between failures in use. Coverage testing may, indeed, discover more errors in failure prone software than usage testing. But it discovers failures of all rates, while usage testing discovers the high rate failures more critical to users.

12.5.4.2 The Power of Usage Testing over Coverage Testing

The insights and data of Adams[1] in the analysis of software testing, and the differences between software errors and failures, give entirely new understandings in software testing. Since Adams has discovered an amazingly wide spectrum in failure rates for software, it is no longer sensible to treat failures as homogeneous objects to find and fix. Finding and fixing failures with high failure rates produces much more reliable software than finding and fixing just any failures, which may have average or low failure rates.

The major surprise in Adams' data is the relative power of finding and fixing failures in usage testing over coverage testing, a factor of 30 in increasing MTTF. That factor of 30 seems incredible until the facts are worked out from Adams' data. But it explains many anecdotes about experiences in testing. In one such experience, an operating systems development group used coverage testing systematically in a major revision and for weeks found mean time to abends in seconds. It reluctantly allowed user tapes in one weekend, but on fixing those failures, found the mean time to abends jumped literally from seconds to minutes.

The Adams data is given in Table 12.10. It describes distributions of failure rates in 9 major IBM products, including the major operating systems, language compilers, data base systems. The uniformity of the failure rate distributions among these very different products is truly amazing. But even more amazing is a spread in failure rates over 4 orders of magnitude, from 19 months to 5000 years (60 K months) calendar time in MTTF, with about a third of the errors having an MTTF of 5000 years, and 1% having an MTTF of 19 months.

After considerable study, it has been concluded that practically none of these failures were due to original programming, but rather were due to attempted fixes. It appears that 15% of fixes lead to new failures. That information changed the way IBM allowed fixes to be made. A simple fix of a low rate failure can easily bring out a high rate failure. So in place of fixing all failures, a rule was installed requiring any single failure to be observed several times before fixing it.

MTTF in K months								
60	19	6	1.9	.6	.19	.06	.019	Product
1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	.7
2	34.2	28.0	18.2	9.7	4.5	3.2	1.5	.7
3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	.4
4	34.2	28.5	18.7	11.9	4.4	2.0	.3	.1
5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	.7
6	32.0	28.2	20.1	11.5	5.0	2.1	.8	.3
7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	.6
8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
9	31.2	27.6	20.4	12.8	5.6	1.9	.5	.0

Table 12.10
Distributions Of Errors (In %) Among
Mean Time To Failure (MTTF) Classes

[1] E. N. Adams, "Optimizing preventive service of software products," IBM Journal of Research and Development, January 1984

With such a range in failure rates, it is easy to see that coverage testing will find the very low failure rate errors a third of the time with practically no effect on the MTTF by the fix, whereas usage testing will find many more of the high failure rate errors with much greater effect. Table 12.11 develops the data, using Table 12.10, that shows the relative effectiveness of fixes in usage testing and coverage testing, in terms of increased MTTF. Table 12.11 develops the change in failure rates for each MTTF class of Table 12.10, because it is the failure rates of the MTTF classes that add up to the failure rate of the product.

Property								
M	60	19	6	1.9	.6	.19	.06	.019
ED	33.2	28.2	18.7	10.6	5.2	2.5	1.1	.5
ED/M	.6	1.5	3.1	5.6	8.7	13.2	18.3	26.3
FD	.8	2.0	3.9	7.3	11.1	17.1	23.6	34.2
FD/M	0	0	1	4	18	90	393	1800

Table 12.11
Error Densities And Failure Densities
In The MTTF Classes Of Table 12.10

First, in Table 12.11, line 1, denoted M (MTTF), is repeated directly from Table 12.10, namely the mean time between failures of the MTTF class. Next, line 2, denoted ED (Error Density), is the average of the error densities of the 9 products of Table 12.10, column by column, which represents a typical software product. Line 3, denoted ED/M, is the contribution of each class, on the average, in reducing the failure rate by fixing the next error found by coverage testing (1/M is the failure rate of the class, ED the probability a member of this class will be found next in coverage testing, so their product, ED/M, is the expected reduction in the total failure rate from that class). Now ED/M is also proportional to the usage failure rate in each class, since failures of that rate will be distributed by just that amount. Therefore, this line 3 is normalized to add to 100% in line 4, denoted FD (Failure Density). It is interesting to note that Error Density (ED) and Failure Density (FD) are almost reverse distributions, Error Density about a third at the high end of MTTFs and Failure Density about a third at the low end of MTTFs. Finally, line 5, denoted FD/M, is the contribution of each class, on the average, in reducing the failure rate by fixing the next error found by usage testing.

