IA

stichting
mathematisch
centrum

AFDELING INFORMATICA IN 2/73 MAY

ANDREW S. TANENBAUM
INTRODUCTION TO ALGOL 68

IA

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

ACM-Computing Reviews-category: 4.22

PREFACE

These notes on ALGOL 68 were originally presented orally to an undergraduate class in the spring of 1973. They were printed and distributed as the second chapter of the course lecture notes. This report is a reprint of that chapter. As a consequence the sections and figures are numbered as they were in the original.

The notes are intended as an introduction to ALGOL 68 for students having already had one introductory course in computer science. It is assumed that they are familiar with an algorithmic language such as BASIC, FORTRAN, PL/1, or ALGOL 60. It is hoped that even freshmen and sophomores can understand this report.

A.S. Tanenbaum
Amsterdam, May 1973

CHAPTER 2 - INTRODUCTION TO ALGOL 68

Throughout this book algorithms are given for various techniques such as evaluating arithmetic expressions and sorting a list of names. It is necessary to have some way of expressing these algorithms in a clear and unambiguous manner. Ordinary English is too verbose and ambiguous to be satisfactory, furthermore we would like to write algorithms in a way such that they can be carried out by a computer.

Anyone who thinks that English is suitable as a method for precisely describing things should ponder the various meanings of "You would never recognize little Freddy. He has grown another foot" or the following bulletin board advertisement "German shepherd dog for sale. He will eat anything and is especially fond of children" or the following newspaper headline, concerning the exhibition of a copy of a painting, "Parents, pupils, faculty enjoy reproduction". If writing unambiguous English were easy, lawyers would not write sentences like "This agreement shall be construed and interpreted according to the laws of the State of New York and shall be binding upon the parties hereto, their heirs, successors, assigns, and personal representatives; and references to the lessor and to the lessee shall include their heirs successors, assigns and personal representations". We will therefore use a programming language to express algorithms.

There already exist hundreds of programming languages, and the number is growing rapidly. Most of these, however, are for special applications, such as making computer generated cartoons, controlling equipment in an automated factory, or helping civil engineers design bridges. To express the algorithms in this book, we need a general purpose programming language in which we can easily describe complicated kinds of data and express operations on the data in a convenient form. FORTRAN and ALGOL 60 are widely used, but both are

restricted to handling very simple kinds of data, and performing very simple operations on that data. For example, neither FORTRAN nor ALGOL 60 can manipulate character strings conveniently.

There are two languages that come closer to what we need: ALGOL 68 and PL/1. PL/1 was designed by a committee, and it shows. Each committee member wanted his favorite feature included, and most of them were accepted. The result was an unwieldy language with a very large collection of features that do not fit together well. As a consequence it is difficult to learn.

ALGOL 68, on the other hand, was designed to be as "orthogonal" as possible. This means that there are a small number of basic ideas that can be combined in many ways to produce a highly expressive language whose parts fit together very well. Orthogonal design also means that almost any construction that both "makes sense" and is unambiguous is allowed. As a consequence ALGOL 68 is easy to learn, comprehensive and therefore is ideally suited to the teaching of computer science. In this book ALGOL 68 will be used for expressing algorithms.

The purpose of this chapter is to explain enough about ALGOL 68 that you can understand the algorithms given in this book. Features of ALGOL 68 not needed in this book, e.g. binary transput, heap generators, unions, and long reals will not be mentioned. You should be aware that the description which follows is by no means complete. If the reader wishes an introduction to the complete language, he should see either "An Informal Introduction to ALGOL 68" by Lindsey and van der Meulen or "an ALGOL 68 Companion" by Peck. (See bibliography) As a final note, we mention that the ALGOL 68 report is already famous for its obscure terminology, thus in some cases the terms used in this book differ from those in the report. We do this in order to make these terms easier to understand.

2.1 Some basic features of ALGOL 68

ALGOL 68 programs consist of a sequence of symbols, including the lower case letters a – z, the digits 0 – 9, certain special characters e.g. + – = ; , : < > ( ) [ ] , and certain words printed in boldface type such as BEGIN IF SKIP THEN TRUE ELSE FALSE FI END. When writing programs with pencil and paper, boldface is indicated by underlining the words e.g. begin if skip then true else false fi end. Comments, which begin and end with the # or ¢ symbol, may be inserted between any two symbols. The ALGOL 68 compiler does not process comments; they are for the purpose of helping other people understand what the program does. They should be used generously. Spaces and carriage returns (end of card) may be inserted between any two symbols to improve readability.

One of the most basic features of a programming language is the kind of data which can be expressed in it. ALGOL 68 provides a rich collection of data types, three of the simplest types being integers, characters and booleans. The computers memory is divided up into a large number of equally sized pieces, usually called words or bytes. The amount of space occupied by a datum (e.g. an integer) varies from computer to computer. Figure 2-37 illustrates a computer whose memory is divided into 8 bit bytes, in which booleans occupy less than 1 byte, characters occupy exactly 1 byte, and integers occupy 4 bytes. The memory space occupied by a boolean, character, integer, or other object is referred to its *location*. Thus there are boolean locations, character locations, integer locations, etc., which in general are different sizes. Each location has a unique numerical address which identifies it. The proper word used to distinguish integers from characters from booleans is *mode*. Integers, characters and booleans are 3 different modes. Thus, a mode is a kind of object. The concept of mode is the most important concept in the entire language, so it is worth dwelling upon this point for a little while. We can make an analogy between the computer's memory, which is divided up into locations of various sizes, each of which can accomodate exactly one object of

the mode appropriate for that location, and a hypothetical city, which is divided up into buildings of various sizes, each of which can hold only 1 object. One kind of location in the city is a doghouse, which can hold one object of mode dog. Another kind of location is a hangar, which may contain one object of mode airplane. Yet another kind of location is a church steeple which contains one object of mode bell. A fourth kind of location is a computer center which holds one object of mode computer.

Just as a memory location for an integer can hold any one of a variety of integers, (0, -3, and 1944 come to mind at the moment) a given doghouse may contain Jack's dog or Susan's dog or Bob's dog, but it may not hold an object of mode airplane. Similarly a computer center may contain an IBM computer or a CDC computer, but not an object of mode dog. An object is not permanently attached to its location. But can be moved to another location of the proper mode whenever needed.

A location in the computer's memory that can hold an integer is called an <u>integer variable</u>, while a location that can hold a character is a character variable, etc. A variable can be given an identifier so the programmer can refer to it by an easy to remember symbolic name instead of its (numerical) address. Thus the identifier is an alias for the address, not the object at that address. This is illustrated in figure 2-38.

The programmer informs the computer how many variables of each mode she needs, and their names, by <u>declarations</u>. A declaration specifies

1. The mode of the objects being declared
2. The identifiers for the objects

Figure 2-1 shows 3 declarations. The first declares 2 integer variables with identifiers "number of girlfriends" and "pelican count". The second declares 1 character variable whose identifier is "grade expected in this course". The third declares 2 boolean variables, with identifiers "voted in last election" and "likes mustard".

There are several things to note about these declarations. First, integer variable declarations start with INT (not INTEGER) while CHAR and BOOL are used for character and boolean variables respectively. Second, identifiers may be as long as desired, and may have spaces "inside" for readability. Third, each declaration declares variables of only 1 mode, but it may declare arbitrarily many variables of that mode. The various identifiers are separated by commas. Fourth, declarations are separated by semicolons. The declaration in figure 2-1 causes 5 locations in the computer's memory to be reserved, one for each of these 5 variables. Associated with each location is an identifier, (given in the declaration) which can be used in subsequent statements. The identifier is equivalent to the address of the location, not the contents of the location. This distinction is absolutely crucial. The identifier is a symbolic name for the numerical address of the location. Sometimes we will refer to "the object whose address is X" or "the object in location X". In all cases it should be remembered that X is a symbolic name for the address of a location containing an object of some specific mode, and not the name of the object itself. Thus is illustrated in figure 2-39.

An object of mode boolean has one of two values, either TRUE or FALSE. An object of mode character has a value equal to some character. The set of characters available is implementation dependent, but includes at least one set of letters, the digits, and some special characters. When characters are used as constants, they are written with quotation marks around them. An object of mode integer has as its value an integer.

2.1.1 Assignment statements

A basic statement is the assignment statement, which assigns a value to a variable. An assignment statement has 3 parts:

1. A destination (written to the left of the := symbol)

2. A becomes symbol, written :=

3. A source (written to the right of the := symbol)

The destination is an expression which when evaluated gives an address of a location. The destination is so named because an object is put in the address specified by it. The source is an expression which when evaluated specifies an object of the required mode. The simplest example of a destination is an identifier. The simplest example of a source is a constant. Thus

number of girlfriends := 3

is an assignment with source = 3 and destination = number of girlfriends. It is pronounced "number of girlfriends becomes 3". When this statement is executed, an integer with value 3 is put in the location whose address is "number of girlfriends". Since "number of girlfriends" identifies a location (i.e. is the address of a location) whose object must be of mode integer, and since 3 is an object of mode integer, everything is fine, and the assignment takes place.

Consider the statement

number of girlfriends := number of girlfriends – 1

The destination of this assignment is the location whose address is "number of girlfriends", as above, but the source is more complicated. The identifier "number of girlfriends" identifies a location, i.e. is equivalent to the address of some location, whereas the constant 1 is an object of mode integer. Surely it is not intended to subtract 1 from the address "number of girlfriends" and use that as the source? of course not. What is intended is that the current value of the object in the location whose address is "number of girlfriends" should be used, not the address itself, and from that 1 is to be subtracted. The result of the subtraction is to be placed in the address given by the destination. The operation of using the contents of a location instead of its address is called <u>dereferencing</u>. Dereferencing is illustrated in figure 2–40. Note that in the above assignment "number of girlfriends" is

dereferenced when used in the source but not when used in the destination. This occurs because the minus operator needs two integers as operands, not an address and an integer, while the destination must be an address. If the destination were dereferenced as well as the source, we would have nonsense:

3 := 3 − 1

One of the nice things about ALGOL 68 is that dereferencing happens automatically where it is needed, and does not happen (in fact is forbidden) where it should not happen. Those points where dereferencing is to occur are carefully specified in the ALGOL 68 report, and agree with one's common sense interpretation.

Most other programming languages do not make such a sharp distinction between the address of a location, and the contents of the location. As a consequence it is usually possible to make disastrous errors such as changing the value of 3 into 2 by an assignment, if done in a sufficiently subtle way (for example, supplying a constant as an actual parameter in a procedure call where an address should be). Because ALGOL 68 makes such a rigid distinction between the address of a location and the contents of the location, this kind of error is caught by the compiler in every case.

2.1.2 Conditional statements

Another kind of statement is the conditional, or IF statement. In this statement a condition is tested. If the condition is found to be true, then the statement following the THEN is executed. If the condition is not true, the statement following the THEN is skipped, and the statement following the ELSE is executed instead. Figure 2-2 shows an IF statement. The statement is executed as follows. "bottle has deposit" is the address of a location containing an object of mode boolean. Boolean objects are either TRUE or FALSE. A condition must be either TRUE or FALSE, so "bottle has deposit" is

dereferenced and the contents of its location is tested. If it is true, then the statement

price := cost of product + amount of deposit

is executed. Before the addition can be carried out, both "cost of product" and "amount of deposit" are dereferenced since we want to add two integers, not two addresses.

If "bottle has deposit" is false the statement

price := cost of product

is executed instead. This statement is executed by first dereferencing "cost of product" to get the object contained in it. A copy of this object is placed in the location whose address is price. Since a new copy of the object is created, the old one remains undisturbed in "cost of product". Figure 2-2 illustrates some other points. First, statements are separated from declarations and from other statements by semicolons. Second, no semicolon is placed at the end of the THEN part or ELSE part, although if either part contained more than 1 statement, these statements would be separated by semicolons. Third, notice that the third line contains 2 statements, and that the IF statement as a whole occupies 4 lines and is indented to improve readability. Fourth, the conditional statement is ended with FI, (IF spelled backwards). The reason for having an explicit symbol to end the IF statement will be discussed later.

Another form of the conditional statement has no ELSE part, as shown in figure 2-3. If the condition is not true, the statement following the THEN is not executed. In this example the condition is

number of legs > 50.

This is evaluated by first dereferencing the identifier "number of legs" to get the value of the integer at that address. This integer is then compared to 50. If it is 51 or more the centipede count is increased. If it is 50 or less, execution continues with the statement following the FI.

2.1.3 Input/output

ALGOL 68 allows a wide range of styles of input and output, ranging from a simple print(X) to sophisticated formatted transput operations on files, with user control over all error handling. In this book we will be content to use the simplest forms namely read(X), which reads in 1 object of the required mode (even an array) and stores it in the location whose address is X, and print(X) which prints the object whose address is X. If the thing in parentheses following print is a string of characters within quotation marks, the characters are printed, but the quotation marks are not printed.

AGOL 68 also allows x to be themselves a list of variables, thus permitting more than 1 variable to be read or printed with a single statement. Such a list must be enclosed in an additional set of parentheses. An example is

```
read((a, b, c, d)).
```

The statement

```
new line;
```

causes subsequent output to be printed beginning at the start of the next line. Successive uses of print cause items to be printed on the same line until there is no room left, in which case printing continues on the next line. The number of characters per print line varies from computer to computer.

2.1.4 Loops

Since performing a sequence of statements repeatedly is very common, ALGOL 68 provides a statement for controlling repetition. One form of it is shown in figure 2-4. The variable after the <u>FOR</u>, called the controlled

variable, is set to 1, a test is made to see if it is less than 100, and if so the statement after the DO is executed. Then i is incremented by 2 and the print statement executed again. For each successive value of "odd number" a test is made to see if it is greater than 100, in which case the repetition stops.

