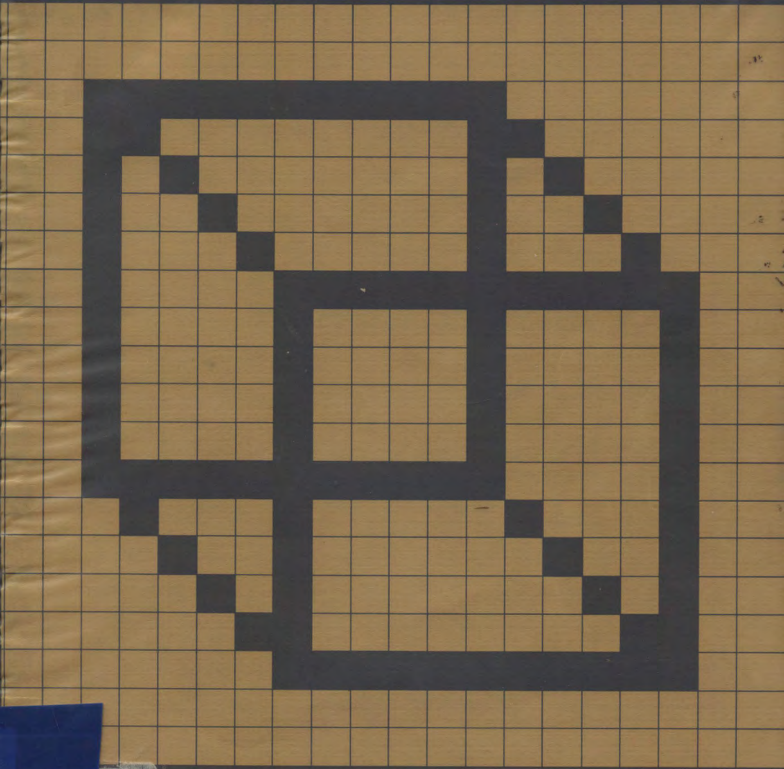
an introduction to FORTRAN



tralian National University
puter Services Centre

The Australian National University Computer Services Centre

An Introduction to FORTRAN

Course Notes

Author Leslie Landau

Revision 6 K. Handel July 1981

ANSI FORTRAN-77 (ANSI X 3.9-1978)

These course notes are designed for use in conjunction with an introductory FORTRAN course on the UNIVAC 1100/82 computer at ANU, using the ASCII FORTRAN compiler in checkout mode (load and go).
Any enquiries regarding attendance at one of these courses should be directed to

The Secretary
Computer Services Centre
Australian National University
P.O. Box 4
CANBERRA 2600
A.C.T.

The Secretary's phone number is 49-4564.

First printing	May 1976
Revised Edition 1	September 1977
Revised Edition 2	December 1977
Revised Edition 3	November 1979
Revised Edition 4	January 1981
Revised Edition 5	March 1981

A note on standards.

In 1966 a standard for Fortran was issued by the American National Standards Institute (ANSI).

In 1977 a revised standard was issued by ANSI. This standard comprised a FULL language specification and also a SUBSET language specification. The subset is more restrictive than the full specification, and as Univac Ascii Fortran level 9R1 (and above) implements the full language specification, reference to the 1977 standard refers to the full specification.

If you wish to use a Fortran other than Univac Ascii Fortran level 9R1 (or higher) then you should consult your manual to see if the version of Fortran you wish to use is the full, subset or non-standard Fortran specification.

1. CHAPTER 1

A computer is a tool used to solve certain classes of problems. Before this can be done we have to solve the problem ourselves and then use the computer to apply the solution to given data. The programmer specifies the solution in the writing of the program - the computer provides speed and accuracy.

1.1. A model of a computer

Figure 1-1 below shows a clerk seated at his desk with a calculating sheet in front of him. He has an IN-TRAY into which slips are placed and an OUT-TRAY into which he places results. He has a calculating machine to do the arithmetic.

```
+----+                                    +----+
|    |                                    |    |
| In |                                    | Out|
|    |                                    |    |
+----+                                    +----+

          +--------------+-----+------------+
          |              |     |            |
          | Instructions |     |            |
          |              |     |            |
          +--------------+     | Calculator |
          |                    |            |
          |                    |            |
          +--------------------+------------+

                    +---------+
                    |         |
                    |  Clerk  |
                    |         |
                    +---------+
```

The clerk's task is to take an input slip which contains the following information about employees:-

NAME
RATE OF PAY
NUMBER OF HOURS WORKED

and he has to prepare an output slip with the above information plus calculations for:-

OVERTIME HOURS
NORMAL PAY
OVERTIME PAY
TOTAL PAY

A section of the calculating sheet contains instructions for him to enable the required calculations to be performed. These instructions are:-

1. Take the next slip from the IN-TRAY and copy the name, rate and total hours.
2. Calculate the overtime hours as (Total hours - 40).
3. Calculate the normal earnings as (40 x Rate).
4. Calculate overtime earnings as (Overtime hours x Rate x 1.5)
5. Calculate gross earnings as (Normal earnings + Overtime earnings).
6. Prepare an output slip and put it in the OUT-TRAY.
7. If there are any more slips in the IN-TRAY go to step 1, otherwise stop.

Now let us automate this system, by using a computer in the following way:-

```
  +---------+                                      +---------+
  |         |                                      |         |
  | Input   |                                      | Output  |
  | Device  |                                      | Device  |
  |         |                                      |         |
  +---------+                                      +---------+

               Main  Storage  Area
       +-----------+----------------+-------------+
       |           |                |             |
       |  Program  |                | Central     |
       |           |                | Processing  |
       +-----------+                | Unit        |
       |                            |             |
       |                            +-------------+
       |           +---------+      |
       |           |         |      |
       |           | Control |      |
       |           |         |      |
       +-----------+---------+------+
```

The IN-TRAY is replaced by an INPUT DEVICE. There are many types of input devices that may be attached to a computer, we will consider it as a convenient means of putting information into the machine.

The OUT-TRAY is replaced by an OUTPUT DEVICE. There are many types of output devices that may be attached to a computer, we will consider it as a convienent means of getting information out of the machine.

The CALCULATING SHEET, with both its instruction area and its calculating area, is replaced by the MAIN STORAGE AREA.

The CALCULATOR is replaced by the Central Processing Unit. This performs the calculations.

Finally, the clerk is replaced by a CONTROL program. The function of the control program is to take each instruction in turn from the main store (memory) and cause the appropriate action to take place. This might be:-

1. To read some information from the outside world via the input device.
2. To write some information to the outside world via the output device.
3. To calculate some quantity, using the central processing unit.

1.2. Solving problems using computers

The steps involved in solving problems by using computers are:-

1. Formulate the problem carefully and determine exactly the objective to be reached.
2. Find a method for solving the problem. This is sometimes referred to as finding an ALGORITHM for the solution of the problem.
3. Organise the information associated with the problem, in a way suitable for processing by the computer. This is called PREPARING THE INPUT DATA.
4. Prepare instructions for the computer. A program is a sequence of instructions, which is a translation of the algorithm found in step 2., written in a computer language.
5. Run the program on the computer, with the input data, to produce an answer to the problem.

If you wanted to solve a problem without using a computer, you would still carry out the first two steps, and some modification of the third.

1.3. Programming languages

Each computer has its own 'private' language called its MACHINE language. Programming in MACHINE LANGUAGE is very tedious because:-

1. Detailed knowledge of how the computer works is necessary.
2. The program will only be able to run on the type of machine for which it was originally written.
3. The machine instructions are very primitive, and very many of them are required for a simple program.

Attempts to overcome such problems have resulted in the so-called HIGH LEVEL LANGUAGES such as FORTRAN, ALGOL, PASCAL, PL/1, COBOL.

A computer cannot directly accept a high level language, so a program, called a COMPILER is used to read the high level language and translate it to machine language. Steps involved in using a high level language are:-

1. Problem analysis, and the formation of an algorithm to be used to solve the problem.
2. Translation of the algorithm to a program written in the high level

language. The program is then transformed into some machine readable form (e.g. punched cards or placed into some storage, say via a terminal). This is referred to as the SOURCE PROGRAM.

3. Compiling the source program into machine language (called the OBJECT PROGRAM or RELOCATABLE PROGRAM) using a compiler. This is referred to as a compilation of a program, and this occurs at COMPILE TIME.
4. Collection of all object programs to form an ABSOLUTE or EXECUTABLE program.
5. Execution of the program, using supplied data (possibly). This occurs at EXECUTION TIME.

The phases are represented below. It is important to understand the different phases that a program will go through before it is actually executed.

1.3.1. Compile time

The compilation (or translation phase) that your program goes through may be represented by the following diagram.

```
+----------+         +-----------+          +-------------+
|          |         |           |          |             |
| Source   |         | Language  |          | Relocatable |
| Program  |--->-----| Compiler  |--->----| Object      |
|          |         |           |          | Program     |
+----------+         +-----+-----+          +-------------+
                           |
                           |
                     +-----+-----+
                     |           |
                     | Program   |
                     | Listing   |
                     |           |
                     +-----------+
```

1.3.2. Collection Time

The output from the compile phase, the relocatable object, is the input for the next phase, that of collecting together all the relocatable objects that are necessary to form an absolute object, which is what the computer will execute.

Relocatable objects come from your program, the system library, and other subprograms that may have been written by other people or by you.

```
+---------+      +----------+      +------------+
| Reloc   |      |          |      |            |
| Object  |      | Collector|      |System Reloc|
| Program |--->--|          |--<---|Library and |
|         |      |          |      |Others      |
+---------+      +----+-----+      +------------+
                      |
                      |
                 +----+-----+
                 |          |
                 | Absolute |
                 |Executable|
                 | Program  |
                 +----------+
```

1.3.3. Execution time

Execution time refers to the execution of your program. Note that much has happened to your program prior to it reaching this point on its way to producing results for you.

Later we will see some Fortran commands that relate to compilation.

1.4. Operating systems

An OPERATING SYSTEM is a large complex program used to control the operation of a computer. All programs execute under the guidance of the operating system. Part of the operating system functions are:-

1. General job scheduling.
2. Identify each user and label his output.
3. Check account numbers
4. Monitor users jobs to ensure they do not exceed a maximum time.
5. Load the required compilers into the computer as needed.
6. Control activities of input and output devices.

The programmer must provide special CONTROL STATEMENTS for the operating system which will specify such things as:-

1. Which compiler to use.
2. Name of the program, and your account number
3. Estimates of total run time.
4. The end of your run.

Such CONTROL STATEMENTS must be provided with each program that is to be run, and they must be prepared according to a fixed format, that is dependent upon the particular computer installation. The operating system control statements are independent of the FORTRAN language, but must be used with each run.

1.5. Program debugging and diagnostics

The source program is entered into the computer and becomes input to the FORTRAN compiler. If the compiler detects any errors it writes messages describing the errors. These messages are called DIAGNOSTICS. Errors in a program are often referred to as BUGS and the process of submitting a program for compilation, examining diagnostics, resubmitting the changed program, and repeating this, is called DEBUGGING a program. The compiler will detect most errors in the SYNTAX or grammar of the Fortran program. The SEMANTICS or meaning of the program may still be 'wrong'. The answers may be wrong in that they may not be the desired ones, but the fault is not in the computer or the compiler. In this case the fault is with the author of the program, who either has a faulty method or has failed to correctly transform the method into a program or has faulty input data.

1.5.1. Debugging tools

1. Intermediate output
A program is normally written to solve a problem and output an answer. If a program is producing incorrect answers, the errors must be located and fixed. In theory all errors can be located by reading the program and examining the input data, but many errors are difficult to locate this way, due to their subtlety and the complexity of some programs.

Sometimes it is difficult to know in which part of a program an error has occurred - indeed, even to know which parts of the program have been executed, and in what order. Intermediate WRITE statements within the program enable you to easily follow what is actually happening - this may then be compared to what should be happening and so errors may be localised, detected and corrected. The WRITE statements may then be removed from the program.

If the value of the answer is incorrect then the programmer may request intermediate output of the values of some variables, at key points in the program such that the programmer has some idea of what their value should be. Then he can, hopefully, detect where the incorrect values first occur and so detect, or at least narrow the search for the location of the error.

2. Simple Test Data
Choose simple input data for which there are known answers and see that the program will produce the desired results. The design of test data is very difficult and is usually not exhaustive. The test data should cover extremes as well as typical data.

3. Desk checking
This is a hand calculation performed by the programmer going through the program step by step. At each step the necessary calculations are performed, and the values of all the variables are recorded. By 'each step' it is meant each instruction or part instruction in the program. In a sense the programmer is to 'play the role of the computer'. The objective is not to check the arithmetic of the computer, but to force the programmer to focus attention on each detail of the program.

4. Manufacturer-supplied debugging aids
Some Fortrans, such as Univac Fortran, come supplied with a debugging package. There are two types of debugging aids available - interactive and static. These aids enable you to trace through the program, print out values of variables when they change, do subscript checking and many other things. The Fortran reference manual has details.

2. CHAPTER 2

2.1. Computer memory

The main memory of a computer consists of entities called WORDS or LOCATIONS. Each word can store a certain amount of information.

Computer memory is vastly different to human memory. Humans tend to display the use of their memory by saying such things as 'I remember when I was last in Thargomindah and all the frogs climbed trees'. The computer memory is quite different. It consists of recall to the extent of:

If you put something into memory, then it will stay there until you put something else in its place.

An analogy to this is recording a piece of music on a cassette tape. The cassette tape now remembers what you recorded on it, to the extent that it will play it back, and this recall will continue until you record something else on it.

Another analogy is if you had a box and you placed a blue sheet of paper in it. If you looked inside this box some time later and saw that the blue sheet was still there, would you say that the box had remembered the sheet of paper? Probably not, and yet this is really how a memory cell on the computer 'remembers'.

2.2. What is stored in computer memory

Some of the things that are stored in the computer memory are:

(a) Integer numbers, e.g. 123, -724. Each memory cell may store one integer number. There is a limit on the size of an integer which varies from computer to computer. On the UNIVAC computer the range is -34 359 738 367 to
34 359 738 367. These ranges are most easily remembered by rounding them to a 10 digit number.