The sums of the two lines ED/M and FD/M turn out to be proportional to the decrease in failure rate from the respective fixes of errors found by coverage testing and usage testing, respectively. Their sums are 77.3 and 2306, with a ratio of about 30 between them. That is the basis for the statement of their relative worth in increasing MTTF. It seems incredible at first glance, but that is the number!

To see that in more detail, consider, first, the relative decreases in failure rate R in the two cases:

```
Fix next error from coverage testing
R -> R - (sum of ED/M values)/(errors remaining)
= R - 77.3/E

Fix next error from usage testing
R -> R - (sum of FD/M values)/(errors remaining)
= R - 2306/E
```

Next, the increase in MTTF in each case will be

```
    1/(R - 77.3/E) - 1/R = 77.3/(R*(E*R - 77.3))
```

and

```
    1/(R - 2306/E) - 1/R = 2306/(R*(E*R - 2306))
```

In these expressions, the numerator values 77.3 and 2306 dominate, and the denominators are nearly equal when E*R is much larger than 77.3 or 2306 (either 77.3/(E*R) or 2306/(E*R) is the fraction of R reduced by the next fix and is supposed to be small in this analysis). As noted above, the ratio of these numerators is about 30 to 1, in favor of the fix with usage testing.

12.5.5 Summary

In summary, the Cleanroom engineering process develops software of certified reliability under statistical quality control in a pipeline of increments that accumulate into the specified software. In the Cleanroom process there is no program debugging before independent statistical usage testing of the increments as they accumulate into the final product. The Cleanroom process requires rigorous methods of software specification, development, and testing, through which disciplined software engineering teams are capable of producing zero defect software of arbitrary size and complexity. Such engineering discipline is not only capable of producing correct software, but also the certification of the software as specified.

As noted before, in Cleanroom engineering a major discovery is the ability of well educated and motivated people to create nearly defect free software before any execution or debugging, less than five defects/KLOC. In this first human generation of software development it has been counter intuitive to expect software with so few defects at the outset. Typical heuristic programming creates fifty defects/KLOC, then reduces that number to five or less by debugging. But such program debugging usually leaves deeper errors behind while doing so.

Software engineering and computer science are new subjects, only a human generation old. In this first generation, two major sacred cows have emerged from the heuristic, error prone software development of this entirely new human activity - namely program debugging and coverage testing. As noted above, program debugging before independent usage testing is unnecessary and creates deeper errors in software than are found and fixed. It is also a surprise to discover that coverage testing is very inefficient for getting reliable software in comparison with statistical usage testing (a factor of 30 in increasing MTTF). In addition, coverage testing provides no capability for scientific certification of reliability in use.

Of course, humans are fallible, even in using sound mathematical processes in functional verification, so software failures are possible during the certification process in the Cleanroom process. But there is a surprising power and synergism between functional verification and statistical usage testing. First, as already noted, functional verification can be scaled up for high productivity and still leave no more failures than heuristic programming often leaves after unit and system testing combined. And second, it turns out that the mathematics based failures left are much easier to find and fix during certification testing than failures left behind in debugging, measured at a factor of five in practice.

12.5.6 Exercises

1. How has mathematics been critical in civil engineering? How would civil engineering be practiced without Arabic numbers or analytic geometry?

2. How will mathematics be critical in software engineering? How is software engineering practiced without formal syntax and semantics for programs?

3. What are the three major activities in Cleanroom engineering? How do they fit together?

4. What is a construction plan in Cleanroom engineering? What are the criteria for building the plan?

5. What are the three main parts of a specification for software? How do they fit together and how are they invented?

6. How will Cleanroom engineering be accepted by people who like to debug programs with lots of failures?