The controlled variable must not be declared. It is always an integer, so the compiler does this automatically. The expressions after FROM, TO and BY may be any expressions yielding an integer as a result. These expressions are evaluated before the loop is begun to determine how many times the statement following the DO is to be executed. If the values of the expressions subsequently change, this has no effect on the number of repetitions.

The FOR and the identifier following it may be omitted if the controlled variable is not needed. If the word FROM and the expression following it are omitted, the counting starts at 1. If the word TO and the expressions following it are omitted, the repeating continues until terminated by some other mechanism. If the BY part is omitted, a default of 1 is used.

In contrast with the above form of the FOR statement which causes a statement to be repeated a fixed number of times, there is another form that causes the statement following the DO to be repeated as long as some condition is true. The condition, written between WHILE and DO is tested; if it is true, the statement following the DO is executed. The test is then repeated, and if still true, the statement following the DO is executed again. This process is repeated until the condition yields false. Figure 2-5 shows how the smallest integer whose square is larger than 1000 can be computed, and stored in the location whose address is n. Common sense requires that the expression following the WHILE yield a boolean value, either TRUE or FALSE. As usual, ALGOL 68 obliges by allowing an arbitrarily complicated expression (even 100 pages long) provided the result is of mode boolean.

These two forms of repetition may be combined into one FOR statement

whose most general form is shown in figure 2-6a, where a, b, and c are expressions whose result is of mode integer, and condition is an expression whose result is of mode boolean. The execution starts with the evaluation of the expressions a, b, and c. The results are then copied to secret memory locations x, y, and z respectively so the programmer can not change them. Then i is set to x. If i is less than or equal to y and the condition is true, the statement is executed. Then i is increased by z and the test of i against y is repeated along with another test of the condition. If $i \leq y$ and the condition is still true the statement following the DO is repeated again. This process continues until either $i > y$ or the condition becomes false. Note that unlike a, b, and c which are evaluated once and for all at the beginning of the FOR statement (with the results stored away in x, y, and z for safe keeping) the condition is evaluated over again before each repetition. If a is initially greater than b or if the condition is initially false, the statement will not be executed at all. Note that if the expression c yields a negative integer, the counting is negative and the test becomes $i \geq y$ instead of $i \leq y$. Figure 2-6 shows several examples of FOR statements.

2.1.5 Compound statements, ranges and programs

The FOR statement allows only 1 statement to be repeated. Frequently it is desired to repeat a whole sequence of statements, not just one, so ALGOL 68 provides a mechanism for forming compound statements, which are treated as single statements. A compound statement, or technically a strong void closed clause, consists of the word BEGIN followed by a series of declarations and statements (which may include conditional statements, FOR statements, and even other compound statements) separated by semicolons. The last statement in the series must be directly followed by END, with no semicolon directly preceeding it. In place of BEGIN and END, left and right parentheses may be used. Figure

2–7 shows examples of FOR statements followed by compound statements.

Figure 2–7a contains a compound statement which will be repeated 10 times, with i taking on the values 1 through 10 on successive iterations. Each iteration prints one line, containing i, $i \uparrow 2$, and $i \uparrow 3$. Note that an arbitrary expression can be printed. The result of executing the FOR statement of figure 2–7a will be to print out a table of the integers from 1 to 10 along with their squares and cubes.

Figure 2–7b contains a FOR statement whose repeatable statement is itself a FOR statement. First i is set to 1, the default when no FROM part is given, then the statement following the DO is executed once. Since that statement is itself a FOR statement, executing it once involves repeated executing the statement after its DO until it is finished. Then i is set to 2, and the second FOR statement is executed once, meaning another 4 executions of the compound statement following it. In all, 8 lines will be printed like this

1 1

1 2

1 3

1 4

2 1

2 2

2 3

2 4

A compound statement may also include declarations as well as statements. A compound statement that includes 1 or more declarations is called a range. The variables declared within a range may only be used within that range (and other ranges that occur inside of it), and may not be used outside the range. Although it is possible to use the same identifier for

different variables declared in different ranges, the practice can be confusing. It is best to give each new variable a unique name.

A complete program consists of a single compound statement or range, i.e. it begins with BEGIN and ends with END (or left and right parentheses respectively, if one prefers). Figure 2-7c is a complete program to print out a series of numbers, each term of which (except the first 2 which are both 1) is equal to the sum of the 2 preceeding terms. The series stops just before a term ≥ 1000 would have been printed.

Figure 2-8 is a complete ALGOL 68 program with 6 conditional statements, 3 repetition statements, 11 assignment statements and 13 input/output statements. Note that print (pennies) prints out the value of the integer variable "pennies", whereas print ("pennies") prints out the 7 letters p e n n i e s.

2.2 Modes, objects and values

In the previous sections the modes integer, character and boolean were introduced. In the following sections a few more simple modes will be introduced and it will be shown how to build up new modes from the simpler ones.

2.2.1 Primitive modes

In addition to the modes integer, boolean, and character, ALGOL 68 provides a mode real, which is an attempt to model the real number system of classical mathematics. Real numbers are objects consisting of 2 integers, f and e, used to represent numbers of the form f × b ⋏ e, where b is the base of the number system. The domain of the objects of mode real is very wide, encompassing tens or even hundreds of orders of magnitude, depending on the

computer. The number of significant digits in a real is also computer dependent, but is frequently in the range 8 to 15 digits.

Another primitive mode is format. Objects of mode format are used to control the formatting of input and output. Another basic mode is bits, which allows the programmer to use machine words conveniently and efficiently for storing and retrieving information.

2.2.2 Procedure modes

A whole class of modes are the procedures. A procedure is a piece of program that takes n input parameters, $n \geq 0$, and (optionally) produces a result. For example, the absolute value function for integers is an object of mode PROC (INT) INT because it takes an integer as input and delivers an integer as result. Just as there are many distinct objects of mode integer, e.g. 0, 6, -17 and 2, there are many distinct objects of mode PROC (INT) INT. A procedure taking 2 integers as parameters and producing an integer as result e.g. a procedure "add" which adds two integers, is of mode PROC (INT,INT) INT. A procedure that has 2 integer parameters and produces a boolean as result (e.g. a procedure "less than" which yields true if the first parameter is less than the second, and false otherwise.) has mode PROC (INT,INT) BOOL. The mode of a procedure is written as the word PROC followed by the modes of its parameters in parentheses, followed by the mode of its result. Procedures with no result have the word VOID in place of the mode of the result. Furthermore, if the procedure has no parameters, the parentheses are omitted. Since there are an infinite number of combinations of parameters (because there is no limit to the number of parameters a procedure can have) there are an infinite number of procedure modes. A few procedure modes and examples of procedures that could be defined for them are shown in figure 2-9.

Unlike most programming languages, procedures in ALGOL 68 are objects

and can be manipulated like other objects. Procedure variables exist and can be assigned values, just like any other variable. If f is declared by

PROC (REAL) REAL f;

then f is the address of a memory location just the proper size for a procedure taking 1 real as parameter and producing 1 real as result. Of course procedures vary in size, but it is the compiler writer's responsibility to solve this problem. He might for example, put the procedure somewhere else in memory, and put only the address of the procedure in f. After the assignation

f := sin

Where sin is the usual trigonometric function, f(x) will compute sin(x). The ability to treat procedures like any other objects is very useful, and follows from the orthogonality of ALGOL 68.

2.2.3 Arrays

Many problems involve data organized into vectors or matrices. By a vector we mean a one dimensional sequence of objects of some mode; by a matrix we mean a two dimensional array of objects. The elements of a vector or matrix may be primitive elements, such as booleans, integers or characters, or they may themselves be composite objects. As an example, consider a model for a computer memory composed of 4096 16-bit words. A word can be regarded as a 16 element boolean vector, and the whole memory can be regarded as a linear sequence of 4096 words, i.e. a vector whose elements are boolean vectors.

The official ALGOL 68 term covering vectors, matrices, and 3 and higher dimensional arrays is multiple value, although we will use the term array for simplicity. An array is a collection of objects, all of which have the same mode. An array is itself an object and has a mode. Array variables exist, and may be declared and assigned values, just as with variables of any other mode.

A 1 dimensional array of integers has mode [] INT pronounced "row of

integer". A 2 dimensional array of integers has mode [,] INT, pronounced "row row of integer". A 3 dimensional array of integers has mode [,,] INT, pronounced "row row row of integer". In general the mode of an array is an open square bracket, followed by a number of commas equal to the dimensionality of the array minus 1, followed by a close square bracket followed by the mode of the objects comprising the array. Objects of different dimensions have different modes.

Declarations of array variables are slightly different than declarations of say integer variables. The reason for this is that to declare an integer variable, specifying the mode of the object and the identifier is enough. The ALGOL 68 compiler knows how much space reserve in memory for the object of mode integer. For an array variable, the situation is different. The compiler can only reserve enough space if told how much space to reserve, i.e. how many elements the array has. To declare a boolean array variable named rb, which is to contain a 1 dimensional boolean array whose elements are numbered 1 to 10 one writes

[1:10] BOOL rb

The lower and upper bounds are written inside the brackets, separated by a colon. The mode of the object contained in the variable rb is [] BOOL, not [1:10] BOOL, i.e. the bounds are not part of the mode.

Figure 2-10 shows several examples of declarations of array variables. In all cases the lower and upper bounds must be given. Each bound is an integer constant, or an expression yielding an integer. Bounds may be positive, negative or zero. In figure 2-10h the size of the array rrb depends upon the values of n1, n2, n3, and n4 at the moment the declaration is executed. The location whose address is rrb will contain enough space for (n2 − n1 + 1) × (n4 − n3 + 1) booleans.

If the word FLEX appears before the sub symbol, the size of the array may be changed during execution of the program. Since the new size may be

larger than the old size, the compiler must provide a means whereby the array can be automatically moved to a new location with enough room. As a consequence flexible arrays, as they are called, are not very efficient. One important use for them however is for variable length character strings.

2.2.3.1 Subscripting

The next step in learning how to use arrays and array variables involves learning how to assign values to array elements. If ri is a 1 dimensional integer array variable as declared in figure 2-10a, then ri[1] is the first element, ri[2] is the second element, and ri[k] is the k-th element. These are integer variables and can be used as destinations in assignment, or they can be dereferenced and used in sources, or as operands, etc. Using only 1 element is called subscripting. Using an element from an n dimensional array requires n subscripts. To set all 10 elements of the integer array ri to 0, we can write

FOR i FROM 1 TO 10 DO ri[i]:= 0

To set all 1000 elements of rrrb declared in figure 2-10c to TRUE, we can write

FOR i FROM 1 TO 10 DO FOR j FROM 1 TO 10 DO FOR k FROM 1 TO 10 DO rrrb[i,j,k]:= TRUE

FOR statements and arrays go together well. Of course individual elements can be assigned values separately. If rc is declared as in figure 2-10d,

rc[6]:= "x"

assigns the character x to the character variable rc[6]. The other elements of the array are unchanged.

It is also possible to assign all the elements of an array at once, e.g.

[1:10] INT x,y; FOR i FROM 1 TO 10 DO x[i]:= i;

y:= x

The latter assignment is equivalent to

```
FOR i FROM 1 TO 10 DO y[i]:= x[i]
```

2.2.3.2 Slicing

In addition to being able to manipulate the individual elements of an array by subscripting, subarrays can also be manipulated as a whole. Subarrays are properly called slices. Subscripting is a special case of slicing. Figure 2-11 shows the declaration of three 1 dimensional integer array variables, a, b and c. First the array a is initialized, then b and c are set equal to a. This means that the location b, which has room for 10 integers, is filled with 10 integers whose values are identical to the integers in location a, and similarly for c. The assignment

```
b[7:10]:= a[7:10]
```

does exactly what you expect; namely, it is equivalent to

```
b[7]:= a[7]; b[8]:= a[8]; b[9]:= a[9]; b[10]:= a[10].
```

The statement c:= b assigns the entire array b to c, i.e. it is equivalent to

```
FOR i FROM 1 TO 10 DO c[i]:= b[i]
```

A column of a matrix can be assigned to a vector as shown in figure 2-12. The reason this makes sense is that the destination is a 1 dimensional integer array variable, so the source must be an object whose mode is that of a 1 dimensional integer array. Unsliced, "mat" is a 2 dimensional integer array, but the value of the slice on the right hand side of the assignment is just the whole 8th column of mat, which is a 1 dimensional integer array, so the source is a 1 dimensional array. The assignment is equivalent to

```
FOR i FROM 0 TO 6 DO vec[i]:= mat[i,8]
```

We note here an important point about slices in assignments: the bounds in the source and the bounds in the destination must match, even if not shown explicity, as above.

2.2.4 Structures

Arrays are used to group together objects of the same mode. Structures are used to group together objects whose modes may or may not be identical. A structure is composed of 1 or more objects called fields each of which has an identifier (more correctly, a tag) associated with it called the field selector. Structures are themselves objects and have modes. The mode of a structure depends upon the mode of its fields and the names of its field selectors. Two structure modes are the same, if and only if the modes and selectors of the fields are the same. Structure variables exist and can be declared, and used, for example, as sources and destinations in assignments. An example of a structure variable declaration is:

```
STRUCT ([1:3] CHAR type, INT seats, BOOL jet, quiet) aircraft
```

This declares aircraft to be a variable capable of holding a structure whose first field is a 3 character string called type, whose second field is an integer called seats, and whose third and fourth fields are both booleans, called jet and quiet, respectively.

To use any of the fields of a structure, e.g. as a source, destination, or operand, one writes the field selector, followed by the word OF, followed by the name of the structured variable, e.g.

```
type OF aircraft := "747";
seats OF aircraft := 350;
jet OF aircraft := TRUE;
quiet OF aircraft := FALSE
```

assigns values to all 4 fields of the variable aircraft. The operation of

selecting one field of a structure is in fact called <u>selecting</u>.

The fields of a structure may be objects of any mode, including arrays, as above, and even other structures. An individual field may be changed without affecting the values of the other fields. Some properties of structures and arrays are compared in figure 2-13.