If an integer constant that is out of the legal range is specified in a program, then Fortran will produce a compilation diagnostic of

```
NUMERIC CONSTANT 'xxxxxxx' IS OUT OF RANGE
```

(b) Real (or floating point) numbers, e.g. 62.43, -0.74. Each memory cell may store one real number. A real number is stored as two parts, viz, mantissa and exponent, e.g. 7.3 x 1000000. The magnitude (sign not considered) of a real number must be zero or lie between limits that vary between computers. On the UNIVAC computer, these limits are approximately

$^{-39}$ ^{38}

1.48 x 10 , 3.37 x 10

Up to 8 significant digits will be kept. The numbers of computations that are performed to evaluate a REAL number determines how many of these 8 digits are accurate. We will see the effect of ROUNDING and TRUNCATION of REAL numbers later.

If a type REAL constant is specified that has too many significant digits, then the excess is just dropped (that is, the value recorded is truncated) without issuing a diagnostic.

(c) Computer instructions. This is done automatically for us by the FORTRAN compiler and so the exact form does not concern us.

(d) Text or messages, e.g. FRED. The computer stores characters by using an integer code for each possible alphabetic (A,B,C,...,Z), numeric (0,1,2,...,9), or special ($,/,*,space) character. Text is stored four characters per word on the Univac computer, in a code called ASCII, or 6 characters per word in a code called FIELDATA. Ascii Fortran stores characters in the Ascii code (although text can be represented in Fieldata).

These four different types are stored as some combination of zeros and ones (i.e. a BINARY code) in a computer word. The representations in each case are different, so the real number 3.0 will be stored in a completely different way to the integer 3. Suppose a word contains an integer number. If this word is then referenced as a real number, we should not expect it to be the correct value. It is therefore important that these different types be used correctly. Memory stores all information unselectively, and information only has meaning when you reference it as a certain type.

2.3. Names

Fortran allows us to reference locations in the computer memory by the use of names. Any reference to this NAME will reference the corresponding LOCATION in the computer memory. The information stored in a location may be of two types.

A CONSTANT references a location in memory whose value remains fixed for the duration of execution of the program.

A VARIABLE refers to a memory location whose stored value may be changed during the execution of the program.

An analogy to this is in a bank vault. A bank vault contains deposit boxes in the names of the bank's customers. The boxes are distinguished by a name (being the customer name and account number). So we can refer to box name:

MACINTOSH551549

and put something in the box. Note the difference between the name of the box and its contents.
This kind of box is analogous to the VARIABLE in Fortran as its contents can

be examined OR changed.

2.3.1. Rules for the formation of constant names

The word containing a constant number is given a symbolic name which is the same as its value. Unless a minus sign precedes a constant, it is assumed to be positive.

(a) An integer constant is written as a signed or unsigned string of digits without a decimal point, e.g. 1, -10, 31234

(b) A real constant is written as a signed or unsigned string of digits containing a decimal point, e.g. 10.23, 6., -72.189

For large numbers (e.g. 128700000.0), the real constant may be written with an integral decimal exponent. In this case, the constant name could be 128.7E6 Other examples are -0.179E24, 1278.0E-10
Note that the number to the left of the E may be an integer, but the whole name is the name of a REAL quantity. For example 42E3 is the same as the REAL number 42000.0

(c) A literal constant is written as a string of characters enclosed by single quote characters, e.g. 'FRED', 'I AM'.

The major difference between an integer and real constant is the absence or presence respectively, of a decimal point (or the use of the E format of naming). Thus 3 is an integer constant, but 3.0 is a real constant. The forms are not interchangeable as they are stored and processed within the computer in entirely different ways. Note that 3.0, 3., and 3.00000 are all equivalent.

2.3.2. Rules for the formation of variable names

A variable name

* consists of one to six
* alphanumeric (i.e. alphabetic and numeric) characters,
* the first of which must be alphabetic.

By default, if the name starts with one of the letters I,J,K,L,M,N, then the variable is of the type INTEGER and refers to a word that may store INTEGER numbers only. If the name begins with one of the other alphabetic characters, A to H, O to Z inclusive, then the variable is of the type REAL and refers to a word that may contain REAL numbers only.

For example,

I, I10, NAME	are integer variable names, while
A, DATA, PLANE	are real variable names.

It should be noted that the compiler places no significance on variable names beyond inspecting the first letter to establish whether the variable is integer or real. A name such as B7 does not mean B times 7, or B raised to the 7th power. Most programmers assign variable names that simplify the recall of the meaning of the variable, but no such meaning is attached by the Fortran system. It should also be noted that every combination of letters and digits is a separate name. Thus the name ABC is not the same as BAC, and A, AB, and AB8 are all distinct.

2.4. Arithmetic expressions

Arithmetic expressions may contain

- variable names,
- constants,
- arithmetic operators, and
- brackets.

The arithmetic operators are

+	representing	addition,
-		subtraction,
*		multiplication,
/		division, and
**		exponentiation.

2.5. Formation of Arithmetic expressions

The basic expression is a single operation and has the form

```
<variable or constant>  <operator>  <variable or constant>

        or              <operator>  <variable or constant>
```

Examples:- A*2, A-B and -B are basic expressions.

The result of evaluating an expression is a variable or constant. Complex expressions are built up from basic expressions, with operations being done one at a time. For example, A + B + C is evaluated in two stages as follows:-

```
        A   +   B   +   C
        |       |
        ---------
            |
         variable   +   C         intermediate variable created.
            |           |
            -------------
                  |
              variable            answer.
```

Variables and constants must be separated by an arithmetic operator.

Two arithmetic operators cannot be adjacent. If this situation arises, then the operators must be separated by brackets. For example, A*-B is illegal and should be A*(-B). The error could also be corrected by using -B*A.

2.5.1. Order of evaluation of an arithmetic expression

Arithmetic expressions, in algebra, may be interpreted in different ways. This ambiguity is not acceptable in a programming language, and so there are strict rules which determine the order of evaluation of an otherwise ambiguous expression.

For example, consider the expression

```
HOURS*RATE + BONUS
```

Do we do the multiplication first, or the addition first?. There is a vast difference in the answer we may get.
In evaluating an expression, the order is

(1) brackets, (innermost first)
(2) exponentiation,
(3) multiplication and division,
(4) addition and subtraction.

For example, the expression A*B-C/D is evaluated as if it were written (A*B)-(C/D). Where two operators have the same priority (e.g. multiplication and division), the order is taken from left to right. For example, the expression A/B*C is evaluated as (A/B)*C which will probably be different from A/(B*C). Exponentiation is the exception : I**J**K is evaluated as I**(J**K), and not (I**J)**K.

A more complex expression is:

```
A + ((B-C)*D + (E-F**K))/3.2*G+H
```

1. The expression ((B-C)*D + (E-F**K)) is in brackets so it will be evaluated first. Within this expression:
   1.1 (B-C) is the leftmost bracketed expression, so it will be done first.
   1.2 (E-F**K) is the next bracketed expression to be evaluated. Within this expression:-
       1.2.1 F**K is evaluated

1.2.2 E - 1.2.1 is evaluated
1.3 1.1 * D is evaluated.

Finally the result for 1. is obtained by adding:
1.3 + 1.2.2

2. The result from 1. is divided by the real constant 3.2

3. The result from 2. is multiplied by G

4. A is added to the result from 3

5. H is added to the result from 4 giving the final answer.

Another way of representing the above explanation is to place a number under the arithmetic operators to indicate the order in which they are applied.

```
 A  +  ((B  -  C)  *  D  +  (E  -  F  **  K)) / 3.2  *  G  +  H
    8        1     4    5      3     2        6      7     9
```

A quick method of checking that an expression has the correct number of right and left brackets is to count each bracket, adding 1 to the total for each left bracket and subtracting 1 for each right bracket. The total should be 0. In the above example:-

```
    A + ((B - C) * D + (E - F ** K)) / 3.2 * G + H
         12    1       2           10                     correct!
```

In the erroneous expression:-

```
    A + ((B - C) * D + (E - F ** K) / 3.2 * G + H
         12    1       2          1
```

A positive total indicates a surplus of left brackets but does not show where the error is.

2.5.2. Expression Mode

Integer expressions and real expressions are arithmetic expressions whose resulting values are of type integer and real respectively. The result of an expression containing one operand, such as -B, has the same type as the operand. The result of a basic expression with two operands, such as A + B, has the same 'type' as its components if they are both the same type. If both components are INTEGER then the resultant variable or constant will be INTEGER. For example:-

```
  I + J  gives an integer result.
  X/Y    gives a real result.
```

2.5.3. Integer division

This is the division of an integer quantity (either a constant or an integer variable) by another integer quantity. The result of the division is an INTEGER.

Where the division is not exact, the result is TRUNCATED to an integer value. That is, the result of 12/5 is 2 and not 2.4, while the result of -9/4 is -2 and not -2.25.

As a result, 10/3*4 is 12 while 10*4/3 is 13. Also, 10/3*3 is 9 and not 10.

N.B. It is NOT rounded.

Note that 2**(-3) is equivalent to 1/(2**3), which is 0 and not 0.125

2.5.4. Mixed mode expressions

Fortran allows mixed mode arithmetic expressions. A mixed mode arithmetic expression is one that contains more than one type of variable or constant (i.e. mixture of integer and real).

If there is a mixture of REAL and INTEGER, then the INTEGER is converted to REAL, internally by Fortran, and the result is of type REAL.

The result of 10/3 + 4.2 is 7.2
The result of 10/3.0 + 4.2 is 7.533

The resultant type of an expression is determined as follows:-

		Right expression	
		Int	Real
Left	Int	Int	Real
Expr.	Real	Real	Real

2.6. The ASSIGNMENT statement

An assignment statement is of the form

```
        variable = expression
```

where variable is a legal variable name, and
expression is a well formed arithmetic expression.

The assignment statement causes the expression on the right to be evaluated and the resulting value to be stored in the variable on the left hand side of the = character. The assignment statement is read as:-

    variable BECOMES the value of the expression

as the variable will become the value of the expression, thus losing any previous value that it may have had.
The expression on the right is evaluated according to the rules we have seen for evaluating expressions. References to variables on the RIGHT cause their values to be fetched from memory. It is impossible to fetch a value of a variable from memory, unless it has already been given a value somehow, prior to this point in the program. A violation of this principle is just as ridiculous as an attempt by us to know the future. So an expression on the RIGHT is said to be EVALUATED. The name on the LEFT is the place in the computer memory in which the result of evaluation of the right side is stored.

For example, after the statements

```
        A = 3.0
        B = 12.3
        C = -10.2
        D = 47.1
        D = A*B+C
```

are executed, the values of the variables will be

```
     A         B         C         D
    3.0       12.3     -10.2      26.7
```

We can now say such things as:

    3.0 has been stored in A
    12.3 is the current value of B
    C has been overwritten by the number -10.2
    D is 26.7

The statement

```
     I=I+1
```

is legal and results in the value of the variable I being incremented by 1. The = character is not an EQUALS sign in the sense of algebra, but is sometimes known as a REPLACEMENT operator as it causes the REPLACEMENT of the current value of the variable on the left hand side with the result of the expression on the right hand side.

2.6.1. Mode conversion across the replacement operator

In the statement

```
        variable = expression
```

the mode of the variable (i.e. INTEGER or REAL) and the expression do not necessarily have to be the same. For example,

(a)
```
      I=A+B
```
(b)
```
      A=I*J
```

are both legal. In these cases, the expression is evaluated according to its mode (viz real for (a) and integer for (b)). The result is then converted to the mode of the variable on the left hand side before being stored. So the statement A=3 results in 3.0 being stored in A. The statement ICE = 4.8 results in the number 4 being stored in the variable ICE.

2.7. OVERFLOW, UNDERFLOW, and DIVIDE CHECK errors

If a result (integer or real) is calculated whose magnitude is greater than the maximum value allowed, (which is specific to a particular machine) than an OVERFLOW is said to have occurred. An UNDERFLOW occurs when the magnitude of a real result is less than the minimum value allowed. Division by zero (either integer or real) is not defined and results in a DIVIDE CHECK condition. All of the above are considered to be errors, but UNDERFLOW is generally considered far less serious.

These errors may be handled differently on different machines, and you should acquaint yourself with the actions taken for these errors on the computer you are using. The actions may vary from ignoring the error (and not telling you), or reporting that it happened somewhere in the program, to stopping the program when one occurs.

Univac Ascii Fortran provides a number of subroutines to trap arithmetic faults. Although these are important, they will not be covered in this course, but the reader is referred to the Univac Ascii Fortran Manual for details.

2.8. Exercises

2.8.1. Exercise 2A

Write FORTRAN statements to evaluate the following algebraic formulae.

(i) $$z = \frac{a+b}{c+d}$$

(ii) $$z = a + \frac{b}{c+d}$$

(iii) $z = \frac{a+b}{c} + d$

(iv) $z = a + \frac{b}{c + \frac{d}{e}}$

(v) $z = \frac{n(n-1)}{2}$

(vi) $z = \frac{(a-b)(c-d)}{e(f+g)}$

(vii) $y = -2.314 + (5.67z - 3.29 \times 10^{-4})z^{3} + 4.13z^{7}$

2.8.2. Exercise 2B

Identify each of the following as either a REAL constant, an INTEGER constant, or neither.

(a) 0.001	(f) 3435.11
(b) .2	(g) $66
(c) 77	(h) 6.1E-.5
(d) 1-23	(i) 467+1
(e) 87E-05	(j) 43.1.2
	(k) 223.

2.8.3. Exercise 2C

Identify which of the following are legal variable names. Which ones are INTEGER variable names, and which are REAL variable names?

(a) QAZ	(g) TUFF.
(b) COMPUTER	(h) BSCBSC
(c) REAL	(i) A+B
(d) 69	(j) 3XY
(e) CD-5	(k) AJKLMN
(f) L9A5Z2	(l) BLOOD

2.8.4. Exercise 2D

What is the error in each of the following Fortran arithmetic statements?