2.2.5 References

We have mentioned addresses of objects quite a few times so far, now it is time to consider them more carefully. An identifier has been considered synonymous with the address of the memory location into which an object of some specific mode can be put. In ALGOL 68 addresses are also objects, and can be handled as such. The mode of any address is "reference to" followed by the mode of the object contained in its location. Thus the address of a memory location containing an integer has mode "reference to integer".

We can now consider assignment statements to involve 2 objects, the destination, which must be of mode "reference to something", and the source which must be of mode " something", where something can be any allowed mode. In particular, something could even be of mode "reference to integer". ALGOL 68 allows "reference to integer" variables, which contain an object of mode "reference to integer". Thus a "reference to integer" variable contains as its object the address of an integer, whereas an integer variable contains an integer, not an address. Now of course, an address is just some bit pattern in the computer's memory and it might be the same bit pattern as some integer, but "reference to integer", "reference to boolean", and integer are all 3 distinct modes and cannot be mixed.

On the CDC Cyber series computers, an integer requires 60 bits and an address 18 bits. On the IBM 370's an integer is 32 bits, and an address is 24 bits. Objects of mode integer and mode "reference to integer" usually have

different sizes, a general characteristic of different modes.

An example of a "reference to boolean" variable declaration is

REF BOOL p

which declares p to be a variable which contains as its object the address of a boolean. Furthermore, p itself is an object of mode "reference to reference to boolean".

2.2.6 Mode declarations

The data used by an ALGOL 68 program consists of a collection of objects. Each object has a mode. Some modes are "built in" to ALGOL 68, e.g. INT, BOOL, CHAR, and REAL, while other modes can be constructed using PROC, REF, [], and STRUCT. To allow programmers to define their own modes and then declare variables of these modes, ALGOL 68 provides a way to invent new modes and add them to the language so they can be used just like the built in modes INT, BOOL, CHAR, and REAL. A mode is added to the language by a mode declaration. An example is:

MODE STRING = FLEX [1:0] CHAR

which declares STRING to be a mode. The bounds 1:0 imply that when a string variable is declared, initially no space is reserved for it, but because it is flexible, a "string" of characters of any size can be assigned to a string variable. This new mode can now be used just as if it were built-in. For example,

STRING s

declares s to be a string variable in exactly the same way that

INT k

declares k to be an integer variable. Since strings are so useful, this definition of string is in fact already built-in, so the programmer need not declare it herself.

Figure 2-14 shows several examples of mode declarations and figure 2-15 shows some declarations of variables using these modes. Variables of any mode may be initialized at the time of their declaration by following the name with a becomes symbol (:=) and an expression yielding an object of the required mode. Arrays and structures may also be initialized by listing their elements or fields inside parentheses. It is possible to partially initialize a newly declared object by writing SKIP in place of any (or all) elements or fields. The value of SKIP is undefined, and is only used when that element or field will later acquire a value by assignment.

The declaration of VECTOR in figure 2-14 causes x in figure 2-15 to become a 1 dimensional real array variable. Since n is 3 at the time x is declared (the value of n at the time the mode VECTOR is declared is immaterial) x has 3 elements. These are initialized by the row display (2.0, 3.0, 4.0) in figure 2-15, and cause assignment of values to the elements of the vector x just as if we had written

x[1]:= 2.0; x[2]:= 3.0; x[3]:= 4.0

The variable xx is a real 3X3 matrix variable, with mode row row of real. It is initialized as shown, where (1.0,2.0,3.0) is the first row, i.e. elements xx[1,1], xx[1,2], and xx[1,3]; (2.0,3.0,4.0) is the second row, i.e. elements xx[2,1], xx[2,2], and xx[2,3], and (3.0,4.0,5.0) is the third row.

The declaration of rat demonstrates the use of a structure display to initialize "numerator OF rat" to 2 and "denominator OF rat" to 7. The declaration of john illustrates another structure display. Since objects of mode PERSON have 5 fields, the structure display also needs 5 fields, each of the proper mode. For the declaration of john to be correct, bill and mary would have to be declared as PERSONs somewhere in the program. The declaration of jones shows how a structure one of whose fields is an array can be initialized. The expression

(nancy, peter)

is itself a row display, and thus can be used to initialize an array such as [1:2] PERSON child.

The declaration of jones raises an important point. Suppose it is desired to print out the name of child [1] of the variable jones. The expression

child OF jones

is a selection from a structured value, and is itself an array variable. As a consequence this array variable can be subscripted. However

child OF jones[1] (wrong)

is wrong becauses subscripting binds more tightly than OF. The expression child OF jones [1] would be correct if jones were an array of structures one of whose fields were child. In that case, jones[1] would be the entire first element (a structure) and the action of selection could be performed on that structure. The correct way to print the name of the first child (nancy) is

print(name OF (child OF jones)[1])

The parentheses force "child OF jones" to be subscripted. Since child OF jones is an array, that is fine. The expression

(child OF jones)[1]

is an object of mode PERSON, and can be selected from using the field selectors name, father, mother, age, and smokes.

It is instructive to compare the mode BRIDGEHAND to the mode FAMILY. North is initialized to a row display containing 13 elements, each of which is a 2 character structure display. Suppose we wish to print the rank of the first card. The expression

north[1]

represents a structure with 2 fields, rank and suit. We can select from it, so

print(rank OF north[1])

is correct. Here no extra set of parentheses is needed because subscripting binds more tightly than OF, and in this case that is what is needed. Extra

parentheses are always allowed however, so if one is unsure they can be used to avoid difficulty.

The variables w, reg, cc, and t are not initialized. The mode INSTRUCTION is a mode all of whose fields are of the same mode, thus it could have been declared as an array

```
MODE INSTRUCTION = [1:4] INT
```

instead of as a structure but to use the first element we must write

```
add[1]
```

instead of

```
opcode OF add
```

Which choice is made depends upon which form the programmer finds most convenient for the problem at hand. ALGOL 68 is very flexible, and often provides several ways of expressing equivalent ideas.

The declaration of twa520 illustrates the use of SKIP to initialize some fields of the structure, but not all of them. In particular 1 field, passenger, is not initialized at the time the variable is declared. When the name and phone of passenger[1] are known, they can be assigned by

```
name OF (passenger OF twa520)[1] := "tarzan";
phone OF (passenger OF twa520)[1] := 914 723 4567
```

What may appear complicated at first, will later be seen to be straightforward and simple. The key to writing expressions involving both selecting and slicing is to carefully note the mode of each expression. The variable twa520 is a structure so it must be selected from, not subscripted. Clearly, passenger is the field selector desired, not number, pilot, movie, or nonstop. The expression

```
(passenger OF twa520)
```

is an array, so we must subscript it, not select from it. We want the first passenger, so we write

```
(passenger OF twa520)[1]
```

Since (passenger OF twa520) is an array of structures, subscripting it gives a structure to be selected from, e.g.

name OF (passenger OF twa520)[1]

which can then be used as a destination, source, operand, etc. Anyone who still thinks this unnecessarily complicated should try expressing the same ideas in FORTRAN, ALGOL 60, or BASIC.

The mode TREE is interesting. It has 3 fields, an integer and 2 addresses. In terms of allocating space for TREE variables, it hardly matters that the addresses are addresses of other objects of mode TREE. Binary trees are a very useful data type in many areas of computer science, so modes such as this are very valuable. A mode which is defined in terms of itself is called a recursive mode. One must exercise some care when declaring recursive modes. For example

MODE BUSH = STRUCT (INT val, BUSH left, right) (wrong)

is wrong. Suppose that an integer requires 1 word of memory and a BUSH requires N words of memory. Then a declaration like

BUSH blueberry

would require enough space to be reserved in the computer's memory for one object of mode INT (1 word) and 2 objects of mode BUSH (2N words). This is a total of 2N+1 word for each object of made BUSH. But this contradicts our statement that a BUSH required only N words. The definition is impossible, since a BUSH can hardly contain an INT plus 2 BUSHes. The mode TREE presents no such problem since it only claims space for an INT and 2 addresses, not 2 objects of mode TREE. As you probably expect by now, ALGOL 68 allows essentially all modes that are reasonable and prohibits those that are not, but the formal test to see if a given mode is allowed is unfortunately too complex to be given here. Interested readers may consult van Wijngaarden (1968).

It is important not to lose sight of the fact that programmer created

modes like PERSON are used just as thought they were built in (see figure 2-15). The ability of a programmer to define modes suitable for her application is the most powerful feature of ALGOL 68. In a later section you will see how to define operators to perform actions on programmer defined modes. For example, in a program to play bridge, a monadic operator to count the number of points in an object of mode BRIDGEHAND might be useful.

Mode declarations can be used to define synonyms for modes. Thus users who like writing INTEGER instead of INT can declare them to be equivalent by MODE INTEGER = INT.

2.3 Units

We have used the term "expression" quite often so far without precisely defining what it is. Now we will examine the concept in detail. The ALGOL 68 term for the intuitive idea of "expression" is unit, which is short for unitary clause. A unit, when evaluated, yields a value which is an object of some specific mode. A unit which when evaluated yields an integer is called an integer unit. A unit which evaluates to a boolean is called a boolean unit; a unit that evaluates to the address of a character is called a reference to character unit, etc.

ALGOL 68 requires certain kinds of units in certain places.,for example, a subscript must be an integer unit. An integer unit is also needed after FROM, TO, and BY in a FOR statement. The condition following the IF in a conditional statement obviously must be a boolean, not an integer. The destination of an integer assignment must be the address of an integer location, i.e. a "reference to integer" unit.

There are many forms a unit can take. We will examine 9 of them. As an example we will show how the form under discussion is used as the source of an assignment, but of course units are used in other places as well. In the

examples i will be assumed to be an integer variable.

2.3.1 Denotations

The simplest form of a unit is a denotation, (called a constant in some programming languages). An example of the constant 3 used as a unit is

i := 3

The expressions on the right hand side of the := symbol in figure 2–15 are all denotations (although other units are also allowed) and therefore units. Constants for structure and row modes are also allowed, and consist of a list of constants. There is no official word for such constants, but we will use the term denotation to include them, although in a strict sense they are not denotations. A unit of a structure or row mode is called a display. Figure 2–16 shows the declarations used by figures 2–17 to 2–25.

2.3.2 Variables

The next simplest kind of unit is a variable. If j is declared by

INT j

then j is an address of an integer, and therefore has mode reference to integer. One might think that

i:= j

would be forbidden, since the source of an assignment to an integer variable must be an integer unit, not a "reference to integer" unit. But our old friend dereferencing comes to the rescue and dereferences j, turning it into an integer. Dereferencing, as you will recall, takes an address as input, and produces the object at the address as the result. If the object is a unit of mode something, then the address has mode "reference to something", so dereferencing turns an object of mode "reference to something" into an object

of mode "something". That is why it is called dereferencing: it removes a reference.

Dereferencing is an example of a coercion. A coercion is an action that replaces an object of one mode (usually the wrong mode) with a different object, hopefully of the right mode. Any object of mode "reference to something" can be dereferenced when it appears as the source in an assignment, or as an operand in a formula etc.

There is one other coercion that is worth mentioning (and 4 others that are not worth mentioning) called deproceduring. Deproceduring starts with a procedure whose mode is procedure something, and produces as its result an object of mode something. For example if r is a real variable, then

r:= random

(where random has mode PROC REAL) does not appear to be proper, since the source should be of mode REAL not mode PROC REAL. Deproceduring, like dereferencing happens automatically when needed. Coercions are a rather subtle idea and have more to do with the syntax of a program than with what it does. If you do not see why i:= j or r:= random would be incorrect without coercions, do not worry about it. It's really not very important. The discussion of coercions was included only because some clever readers may notice the apparent inconsistency which coercions solve. People who like this sort of thing will probably like the other 4 coercions as well. Figure 2-18 shows examples of variables of various modes.

2.3.3 Slices

The third form of a unit is a slice, which includes subscrpted expressions such as

(child OF jones)[1].

A slice has 2 parts, an array to be sliced, and the index or indices, enclosed

within square brackets. The expression above is considered to be a slice rather than a selection because slicing is the last operation performed, i.e. child OF jones is an array and it is sliced. If len is a 1 dimensional integer array, then

i:= len[1]

is an example of a slice being used as a unit. More examples are shown in figure 2-19.

2.3.4 Selections

The fourth form of a unit is a selection. Like a slice, a selection also has 2 parts, separated by OF. These parts are the field selector and the structure being selected from. The field selector must be an identifier and cannot be computed (because it is not an object). The structure being selected from may be the result of evaluating an expression. An example of a selection being used as a unit is

i:= age OF john

where john is declared in figure 2-15. More examples are shown in figure 2-20.

2.3.5 Procedure calls

The fifth kind of unit is a procedure call. A procedure call causes a procedure to be executed and (optionally) return a result. If the result is an integer unit, the procedure call can be used anywhere an integer unit is allowed. If the call yields a reference to boolean, the call can be used wherever a reference to boolean unit is required, or even where a boolean unit is required, because the result can be dereferenced.

A call has 2 parts, the procedure to be called, and the parameter list. The mode of the result of a procedure call can be found by looking at the mode

of the procedure called. Procedures with parameters always have modes of the form

PROC (mode of parameter 1, mode of parameter 2, etc) mode of result

or

PROC (mode of parameter 1, mode of parameter 2, etc) VOID

Calls of procedures of the second form cannot be used as units in sources, subscripts etc. Calls of procedures of the first form may be used anywhere a unit of "mode of result" is needed. As an example, if count is a procedure of mode PROC (INT) INT, i.e. it takes 1 integer as parameter and delivers an integer as result, then

i:= count(j)

illustrates a procedure call being used as an integer unit.

A procedure call is very similar to a deprocedured variable. The only difference is that procedure calls always have parameters, and deproceduring occurs only for procedures with no parameters. This distinction is needed to avoid certain ambiguities which can result if the result of a procedure is another procedure.