Notice that although we may recognise errors in the statements below, we cannot correct them, as there are many possible changes that may be made, which would correct them. The Fortran compiler will also adopt this approach, and these statements would result in errors occurring at compile time.

(i) `X = A+3B`

(ii) `Y = ((A+B)*(C+D)-(A-D)*B-C))`

(iii) `3.14159 = PI`

(iv) `J = M-4.5*N**-2`

(v) `AMOUNT = BALANCE + RECEIPT - SALES`

(vi) `X+Y = (A+B)/Z-2*PI*R`

(vii) `TOAST = (TOAST + BUTTER)(BREAD/TOAST)`

2.8.5. Exercise 2E

What would be the result of executing the following statements? The initial values of the variables are

```
A = 6.0      X = -9.0      J = 2
B = 3.6      I = -3        N = 5
```

(i) `Z = (A+B)/X`

(ii) `Z = J+I+N`

(iii) `Z = (J-I)/N`

(iv) `Z = N-I/J`

(v) `Z = N/J+I+2`

(vi) `K = B`

(vii) `K = N/J`

(viii) `K = B*3.0 + X`

(ix) `K = (A+B-X)/3.0`

(x) `K = I*J/N`

(xi) `K = I/N*J`

3. CHAPTER 3

3.1. The Fortran character set

A Fortran program is written using the following characters.

(1) The twenty-six upper case alphabetic characters A through Z.

(2) The ten numeric digits 0 through 9.

(3) The thirteen special characters:

	blank
=	equals
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
$	currency symbol
'	apostrophe
:	colon

In addition to the standard character set above, Univac Ascii Fortran includes lower case alphabetics and the special characters:-

<	less than
>	greater than
&	ampersand

The full Ascii character set is legal in literal constants, eg. 'why?' , '[...]'
See Appendix 3 for the table of Ascii characters.

3.2. Spaces in Fortran

Except for certain specified uses, e.g. as part of a literal constant, spaces or blanks have no meaning and may be used freely to improve the appearance of the program. For example,

```
        AMP = AJ + 3.6
```

and

```
        A    MP  = AJ+  3.   6
```

and

```
        AMP=AJ+3.6
```

are equivalent, the first being the most desirable.

3.3. Fortran Program Layout

A program is a sequence of statements and comments written on 80-column lines. Each statement of a program is written on a separate line. The 80 columns are divided into a number of fields with different uses. Only the first 72 columns are read by the computer.

3.3.1. STATEMENT line

A statement is written in columns 7 through 72 . The only Fortran statement covered so far is an assignment statement. So an example of a statement is:-

```
  column:-       7

                 AMA = (BAG-12.6) + 2345.2*FEE - DOC
```

For good program layout you should start each Fortran command on column 7 (rather than after column 7). This makes your program easier to understand, easier to read, fix and alter.

3.3.2. CONTINUATION line

Often a statement may require more than columns 7 through 72 and so a method of continuing the statement is required. This may be done by putting a non-zero, non-blank character in column 6 of the line that is the continuation of the previous line. Up to 19 continuation lines are allowed for a single statement. An example of this is if we wanted to spread the following statement over 3 lines

```
      INCHES = MILES*1760*3*12 + YARDS*3*12 + FEET*12 + INS
```

This is not normally done if it is not necessary, however the following three lines have an identical effect to the one line above.

```
column:         67
                 INCHES = MILES*1760*3*12

                $         + YARDS*3*12

                $         + FEET*12 + INS
```

When using continuation lines, it is best to indent the continued lines to highlight the continuation, as in the above example. This practice makes your program easier to read and understand.

3.3.3. COMMENT line

The letter C (or an asterisk) in column 1 designates that line as a comment, and whatever follows on that line is the text of the comment. A comment line does not affect the program in any way, and is available as a means of documentation and a convenience for the programmer. A line containing blank characters in columns 1 to 72 is also treated as a comment, and may be used to space out the program. Comments are allowed between the lines of a statement which has continuations. In the above example, there is a blank line between each line of the statement.

If a line has a C in column 1 the rest of the line is IGNORED by Fortran, but it will be printed in the listing of the program. Comment lines may not be continued. To have a multi-line comment, put a C in column 1 of each additional line. All programs should commence with a series of comments to detail the following things:-

1. The name of the author of the program.

2. The date

3. What the program does from the point of view of a user of the program - including information as to how the user should prepare any required input data for the program.

4. What the program does from the point of view of a person trying to understand the Fortran program.

5. Any limitations of the program, and how they may be overcome.

The comments at the program head should be made using English sentences rather than pseudo Fortran sentences.

It is easier to locate the different comment sections if standard headings are used and if the comments themselves are indented (as shown below).
For example, you may find the head of a program has:-

```
C
C    Author:  Fortesque Quincy Zladinov Mc Smith The Third
C
C      Date:  27th June 1976
C  Modified:  January 1981  by Les Landau
C             Modifications were to clarify comment headings
C
C  Program Description:
C             This program finds the average of numbers read in
C             from input data. Any number of data lines may be
C             read. The last line
C             contains a sentinel (the number -999) and so
C             no other line may contain this value.
C             To change the ending indicator, change the value
C             of IFIN in the program.
C
```

```
C  Input Description:
C           The numbers to be read in must be integers, one per
C           line right justified in columns 1 - 6.
C           The last line to be read in is a line with -999 in
C           it. This special ending indicator may be altered by
C           changing the value of IFIN in the program below.
C
C   List of Variables Used:
C
C   AV........CONTAINS THE AVERAGE OF THE NUMBERS
C   IFIN......CONTAINS THE ENDING INDICATOR
C   N.........CONTAINS THE NUMBER JUST READ IN
C   NUM.......CONTAINS THE NUMBER OF DATA LINES READ
C   TOT.......CONTAINS THE TOTAL OF THE NUMBERS
C
C
```

The above may seem to be rather verbose and unnecessary, however it does explain all that a person needs to know to either USE or MODIFY the program. Remember that INCORRECT or MISLEADING comments are far worse than none, so ensure that those that you use are accurate!

In addition to the above program header, there should also be comments within the program, describing each logical section.

Appendix 1 contains information regarding the commenting of programs in general and specifically on what is expected for your assignments. You should read this appendix before attempting your assignments.

3.3.4. Statement Label

Optionally, a statement other than a comment may be labelled so that it may be referenced in other statements. In Fortran, a statement should not have a label unless it is referenced by some other statement in the program (we will see such commands as DO and GO TO later). In the programming language called BASIC, all statements must be labelled.

A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the first line of a statement, i.e. not on continuation lines. Leading zeros are not significant, nor are blanks.

The same statement label may not be given to more than one statement in a program, otherwise the label is not unique and a reference from another statement to this label will be ambiguous.

A statement label is also known as a statement number.

Although not required, statement numbers are generally allocated in ascending order so that a referenced statement number may be easily found without having to search the complete program.

For example:-

```
column:  1        7
         65       RAIN = PERIOD*MOIST + 0.1*PROB
```

3.3.5. Sequencing

Columns 73 through 80 are ignored by the Fortran compiler but are printed as part of the compilation listing. These columns may be used to contain sequence numbers, so that if the program is on cards and the card deck is dropped, it may be easily put back in order by sorting on the sequence numbers. The program name may be put in the first four columns of this field, with numbers in the next four columns. As in the case of statement labels, the numbers may be incremented by 5 or 10 to allow for lines that may need to be inserted at a later stage.

This field is normally not used for small programs (of less than 50 lines). Programs for this course do not require sequencing.

3.4. Structure of a Fortran Program

A program consists of a number of lines of Fortran statements. The program is executed sequentially, starting at the first executable statement. A comment is not executable. The execution of each statement is completed before going on to the next one.

3.4.1. END Statement

The physically last statement in a program must be an END statement which contains the characters END anywhere in columns 7 through 72. This statement indicates to the compiler that this is the last statement to be compiled.

3.4.2. STOP statement

When a STOP statement is encountered, the execution of the program terminates. There must be at least one STOP command in a Fortran program.

Remember the difference between compile time and execution time, as covered in Chapter 1? Understanding this difference will help in understanding the different functions of the END and STOP statements.

3.4.3. Simple Fortran Program

So a valid Fortran program is

```
C     THIS IS A SAMPLE PROGRAM
C
      A=2.0
      B=3.0/A
      STOP
      END
```

Each of these statements is entered on a separate line.

A program may have several STOP commands, but only one END directive. Note that there is no output from this program, and that values of variables are not kept after the program stops.

3.5. Reading and writing in Fortran

Programs must converse with the outside world, to locate any data to operate on, and to display the results of any calculation. In batch mode this is done via the card reader for input and the line printer for output. When running interactively you can also read and write to a terminal that is executing the program, and also read and write to a data file (or files).

There are Fortran statements to do this, and they are known as INPUT/OUTPUT or I/O statements. Initially we will use free format (sometimes called list directed) input and output, and only refer to the card reader (or terminal) and the line printer (or terminal). Later we will discuss fixed format I/O and reading and writing data files.

3.5.1. READ statement

The READ statement is of the form

```
      READ (5,*,END=n) input list
```

where input list is a list of variable names separated by commas.

Execution of this statement causes some numbers to be read from data . The number of numbers to be read is determined by the number of variables on the input list. The numbers read are stored in the respective variables on the input list.

The first parameter in brackets is the 'unit number' for I/O. Unit 5 means that data is to be read from the card reader (in batch) or the terminal (on-line). Up to 80 columns per line are read.

The END=n clause (and the preceding comma) may be omitted. If present, and an end-of-file condition is encountered (by reading in a line of data with an

@-sign in column 1, apart from @ADD, on the Univac) then the program will jump to the statement whose number is n. If the END=n clause is missing, and you attempt to read a line of data with an @-sign in column 1, then you will get a run-time error and your program will terminate.

If the end-of-file is indicated by an @EOF image, then reading may continue at some other point in the program. If it is any other image (apart from @ADD) then any attempt to read more will result in an error.

3.5.2. Layout of data

The data values themselves appear in the RUN on the lines following the @FTN,C and @EOF commands. The numbers are separated by commas or blanks. If you use commas to separate values, then do not have any blanks in the data. If you use blanks to separate data values, then do not have any commas in the data.

3.5.3. Examples of read

1. For example, the statements

```
      READ (5,*) WEIGHT,AGE,MONEY
```

will read three numbers where

(i) the first number is a REAL and will be assigned to the variable WEIGHT,

(ii) the second is also REAL and will be assigned to AGE,

(iii) the third is INTEGER and will be assigned to MONEY.

The data will contain these three numbers separated by commas. Note that real numbers MUST contain a decimal point, and that integer numbers must NOT contain a decimal point. The data line for this example could be

```
17.23,-150.,-7
```

The above read statement would cause the variables WEIGHT, AGE, MONEY to contain the numbers 17.23, -150.0, and -7 respectively.

2. To read in a value for the variable NUM. If no value is in the input stream, then assume a value of 1.

```
      NUM = 1
      READ (5,*,END=10) NUM
   10 WRITE (6,*) 'VALUE FOR NUM IS ', NUM
```

3.5.4. WRITE statement

The WRITE statement is very similar to the READ statement and is of the form

```
        WRITE (6,*) output list
```

where output list is a list of variable names separated by commas, whose values are to be written out by the program. Analogous to the READ statement, the list indicates the variable names whose values are to be printed.

Unit 6 means that output is to be written to the line printer (in batch) or terminal (on-line). Up to 132 columns per line are written.

For example, to print the numbers that were read in the previous example, the statements could be

```
        READ (5,*) WEIGHT,AGE,MONEY
        WRITE (6,*) WEIGHT,AGE,MONEY
        STOP
        END
```

Try this program on the computer so that you may see what happens. Don't forget to include a line of data with the three numbers on it.

3.5.5. Examples of READ and WRITE statements

READ statements should be used to read data that may vary from one run of the program to another. Consider the following problems.

Find the average of the ten numbers 1.3, 2.4, 5.4, 6.3, -3.7, 0.0, 13.4, -12.0, 17.7, -21.3.

In this problem, all data has already been defined and so there is no need to read any data. The program would be

```
C
C     FIND THE AVERAGE OF TEN SPECIFIED NUMBERS
C
      FNUM = 10.0
      AVER = (1.3 + 2.4 + 5.4 + 6.3 - 3.7 + 0.0 + 13.4 - 12.0 +
     $       17.7 -21.3)/FNUM
      WRITE (6,*) AVER
      STOP
      END
```

Find the average of any ten numbers.

In this problem, the values of the ten numbers are not known when the program is written. Thus, they must be supplied as data and so must be read by the program which could be as follows:

```
C     FIND THE AVERAGE OF ANY TEN NUMBERS
C
      FNUM = 10.0
      READ (5,*) VAL1,VAL2,VAL3,VAL4,VAL5,VAL6,VAL7,
     $           VAL8,VAL9,VAL10
      AVER = (VAL1+VAL2+VAL3+VAL4+VAL5+VAL6+VAL7+
     $       VAL8+VAL9+VAL10)/FNUM
      WRITE (6,*) AVER
      STOP
      END
```

Find the average of any number of numbers.

Now, not only the values of the numbers, but also the number of numbers, are not known when the program is written. The program would thus have to read a value indicating how many values there were, and then read that many values. It is not yet possible to write this program, but it will be set as an exercise later.

3.6. Writing out heading information

The method below of writing out headings is very non-standard, but because it is also very easy we will use it initially, and will discuss the more standard methods later.

The output of numbers on their own is quite often confusing to interpret. Tables of figures in books and even single results usually have some accompanying text to indicate what the results mean, or how they were obtained.

Fortran provides a mechanism to allow headings to be output very easily by means of a WRITE statement.

To write a heading that will appear at the beginning of a list of values that are to be written out, include the heading in the output list, enclosed in single quotes. If you want to include a quote sign within the heading itself, then the quote must be followed immediately by a quote.

For example, to write out values for ECCLES and JIM to appear after a heading of GOON WITH THE WAND

```
      WRITE(6,*)' GOON WITH THE WAND',ECCLES,JIM
```

Headings may be written out on their own, without any variable values. In this case the corresponding write statement simply has the heading on its own.