A procedure is the ALGOL 68 method of implementing the idea of a function in classical mathematics. A function in classical mathematics has 0 or more parameters and delivers a result. The ALGOL 68 concept of a procedure is more general, since a procedure may have no parameters and yield VOID instead of a value.

Examples of calls used as units can be found in figure 2-21.

2.3.6 Formulas

The sixth kind of unit is a formula. A formula consists of an operator and its operands(s). Monadic operators have only 1 operand, for example, ABS i has the value of the absolute value of i. Dyadic operators have 2 operands,

for example j+k is a formula in which the operator + has 2 operands. ALGOL 68 has well over 100 built-in operators, and the programmer can define new ones just as she can define new modes. An operand in a formula may itself be a formula, for example the formula 2×n, may be used as the right operand of + to yield another formula, e.g. j+2×n. Denotations, variables, slices, selections, and procedure calls (among other things) may also be used as operands, allowing very general formulas to be expressed. An example of a formula used as a unit is

i:= j+k

More examples of formulas as units are shown in figure 2-22.

2.3.7 Assignments

The seventh kind of unit is an assignment. An assignment can stand by itself as a statement, but it can also be used as a unit. When used as a unit, the value of an assignment is the value of its destination, not its source. In j:= k the value of the assignment is j, which is of mode "reference to integer".Thus j:= k can be used anywhere a "reference to integer" unit needed, or because it can be dereferenced, it can also be used anywhere an integer unit is needed. Consider the assignment

i:= j:= k

Here i is the destination and j:= k is the source. As a consequence of allowing assignments as units, ALGOL 68 gets multiple assignment statements as an extra added attraction, for free. The above statement is equivalent to the 2 assignments

j:= k; i:= j

but is easier to write. The reason that k is first assigned to j, then j is assigned to i, is that j:= k is the source of the assignment to i. Before an assignment can be performed, the source and destination of the assignment must

be evaluated, and as a "byproduct" of evaluating the source j:= k, k is assigned to j. See figure 2-23 for more examples of assignments as units.

2.3.8 Closed clauses

The eighth kind of unit is a closed clause. A serial clause consists of a series of zero or more statements and/or declarations followed by a unit. A serial clause has the mode and value of its final unit. A closed clause is a serial clause enclosed by either BEGIN END or by parentheses. A closed clause has a mode and a value, namely the mode and value of the unit at the end of its serial clause. For example

```
(read(j); j:= j+3; IF j < 0 THEN j:= 0 FI; j+1)
```

is a closed clause, hence a unit. It's serial clause ends with the formula j+1, so the value of the closed clause is j+1, which is an integer unit. This closed clause may be used anywhere an integer unit may be used (even if it seems somewhat strange at first). For example,

```
i:= (read(j); j:= j+3; IF j < 0 THEN j:= 0 FI; j+1)
```

is a perfectly valid assignment, which may either stand alone as a statement, or be used as a unit. The above assignment is evaluated in 5 steps

1. j is read in
2. j is increased by 3
3. if j is negative it is set to 0
4. j+1 is computed (but j is not changed)
5. the integer computed in step 4 is stored in i

It should be noted that a serial clause need not have any statements or declarations, therefore a unit all by itself is also a serial clause. A

unit in parentheses is a serial clause in parentheses, so it is a closed clause hence a unit. Therefore unnecessary parentheses around units are allowed, and are sometimes useful for improving readability. Figure 2-41 illustrates the relation between a unit, a serial clause, and a closed clause. Examples of closed clauses are shown in figure 2-24.

2.3.9 Conditional clauses

The ninth kind of unit is a conditional clause. A conditional clause consists of an IF part, a THEN part, and an ELSE part. The THEN and ELSE parts consist of the words THEN and ELSE respectively, followed in each case by a serial clause. The two serial clauses must either be of the same mode, or be coerceable to the same mode. The mode of a conditional clause is the mode of its serial clauses, or if they have different modes, the mode to which they may be both coerced. For example,

i:= IF i < j THEN k ELSE read(n); n+1 FI

is a valid assignment. If i is less than j, k is assigned to i, otherwise n is read in and n+1 is assigned to i. The formula n+1 at the end of the ELSE part does not change the value of n of course, anymore than i:= n+1 would change n.

The restriction that both serial clauses must be or be coerceable to the same mode is needed to avoid nonsensical assignments. Consider the meaning of

i:= IF k<0 THEN 4 ELSE TRUE FI (wrong)

If k is less than 0, i becomes 4. If k is greater than or equal to 0, the statement requires assigning a boolean value, TRUE, to an integer variable, which is impossible. On the other hand,

i:= IF k<0 THEN 4 ELSE i FI

is fine. The denotation 4 is of mode integer, while the object i is of mode "reference to integer", but it can be dereferenced to produce an object of mode integer, so both serial clauses can be coerced to mode integer.

A conditional clause and a conditional statement are really slightly different forms of the same beast. The difference is that while a conditional clause always has a value, a conditional statement stands alone, so it does not need a value. The ELSE part of a conditional clause is not strictly required, but it hardly makes much sense to write

```
i:= IF i<j THEN k FI
```

since if $j \geq i$ the result will be undefined.

The reason that FI is required to terminate conditional clauses and statements can now be given. If no FI were required, then

```
IF i=0 THEN IF j=0 THEN j:= j+1 ELSE i:= i+1
```

would be ambiguous. It might mean

```
IF i=0
    THEN IF j=0 THEN j:= j+1 ELSE i:= i+1 FI
FI
```

or it might mean

```
IF i=0
    THEN IF j=0 THEN j:= j+1 FI
    ELSE i:= i+1
FI
```

which have very different meanings. In the first interpretation, if i is not zero the statement is finished and nothing happens. In the second interpretation, if i is not zero then it is increased by 1. This is the famous "dangling else" problem. Some programming languages solve it by not permitting IF statements in then parts. Others solve it by arbitrarily declaring one interpretation or the other to be correct. FORTRAN solves the problem by not allowing else parts at all. That certainly avoids the ambiguity, but unfortunately it also makes programming very difficult. The ALGOL 68 solution of requiring conditional statements and to end in FI is symmetric, elegant, and always unambiguous.

Examples of conditional clauses are shown in figure 2-25.

2.4 Where units and serial clauses are allowed

In the preceeding sections, 10 different kinds of expressions have been introduced: denotations, variables, slices, selections, procedure calls, formulas, assignments, closed clauses, conditional clauses and serial clauses. ALGOL 68 programs are built from statements that use these and a few relatively unimportant other kinds of expressions. Not every kind of expression can be used everywhere, however, since ambiguities would result if this were allowed. For example, assignments are not allowed as operands of formulas. Consider what would happen to the assignment

i:= j+1

if j, which is an operand of +, were replaced by the assignment k:= 1. We would have

i:= k:= 1+1

which is allowed but is not what was intended. It sets k to 1+1, i.e. to 2, then sets i to k, also 2. If instead of writing the assignment k:= 1 as the left operand of + we write the closed clause (k:= 1) we get

i:= (k:= 1) + 1.

The above first sets k to 1, and then i to 2, which is quite different than the previous expression. To avoid this sort of ambiguity, ALGOL 68 only allows constructions in positions where no confusion can arise.

Figure 2-26 shows a number of syntactic positions within programs where expressions are required. For each syntactic position the kinds of constructions that are allowed are shown at the right. Thus after IF or WHILE a boolean serial clause, or any kind of boolean unit (or something coerceable to a boolean unit) will suffice. We emphasize that a unit yielding a "reference to boolean" object, e.g. a variable such as q, is quite acceptable in a position requiring a boolean unit, since it can be dereferenced, yielding

a boolean. An expression which can be coerced to the proper mode is always acceptable.

A serial clause consists of zero or more statements followed by an unit, so every unit is also a serial clause, although serial clauses are in general not units. In other words, "serial clause" encompasses more than "unit" so it is redundant to list both of them together. Whenever a serial clause is allowed, a unit is certainly allowed. Nevertheless unit is listed too as a reminder, where appropriate.

2.5 Procedures

The most powerful technique for writing a large or complicated program is breaking it up into a number of smaller, and conceptually simpler pieces, called procedures. Some people prefer the term subroutine instead of procedure; both are widely used. They will be used interchangeably in this book, in accordance with common usage.

A procedure is used to perform some logical task, for example computing the value of sin or arctan or the cube root of some input value, called a parameter or argument. If the procedure has only 1 result, the result can be returned as the value of the procedure. Alternately, the procedure can change one of its parameters, e.g. set a variable to the answer.

In ALGOL 68, procedures are objects and have modes and values just as other objects. The value of a procedure is a piece of program that performs some computation and possibly returns some value. An example of a procedure variable declaration initialized to a procedure that determines if its second parameter is 1 larger than its first parameter is

```
PROC (INT,INT) BOOL adjacent:= (INT i,INT j) BOOL: i+1 = j
```

Notice that this declaration has the same form as all the other declarations. First is the mode, in this case PROC(INT,INT) BOOL because the procedure has 2

integers as parameters and delivers a boolean as result (meaning a call to this procedure may be used anywhere a boolean unit is required). Following the mode is (as usual in all declarations) the identifier which identifies the procedure. This is then followed by a becomes symbol and the initial value of the procedure. Compare the structure of the above procedure declaration to the integer declaration

```
INT j:= i + 1
```

The right hand side of a becomes symbol in an integer declaration is an integer unit. Logically the right hand side of a becomes symbol in a PROC(INT,INT) BOOL declaration should be a PROC(INT,INT) BOOL unit, which it is. We will now describe what a unit for a procedure is.

Units for procedures begin with a list of the modes of the parameters, each of which is followed by an identifier called a formal parameter. The entire list of formal parameters is enclosed in parentheses and followed by the mode of the procedure's result. This is followed by a colon. The colon is followed by a unit of the mode of the result. The value of this unit is the value of the procedure, so naturally it must be the same mode as the result. In the above procedure declaration, j = i+1 is a formula, hence a unit, which has the value TRUE if j and i+1 are equal and FALSE if j and i+1 are unequal. Thus j = i+1 is a boolean unit, which it should be (since the mode of the procedure's result is boolean). Note that := is the becomes symbol, while = is the equality symbol.

An example of a procedure call to adjacent is

```
read(n); read(k); IF adjacent(n,k) THEN print("ok") FI
```

which reads in 2 integers and prints ok if the second integer is equal to the first integer plus one. The call adjacent(n,k) produces a boolean, so it can be used after IF, where any boolean unit or serial clause may be placed.

The call adjacent(n,k) tests to see if n+1 = k. It does not test to see if k+1 = n. The reason has to do with the way actual parameters are accessed

by procedures. At the time a procedure is called, space is reserved in the computer's memory for the parameters. The number and mode of the parameters can be determined from the mode of the procedure. Thus a

PROC(INT,INT) BOOL

has 2 integers as parameters, whereas a

PROC([] BOOL,CHAR,REF INT) INT

has a 1 dimensional boolean array, a character, and the address of an integer as its 3 parameters. At the time of the call, copies of the actual parameters are made and put into the space reserved for them. The first actual parameter can be accessed by using the identifier of the first formal parameter. The second actual parameter can be accessed by using the identifier of the second formal parameter, and so forth. The order in which the actual parameters are listed is thus very important.

A very important point is that parameters are objects not variables. In the declaration

INT i

i is an integer variable, that is i itself is an address of an integer, not an integer. Because it is an address of an integer, i.e. i has mode "reference to integer", it can be used as a destination in an assignment, however, the declaration

PROC(INT) INT p1:= (INT i) INT : i:= i+1 (wrong)

is incorrect because here i is an integer, not an integer variable. Suppose p1 is called by p1(n). At the time of the call the following things happen (conceptually a clever compiler may be able to do some optimization). First n, which is the address of an integer, is dereferenced yielding an integer. A copy of this integer is then placed in the space reserved for it by procedure p1. The identifier i identifies the integer itself, and not its address. As a consequence, the object i has mode integer and not mode "reference to integer", so it cannot be changed. By declaring a formal parameter to be of a

mode not starting with "reference to", one can protect the corresponding actual parameter from being accidently changed. This helps catch programming errors.

Of course if it is desired to change i, we can write

```
PROC(REF INT) INT p2:= (REF INT i) INT : i:= i+1
```

A call of p2(n) will cause the address of n to be copied into the space reserved for i. No dereferencing happens in this case because the formal parameter is of mode "reference to integer" and the actual parameter is also of mode "reference to integer". Thus a copy of the address of n is made, not a copy of the integer of which n is the address. When p2 is executed the formula i+1 is evaluated by fetching i, i.e. the copy of the address of n. This object is then dereferenced because + requires integers not addresses, as operands. The process of dereferencing i fetches the object whose address is i, namely the contents of n. The addition is performed and the result is stored back into n. The result of p2(n) is n:= n+1.

Three facts about parameters will be repeated for emphasis:

1. An actual parameter (after coercion) must be of the same mode as the corresponding formal parameter.

2. A formal parameter has the mode appearing in front of it. A formal parameter written as INT i really is an integer, not a "reference to integer".

3. A copy is made of the actual parameter, after coercion. Accesses by the procedure to the formal parameter are to this copy, not the original.

The examples given above are all very simple. A more common situation is a procedure whose unit (i.e. its body) is a closed clause. Figure 2-27 shows a complete program that reads in 20 integers comprising 2 vectors of

length 10 and prints their inner product, i.e.

a[1] × b[1] + a[2] × b[2] + a[3] × b[3] + ... + a[10] × b[10]

We note several things about this program. First, the formal parameters y and z are written after the same mode declarer. This is an alternate form which is easier to write than

([] INT y,[] INT z).

Second, the modes of y and z do not have upper and lower bounds specified. Modes never have bounds, although of course variable declarations such as a and b do have bounds. Third, the integer unit comprising the procedure body is a closed clause, whose serial clause ends in a variable, answer. The mode of answer is reference to integer, but is dereferenced to give an integer.

The declaration of a procedure is somewhat wordy, since the modes of the formal parameters are listed twice. If there are many parameters, this can be a nuisance. Thus ALGOL 68 provides another form for procedure declarations, namely the left hand side of the := is replaced by the word PROC and the procedure name, and the := is changed into an = to indicate the alternate form is being used. Strictly speaking, an object declared by this alternative is not a procedure variable, i.e. it does not have mode "reference to procedure something", but is an object of mode "procedure something". For our purposes the 2 forms are close enough. Figure 2-28 shows innerproduct in this alternate form. We will use this simplified form throughout the book.

We now consider a final item. A procedure which returns no explicit value has mode VOID for its result. A call of such a procedure cannot be used as a source, or a destination, or an operand, or anywhere an object of some mode is needed, however, it can be used where a statement is needed such as in a serial clause.

2.6 Operators

Consider the program of figure 2-29. It declares a mode VECTOR, and a procedure to add 2 vectors, and then uses those definitions to add 4 vectors and print the result. The statement

```
v5:= vectoradd(vectoradd(vectoradd(v1,v2),v3),v4)
```

although ghastly to look at is quite correct, since an actual parameter may be any kind of unit, (see figure 2-26) and a procedure call, such as vectoradd(v1,v2) is certainly a unit. Hence vectoradd(v1,v2) may be used as an actual parameter.

The difficulty with the above expression is that although perfectly acceptable to the computer, we mere humans are accustomed to a notation in which the operator comes between the operands, not in front of them. ALGOL 68 comes to the rescue once more by allowing us to define new operators. A formula in ALGOL 68 is really just a procedure call in a different notation.

Consider the operator + in the integer formula

```
1 + 2
```

as compared to the operator + in the real formula

```
1.5 + 2.5
```

They are actually different operators with the same symbol, +. The first operator takes 2 integers and performs an integer addition, yielding an integer. The second operator adds 2 reals, yielding another real. On almost all computers different hardware instructions are provided for performing integer and real arithmetic, but this causes no problem for the ALGOL 68 compiler, which merely examines the modes of the operands to determine which operator is intended.

There are 2 kinds of operators, monadic, which take 1 operand (as in ABS i) and dyadic, which take 2 operands (as in i-j). The definition of a

monadic operator specifies the mode of its operand and the mode of its result. The definition of a dyadic operator specifies the modes of both of its operands, and the mode of its result. When the ALGOL 68 compiler finds an operator it must inspect its operator definition table to find out which definition is appropriate for that operator's operands. In this way + can be defined to signify one operation for integer operands, another operation for real operands, a different operation for 1 dimensional integer array operands, and still another operation for 2 dimensional boolean arrays. In fact, the programmer may define as many other operations on as many other distinct pairs of modes as she wishes. Note that + defined to operate on an INT as left operand and a REAL as right operand is distinct from + defined to operate on a REAL as left operand and a INT as right operand. Both of these operators are again different from + defined to operate on 2 INTs or 2 REALs. If i is an INT variable and x a REAL variable, the 4 formulas

i+1, i+x, x+i, x+x

all use distinct built-in operator definitions. As if this generality and power were not enough, ALGOL 68 also allows the programmer to change any of the built in definitions. Thus a programmer who desired could redefine + to mean addition on integers and subtraction on reals.

Figure 2-30 shows figure 2-29 redone using an operator instead of a procedure. An operator definition consists of the word OP, followed by the operator (which may be a BOLDFACE word), followed by an equals sign. The text to the right of the equals sign is exactly the same as that for a procedure definition.

As another example, below is an operator definition which declares ∀ to be the same as × for integers, so i∀j means the same as i×j (although ∀ is still undefined for reals).

```
OP ∀ = (int i,j) int : i×j
```

In formulas like

i+j × k

we know that the multiplication is performed before the addition because multiplication has a higher priority than addition. ALGOL 68 allows the priority of dyadic operators to be changed, and the priority of new dyadic operators to be defined. Dyadic operators have priority between 1 and 9, inclusive. Monadic operators have priority 10, which implies that

–1 ⩚ 2

(where ⩚ means exponentiation) has the meaning

(–1) ⩚ 2 and not – (1 ⩚ 2)

The following priority declarations have the effect of causing addition to bind more tightly than multiplication

PRIORITY + = 7, × = 6

Thus after these declarations, 2+3×4 evaluates as (2+3)×4 which is 20.

A very large number of operator definitions in ALGOL 68 are built-in. Figure 2–31 lists a few of the more important ones.

There are 2 quasi operators LWB and UPB that are also useful. Both are monadic and both operate on all 1 dimensional arrays. The value of LWB i1 is the lower bound of i1, and the value of UPB i1 is the upper bound of i1. These are particularly useful in procedures and operators that have a 1 dimensional array as parameter. They allow the procedure to determine the bounds of the array, so that every element of the array may be accessed. Figure 2–32 illustrates their use. First n is read in, and then i1 is declared to have n elements. The monadic operator BIGGEST needs to know how many elements are contained in its parameter so it will be able to test all of them to find the biggest one. The value of the operator is the value of the largest element. These operators are quasi operators because they are automatically defined on all 1 dimensional array modes. Normal operators have to be defined separately for each mode. For example

[4:9] STRUCT (INT a,b) s

is an array of 6 structures and we can write LWB s, which has the value 4.

2.7 Serial, collateral, and parallel actions

In general, statements are executed one after another in the order written. The semicolon can be regarded as a go-on operator, which causes execution to continue. In some situations, however, there is no inherent sequencing. For example there is no reason for the first unit in a row display to be evaluated before the last one. Nor is there any reason why the left operand of a dyadic operator should be evaluated before the right operand. In some other programming languages units are evaluated left to right but nothing in classical mathematics suggests any precedent for this.

In formulating ALGOL 68, the designers intentionally specified that the order of evaluating certain things, such as the left and right operands of an operator, be undefined. This was done for three reasons. First, it discourages programmers from making use of the order, since they do not know what it is, and in fact it need not be consistent. Programs which execute differently depending on the order in which things are evaluated are bad programs. They are difficult to understand and are not likely to give the same result on all computers. An example of a program whose result depends upon the order of evaluation of operands is

```
BEGIN INT i,j; print((read(i); i) - (read(j); j) END
```

If the input data is 1 followed by 2, then -1 will be printed if the left operand of the - operator is evaluated first and +1 if the right operator is evaluated first.

The second reason the order of evaluation is intentionally undefined is to give the ALGOL 68 compiler writer the freedom to do evaluations in the most efficient order. In some situations doing something in one order may be

preferable to doing them in another order, and if the compiler writer were forced to do everything strictly left to right then he could not take advantage of these situations to produce faster machine code. For example, consider the serial clause

<u>INT</u> i,j,k; read(i); j:= −1; k:= p(i) + p(j)

On a computer with one fast register used for arithmetic, after evaluating j:= −1 the register contains the value of j. It might be more efficient to evaluate the right operand of + before the left, since j is already in the fast register. However, if the language had specified that operands are evaluated left to right, the compiler writer would have no choice but to evaluate the procedure call p(i) first, even though it is less efficient.

The third reason for having the order of evaluation of certain constructions undefined is that some computers have more than 1 processor, and are thus capable of performing more than 1 computation at a given time. The evaluation of i−j is a trivial case, since the only action required to evaluate each operand is dereferencing, but it is quite possible that each operand of some dyadic operator could be a closed clause 100 pages long. Or more importantly a row display with 64 units might consist of 64 closed clauses, each 10 pages long. If such a program were run on a computer with 64 processors, it would clearly be terribly inefficient to require that closed clause n be completely evaluated before the evaluation of closed clause n+1 begins. Obviously it would be much better to give each processor its own closed clause to evaluate, so they could be evaluated in parallel.

Actions that have no specified ordering in time are said to be evaluated <u>collaterally</u>. A <u>collateral clause</u> is a list of units separated by commas, and enclosed by <u>BEGIN</u> <u>END</u> or parentheses. The order in which the units of a collateral clause are evaluated is expressly undefined. Figure 2–23 lists some evaluations that are performed collaterally. If the elements of the collateral clause are all statements, i.e. void units, the collateral clause

may be used in any position in which a statement is allowed. Thus

```
BEGIN i:= 1, j:= 2, k:= 3 END
```

is a void collateral clause and can be used just like an ordinary statement, just as though the commas were semicolons.

Consider the following two programs, the first of which contains a closed clause and the second of which contains a collateral clause.

```
BEGIN INT i:= 0; (i:= i+1; i:= i+1); print(i) END
BEGIN INT i:= 0; (i:= i+1, i:= i+1); print(i) END
```

The only difference is that in the first clause the assignment statements are separated by a semicolon and in the second one they are separated by a comma, only one tiny spot of ink difference in appearance, but a very large difference in meaning as we shall see.

The first program prints 2 just as you expect, but the second requires closer scrutiny. Since the order of evaluation of the units in a collateral clause is undefined. The first one might be completed first, then the second one begun, giving the same result as the closed clause. However, on a computer with 2 processors, the sequence of actions might be as follows

1. Processor 1 fetches i into a register local to itself
2. Processor 2 fetches i into a register local to itself
3. Processor 1 adds 1 to its register containing i
4. Processor 2 adds 1 to its register containing i
5. Processor 1 stores its register back into memory location i
6. Processor 2 stores its register back into memory location i

The result is that i becomes 1, into i, instead of 2. As a consequence, the second program may print 2 or it may print 1. Random numbers are very useful in computer science, but this is not a recommended technique for printing them. Note that if the second unit were evaluated before the first, the result would have been 2. The difficulty only arises when the units actually are evaluated collaterally. Note that

```
BEGIN INT i:= 0,j:= 0; (i:= i+1,j:= j+1); print(i); print(j) END
```

produces identical results independent of the order of evaluation of the units. The moral of the story is: collateral clauses are an important programming technique but some care is required in their use.

There are some applications in which 2 processors running in parallel must cooperate with each other. For example, a computer with 2 processors might use one processor to compute values of some function and store them in an area of memory. The other processor might be removing these values and printing them. These two activities must be synchronized to avoid having the first processor continue generating values when there is no room left to store them. Similarly the second processor must stop running when the memory is temporarily empty and wait for the next value to be computed and made available in the memory. ALGOL 68 provides a mechanism for synchronizing collateral clauses. Synchronized collateral clauses are called parallel clauses, and will be discussed in detail in chapter 7.

2.8 Miscellaneous statements

It occurs occasionally in programming that 1 out of a large group of statements is to be executed, depending upon the value of some variable ALGOL 68 provides a CASE statement for this purpose. A CASE statement is of the form

```
CASE integer unit IN s ↓ 1, s ↓ 2, ... s ↓ n OUT s ESAC
```

The CASE statement is executed as follows. The integer unit is evaluated. If its value is 1, s ↓ 1 is executed, if its value is 2, s ↓ 2 is executed, etc. If the value of the integer unit is less than 1 or greater than n i.e. it is out of range, s is executed. After the selected statement is executed, the CASE statement is finished and execution continues with the statement following the ESAC. The CASE statement

```
CASE f IN s1 OUT s2 ESAC
```

is identical to

IF i = 1 THEN s1 ELSE s2 FI

The word OUT and the statement following it may be omitted if there is no possibility that the integer unit will be less than 1 or more than the number of statements between IN and OUT.

There is one final item to be mentioned about ALGOL 68. It is possible to label any statement with an identifier followed by a colon. There exists a GOTO statement which can be used to jump to a label. In recent years it has become increasingly clear that having GOTO statements in programming language is bad, in that programs with many jumps usually have many errors as well. A few references to the continuing GOTO controversy are given in the bibliography. The need for many GOTO statements usually indicates a poorly structured program. Upon finding the apparent need for a GOTO statement, the programmer should examine the program very carefully to see if perhaps the use of a FOR statement or a procedure would not make the program logically clearer. Using GOTO's should be compared to parachuting out of an airplane: it can be done, but there is usually a better way.

2.9 Summary

In this section some of the major features of ALGOL 68 will be reviewed. The language is built around the concept that in the computer's memory there exist objects. Each object has an address, a mode, and a value. There exist many actions that can be performed on these objects, such as slicing, selecting, adding and assigning. The execution of a program consists of carrying out a sequence of actions.

Figure 2-34 shows a summary of the ALGOL 68 modes. Figure 2-35 gives a summary of the kinds of statements and declarations available. The statements in a program and not the declarations really do the work, and it is

interesting to note how few ALGOL 68 has. Of the 7 types of statements, only the first 4 really count, since read and print are actually procedure calls and GOTO should be avoided. Despite the fact, or more accurately, because of the fact, that ALGOL 68 has so few statements, it is possible to express a very wide class of algorithms very conveniently in it. This is an important recognition, and the reader would do well to ponder its meaning. Figure 2-36 summarizes the 9 types of units. Figure 2-42 summarizes the grammatical structure of ALGOL [(:

A glossary of important terms introduced in this chapter follows.

Actual parameter – An object supplied to a procedure as input. In the procedure call sin(x), x is an actual parameter.

Assignment – Variables are given values by assigning to them. In ALGOL 68 i:= 2 is an example of an assignment. It can either be used as an ordinary statement by itself, or it can be used as a unit, as in a[i:= 2], in which statement by itself, or it can be used as a unit.

Closed clause – A serial clause enclosed by BEGIN END or parentheses. For example (INT i,j; read(i); i+j) is a closed clause. A closed clause is a unit.

Coercion – An implicit process of changing an object of one mode into an object of another mode. Dereferencing and deproceduring are two types of coercions.

Collateral clause – A series of units separated by commas and enclosed by BEGIN END or parentheses. The order in which the units of a collateral clause are evaluated is undefined, thus leaving open the possibility that on a computer with more than 1 processor several units may be evaluated concurrently.

Conditional clause — A construction of the form IF condition THEN ... ELSE ... FI. The value of the conditional clause depends upon whether the condition is true or false. This construction may also be used by itself as a conditional statement.