For example, to write out:

```
MIN, MIN, MIGHTY MODERN MIN
HENRY, WHAT HAVE YOU DONE WITH THE PAPER?
AH MIN, IT'S IN THE BOX MARKED        3
```

the following lines of Fortran may be used.

```
          I = 3
          WRITE(6,*)' MIN, MIN, MIGHTY MODERN MIN'
          WRITE(6,*)' HENRY, WHAT HAVE YOU DONE WITH THE PAPER?'
          WRITE(6,*) ' AH MIN, IT''S IN THE BOX MARKED ', I
```

3.7. Exercises

3.7.1. Exercise 3A

Read in 4 numbers. The first two should be real, the next two should be integer.
Write out the numbers in reverse order.

3.7.2. Exercise 3B

Read two integers i and j. Do the following calculations and write out the results.

$k = i + j$

$l = i \times j$

$m = i / j$

$n = i^{j}$

3.7.3. Exercise 3C

Read a positive real number. Place the integral part (ie, the whole number part) into a variable called INT and the fractional part into a variable called FRACT. Write out the number, INT, and FRACT.

For example, if the number is 53.261, then INT = 53 and FRACT = 0.261.

3.7.4. Exercise 3D

Read in two real numbers. Calculate the difference between the first one squared and the second one squared. Write out the the two numbers, their squares and the difference.

4. CHAPTER 4

4.1. Relational expressions

A RELATIONAL EXPRESSION compares the values of two arithmetic expressions. The expressions are separated by a RELATIONAL OPERATOR and the result will have the value TRUE or FALSE, as the relation is TRUE or FALSE. The relational operators are

.LT.	meaning	less than
.LE.		less than or equal to
.EQ.		equal to
.NE.		not equal to
.GE.		greater than or equal to
.GT.		greater than

Some examples of relational expressions are

(a) PAY .LT. CREDIT

The arithmetic expressions involved here are simply single variables. The relational expression will have the value TRUE if the value of the real number stored in PAY is less than the real value stored in CREDIT. Otherwise (i.e., if PAY is greater than or equal to CREDIT), it will have the value FALSE.

(b) NURKE*2.GE.MIN**3/KO

Here the arithmetic expressions are a little more involved. The relational expression will have one of the following values.

TRUE if NURKE*2 is greater than or equal to MIN**3/KO

or

FALSE if NURKE*2 is less than MIN**3/KO

(c) Brackets may be used within the arithmetic expressions involved.

```
(KOLD*ICE)/(MAN-KG)**4 + 7.EQ.((MINE - IODINE)*3)**JEWEL
```

Relational operators have a lower order of precedence than arithmetic operators, so the arithmetic expressions are evaluated first.

4.1.1. Use of .EQ. with REAL variables and constants

The computer cannot represent most real numbers exactly due to the nature of storage of these numbers in a fixed size memory word. For example, in decimal arithmetic, 1/3 cannot be represented exactly in 6 significant digits, and is approximated by 0.333333. If this is then multiplied by 3, then you have (approximately) 0.999999 which is not exactly equal to 1.0

Also, errors may be introduced when performing arithmetic operations on REAL numbers. For these reasons, it may not be meaningful to form a relational expression using .EQ. between two REAL arithmetic expressions.
For example, 0.1 cannot be represented exactly in a computer word and in fact could be 0.1 + e where e is a very small amount. If this is then multiplied by 10, the result is

$$10 \times (0.1 + e) = 1.0 + 10 \times e$$

which has ten times e. Hence, 10 x 0.1 is not EXACTLY equal to 1.0

4.2. Logical Expressions.

A logical expression is a relational expression, or a combination of relational expressions. These expressions may be combined with logical operators. The logical operators are:-

.OR.	meaning logical disjunction, and
.AND.	logical conjunction.

The logical expression a.AND.b is TRUE if and only if both the logical expressions a and b are TRUE. The logical expression a.OR.b is TRUE if and only if at least one of the logical expressions a and b is TRUE. (i.e. it is an INCLUSIVE OR).

AND. has precedence over .OR.
Arithmetic operators have precedence over relational operators, which have precedence over logical operators. If in doubt, always use brackets.

For example, the logical expression:-

```
      JILL .EQ. JACK*2 .OR.  JACK .GT. 0 .AND. JACK .LE. 100
```

is equivalent to

```
      (JILL .EQ. (JACK*2)) .OR. ((JACK .GT. 0) .AND. (JACK .LE. 100))
              2       1     6           3       5          4
```

The numbers show the order of evaluation.

4.3. Logical IF statement

A logical IF statement is of the form

```
      IF(logical expression) S
```

The logical expression is as described above.

S ... is any executable statement except a DO statement, another logical IF statement or any block-IF command.

The logical expression is evaluated, and if it has the value TRUE, the statement S is executed. If the value of the expression is FALSE, the statement S is not executed. In either case, control then passes to the next statement.

4.3.1. Examples of Logical IF statement

(a)
If PAY is less than 4000.00, then increase PAY by 275.25

```
        IF(PAY.LT.4000.0) PAY = PAY + 275.25
```

(b)
Read a card. If the integer on it is equal to 7, then read another card.

```
        READ (5,*) NUMBER
        IF(NUMBER.EQ.7) READ (5,*) NUM2
```

(c)
Read a card. If the number on the card was 99 then stop.

```
        READ(5,*)NUM
        IF(NUM.EQ.99)STOP
```

(d)
Test if SALARY is in the range 30,000 to 100,000 and write out a message saying FAT CAT if it is.

```
 IF (SALARY.GE.30000.0 .AND. SALARY.LE.100000.0) WRITE(6,*)'FAT CAT'
```

4.3.2. Common errors with IF statements

(a) Where you want to do one of two things depending on a test.

Suppose that you want to test to see if PAY is less than STARVE then add 250.0 to PAY, but if it is not less than STARVE then only add 100.0 to PAY

One solution, which is incorrect, may be:

```
        IF( PAY.LT.STARVE ) PAY = PAY + 250.0
        PAY = PAY + 100.0
```

This will not work of course because if PAY was less than STARVE then we would add on 250.0 to PAY, but then we would also add on 100.0. It is true that it would work in the case of PAY being greater than (or equal to) STARVE, as we would only add 100.0

So, another attempt at a solution leads us to another erroneous answer:

```
        IF( PAY.LT.STARVE ) PAY = PAY + 250.0
        IF( PAY.GE.STARVE ) PAY = PAY + 100.0
```

Why is this wrong? It seems to work: if PAY is less than STARVE then we will add on 250.0 and then we test again and only if PAY is greater than or equal to STARVE do we add on 100.0

Well this WILL work, but if PAY lies between STARVE and STARVE - 250.0 then PAY will be increased by 350.0! Try it in the case of STARVE having the value 4000.0 and PAY having the value 3950.0

Assuming that this was not the intention of the exercise, the following solution will work:

```
        IF( PAY.LT.STARVE ) ADDON = 250.0
        IF( PAY.GE.STARVE ) ADDON = 100.0
        PAY = PAY + ADDON
```

compare this with:

```
        ADDON = 100.0
        IF( PAY.LT.STARVE ) ADDON = 250.0
        PAY = PAY + ADDON
```

4.4. Block-IF statements

The logical IF statement above is limiting in that only one statement is allowed after the logical expression on the IF.

1977 standard Fortran introduced a construction known as a block-IF, that allows several Fortran statements to be executed as a result of one logical test. Further, it incorporates an ELSE mechanism which can be used to specify a block of statements that are to be performed if the logical test comes out false. This reduces the number of GO TOs in the program, thus making the program easier to read and debug.

4.5. Block-IF terminology

Block-IF	The name of the IF statement. It starts with an IF......THEN statement (see below) and ends with an END IF statement.
IF-block	The group of lines that lie between an IF......THEN and the next ELSE IF, ELSE or END IF statement.
ELSE-IF-block	The lines that lie between an ELSE IF statement and the next ELSE IF, ELSE or END IF.

ELSE-block
: The lines between an ELSE statement and the following END IF statement.

4.6. Basic Block-IF

The form of the block-IF is:

```
      IF (logical expression) THEN

          +------------------------+    --+
          |                        |      |
          |    lines of Fortran    |      >  The IF-block
          |                        |      |
          +------------------------+    --+

      END IF
```

4.6.1. Evaluation

The logical expression is evaluated. If it has the value of TRUE then all the lines of Fortran in the IF-block are executed.
If the value of the logical expression is FALSE then the program skips the IF-block and continues execution with the statement immediately following the END IF statement.

Any executable statement may appear within the IF-block, including another block-IF. In this case of nested block-IFs there must be an END IF that corresponds to each IF.......THEN

4.6.2. Examples of block-IF

(a) Read in a value for SALARY and if it is negative then write out a message and stop the program. If it is non-negative, then write it out and go on with the program.

```
      READ (5,*) SALARY
      IF (SALARY .LT. 0) THEN
          WRITE (6,*) ' Negative salary is illegal ',
     $                ' A value of ', SALARY,' was read'
          STOP
      END IF
      WRITE (6,*) ' Salary: ', SALARY
```

(b) Read in a value for NUM. If NUM equals 0 then read in another value for NUM, write out a message saying that has been done and calculate values for HALF (half of NUM) and TWICE (2 times NUM).

```
      READ (5,*) NUM
      IF (NUM .EQ. 0) THEN
          READ (5,*) NUM
          HALF  = NUM/2.0
          TWICE = NUM*2
          WRITE (6,*) 'Second value of ', NUM, ' read'
      END IF
```

In this program, if NUM was originally read in as non-zero then HALF (and TWICE also) would not have a value.

(c) Same exercise as above, but if the second value of NUM is less than 10, then do not calculate HALF or TWICE.

```
      READ (5,*) NUM
      IF (NUM .EQ. 0) THEN
          READ (5,*) NUM
          IF (NUM. GE. 10) THEN
              HALF  = NUM/2.0
              TWICE = NUM*2
          END IF
          WRITE (6,*) 'Second value of ', NUM, ' read'
      END IF
```

4.7. ELSE-IF statement

This forms part of the block-IF. Any number of ELSE IFs may be part of the block-IF.

4.7.1. Syntax of the ELSE IF

```
      IF (logical expression-1) THEN

        +--------------------------+  --+
        |                          |    |
        |     lines of Fortran     |     > An IF-block
        |                          |    |
        +--------------------------+  --+

      ELSE IF (logical expression-2) THEN

        +--------------------------+  --+
        |                          |    |
        |     lines of Fortran     |     > An ELSE IF-block
        |                          |    |
        +--------------------------+  --+

      ELSE IF (logical expression-3) THEN
```

```
        +-----------------------+  --+
        |                       |    |
        |    lines of Fortran   |    > An ELSE IF-block
        |                       |    |
        +-----------------------+  --+

                  .
                  .
                  .
                 etc

     END IF
```

There is only one END IF and that END IF corresponds to the block-IF at the top.

4.7.2. Interpretation of ELSE IF

The use of ELSE IF within a block-IF is to test a logical expression and if it is true then execute the Fortran in the IF-block. If the logical expression is false, then go to the next ELSE IF and test that expression. When eventually a value of TRUE is found, the corresponding block of statements is executed, and then the program skips to the END IF statement.

More exactly:

1. Logical expression-1 is evaluated

2. If it is TRUE then:
   (a) the statements in the IF-block are executed.
   (b) the program skips all the ELSE IF blocks and continues execution following the END IF.

3. If logical expression-1 is FALSE then skip to the next ELSE IF (or END IF if there isn't an ELSE IF) and evaluate the logical expression there (logical expression-2).

4. If the logical expression is TRUE then:
   (a) the statements in the ELSE IF block are executed.
   (b) the program skips to the END IF

5. If the logical expression is FALSE then skip to the next ELSE IF (or END IF if there isn't an ELSE IF) and evaluate the logical expression there.

6. Repeat steps 4 and 5 until eventually arriving at an END IF.

Any of these blocks can contain other (nested) block-IF expressions, but one and only one END IF statement exists for each IF....THEN expression. That is, there is NOT any END IF statement that corresponds to an ELSE IF......THEN statement.

4.7.3. Example of ELSE IF

(a) Test for PAY less than STARVE. If it is, then add 250.0 to PAY, otherwise add 100.0 to PAY.

```
       IF (PAY .LT. STARVE) THEN
           PAY = PAY + 250.0
       ELSE IF (PAY .GE. STARVE) THEN
           PAY = PAY + 100.0
       END IF
```

(b) Read in a number.

Value	Action
1	Read in 3 numbers and write out their average
2	Read in 2 numbers and write out the result of raising the first to the power of the second.
4	Read in two numbers and if the second number is zero, then write out an error message and stop. If the second number > 0 then calculate the value of the first divided by the second.

All input is integer.

```
       READ (5,*) NUM
       IF (NUM .EQ. 1) THEN
           READ (5,*) I1, I2, I3
           AVE = (I1+I2+I3)/3.0
           WRITE (6,*) ' Average is ', AVE
       ELSE IF (NUM .EQ. 2) THEN
           READ (5,*) I1, IPOWER
           IVAL = I1**IPOWER
           WRITE (6,*) ' Power calculation: ', IVAL
       ELSE IF (NUM .EQ. 4) THEN
           READ (5,*) N1, N2
           IF (N2 .GT. 0) THEN
               IQUOT = N1/N2
               WRITE (6,*) ' Integer quotient: ', IQUOT
           ELSE IF (N2 .EQ. 0) THEN
               WRITE (6,*) ' Divisor of zero found'
               STOP
           END IF
       END IF
```

4.7.4. ELSE statement

The form of an ELSE statement is just the command:

```
         ELSE
```

on a line of its own

4.7.5. Use of ELSE

The ELSE block (if present) must come after all ELSE IF blocks (if there are any). Execution of the ELSE block will be done if all logical expressions (at that level of nesting) evaluate to FALSE.