Denotation — A constant. Examples 4, TRUE, "x".

Deproceduring — The process of replacing a procedure name by its result. This is simply a special name for a procedure call for the special case of a procedure with no parameters. For example, in REAL x:= random, the procedure random cannot be assigned to x, since the source must be of mode real, so random is deprocedured, i.e. "called" to deliver a real which can be assigned to x. Deproceduring is a coercion. In this example although random has mode PROC REAL, it can be written in a position requiring an object of mode REAL because it can be coerced to REAL by deproceduring.

Dereferencing — The process of replacing an object of mode "reference to something" by an object of mode "something". In i:= j both i and j are of mode "reference to integer". However, an assignment to an integer variable i such as i requires an object of mode integer as source, not the address of an integer. To solve this syntactic problem and allow the assignment to be meaningful, the integer whose address is j is used instead of the address itself.

Dyadic — A dyadic operator is one with 2 operands, such as + in i+j.

Field selector — A structure is an object containing 1 or more objects called fields. Each field has a name, called a field selector. In STRUCT (STRING

breed, INT weight) breed and weight are field selectors.

Formal parameter – In a procedure declaration parameters are declared by specifying their modes, and giving them symbolic names to be used in the procedure. These are formal parameters. In PROC bump = (REF INT k) INT : k:= k+1, k is a formal parameter.

Mode – The property of an object which specifies the type of object it is. Examples of modes are INT, BOOL, REAL, CHAR, []INT, [,]INT, STRUCT(INT num,denom), PROC(INT) INT and STRING.

Mode declaration – A definition of a new mode. Examples: MODE REGISTER = [0:15] BOOL and MODE MATRIX = [1:n,1:n] REAL.

Monadic – A monadic operator is one with only 1 operand. In i:= ABS j, ABS is a monadic operator.

Operator declaration – A definition of a new operator, e.g. OP HALF = (INT i) INT : i : 2 defines HALF to be a monadic operator so j:= HALF 6 will assign 3 to j.

Row display – A collateral clause of mode row of something used as a unit. For example, in [1:5] INT i1:= (0,1,3,9,–2) the collateral clause (0,7,3,9,–2) is used as a row display.

Selection – The use of one field of a structure as a unit. If house is declared STRUCT(INT price, STRING style) house then "price OF house" and

"style OF house" are selections. Selections can be used as sources or as destinations in assignment statements and in many other positions.

Serial clause – A series of statements and declarations followed by a unit, or a unit all by itself. Example: INT i,j; i:= 0; j:= 2.

Slice – One or more elements of an array. If i1 is declared [1:6] INT i1 then i1[2], i1[6] and i1[2:5] are all slices.

Structure display – A collateral clause used as a denotation. For example in STRUCT(STRING name, INT length, BOOL filthy) lake:= ("erie",400,TRUE) the collateral clause ("erie",400,TRUE) is a structure display.

Unit – An expression that yields a value. Denotations, variables, slices, selections, formulas, assignations, calls, closed clauses, and conditional clauses are all units.

PROBLEMS CHAPTER 2

Variables mentioned in the problems but not explicitly declared are assumed to be declared in figure 2-16.

1. What are the modes of: i, p, c, s, i1 and i2?

2. What are the modes of 2, TRUE, "piggy"?

3. Is it possible to uniquely determine the mode of (0,1,2)?
   If so, what is it? If not, give 2 modes it might be.

4. Does every object have a mode? If not give an example of an object with no mode.

5. The procedures sin and sqrt both take a real parameter and deliver a real result. Do they have the same mode? If so, what is it?

6. Is it possible to declare an operator which changes the mode of a given object? If so give an example.

7. How many distinct modes are there?

8. What is another word for denotation?

9. Give an example of a denotation for each of the following modes: INT, [] CHAR, STRUCT(INT i, BOOL pig).

10. How many boolean denotations are there?

11. Is ";" a denotation?

12. Which of the following are integer denotations? 4, (10), −2, 0001.

13. Give an example of an assignment statement, an IF statement, and a FOR statement.

14. Which of the following are valid statements?
    a) IF i<0 THEN k:= 1 FI
    b) IF i≠0 THEN k:= 1
    c) IF i=0 THEN i:= 0 ELSE i:= 0 FI
    d) IF i>0 ∨ j≠ 4 ∧ k=3 THEN i:= 0; j:= 0 ELSE k:= 0 FI
    e) IF k=1 THEN i>0 ELSE J:= 0 FI
    f) FOR i:= 1 TO 3 BY 1 DO j:= j+1
    g) WHILE i>0 DO i:= −i
    h) FROM IF i<0 THEN 1 ELSE 2 FI TO 3 DO n:= n+1
    i) FOR i WHILE P:= q DO q:= FALSE
    j) FOR i FROM time OF album[2] TO i1[2] DO i1[i]:= 0

15. If i:= 1, j:= 2, k:= 50, i1:= (2,1,2,1,2,5,9,3,−1,−6) and n:= 0, what is the value of n after each of the following statements is executed.
    a) FOR i FROM 4 TO j DO n:= n + 1
    b) FROM i1[2] TO i1[6] DO n:= n+1
    c) WHILE(i:= −i; i<0) DO n:= n +1
    d) FROM i TO k DO BEGIN k:= 10; n:= n+1 END
    e) TO j DO n:= n+1

16. Which of the following are valid variable declarations?
a) INT k:= i
b) REF INT k:= 1
c) REF INT k:= i
d) [] BOOL b
e) STRUCT(INT,BOOL) s

17. You are to agree or disagree with the following statement and defend your viewpoint: The text INT i,j; j:= 0; i:= j is incorrect because in an integer assignment the destination must be an object of mode reference to integer, i.e. the address of an integer (which it is) and the source must be an object of mode integer (which it is not, since j is also of mode reference to integer). Therefore i:= j is incorrect.

18. What is the essential difference between i:= 1 and i:= j?

19. What are the 9 kinds of units? Give an example of each.

20. Which of the following constructions are correct?
a) i1[IF p THEN 2 ELSE 3 fi]
b) FROM(INT i; read(i); i) TO 10 DO n:= n+1
c) i:= j:= k
d) FROM abscissa OF gp TO ordinate OF gp DO n:= n+1
e) IF i<0 THEN i ELSE j FI := IF k≠2 THEN i+1 ELSE 3 FI

21. Consider the declaration.
[1:50] STRUCT(STRING symbol,INT value,[1:10] BOOL attributes)h
What mode do each of the following have?
a) h[4]
b) value OF h[4]
c) symbol OF h[29]
d) attributes OF h[1]
e) (attributes OF h[1])[10]

22. Is the following statement valid, and if so, what does it do +0 k?
(read(j); IF j<0 THEN j:= 0 FI;k) := (n:= 1;2)

23. Is the following a unit, and if so what is its value and mode?
(read(i); i:= i×i; (read(j); j:= j+1; "."))

24. Which of the following are correct assignments?
Give the values of all variables assigned to
a) i:= j:= 2
b) i:= 3:= 2
c) (i:= j):= 3
d) i:= (j:= 3)
e) i:= i

25. After the declarations
[1:5] BOOL pp:= (TRUE,TRUE,FALSE,TRUE,FALSE);
[1:5] BOOL qq:= (FALSE,TRUE,FALSE,FALSE,TRUE)
what are the values of pp[1] to pp[5] after
a) pp:= qq
b) pp[1:5]:= qq[1:5]
c) pp[3:5]:= qq[3:5]
d) pp[2]:= qq[3]
e) pp[1]:= qq[4]:= TRUE

26. Which of the following operators are dyadic?
    a) OP × = (INT i) INT : i
    b) OP × = (INT k) INT : k+1
    c) OP × = (INT i,j) INT : i×j
    d) OP × = (INT i, PROC VOID p) VOID : TO i DO p
    e) OP × = (INT i,j) BOOL : i=j

27. If ⋔ is defined by
    OP ⋔ = (INT k) INT : k+1
    what is the value assigned to n in n:= ⋔ ⋔ ⋔ 2?

28. If + is defined by OP + = (INT i,j) INT : i−j
    what is the value assigned to n in n:= 3+2

29. After the declarations
    PROC p = (INT i, STRIN s) CHAR : s[i];
    STRUCT(INT employees, STRING name) company:= (1000, "general widget")
    what is the value of the call p(3,name OF company)?

30. Write a declaration for a procedure "biggest" with 3 integers as parameters. The value of the procedure is the largest of the 3 integers given as parameters.

31. Write a procedure of mode PROC([] INT,INT) BOOL that searches its first parameter element by element and returns TRUE if some element matches the second parameter and FALSE if not.

32. Write a procedure taking as parameter an 8 × 8 character matrix, each of whose elements is "r", "b", or "e" (red,black,empty) corresponding to a checkerboard and which returns the number of black pieces minus the number of red pieces.

```
int  number of girlfriends, pelican count;
char grade expected in this course;
bool voted in last election, likes mustard
```

Figure 2-1. Declarations for 2 integer variables, 1 character variable, and 2 boolean variables.

```
bool bottle has deposit;
int  price, cost of product, amount of deposit;
if bottle has deposit
   then price:= cost of product + amount of deposit
   else price:= cost of product
fi
```

Figure 2-2. Use of conditional statement.

```
int  number of legs,centipede count;

if   number of legs > 50
     then centipede count:= centipede count + 1
fi
```

Figure 2-3. Conditional statement with no else part.

```
for odd number from 1 by 2 to 100 do print(odd number)
```

Figure 2-4. A for statement that prints the first 50 odd numbers.

```
int n;
n:= 1;
while n × n < 1000 do n:= n + 1
```

Figure 2-5. Use of a while clause to find the smallest integer whose square is > 1000.

```
a) for i from a by c to b while condition do S

b) for i from 0 to n do S

c) to 4 do new line

d) while x + y ≠ i + j do S

e) for k from 10 to 50 while p < 0 do S

f) for j from -100 to 100 do S

g) from -60 to 4096 + i x j - 40 x k by m x m do S

h) for in
     from  i + i x i - 4 x k + n x n x n
     by    m x n x o x p - a x b x c
     to    a + b + c + d + e + f
     while a + i + 3 x j - 6 ≥ k + 4096
     do S

i) for i to n do if i ≠ k then print(i) fi
```

Figure 2-6. Examples of for statements. S represents some statement.

```
a) for i to 10 do
     begin  print(i);
            print(i × i);
            print(i × i × i);
            newline
     end

b) for i to 2 do
   for j to 4 do
     begin  print(i);
            print(j);
            newline
     end

c) begin  ¢ This program prints the Fibonacci numbers up to 1000¢
          int first,second,third;
          first:= 1;
          second:= 1;
          while first < 1000 do
          begin  print(first);
                 newline;
                 third:= first + second;
                 first:= second;
                 second:= third
            end
   end
```

Figure 2-7. Use of compound statements. Figure 2-7c is a complete program.

```
begin   ¢ This program reads two numbers, the price of an item and the amount
        the customer paid for it. It then calculates how much change he should
        get, and prints out the correct number of quarters, dimes nickels, and
        pennies to minimize the number of coins to be returned. The program
        only handles change up to 99 cents, and prints a message if the change
        is too much¢
        int price, amount paid, change, quarters, dimes, nickels, pennies;
        ¢first read in the price and payment and compute the change¢

        read(price); read(amount paid);
        change:= amount paid - price;
        if change > 99
           then print("change > 99 cents")
           else ¢test to see if the payment was exact¢
                if change = 0
                   then print("no change")
                   else ¢compute how many of each coin¢
                        quarters:= 0;
                        dimes:= 0;
                        nickels:= 0;

                        while change ≥ 25 do
                              (quarters:= quarters + 1; change:= change - 25);
                        while change ≥ 10 do
                              (dimes:= dimes + 1; change:= change - 10);
                        while change ≥ 5 do
                              (nickels:= nickels + 1); change:= change - 5);
                        pennies:= change;

                        ¢print results¢

                        if quarters > 0
                           then print(quarters);
                                print("quarters");
                           newline
                        fi;

                        if dimes > 0
                           then print(dimes);
                                print("dimes");
                                newline
                        fi;

                        if nickels > 0
                           then print(nickels);
                                print("nickels");
                                newline
                        fi;

                        if pennies > 0
                           then print(pennies);
                                print("pennies")
                        fi
                fi ¢This matches the if change = 0¢
        fi ¢This matches the if change > 99¢
end
```

Figure 2-8. A complete ALGOL 68 program to calculate how to pay out change with the minimum number of coins, using only quarters, dimes, nickels, and pennies.

mode	possible procedure
proc void	newline
proc(real) real	sin
proc(int,int) int	integer multiply
proc(int) void	skip the next n records on a tape
proc(int,int) bool	compare the parameters for equality
proc(real,real) bool	compare the parameters for equality
proc(char,char) bool	compare the parameters for equality

Figure 2-9. Examples of procedure modes.

declaration	meaning
a) [1:10] int ri;	declares 1 dim. array of 10 integers
b) [1:5,1:7] int rri;	declares 2 dim. array of 35 integers
c) [1:10,1:10,1:10] bool rrrb;	declares 3 dim. array of 1000 booleans
d) [0:100] char rc;	declares 1 dim. array of 101 characters
e) [-20:-6,-4:2] real rrr;	declares 2 dim. array of 105 reals
f) [1:20] proc void rpv;	declares 1 dim. array of 20 procedures
g) [1:n] int ri2;	declares 1 dim. array of n ints
h) [n1:n2,n3:n4] bool rrb;	declares 2 dim. array of booleans

Figure 2-10. Declarations of arrays.

```
begin   ¢Examples of slicing and assigning together¢
        [1:10] int a,b,c;
        for i from 1 to 10 do a[i]:= i × i;
        b[7:10]:= a[7:10];
        b[4:6]:= a[4:6];
        b[1:3]:= a[1:3];
        c:= b
end
```

Figure 2-11. Examples of the use of slices.

```
begin   [0:6,2:8] int mat;
        [0:6] int vec;
        for i from 0 to 6 do
        for j from 2 to 8 do mat[i,j]:= i×j;
        vec[0:6]:= mat[0:6,8]
end
```

Figure 2-12. Assigning a column of a matrix to a vector.

Property	Array	Structure
May the elements have different modes?	No	Yes
How is an element accessed?	subscripting e.g. a[4]	selecting e.g. type of aircraft
Can several elements be accessed together?	Yes, by slicing e.g. a[2:4]	no
What is the largest number of elements?	no limit	no limit

Figure 2-13. Comparison of arrays and structures.

```
mode vector          = [1:n] real
mode matrix          = [1:n,1:n] real
mode rational        = struct(int numerator,denominator);
mode person          = struct(string name,ref person father, mother,
                       int age,bool smokes);
mode family          = struct(person mommy,daddy,[1:2] person child);
mode bridgehand      = [1:13] struct(char rank,suit);
mode word            = [0:15] bool;
mode registers       = [0:7] word;
mode conditioncodes  = struct(bool n,z,v,c);
mode table           = [1:1000] struct(string symbol,int value);
mode instruction     = struct(int opcode,addr1,addr2,addr3);
mode flight          = struct(int number,string pilot,movie,bool nonstop,
                       [1:350] struct(string name,int phone) passenger);
mode tree            = struct(int value,ref tree left,right);
mode booladdress     = ref bool
```

Figure 2–14. Sample mode declarations.

```
bool b:= true;
int n:= 3;
vector x:= (2.0,3.0,4.0);
matrix xx:= ((1.0,2.0,3.0),(2.0,3.0,4.0),(3.0,4.0,5.0));
rational rat:= (2,7);
person john:= ("smith",bill,mary,20,false);
family jones:= (john,linda,(nancy,peter));
bridgehand north:= (("A","S"),("K","S"),("9","S"),("7","S"),("5","S"),
                    ("3","S"),("Q","H"),("J","H"),("T","H"),("3","H"),
                    ("7","D"),("4","D"),("2","C"));
word w; registers reg; conditioncodes cc; table t;
instruction add:= (4,6,8,2);
flight twa666:= (520,"warthog","trash",true,skip);
tree t4;
booladdress bad
```

Figure 2–15 variable declarations using the modes of figure 2–14.

```
mode song      = struct(string title,int time,bool folksong);

mode gridpoint = struct(int abscissa,ordinate);

mode student   = struct(char grade,bool virgin,
                 string name,int year of graduation);

mode course    = struct([1:n] student kids,string prof);

mode book      = struct(string title,author,bool paperback);

mode library   = [1:10000000] book
```

```
real x,y;
int i,j,k,n;
bool p,q;
char c;
[1:100] char S;
[1:10] int i1;
[1:3,1:2] int i2;
[1:4] bool boole;
[1:8] song album;
course cs101;
proc(int) int p1;
proc(real,real) bool p2;
proc(string,int) char p3;
proc([,] int,int) [] int p4;
proc(bool) bool p5;
[-20:+20] struct(real coef,int exp) polynomial;
gridpoint gp;
library harvard;
```

Figure 2-16. Declarations used in figures 2-17 to 2-25 and the problems.

Mode	Denotations
int	1,3,0,1492
bool	true,false
char	"a","4","+"
[] int	(1776,1812,1861,1917,1941,1954)
song	("barbara allen",183,true)
[,] real	((1.0,2.0),(2.0,7.24))
[,,] int	(((1,2),(1,4)),((2,6),(9,-4)))
struct(bool b,c)	(true,false)
real	3.14
struct(real re,im)	(2.78,3.14)

Figure 2-17. Examples of denotations, including constant displays, which strictly speaking are not called denotations.

Mode	Variables
int	i,j,k,n
bool	p,q
char	c
[] int	i1
string	S
[] song	album
gridpoint	gp

Figure 2–18. Examples of variables.

Mode m	Slice yielding object of Mode m
int	i1[2],i1[4],i2[1,3]
bool	boole[i + 3 × j]
char	S[99],S[100]
song	album[1+3]
struct	polynomial[0]
student	(kids of cs101)[1]
book	harvard[i]

Figure 2–19. Examples of slices.

Mode m	Selection yielding object of Mode m
int	time of album[4],exp of polynomial[0]
bool	folksong of album[1]
char	grade of student
[] student	kids of cs101
string	title of album[2]
bool	virgin of (kids of cs101)[2×j]
bool	paperback of harvard[i]

Figure 2–20. Examples of selections.

Mode m	Call yielding object of Mode m
int	p1(n)
bool	p2(3.14,2.78)
char	p3("doggy",2)
[] int	p4(i2,i1[3])
bool	p5(folksong of album[4])

Figure 2–21. Examples of procedure calls.

Mode m	Formula yielding object of Mode m
int	i+2, j+4, 3x2+1, 4+n+k
bool	p∧q, p∨q, p∧(p∨q)
real	2.0+2.5, x+3.141592, x×y/2.0

Figure 2-22. Examples of formulas.

Mode m	Assignment yielding object of mode m
int	i:= 2, i:= j, i:= j:= k:= 0
bool	p:= true, q:= p ∨ q
char	S[3]:= "x"
[] int	i1:= (0,1,2,3,4,5,6,7,8,9)
song	album[1]:= ("silkie",206,true)
book	harvard[21416]:= ("august 1914","solzhenitsyn",true)
real	coef of polynomial[0]:= 2.55
string	prof of cs101:= "barry bigbrain"

Figure 2-23. Examples of assignment. In all cases the result of the assignment must be dereferenced before yielding the specified mode.

Mode m	Closed clause yielding object of Mode m
int	(x:= 1; if p then i else j fi; k)
bool	(p)
char	(S[1:4]:= "love"; S[1])
[] int	(for i to 10 do i1[i]:= 0; i1)
student	(read(cs101); (kids of cs101)[1])
gridpoint	(abscissa of gp:= ordinate of gp:= 1; gp)
int	begin p1(k) end
bool	begin p1(1); p2(3.14,3.14); p5(q) end
int	begin read(j); i:= j end
int	((((((1066))))))

Figure 2-24. Examples of closed clauses. Note that is some cases the value of the closed clause may have to be dereferenced before yielding the specified mode.

Mode m	Conditional clause yielding object of Mode m
int	if x<y then i else 3 fi
bool	if p then p else q∨p fi
char	if S[1] = "x" then S[3] else "y" fi
[ ] int	if x[1] < 4 then x1[1:3] else x1[1:2] fi
song	if i = 2 then album[1] else album[2] fi
book	if i<j then harvard[i] else harvard[j] fi

Figure 2-25. Examples of conditional clauses. Note that in some cases the value of the conditional clause must be dereferenced before yielding the specified mode.

Position	Allowed constructions
subscript	integer unit
lower bound in array declaration	integer unit
upper bound in array declaration	integer unit
after CASE	integer unit
after from	integer unit
after to	integer unit
after by	integer unit
condition following if	boolean unit or boolean serial clause
condition following while	boolean unit or boolean serial clause
procedure body	unit
source in assignment	unit
initial value of declared variable	unit
actual parameter of a call	unit
element of row display	unit
element of structure display	unit
operand of formula	any unit except an assignment
following of in a selection	any unit except assignment or formula
procedure to be called	any unit except assignment, formula, or selection
array to be sliced	any unit except assignment, formula, or selection
after then or after else	serial clause, unit
destination in assignment	any unit except denotation or assignment, providing it yields reference to something

Figure 2-26. Kinds of constructions allowed in different positions in the program.

```
begin   ¢This program reads in a 10 element integer array "a", and a 10
        element integer array "b", and calls innerproduct to form the sum
        a[1] x b[1] + a[2] x b[2] + ... a[10] x b[10].
        The sum is printed.¢

        [1:10] int a,b;

        proc([] int,[] int) int innerproduct:= ([] int y,z) int:
        begin   int answer:= 0;
                for k from 1 to 10 do answer:= answer + y[i] × z[i];
                answer
        end;
        read(a);
        read(b);
        print(innerproduct(a,b))
end
```

Figure 2-27. A program using a procedure.

```
proc innerproduct = ([] int y,z) int:
begin   int answer:= 0;
        for k from 1 to 10 do answer:= answer + y[i] x z[i];
        answer
end
```

Figure 2-28. The alternative form for a procedure declaration.

```
begin   ¢This program defines a procedure vectoradd that adds together
        2 1 dimensional integer arrays with 10 elements. The use of the
        procedure is demonstrated.¢

        mode     vector = [1:10] int;
        vector   v1,v2,v3,v4,v5;

        procedure vectoradd = (vector a,b) vector:
        begin    vector sum;
                 for i from 1 to 10 do sum[i]:= a[i] + b[i];
                 sum
        end;

        ¢read in v1, v2, v3, and v4 and compute the sum v1 + v2 + v3 + v4
        and store it in v5.¢

        read(v1);
        read(v2);
        read(v3);
        read(v4);
        v5:= vectoradd(vectoradd(vectoradd(v1,v2),v3),v4);
        print(v5)
end
```

Figure 2-29. A program to read in and add 4 vectors using a procedure.

```
begin   ¢This program defines an operator + that adds together
        2 1 dimensional integer arrays, each with 10 elements. The use
        of the operator is demonstrated.¢

        mode    vector = [1:10] int;
        vector  v1,v2,v3,v4,v5;

        op + = (vector a,b) vector:
        begin   vector sum;
                for i from 1 to 10 do sum[i]:= a[i] + b[i];
                sum
        end;

        ¢read in v1, v2, v3, and v4 and compute. The sum v1 + v2 + v3 + v4
        and store it in v5.¢

        read(v1);
        read(v2);
        read(v3);
        read(v4);
        v5:= v1 + v2 + v3 + v4;
        print(v5)
end
```

Figure 2-30. A program to read in and add 4 vectors using an operator.

operators	mode of operands	mode of result	meaning
dyadic + − × :	int,int	int	add,subtract,multiply,divide
≤ < = ≠ > ≥	int int	bool	the usual meanings, e.g. i = j is true if and only if i and j have the same value
↑	int int	int	exponentiation, i.e. i ↑ j = ij
∧	bool bool	bool	logical and
∨	bool bool	bool	logical or
= ≠	bool bool	bool	tests operands for (in) equality
≤ < = ≠ > ≥	char char	bool	tests the (implementation dependent) character codes for equality, less than etc
≤ ≤ = ≠ > ≥	string string	bool	test for ordering using the character codes. If the character codes are in alphabetical order as in ASCII, s1 < s2 means s1 is alphabetically before s2, etc.
+	string string	string	concatenation. Thus
+	char string	string	string s1:= "hot";
+	string char	string	string s2 = "dog";
+	char char	string	print(s1 + s2) will print hotdog
monadic ¬	bool	bool	logical not
abs	int	int	absolute value

Figure 2-31. Some of the built in operators.

```
begin   ¢This program reads in the length of a list of integers, then
        reads the integers themselves. The operator biggest takes a 1
        dimensional integer array of arbitrary length as parameter,
        and returns the largest element.¢

        op biggest = ([] int a) int:
        begin   int biggie:= a[lwb a]; ¢declare and initialize biggie to
                first element¢
                for i from lwb a + 1 to upb a do
                if a[i] > biggie then biggie:= a[i] fi;
                biggie
        end;
        int n;                  ¢declare n as integer variable¢
        read(n);                ¢read in n¢
        [1:n] int i1;           ¢algol 68 declarations need not precede the
                                 executable statements¢
        read(i1);               ¢read in the entire array¢
        print(biggie i1)        ¢compute and print the biggest¢
end
```

Figure 2-32. Use of the monadic operators lwb and upb.

1. Source and destination in an assignment
2. Operands of a dyadic operator
3. Elements in a row display
4. Fields in a structure display
5. Units in a collateral clause
6. Units in an actual parameter list
7. Integral units after from, to and by in a for statement
8. Subscripts in a slice
9. Upper and lower bounds in an array declaration
10. Array to be sliced and its subscripts (in a slice)
11. Procedure to be called and its parameters (in a call)
12. Declarations separated by commas

Figure 2-33. Constructions evaluated collaterally.

1. Built in modes: INT, BOOL, CHAR, REAL, FORMAT

2. Array modes:
   [] M is a row of M (1 dimensional array)
   [,] M is a row row of M (2 dimensional array)
   [,,] M is a row row row of M (3 dimensional array)
   etc.

3. Structure modes:
STRUCT(M1 id1, M2 id2, ...) is a mode
   The first field has mode M1 and field selector id1, etc.

4. Reference to modes:
   REF M is of mode reference to M. An object of mode REF M is called an M variable.

5. Procedure modes:

PROC M	PROC VOID
PROC (M1) M	PROC (M1) VOID
PROC (M1,M2) M	PROC (M1,M2) VOID
PROC (M1,M2,M3) M	PROC (M1,M2,M3) VOID
PROC (M1,M2,M3,...) M	PROC (M1,M2,M3,...) VOID

are all modes.

Figure 2-34. Summary of ALGOL 68 modes. M1, M2, M3 and M are all arbitrary modes, and id1, id2 are arbitrary identifiers.

Statements and examples

IF statement: IF i<0 THEN i:= 0 ELSE i:= i+1 FI

FOR statement: FOR i FROM 1 TO 10 BY 3 WHILE j<0 DO S

assignment statement: i := j

CASE statement: CASE i+j IN i:= 0,read(j) OUT print(i+j) ESAC

input statement(really a procedure call): read(n)
output statement(really a procedure call): print(n)

GOTO statement: GOTO jail

Declarations and examples

variable declaration: BOOL imbibes

procedure declaration: PROC and = (BOOL p,q) BOOL: IF p THEN q ELSE FALSE FI

mode declaration: MODE INTEGER = INT

operator declaration: OP × = (INT i,j) INT: i×j

priority declaration: PRIORITY, + = 5

Figure 2-35. Summary of ALGOL 68 statements and declarations.

kind	examples
denotation:	2,TRUE,"c",(3.14)
variable:	i,x,house
slice:	a[2],a[4:5]
selection:	name OF john, filthy OF erie
formula:	a+b, ¬ p ∧ q
procedure call:	sin(x); sqrt(3.14)
assignation:	i:= j, filthy OF erie:= TRUE
closed clause:	(FOR i TO N DO a[i]:= 0; a),(4)
conditional clause:	IF i < 0 THEN 1 ELSE 2 FI

Figure 2-36. Types of units.