For example:

```
        IF (logical expression-1) THEN

          +-----------------------+  --+
          |                       |    |
          |    lines of Fortran   |     > An IF-block
          |                       |    |
          +-----------------------+  --+

        ELSE IF (logical expression-2) THEN

          +-----------------------+  --+
          |                       |    |
          |    lines of Fortran   |     > An ELSE IF-block
          |                       |    |
          +-----------------------+  --+

        ELSE IF (logical expression-3) THEN

          +-----------------------+  --+
          |                       |    |
          |    lines of Fortran   |     > An ELSE IF-block
          |                       |    |
          +-----------------------+  --+

                      .
                      .
                      .
                     etc

        ELSE

          +-----------------------+  --+
          |                       |    |
          |    lines of Fortran   |     > An ELSE block
          |                       |    |
          +-----------------------+  --+

        END IF
```

The ELSE block will only be executed if logical expressions 1, 2, 3 etc all evaluate to FALSE.

So, each logical expression is tested and if one is found that is TRUE, then the corresponding block is evaluated. Otherwise, the ELSE block is evaluated.

4.8. Notes on block-IF in general

(a) To facilitate the understanding of programs, use indentation within a block-IF. This way statements that fall into the block stand out from those that are not in the block. An indentation of 3 or 4 characters is advised (for each level of nesting).

(b) Any executable statement, comment or FORMAT statement (see chapter 8) is allowed within any of the three different types of blocks. A statement must be wholly contained within a block.

(c) In the next chapter, DO statements are covered. If a DO statement appears in a block, it must be wholly contained in that block.

(d) Chapter 7 introduces a GO TO statement, which transfers control to another part of the program. It is not allowed to transfer control to within a block from outside that block.

(e) Do not use a block-IF statement if there is only one thing that is to be done as a result of a test. In such cases use a simple logical IF statement.

(f) Try to avoid very large block-IFs. This may be done by using a GO TO statement (see chapter 7). Large ranging block-IFs make it difficult to keep track of where you are in a program.

(g) Try to avoid very deep nesting of block-IFs (you are allowed 25 nested levels).

4.9. Exercises

4.9.1. Exercise 4A

Identify the following relational expressions as valid or invalid.

```
(a)    MPX.LE.19+K
(b)    LAX.GT.AMA
(c)    COB.LT..EQ.CORN
(d)    14.NE.LS
(e)    .EQ.6
(f)    (I+19)*K .EQ. J*L*(JJ+I)/K
(g)    P.GRT.Q
(h)    L.=.77
```

4.9.2. Exercise 4B

Read a card containing two integers. Determine which is the larger one and write out the two integers with the larger one appearing first.

4.9.3. Exercise 4C

Rewrite the following using block-IF statement(s).

```
      READ (5,*) MAXIN
      IF (MAXIN .EQ. 16) WRITE (6,*) 'Gotit'
      IF (MAXIN .EQ. 19) FORGET = 0.5
      IF (MAXIN .EQ. 10) FORGET = 0.0
      IF (MAXIN .EQ. 16) STOP
      IF (MAXIN .EQ. 19) WRITE (6,*) ' Found one'
```

4.9.4. Exercise 4D

Write a logical IF statement that will test if A is in the range (-0.00001,0.00001) and if it is, set A to 0.0

4.9.5. Exercise 4E

Rewrite the following using only one level of block-IF (and so only one END IF statement)

```
      IF (IND .EQ 16) THEN
          K = 6
          I = 9
      ELSE
          IF (L .GT. J+4) THEN
              X = 19.6
              READ (5,*) Y
              L = 0
          ELSE
              IF (MAIN .LT. I) THEN
                  COST = 19.0
                  TRY  = 14.2
              ELSE
                  PAY  = 0.0
              END IF
          END IF
      END IF
```

5. CHAPTER 5

5.1. DO statement

The DO statement is a mechanism which enables repeated execution of a block of code (i.e., a group of Fortran statements) a number of times, without having to write the statements repeatedly.

The form of the DO statement is

```
        DO n i = m1,m2,m3
```

where

(i) n is the statement label of the terminal statement of the DO, which must be physically later in the program than the corresponding DO statement.

The range of a DO statement is the block of statements following the DO statement, up to and including the statement labelled n.

(ii) i is an integer or real variable and is called the DO-variable. The value of i may not be changed within the range of the DO-loop.

(iii) m1, called the initial parameter, m2, called the terminal parameter, and m3, called the incrementation parameter, are each an integer or real expression.

m3 is optional and it (and its preceding comma) may be omitted. In that case a value of 1 is assumed for m3. At the time of execution of the DO statement m3 must not equal 0. m1, m2 and m3 may be changed during execution of the loop, but this will not change the number of times that the loop is executed.

5.1.1. Restrictions on terminal statements

The terminal statement may not certain kinds of commands, some of which we have already done. The list of commands that the terminal statement MUST NOT be is

unconditional GO TO
assigned GO TO
END
arithmetic IF
block-IF
ELSE IF
ELSE
END IF
RETURN
STOP

DO

5.2. CONTINUE statement

The CONTINUE statement is an executable Fortran statement which does nothing. When a CONTINUE statement is executed, its effect is a 'no operation' effect, and the program will simply go on to execute the next statement.
The main use of the CONTINUE statement is as the terminal statement of a DO loop, to avoid the illegal terminal statements listed above.

It is a good idea to always end your DO loops on a CONTINUE statement. This way

(a) The loop is easier to alter
(b) The terminal statement stands out more
(c) You always end on a legal statement

5.2.1. The DO statement and 1977 standards

A major change in the semantics of the DO statement occurred when 1977 standard Fortran was announced.

In 1966 standard Fortran, the testing of the DO-variable was done at the end of executing the statements in the range of the DO. This means that the statements in the range of the DO will always be done at least once, even if the finish value of the DO (m2) is less than the start value (m1) with a positive increment (m3).

In 1977 standard Fortran, the test is done BEFORE executing any statements in the range of the DO. This means that you can have a null DO loop (that is, one in which the statements in the range of the DO are not executed at all).

In 1966 Fortran, the DO-variable became undefined when the loop became inactive. In 1977 Fortran, the DO-variable retains its last defined value, i.e. the value after the increment which caused the loop to terminate.

Univac Ascii Fortran level 9R1 (and higher) introduced the 1977 version. If you use a compiler other than this, then you should check to see which version it adheres to.

In the examples below, it is assumed that the 1977 standard is in effect, but for compatability the way of interpreting the 1966 standard is also presented (see Appendix 5).

5.2.2. Evaluation of the DO statement

A DO statement is used to define a loop.

The action following the execution of a DO statement is described in the following steps.

(i) The initial parameter, the termination parameter and the increment parameter are converted to the type of the DO-variable, and the DO-variable is given a value of the initial parameter.

(ii) The iteration count is established and is the value of the expression
```
    maximum of: (a) truncation of  ((m2-m1+m3)/m3)
                and
                (b) 0
```

(iii) The iteration count is tested. If the iteration count is zero, then the DO-loop becomes inactive, and the program continues with the statement immediately following the terminal statement.

If the iteration count is greater than zero, then all the statements within the range of the DO (i.e. the statements following the DO line up to and including the terminal statement) are executed.

(iv) The value of the DO-variable is incremented by the incrementation parameter (m3), and the iteration count is decremented by 1.

(v) The action starting at step (iii) of this procedure is then commenced, and so on around the loop until eventually the test in step (iii) leads to the DO-loop becoming inactive.

So, effectively a DO statement will repeat all the statements from the one immediately after the DO line, up to and including the terminal statement. This will be done a number of times, determined by the interactions of m1, m2, and m3 (bearing in mind that the number of times could be zero).

5.2.3. Examples of DO statements

(a) Read 3 cards and write them out. This will produce 3 lines of output.

```
      DO 25 I = 1,3,1
         READ (5,*) VAL1,TIME,LIMIT
         WRITE(6,*) ' INPUT VALUES WERE:', VAL1,TIME,LIMIT
   25 CONTINUE
```

(b) Write all the even numbers between 1 and 100 inclusive.

```
      DO 6 K = 2,100,2
         WRITE (6,*) K
    6 CONTINUE
```

(c) Read a card containing an integer into the variable INT. Now read INT more cards and write them out. This is a very common method used to enable a program to process a variable amount of data.

```
      READ (5,*) INT
      DO 12 I = 1,INT
         READ (5,*) NUMBER
         WRITE (6,*) NUMBER
  12  CONTINUE
```

(d) Read up to 25 cards and write them out, until a card containing -9 is found, and then stop. This is another method of determining when there is no more data to read.

```
      DO 16 I = 1,25
         READ (5,*) NUMBER
         IF(NUMBER.EQ.-9) STOP
         WRITE (6,*)' INPUT VALUE IS ', NUMBER
  16  CONTINUE
```

(e) Suppose that we read in some population statistics for the years from 1960 back to 1950 (in that order) and we wanted to write them out prefixed by the year to which they correspond.

```
      DO 20 IYEAR = 1960,1950,-1
         READ(5,*)IPOP
         WRITE(6,*)' IN ',IYEAR,' THE POPULATION WAS ',IPOP
  20  CONTINUE
```

This would write out:

```
  IN     1960  THE POPULATION WAS    xxxxxxxx
  IN     1959  THE POPULATION WAS    xxxxxxxx
  IN     1958  THE POPULATION WAS    xxxxxxxx
  etc
```

If you were using 1966 standard Fortran (which does not allow DO loops to go backwards) then you would have the program

```
      DO 24 I = 1950,1960
         IYEAR = 1950 + 1960 - I
         READ(5,*)IPOP
         WRITE(6,*)' IN ',IYEAR,' THE POPULATION WAS ',IPOP
  24  CONTINUE
```

(f) Read 10 cards each containing an integer, and print each number. When a card contains the number -7, stop AFTER printing the number.

```
      DO 18 I = 1,10
         READ(5,*)NUMBER
         WRITE(6,*)' INPUT NUMBER = ', NUMBER
         IF(NUMBER.EQ.-7) STOP
  18  CONTINUE
```

5.3. Nested DO statements

It is possible to have a DO loop wholly contained within another DO loop. This is known as 'nesting DO loops.' For example,

```
      DO 42 I = 1,15              -------------------------+
                                                           |
         . . . code A . . .                                |
                                                           |
         DO 5 J = 3,30,3          -------+                 |
                                         |                 |
            . . .       . . .            |    Inner        |
                                         |                 |
            READ(5,*) L                  >     DO          |     Outer
                                         |                 |
            . . .       . . .            |    LOOP         >      DO
                                         |                 |
   5     CONTINUE                 -------+                 |     LOOP
                                                           |
         . . . code B . . .                                |
                                                           |
 42   CONTINUE                    -------------------------+

      . . . code C . . .
```

The INNER loop (down to statement number 5) is said to be nested within the OUTER loop (which ranges down to statement number 42). The operation of this example is as follows.

1. Set I to its initial value (1)

2. Calculate the iteration count for the outer DO

3. Test the iteration count. If > 0 then go on. If <= 0 then go to step 12.

4. Execute the code marked A

5. Set J to its initial value

6. Calculate the iteration count for the inner DO

7. Test the iteration count. If > 0 then go on. If <= 0 then go to step 10.

8. Execute the code in the inner DO

9. At label 5:
   increment J by 3
   decrement the iteration count of the inner DO
   go to step 7 above.

10. Execute the code marked B

```
11. At label 42:
                 increment I by 1
                 decrement the iteration count of the outer DO
                 go to step 3 above.

12. Execute code marked C
```

5.3.1. Rules for nested DO loops

An inner DO must terminate ON OR BEFORE the terminal statement of the outer DO. This means that DO loops must not 'cross over'. The following nesting construction is ILLEGAL.

```
          DO 12 I = 3,17
          . . . . .
          DO 88 J = 2,37,3
          . . . . .
     12   CONTINUE
          . . . . .
     88   CONTINUE
```

The following construction, where the inner and outer DO loops finish on the same statement, is legal.

```
          DO 16 J = 1,12
          DO 16 K = 3,17,2
          . . . . .
     16   CONTINUE
```

In this example the INNER DO will be completed before control is returned to the outer DO, even though they have the same terminal statement number.
There is a limit to the depth of nesting of DO loops. This limit is not defined in the ANSI standard, but most computers allow at least 5 levels, and usually a lot more, although very few programs ever need more than a depth of 3.

5.3.2. Final value of the DO-variable

When the range of a DO loop is exhausted (ie, when the loop variable has a value that exceeds the terminal parameter on the DO statement), then control is passed 'out the bottom' of the DO loop. When this occurs, the value of the DO-variable retains its last defined value, which is the value after the last increment, when the iteration count is zero.