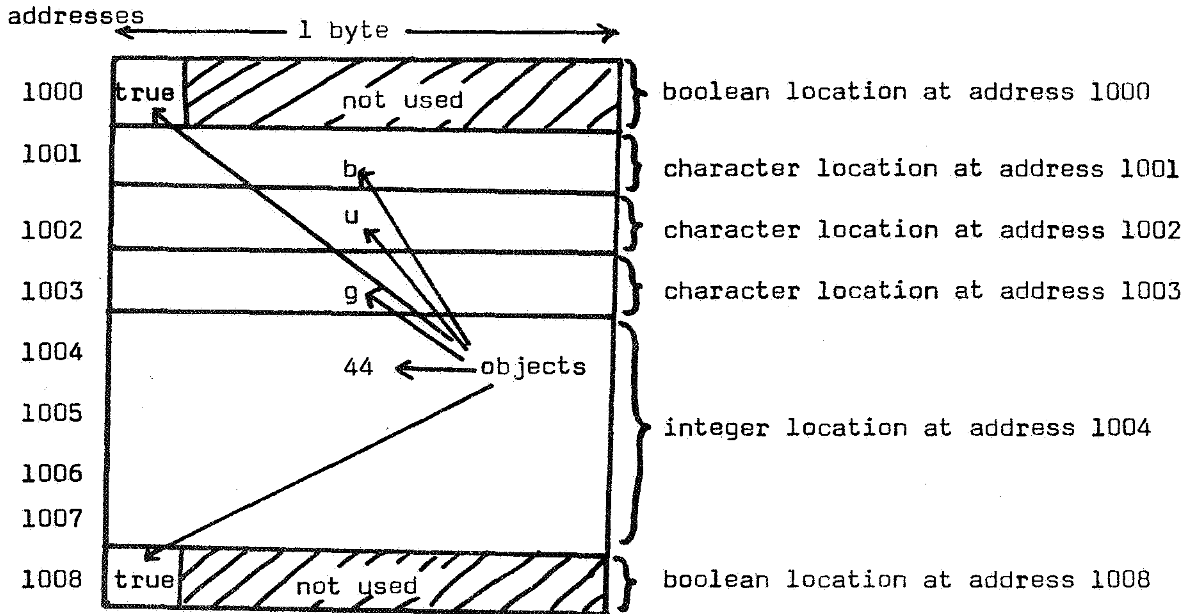

Figure 2-37. A memory organization in which boolean locations occupy less than one byte, character locations occupy exactly one byte, and integer locations occupy four bytes. A byte is 8 bits.

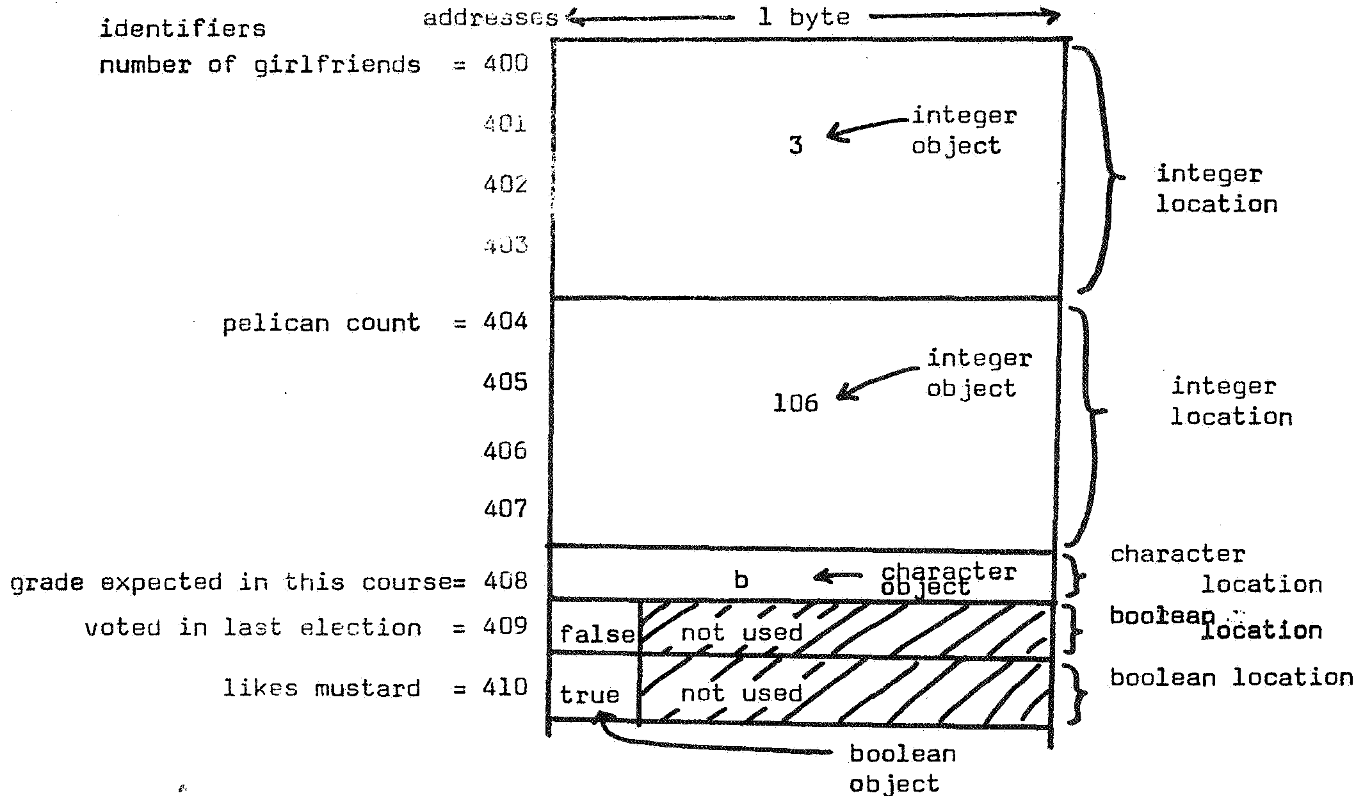

Figure 2-38. The relation between identifiers, addresses, objects, locations, and variables. Five variables are shown. Each variable has a numerical address (400,404,408,409, or 410) and an identifier which is equivalent to that address, as well as a location (a region in memory) in which an object can be kept. For example, the integer variable "pelican count" is at address 404 and occupies 4 bytes.

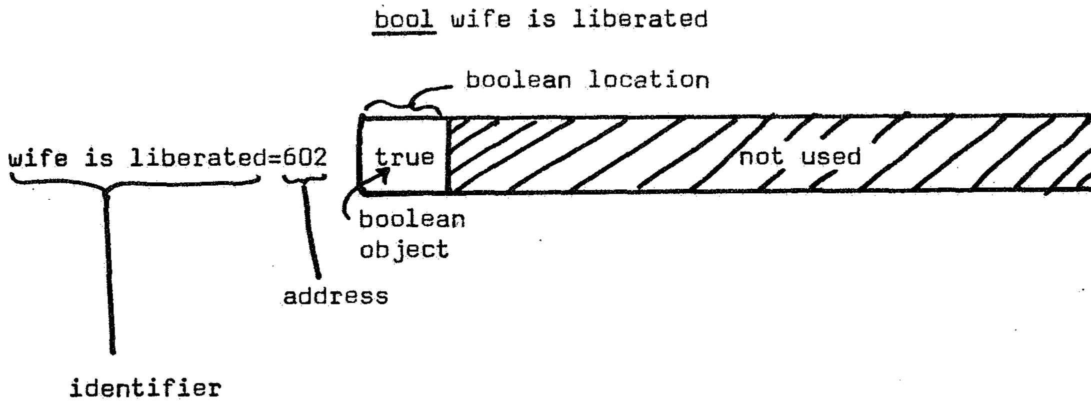

Figure 2-39. Declaration of a boolean variable. The identifier wife is liberated is equivalent to the address 602. It is NOT equivalent to the boolean object at that location (in this case, true). We will often refer to "the boolean whose address is 'wife is liberated'" instead of the clumsy expression "the boolean in the location whose address is equivalent to 'wife is liberated'".

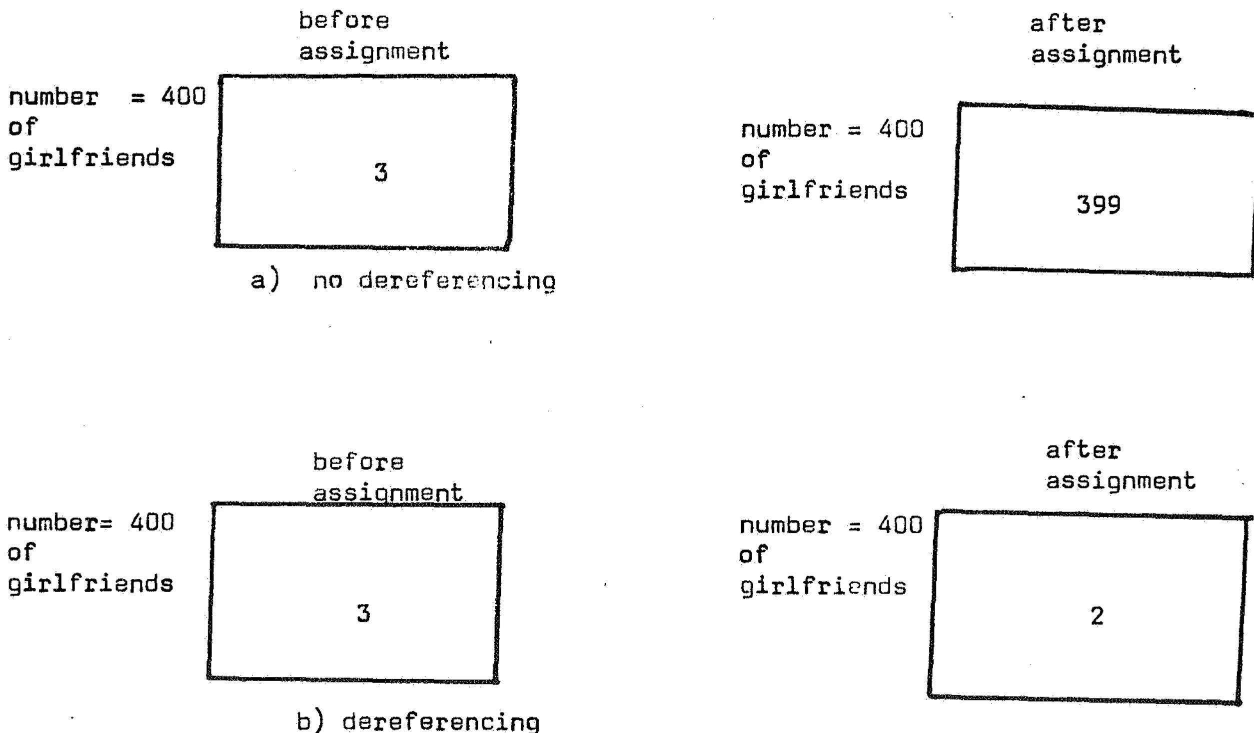

Figure 2-40. The assignment number of girlfriends := number of girlfriends - 1 with and without dereferencing. a) If no dereferencing took place, the expression "number of girlfriends-1" would have the value 400-1, which is 399. b) With dereferencing, the value of the integer at location 400 is used instead of 400, thus giving 3-1 as the source of the assignment.

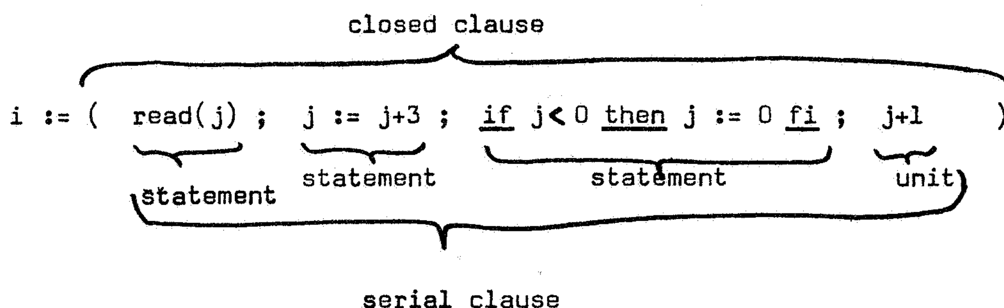

Figure 2-41. Relation between serial clause, unit, and closed clause. A serial clause is a series of 0 or more declarations and/or statements followed by a unit. In this figure, the serial clause consists of 3 statements followed by the unit j+1. A closed clause is a serial clause enclosed by parentheses or begin and end. The entire right hand side of the above assignment is a closed clause. A closed clause is itself a unit, but a serial clause is not a unit.

---

assignment statement: "reference to some mode" unit,
becomes symbol,
"some mode" unit

conditional statement: if, boolean serial clause,
then, serial clause,
{ else, serial clause, }
fi

for statement: { for, identifier, }
{ from, integer unit, }
{ to, integer unit, }
{ by, integer unit, }
{ while, boolean serial clause, }
do, statement (including a compound statement or range)

case statement: case, integer serial clause,
in, list of statements
{ out, statement }
esac

go to statement: goto, label

Figure 2-42. Definitions of statements. {-} means optional.