5.4. Examples of nested DO loops

5.4.1. Example 1

Suppose we want to find out the average salary earned in different electorates. We have a number of electorates to process and a variable number of people in each electorate.
The program below will first read in a card indicating how many electorates there are. Following this there will be sets of salary data for each person in each electorate, one salary per card. The first line in each electorate set indicates the number of people in that electorate, and then following this card will be that number of salary lines. Each salary is a real number.
For example if there were 3 electorates, with the following numbers of people in each:

electorate 1	2 people	earning	\$256.50
			\$291.85
electorate 2	4 people	earning	\$196.45
			\$202.44
			\$180.00
			\$175.10
electorate 3	3 people	earning	\$120.45
			\$133.22
			\$110.90

Then the data deck would look like:

```
3
2
256.50
291.85
4
196.45
202.44
180.00
175.10
3
120.45
133.22
110.90
```

```
C     AUTHOR:       G. MANDER
C
C     DATE:         NOVEMBER 1979
C
C     PURPOSE:
C
C      TO FIND THE AVERAGE SALARY OF PEOPLE IN A NUMBER OF
C      ELECTORATES. THE AVERAGE IN EACH ELECTORATE IS PRINTED,
C      AND AT THE END THE AVERAGE OF ALL SALARIES IS PRINTED.
C
C
C     INPUT DATA:
C     ----------
C
```

```
C     ALL INPUT IS FREE FORMAT
C
C     (A)   FIRST CARD
C           CONTAINS THE TOTAL NUMBER OF ELECTORATES (INTEGER)
C
C     (B)   FOLLOWING CARDS
C           CONTAIN ELECTORAL SALARY DATA. THE FIRST CARD
C           INDICATES THE NUMBER OF PEOPLE IN AN ELECTORATE,
C           AND THEN ONE SALARY PER CARD, WHICH REPRESENTS
C           THE EARNINGS OF A RESIDENT IN THE ELECTORATE.
C
C
C       VARIABLES USED:
C       --------------
C         GRAND      the total salary earned by all
C                    the people
C         NELECT     the number of electorates
C         NPEEP      the number of people in an electorate
C         NPOP       the total number of people in all
C                    electorates
C         SALARY     the salary earned by a person
C         TOTAL      the total salary earned by an electorate
C
C
C
      READ (5,*) NELECT
      GRAND = 0.0
      NPOP  = 0
      DO 10 I = 1,NELECT
C
C   READ THE NUMBER OF PEOPLE IN EACH ELECTORATE
C
      READ (5,*) NPEEP
C    READ AND TOTAL THE SALARIES IN AN ELECTORATE
C
      TOTAL = 0.0
      DO 5 J = 1,NPEEP
      READ (5,*) SALARY
      TOTAL = TOTAL + SALARY
 5    CONTINUE
C
C   FORM THE AVERAGE FOR THIS ELECTORATE
C
      AVER = TOTAL/NPEEP
      WRITE (6,*)' THE AVERAGE FOR ELECTORATE ',I,' IS ',AVER
C
C    ADD TO GRAND TOTAL
C
      GRAND = GRAND + TOTAL
      NPOP  = NPOP  + NPEEP
10    CONTINUE
C
C    CALCULATE AND WRITE OUT GRAND AVERAGE
C
      GAVE = GRAND/NPOP
      WRITE(6,*)' AVERAGE OF ALL',NELECT,' ELECTORATES IS ',GAVE
      STOP
      END
```

The output from this program, using the data above would be:

```
 THE AVERAGE FOR ELECTORATE              1 IS   274.17500
 THE AVERAGE FOR ELECTORATE              2 IS   188.49750
 THE AVERAGE FOR ELECTORATE              3 IS   121.52333
 AVERAGE OF ALL            3 ELECTORATES IS    185.21222
```

5.4.2. Example 2

To write out a series of headings for a monthly diary. There is to be a one line heading for each month of each year between 1975 and 1982.

```
C
C      AUTHOR:  H. MOOD
C        DATE:  OCTOBER 1979
C       INPUT:  THERE IS NO INPUT
C     PURPOSE:
C        TO WRITE OUT A HEADING SAYING THE YEAR AND MONTH FOR
C        EACH MONTH BETWEEN 1975 AND 1982
C
C
      DO 15 IYR = 1975,1982
         WRITE(6,*)' ------------------------'
         DO 10 MONTH = 1,12
            WRITE(6,*) IYR, MONTH
            WRITE(6,*)
            WRITE(6,*)' +++++++++++++++++++++++'
 10      CONTINUE
 15   CONTINUE
      STOP
      END
```

5.5. Exercises

5.5.1. Exercise 5A

Identify the following statements as being TRUE or FALSE.

(i) Statement labels must be assigned sequentially.
(ii) The largest statement number is 99999.
(iii) Every FORTRAN statement must be assigned a statement label.
(iv) Statement numbers may be variable quantities.
(v) Statement numbers do not have to start in column 1.
(vi) It is valid to assign the same statement label to several statements

in a program.
(vii) A CONTINUE statement must be the last statement of a DO loop.
(viii) A CONTINUE statement may be used outside a DO loop.
(ix) A DO loop must finish before another one may start.
(x) A CONTINUE statement must be labelled.

5.5.2. Exercise 5B

Identify the following statements as correct or incorrect DO statements.

```
(i)    DO 8 KAN = J,K,L           (vii)  DO 88 KIRSH = 1,K+4
(ii)   DO N K6 = 5,N,2            (viii) DO 2 L = S,2
(iii)  DO 5 IY = 1,12             (ix)   DO 7453 FRED = 1,N4A2,2
(iv)   DO 692 3 = 1,K             (x)    DO 222 K = J,6
(v)    DO 99 N1234 = 1,3,K        (xi)   DO I = 1,9,2
(vi)   DO 9 K = 1,9.5             (xii)  DO 60 J = I,J,K
```

5.5.3. Exercise 5C

What is written out by the following groups of statements:

(a)
```
      VP = 98.6
      J  = 16
      IF(VP.LE.62.3)WRITE(6,*)J
      J = 14
      WRITE(6,*)J
```

(b)
```
      DO 16 KX = 1,30,4
      IF(KX.GT.16)WRITE(6,*)KX
      LX = KX-1
   16 IF(KX.LE.13)WRITE(6,*)LX
```

5.5.4. Exercise 5D

Using a DO statement
add up all the even integers between 98 and 224 inclusive
and write out the total.

5.5.5. Exercise 5E

ead a line of data containing an integer indicating the number of lines to follow. Read in the rest of the data, one number per line, and determine the largest and smallest number. For example, if the data were

```
4
12.2
```

```
        16.4
        -7.1
        4.4
```

the output would be

```
    THE LARGEST  NUMBER WAS      16.4
    THE SMALLEST NUMBER WAS      -7.1
```

5.5.6. Exercise 5F

Assuming that there is one integer punched per data card, how many cards will be read in the following?

```
(i)               DO 16 I = 1,3
                  DO 16 J = 1,4
            16    READ(5,*) L

(ii)              DO 16 I = 1,3
                  DO 14 J = 1,4
                     READ(5,*) L
            14    CONTINUE
            16    CONTINUE

(iii)             DO 16 K = 1,3
                     DO 14 J = K,4
            14       READ(5,*) L
            16    CONTINUE

(iv)              DO 16 K = 1,3
                     READ(5,*) L
                     DO 14 J = 1,4
            14       READ(5,*) L
            16    CONTINUE

(v)               DO 16 KK = 3,6,1
                  DO 16 LM = 8,11,4
            16    READ(5,*) L
```

5.5.7. Exercise 5G

What number would be written out in the following?

```
          DO 17 J = 17,38,9
          . . . . .
    17    CONTINUE
          WRITE (6,*) J
```

5.5.8. Exercise 5H

Write a program to find the average of any number of numbers.

6. CHAPTER 6

6.1. Supplied FUNCTIONS

There are some special routines available to the FORTRAN programmer that result in certain actions taking place. These routines are invoked by mentioning a key name, called the function name, followed by a list of parameters for that function.

For example, the function ABS will find the absolute value of a real variable. The only parameter is the name of the real variable whose absolute value is required. To find the absolute value of the variable BILL and to store that value in the variable FRED, the statement is

```
        FRED = ABS(BILL)
```

This could also be achieved by using a logical IF statement, viz,

```
        FRED = BILL
        IF(BILL.LT.0.0) FRED = -BILL
```

The use of the ABS function is a little clearer and certainly more concise. You should try to use FUNCTIONs extensively for this reason.

A function is said to RETURN a value. In the above example, the function ABS returns a value which is the absolute value of its parameter.

Another function is MAX0 which requires two integer parameters. The function returns an integer value which is equal to the larger of the two parameters.

The statement

```
        L = 12
        LARGE = MAX0(L,6)
```

results in LARGE having the value 12.

1977 standard Fortran introduced the concept of generic functions. These are functions that may be used with different data types as parameters to the function. Prior to this, you had to have specific data types for any specific function, and the function to find the larger of two integers (MAX0) had a different name to the function that found the larger of two reals (AMAX1). Now, you can just use the generic name MAX in either case, however, you still cannot mix data types.

Later on we will see how you can write your own functions, but in that case you can only write specific functions, that must ALWAYS have the mix of data types you expect as their parameters. More of that later.

The list below has some of the more common generic functions. The abbreviations mean

I	type integer
R	type real
P	the type of the parameters used

Function Name	Result Type	Number of Parameters	Description
INT	I	1	Truncated value of its parameter.
REAL	R	1	Value of its parameter converted to type REAL representation.
ABS	P	1	Absolute value of its parameter.
MOD	P	2	Performs modulus arithmetic. p1...first parameter p2...second parameter computes: p1-int(p1/p2)*p2.
MAX	P	>1	Largest of its parameters.
MIN	P	>1	Smallest of its parameters.

For a more complete list of functions see your Fortran manual.

6.2. More FUNCTIONS

There are some more complicated functions such as the trigonometric functions sine and cosine that are used in the same way as the functions described above.

Some of the more common ones are

SIN(real)	returns the sine of real number expressed in radians
COS(real)	returns the cosine of a real number expressed in radians
EXP(real)	returns the exponential of a real number
SQRT(real)	returns the square root of a positive real number
LOG(real)	returns the natural logarithm of a real number

Using a function to return a value is called a FUNCTION REFERENCE.

Note that some functions return real values (eg, the function REAL) while others return integer values (eg, INT). The NUMBER and TYPE of the parameters in a function reference are important and MUST be exactly what the function expects. In the case of generic functions, the parameters may be REAL or INTEGER (but not a mixture).

The parameters in a function reference may be arithmetic expressions of the same type that the function expects. For example, to find the square root of

$$(X1 - X2)^2 + (Y1 - Y2)^2$$

the statement may be

```
   ANS = SQRT( (X1 -X2) ** 2 + (Y1 - Y2) ** 2)
```

Function references may be used wherever arithmetic expressions are legal.

For example, to find the sum of the sine and cosine of the variable A, the statement would be

```
   ANS = SIN(A) + COS(A)
```

Another example may be to find the square root of the larger of two real numbers. This may be done by:

```
   ANS = SQRT (MAX(VAL1,VAL2))
```

So a parameter in a function reference may be another function reference, as long as it is a different function.

6.3. LOGICAL variables

So far the only variable types we have introduced are INTEGER and REAL. Integer variables can store integer numbers and real variables can store real numbers. Now, we introduce LOGICAL variables, which can store LOGICAL values. A LOGICAL value is one of:

.TRUE.	meaning true
.FALSE.	meaning false

Logical variables may be assigned values that are either .TRUE. or .FALSE. They may also appear in IF statements as part, or all of the logical expression in parentheses.

For example, if NOGOOD was a LOGICAL variable we could use it as:

```
         IF( NOGOOD ) WRITE(6,*)' ERROR FOUND IN DATA'
```

The way to read the above statement is to say: 'If NOGOOD is true then write out the message'. Note that the value of .TRUE. does NOT appear in the expression.

6.3.1. Declaring LOGICAL variables

Before using LOGICAL variables they must be declared by appearing on a LOGICAL declaration statement. These DECLARATION or TYPE statements should appear at the top of a program before any executable statements. They may be in any order if there is more than one.
The form of a logical type statement is:

```
         LOGICAL  <list of variables>
```

For example:

```
         LOGICAL POOR, MID, HIGH, MALE, FEMALE
```

The variables appearing on the list are called logical variables and must be treated as such throughout the program. Note that the first letter of the variable name has no special significance as the variable appears on a type statement.

6.3.2. Assignment

Logical variables may be assigned values that are true or false. For example

```
      LOGICAL SWAP, ENDATA
      SWAP = .FALSE.
      IF(NUM.LT.0) ENDATA = .TRUE.
```

In this last case, if NUM was not less than zero then ENDATA would not have a value. A much better way, that would give ENDATA a value of .FALSE. if NUM was not less than zero and .TRUE. otherwise is:

```
      ENDATA = NUM.LT.0
```

6.3.3. Using logical variables

Suppose a program reads in a salary and that salary is classified as:

0 - 5000	poor
5001 - 12000	medium
12001 - 20,000	high
20,001 - 100,000	fat cat
100,001 onwards	suspect error

In a program we can classify this by:

```
      LOGICAL POOR, MID, HIGH, FATCAT, SUSP
      READ(5,*) IWAGES
      POOR = IWAGES .LE. 5000
      MID    = IWAGES .GT. 5000  .AND. IWAGES .LE. 12000
      HIGH   = IWAGES .GT. 12000 .AND. IWAGES .LE. 20000
      FATCAT= IWAGES .GT. 20000 .AND. IWAGES .LE. 100000
      SUSP   = IWAGES .GT. 100000
```

and then later in the program we can test these values and take different action depending on the wage classification as:

```
      IF(POOR)          TAXHIT = TAXHIT - 5.0
      IF(MID .OR. HIGH) TAXHIT = TAXHIT + 0.75
      IF(FATCAT)        TAXHIT = TAXHIT + 7.5
```

This way we localise the classification ranges and then can refer to them using meaningful names. If in the future we want to change this program by altering the range classifications then we only have to make the change in one place, no matter how many times we refer to the ranges.

Another example is in the processing of SEX. Suppose that we read in a classification of 1 meaning FEMALE and 0 meaning MALE.

```
      LOGICAL FEMALE, MALE
         .     .      .
         .     .      .
      READ(5,*) ISEX
      FEMALE = ISEX .EQ. 1
      MALE   = ISEX .EQ. 0
         .     .      .
        later in the program

      IF(FEMALE) JOBFEM = JOBFEM + 1
      IF(MALE)   JOBMAL = JOBMAL + 1
```

6.3.4. The .NOT. operator

The .NOT. operator is used to turn logical values (i.e. .TRUE. and .FALSE.) into their opposite.
For example:

```
      LOGICAL POS, NEG
      POS = NUM .GE. 0
      NEG = .NOT. POS
```

Now if POS is .TRUE. then NEG will be .FALSE., but if POS was .FALSE. then NEG will be .TRUE.

The .NOT. operator can also be used within an IF statement. For example:

```
      IF(.NOT. ENDATA) READ(5,*) MORE
```

The READ will be done only if ENDATA is .FALSE.

Consider the setting of MALE and FEMALE above. Suppose that we wanted to set the logical variable ERRSEX to .TRUE. for any code other than 0 or 1, and to set it to .FALSE. otherwise.

```
      ERRSEX = .NOT. (MALE .OR. FEMALE)
```

this is equivalent to

```
      ERRSEX = .NOT. MALE .AND. .NOT. FEMALE
```

6.4. TYPE statements for integers and reals

So far, the IJKLMN naming convention has been used for distinguishing between integer and real variables. This convention may be overridden so that, for example, ABC may be the name of an integer variable and IM the name of a real variable.

A type statement consists of one of the declarations INTEGER or REAL followed by a list of variable names separated by commas, specifying those variables as being of type INTEGER or REAL. Some examples are

```
        INTEGER B
        REAL J
        INTEGER I,ABC,ROOT8
        REAL MATRIX,NUMBER,X
```

Type statements must precede all executable statements.

The variable I could have been omitted from the INTEGER statment and X from the REAL statement without effect. These names are already identified as integer and real respectively by their first letters. On the other hand, there is no harm in such 'unnecessary' inclusions in type statements. This may help to guard against failure to give the correct type to a variable whose name does not agree with the IJKLMN naming convention.

There are various arguments for and against violating the default naming convention.

If you stick to the I to N default typing, then it is much easier to follow the program, from the point of view of mixed mode arithmetic problems, relating I/O lists to data, and typing of parameters to functions.

If you declare variables as being a particular type, then you can use more meaningful variable names to describe the contents of variables.

6.5. Exercises

6.5.1. Exercise 6A

Use function references to return answers to be stored in either IANS or ANS depending on whether the function returns an integer or real result.

(i) find the square root of B*B - 4.0*A*C
(ii) find the sine of 2.4
(iii) find the larger of J and LARG
(iv) find the largest of A and BIG
(v) find the absolute value of ECC
(vi) find the absolute values of I, M, and A
(vii) convert KKK to real
(viii) find the square root of INT
(ix) find the largest of I and X

6.5.2. Exercise 6B

Write a program to calculate the factorial of a number that is read in. Factorial n is defined to be n! = n*(n-1)*(n-2)* ... *3*2*1.
For example, 5! = 5*4*3*2*1.

6.5.3. Exercise 6C

Replace the line(s) of FORTRAN following by a single line that will produce the same answer in the indicated variable:

```
 (i)     M
         M = IJ/K
         M = IJ - M*K

 (ii)    SMALL
         IF(A.LE.B) SMALL = A
         IF(A.GT.B) SMALL = B

 (iii)   WARM
         QWERK = ION
         YIPE  = KAN
         WARM  = QWERK/YIPE

 (iv)    ADAM
         NUM = VAL/WORLD
         A = ABS(VAL)
         B = ABS(WORLD)
         IF(NUM.LT.0)NUM = -NUM
         DO 10 I = 1,NUM
    10   A = A - B
         IF(VAL.LT.0.0)ADAM = -A
         IF(VAL.GE.0.0)ADAM = A
```

7. CHAPTER 7

7.1. GO TO statement

Fortran statements are executed in order of occurrence, starting with the first executable statement. Each statement is then executed in turn, and some function is performed, depending on the statement concerned.

The GO TO statement is of the form

```
GO TO n
```

where n is the statement label of an executable statement. Execution of the GO TO statement causes the statement identified by the statement label n to be executed next. The program would then continue on from that point. Thus the GO TO statement may be used to transfer control to a statement other than the next one in sequence.

7.1.1. Examples of GO TO statements

Example 1:

Suppose we want to find the average salary of people who are defined as low salary earners. A low salary is defined as one below $5,000. The data that we have to process is entered as one salary per line, and we are to process data until an END OF FILE (@EOF) is encountered.

7.1.1.1. Algorithm

One method for solving the problem may be outlined as:

```
 1. Initialise a salary total and worker number total to zero
 2. Repeat for as many salaries as there are:
    2a.  Read in a salary
    2b.  If the salary is greater than 5000 ignore it
         If the salary is less than 5000 then:
         (i) add it to the salary total
        (ii) add 1 to the number of workers total
 3. When all salaries are processed calculate the average
```

A program to do the above may be:

```
C  AUTHOR:   L. YAMAHA
C  DATE:     SEPT 1979
C
```

```
C  LANGUAGE: UNIVAC ASCII FORTRAN LEVEL 9R1
C
C  INPUT DESCRIPTION:
C
C      EACH LINE CONTAINS AN INTEGER IN FREE FORMAT INDICATING THE
C      YEARLY SALARY OF A WORKER (IN DOLLARS). THE INPUT IS
C      TERMINATED BY AN @EOF.
C
C  PURPOSE:
C
C     TO CALCULATE THE AVERAGE SALARY OF PEOPLE WHO EARN LESS
C     THAN $5000.
C
      NTOT = 0
      NPEEP = 0
C
C   READ IN AND PROCESS THE WORKERS
C
 10   CONTINUE
      READ(5,*,END=20) ISAL
      IF (ISAL .LT. 5000) THEN
          NTOT  = NTOT  + ISAL
          NPEEP = NPEEP + 1
      END IF
      GO TO 10
C
C    END OF DATA, NOW WORK OUT THE AVERAGE
C    CHECK FOR ZERO IN CASE THERE ARE NO LOW WAGE EARNERS
C
 20   CONTINUE
      IF (NPEEP .EQ. 0) THEN
          WRITE (6,*) 'NO LOW INCOME EARNERS FOUND'
      ELSE
          AVE = NTOT/REAL(NPEEP)
          WRITE(6,*) 'THE AVERAGE SALARY OF THE ',NPEEP,
     $               ' LOW INCOME EARNERS FOUND IS ',AVE
      END IF
      STOP
      END
```

Notice that there is only one GO TO in the above program. If it were not for the use of the block-IF statement there would be at least two more GO TOs.

Example 2:

Read an integer from a card and write it out. Continue this process until a card with -99 is read. When this happens, write out a message and then stop.

```
C
C   AUTHOR:   DEE
C   DATE:     SEPT 1973
C   PURPOSE:  TO READ INTEGERS FROM CARDS AND TO
C             LIST THEM OUT
C
C   INPUT DESCRIPTION:
```

```
C
C           FREE FORMAT INPUT
C           ONE INTEGER PER CARD, LAST INTEGER MUST
C           BE -99
C
C   RESTRICTIONS:
C           ONLY THE LAST DATA CARD MAY CONTAIN -99.
C           IF THIS IS NOT POSSIBLE, THEN CHANGE
C           THE VALUE OF ISTOP
C
      ISTOP = -99
 10   CONTINUE
      READ(5,*) INT
C
C     TEST FOR THE END OF DATA
C
      IF (INT .EQ. ISTOP) THEN
         WRITE (6,*) ' END OF JOB '
         STOP
      ELSE
         WRITE (6,*) INT
         GO TO 10
      END IF
      END
```

7.1.2. Using GO TO statements with DO loops

The terminal statement of a DO loop cannot be a GO TO statement or a logical IF statement that contains a GO TO statement.

It is not legal to jump into the middle of a DO loop from outside the DO loop. For example, the following is NOT allowed.

```
      DO 40 I =1,IFREQ
      . . . . .
      . . . . .
 20   CONTINUE
      . . . . .
      . . . . .
 40   CONTINUE
      . . . . .
      . . . . .
      GO TO 20
```

It is possible to leave a DO loop by using a GO TO statement. If this is done, the control variable of the DO is defined and is equal to the most recent value attained. For example,

```
      MAX = 4
      LOW = 2
      DO 20 I = 1,4
         LOW = LOW+1
         IF(MAX.EQ.LOW) GO TO 30
 20   CONTINUE
 30   WRITE(6,*) ' THE VALUE OF I IS ',I
```

will print the line

```
 THE VALUE OF I IS   2
```

7.1.3. Using GO TO statements with block-IF

It is illegal to jump into the middle of a block-IF (any of the three types of blocks) from outside of a block. A GO TO that transfers control within a block is allowed as is a transfer of control from within a block to outside (provided that you do not GO TO the middle of another block or a DO).

7.1.4. Reachability of statements

Every executable statement must be 'reachable' along some logic path of the program. If some statement is not reachable, then an error is said to have occurred. Some computers will warn you of this but proceed and try to execute anyhow, or they may not allow an attempt at execution until the problem has been rectified. It is always best to eliminate all diagnostics from your program, even if they are only warnings. For example, in the program

```
      ISUM = 0
      DO 20 I = 1,12
 20   ISUM = ISUM + I
      GO TO 60
      WRITE(6,*) ' JUST A HEADING '
 60   STOP
      END
```

the WRITE statement can never be reached.

7.1.5. Some common errors associated with GO TO statements

(i) Two statements having the same label.

(ii) The label referred to by the GO TO is missing.

(iii) There is an unlabelled statement after a GO TO statement, and hence unreachable.

(iv) Generation of an infinite loop by a GO TO an earlier statement without any proper 'escape line' in between, to get out of the loop. For example,

```
      I = 5
 10   I = I+1
      IF(I.EQ.0) GO TO 20
      GO TO 10
 20   CONTINUE
```

is an infinite loop.

7.1.6. When and how to use the GO TO

A program is easiest to follow and thus easier to get running successfully if it has 'forward control' which refers to branches always going down the program. This is sometimes not possible, nor desirable, as shown in the first example in this chapter.

Given that there are valid cases for using the GO TO there are still different program designs available, some of which are considered better than others.

For example, consider the two blocks of code below (which are equivalent):

```
      IF(PAY.GE.6500.0)GO TO 10
      GO TO 20
 10   PAY = PAY + 250.5
      NRICH = NRICH + 1
 20   CONTINUE

      IF(PAY.LT.6500.0)GO TO 20
      PAY = PAY + 250.5
      NRICH = NRICH + 1
 20   CONTINUE
```

By reversing the sense of the test in the logical IF the second block of code contains one less GO TO and one less statement label.

Of course a much better way of doing this is to avoid using GO TOs entirely by:

```
      IF (PAY .GE. 6500.0) THEN
          PAY = PAY + 250.5
          NRICH = NRICH + 1
      END IF
```

7.1.7. COMMENTS and GOTO'S

The GO TO statement may transfer control a long way away from where the test that caused the branch of control occurred. If this does occur, then it is desirable for a comment to appear above the statement gone to detailing why it is that you have come here. For example, suppose a program reads in cards each containing an integer. If a value of 1 is read, then this will indicate that the cards following are personnel records. If a value of 2 is read this will indicate that the cards following are required for inventory control.

```
    C  READ AN INDICATOR CARD
    C
          READ(5,*)ITYPE
    C
    C  TEST THE TYPE
    C
          IF(ITYPE.EQ.1)GO TO 25
          IF(ITYPE.EQ.2)GO TO 30
            .     .     .     .
            .     .     .     .
           test for further types
            .     .     .     .
            .     .     .     .
    C
    C  TYPE 1 RECORD FOUND, PROCESS PERSONNEL RECORDS
    C
    25    CONTINUE
          .     .     .     .
          .     .     .     .
    C
    C  TYPE 2 RECORD FOUND, PROCESS INVENTORY RECORDS
    C
    30    CONTINUE
          .     .     .     .     .
```

7.2. Exercises

Some of these exercises do NOT require the use of GO TO statements. Where GO TO statements are not required, don't use them. It can become a bad habit to use GO TO statements excessively.

7.2.1. Exercise 7A

Write a program to evaluate

$$y = (a^2 + \sin^2 p)^{1/2} \quad \text{if k is negative,}$$

$$y = (b^2 - \sin^2 p)^{1/2} \quad \text{if k is positive,}$$

$$y = (a^2 + b^2)^{1/2} \quad \text{if k is zero.}$$

a, b, p and k are to be read from a line of data.

7.2.2. Exercise 7B

Write a program to read x and evaluate

$$f = 1 + \cos x + \frac{\cos^{2} x}{2!} + \frac{\cos^{3} x}{3!} + \ldots + \frac{\cos^{20} x}{20!}$$

where n! is called factorial n and is defined as the product n*(n-1)*(n-2)* ... *1. For example, 5! = 5*4*3*2*1.

7.2.3. Exercise 7C

Write a program to evaluate the exponential of x for every integer from 1 to 100, printing x and its exponential side by side.

Run the program and see what happens !

8. CHAPTER 8

8.1. FIXED FORMAT with WRITE statement

So far, free field format has been used with both READ and WRITE statements. When using WRITE statements with this type of format on the Univac, up to ten numbers may be printed per line in a form dictated by Fortran. It is possible to specify (in a FORMAT statement) the layout of each line to be printed.
In order to do this the asterisk in the WRITE(6,*) is replaced by a statement number, that is the number of a FORMAT statement.
The form of a FORMAT statement is:

```
  n      FORMAT  (field descriptors)
```

where n is a statement label.
When using this type of output, the headings may no longer appear on the output list, but must be placed within the 'field descriptors' part of the FORMAT.

The FORMAT statement is referenced when the corresponding WRITE statement is executed. It provides information (via field descriptors) as to how the information to output, should be written out.

The FORMAT statement itself is called NON-EXECUTABLE. It may appear anywhere within the program, but it is not good practice to terminate a DO-loop with a FORMAT. It may appear either before, after or nowhere near its corresponding WRITE statement(s). The correspondence between the two statements is achieved via the statement number. For example

```
      WRITE(6,103)K,A
  103 FORMAT(field descriptors)
```

The FORMAT statement will contain FIELD DESCRIPTORS which will describe the fields for integers, real numbers, headings, etc. Field descriptors are inserted between the brackets of the FORMAT statement and are separated from other field descriptors by commas.

A FORMAT statement may be referenced by several WRITE statements.

8.1.1. Spacing across a line, leaving blanks

This may be done with the X field descriptor which is of the form

```
      nX
```

where n is the number of spaces to be placed in the printed line.

The first character on an output line is not printed, but treated specially. For the moment, let us put 1X as the first field descriptor so that the first character is a blank (which is not printed).

8.1.2. Writing headings using FIXED FORMAT

The headings should appear within the FORMAT statement enclosed in quotes. For example

```
      WRITE(6,104)
  104 FORMAT(1X, 'SHOP INVENTORY',25X,'GROCERY SECTION')
```

8.1.3. INTEGER FIELD DESCRIPTOR

The integer field descriptor is of the form

Iw

where	I	is the letter I and indicates that an integer value will be printed in this field. This MUST be the letter I.
	w	is a number (that you must supply) and refers to the width of the field. The width of the field is the maximum number of digits that may be printed using this field descriptor.

For example, the statements

```
      INT = -27
      WRITE (6,100) INT
  100 FORMAT(1X,I5)
```

will print the line

```
bb-27
```

where b indicates the blank character.

The 1X in the FORMAT statement is not printed as a space. The 5 characters (viz, bb-27) come from the I5 in the FORMAT statement. The number is printed right justified in the field with leading blanks to make up the required field width.

If more than one value is to be written out, then there will be more than one variable in the output list. In this case there must also be field descriptors that correspond to each variable in the output list.

For example, if MEN has value 12345
and LIB has value 0

```
      WRITE(6,100) MEN, LIB
  100 FORMAT(1X,I8,I4)
```

produces the output

```
bbb12345bbb0
```

The I8 is used to write out a value for MEN, and the I4 is used to write out a value for LIB.

8.1.4. REAL FIELD DESCRIPTOR

The real field descriptor is of the form

```
        Fw.d
```

where	F	is the letter F and indicates that a real value will be printed in this field
	w	is a number and refers to the width of the field
	d	is a number indicating the number of decimal places to be printed.

For example, the statements

```
        VICE = -33.47
        WRITE (6,110) VICE
    110 FORMAT(1X,F10.3)
```

will print the line

```
bbb-33.470
```

Again, the 1X is not printed. The ten characters (viz, bbb-33.470) come from the 10 in F10.3. The three characters 470 come from the 3 in F10.3. Note that the minus sign (if there is one) and the decimal point in the printed line are included in the field width.

8.1.5. Mixing REAL and INTEGER FIELD DESCRIPTORS

Of course, it is possible to print out more than one number per line, and to have both real and integer numbers on the same line. When a mixture of reals and integers are written out in the one WRITE statement there must be an exact correspondence between the types of variables on the output list (integers or reals) and the type of field descriptor used (I or F). An F field descriptor in the FORMAT statement must correspond to a real variable in the output list in the WRITE statement, and an I field descriptor in the FORMAT statement must correspond to an integer variable in the output list in the WRITE statement. Field descriptors are separated by commas in the FORMAT statement.

For example, the statements

```
      AVE   = 10.3
      SMALL = -7.53
      NUM   = 23
      MINT  = -10
      WRITE (6,120) AVE, SMALL, NUM, MINT
  120 FORMAT(1X,F7.2,F7.1,I5,I6)
```

will print the line

```
bb10.30bbb-7.5bbb23bbb-10
```

also

```
      WRITE (6,140) AVE, SMALL, NUM, MINT
  140 FORMAT(1X,F7.2,3X,F7.1,1X,I5,20X,I6)
```

will print the line

```
bb10.30bbbbbb-7.5bbbb23bbbbbbbbbbbbbbbbbbbbbbbb-10
```

8.1.6. Repetition of a FIELD DESCRIPTOR

Where two or more consecutive field descriptors are identical in every respect, then a shorthand notation may be used. This is accomplished by writing the field descriptor only once and prefixing it with a number indicating the desired number of repetitions. For example, the following FORMAT statements are equivalent:

```
    (a)   160  FORMAT(1X,I5,I5)
       and
          160  FORMAT(1X,2I5)

    (b)   130  FORMAT(1X,F10.3,F10.3,F10.3,I4,I3,I3,F10.3)
       and
          130  FORMAT(1X,3F10.3,I4,2I3,F10.3)
```

8.1.7. Combining headings with numbers output

The heading is placed between quotes symbols (as before) in the FORMAT statement, and will be printed 'in place'. This means that it will be printed on the line immediately prior to the number whose corresponding field descriptor follows the heading.

For example, the statements

```
      WRITE (6,150)
  150 FORMAT(1X,'THIS IS A HEADING')
      WRITE (6,160) AVE, SMALL, NUM, MINT
  160 FORMAT(1X,'VALUE ',F7.2,' HI THERE ',
     *      F7.1,I5,I6,' MESSAGE')
```

will print the lines

```
 THIS IS A HEADING
 VALUE   10.30 HI THERE    -7.5   23   -10 MESSAGE
```

Note that 5X in a FORMAT statement is equivalent to 'bbbbb'.

8.1.8. Print Control of the printer

So far, the first field descriptor in a FORMAT statement has been 1X. This first character of EVERY output line has the special function of controlling paper spacing and is NEVER actually printed. In 1966 standard Fortran, this character was actually part of the output line, and so you could only ever print 131 columns. In 1977 standard Fortran, you can print 132 columns as the print control character is not part of the output image. The character that is in the first column is known as a print control character and has the following effect.

character	vertical spacing before printing
blank	one line, ie single spacing
0 (zero)	two lines, ie double spacing
1 (one)	to first line of next page (printer only)
+	no advance, ie print on the same line.

These characters are usually provided using a literal such as 'b' (or 1X), '0', '1' and '+'. If a character other than one of those defined above is used, the result is unpredictable. The Univac is kind, and assumes a blank, but the erroneous character is still not printed. A '+' should only be used to underline headings, and not to set up a line of numbers with different WRITE statements.

When printing output at a terminal, you must still allow for the print control character. For example:-

Print a line on the top of a new page, and write out the values for variables MIN and MAX. Then write a blank line followed by values for the variables DOLLAS and KOST on one line. Suppose the values had been set up as:

```
MIN    = -126
```

```
        MAX    = 2
        DOLLAS = 77.2576
        KOST   = 10301
```

then the lines following, are those required.

```
        WRITE(6,170) MIN, MAX
   170  FORMAT('1',I6,3X,I5)
        WRITE(6,180) DOLLAS, KOST
   180  FORMAT('0',F10.2,I7)
```

The output (on a new page) would be:

```
bb-126bbbbbbb2

bbbbb77.26bb10301
```

8.1.9. MULTI-LINE FORMAT

A slash (ie, /) in a FORMAT statement indicates that the current output line is complete and the next one is to begin. It does not need to be separated from field descriptors with commas.

For example,

```
        WRITE (6,190) AVE, SMALL, NUM, MINT
    190 FORMAT(1X,2F10.3/1X,2I4)
```

will print the lines

```
     10.300    -7.530
   23 -10
```

Note that the 1X after the slash is required as it will be at the start of a new line and is therefore taken as carriage control.

If n consecutive slashes are written in sequence at the start of a FORMAT statement, n lines are left blank before printing the first line.
If n consecutive slashes are written in sequence at the end of a FORMAT statement, n lines are left blank after printing the last line.
If n consecutive slashes are written in sequence between other field descriptors in a FORMAT statement, n-1 lines are left blank between the two sets of printed lines.

8.2. Example of FORMATTED WRITE statement

Suppose the desired output is

```
(new page)
GROSS INCOME = xxxxx.xx
TAX PAID     = xxxxx.xx
SUPER  = xxx.xx      DEDUCTIONS = xxx.xx
MONTHS OF SERVICE = xxx
```

where x represents a digit.

This will require 5 variables, the first four will be real and the last an integer. Assume the variables have these values.

```
         GROSS = 6847.2
         TAX   = 948.353
         SUPER = 66.1
         DED   = 2.65
         MONTH = 95
```

Then the WRITE and FORMAT statements are

```
         WRITE (6,200) GROSS,TAX,SUPER,DED,MONTH
    200  FORMAT('1','GROSS INCOME = ',F8.2/
        $      1X,'TAX PAID',5X,'= ',F8.2/
        $      1X,'SUPER  = ',F6.2,5X,'DEDUCTIONS = ',F6.2/
        $      1X,'MONTHS OF SERVICE = ',I3)
```

8.3. Design of FORMAT statements

Format statements are often the source of errors as they can become quite complex. There are two things that may be done to simplify the task of format design.

1. Before starting to write out the format statement you should first draw out the form of output that you require, marking in all the headings and blanks that are required, as in the example above.

2. Use sensible continuation points. There is no need to go right up to column 72 before going to a continuation card. Pick an easy break between field descriptors at least.

3. Never continue a FORMAT statement (onto a continuation line) in the middle of a heading - rather terminate the heading in a convenient place (e.g. on a word boundary) and then start again on the continuation line. So instead of:

```
          WRITE(6,100)
      100 FORMAT(1X,'THE NUMBER OF POLITICIANS IN THE UPPER
         +HOUSE IS LIMITED')
```

put:

```
          WRITE(6,100)
      100 FORMAT(1X,'THE NUMBER OF POLITICIANS IN THE UPPER ',
         +       'HOUSE IS LIMITED')
```

Note the space after the word UPPER and the comma between the two headings.

The output in both cases would all be on the same line.

In the first case, the first line of the FORMAT statement uses all 72 characters even though 'UPPER' ends in column 55, so 17 spaces will be printed between 'UPPER' and 'HOUSE'. Remember that any characters after column 72 are ignored.

8.4. Common errors and points to note

(a) Trying to print a number that is too large for the space allowed.
The UNIVAC computer will fill the entire field with asterisks. The statements

```
          OOPS = -6.235
          WRITE (6,210) OOPS
      210 FORMAT(1X,F5.3)
```

will print the line

```
*****
```

(b) Omitting the print control character.
Remember the FIRST character of EVERY output line is used for print control and is not printed.

```
          I = 1234
          WRITE (6,220) I
      220 FORMAT(I4)
```

will print the following line at the top of the next page.

```
234
```

(c) Using variables in a FORMAT statement.

```
          legal     illegal
          -----     -------

          F5.3      FJ.K
          3I10      NI10
```

(d) Trying to WRITE an integer using an F field descriptor and a real using an I field descriptor.
The order and type of variables in the list of the WRITE statement MUST correspond with the order and type of field descriptors in the FORMAT

statement.

(e) Trying to print more than 132 characters on a line.
If this is attempted, an error occurs.

8.5. Exercises

8.5.1. Exercise 8A

Find the errors in the following FORTRAN statements and suggest ways in which they might be corrected.

(i)

```
      WRITE (6,230) K
  230 FORMAT(' FINISHED THE JOB')
```

(ii)

```
      WRITE (6,240) ROLLED,STONES,GATHER,NO,MOSS
  240 FORMAT(1X,2F10.2,2I5,F3.5)
```

(iii)

```
 250  I = 17
      DO 250 K=3,49
      READ (5,*) I,F,K
      WRITE (6,270) I
  270 FORMAT(1X,F7)
      IF(I.=.26) STOP
  280 CONTINUE
```

8.5.2. Exercise 8B

How many lines will be printed?

```
      WRITE (6,290) ZAP,BAM,ZOWIE,POW
  290 FORMAT(1X,'A = ',/,' ',2F10.3,' B = ',F6.1/
     +       '0','C = ',F9.1)
```

8.5.3. Exercise 8C

What is the output from the following?

```
      JILL = 0
      ZERO = 0.0
      YOURS = 16.77
      TAX = -586.21
      MARVIN = 90062
      MIN = -587
      JACK = 10
      IN = 123456
      OUT = 5.4827
      WHYNOT = 131.1
      WRITE (6,300) JILL,JACK,MIN,MARVIN,IN
  300 FORMAT(1X,5I5)
      WRITE (6,310) ZERO,YOURS,TAX,OUT,WHYNOT
  310 FORMAT(1X,5F7.2)
```

8.5.4. Exercise 8D

Indicate whether the field descriptors on the left are sufficient to print the numbers on the right and show what would be printed.

field descriptor	number
F5.1	676.71
I4	6381
F10.1	132.63
F4.2	12.16
I10	99999

8.5.5. Exercise 8E

Consider two blocks a and b. Block a is a cube of side length h and block b has sides k, 2k, 3k. Read in the values of h and k. Print your name and, after two blank lines, write the heading

COMPARISON OF SURFACE AREAS

in the middle of the line. After three blank lines, write the following lines with their calculated values.

```
SIDE OF CUBE A:
WIDTH OF BLOCK B:
HEIGHT OF BLOCK B:
LENGTH OF BLOCK B:

SURFACE AREA OF A:
SURFACE AREA OF B:

DIFFERENCE IN SURFACE AREA:

LARGER SURFACE AREA:
```

8.5.6. Exercise 8F

Write a program to evaluate the sine, cosine, tangent, secant, cosecant, and cotangent of every integral angle from 1 to 89 degrees inclusive.

9. CHAPTER 9

9.1. READ statement with FIXED FIELD FORMAT

Fixed field format may also be used with READ statements. The data line is divided into fixed length fields as defined by field descriptors in the FORMAT statement.

The field descriptors will specify particular columns on a data line that will contain the number to be read. This now means that numbers will no longer be separated by commas in the data, but must be exactly within the columns specified. As blanks take up columns, they are significant, and in fact act as though they were zeros.

9.1.1. INTEGER FIELD DESCRIPTOR

The integer field descriptor is of the form

```
        Iw
```

where w is an integer constant indicating the width of the field, which includes any leading sign that may be present.

Consider the statements

```
        READ (5,100) NUMBER,NEXT
    100 FORMAT(I5,I4)
```

If the input were

```
bb345b-12
```

then NUMBER would be set to 345 and NEXT would be set to -12.

For integer field descriptors, blanks in the data are taken as zeros, so if the input was

```
b345bb-12
```

then NUMBER and NEXT would be set to 3450 and -12 respectively.

To read in 4 integers from input, that are laid out on one line as:-

number	columns	width
1	1 - 4	4
2	5 - 10	6
3	11 - 17	7
4	18 - 19	2

the Fortran statements would be:-

```
        READ(5,122)JANE,KINDA,LIKES,NUDES
  122   FORMAT(I4,I6,I7,I2)
```

The data may look like:-

```
  column 1
        9876bb-129bbbbbb320          (a 'b' indicates a blank)
```

The effect of this READ statement is the same as the four assignment statements:-

```
        JANE  = 9876
        KINDA = -129
        LIKES = 3
        NUDES = 20
```

WARNING

BLANKS IN INPUT DATA ARE INTERPRETED AS ZEROS. This means that the READ statement:-

```
        READ(5,134)KOOL
  134   FORMAT(I5)
```

would read the data card:

b2bbbbb

causing KOOL to become the value 2000

9.1.2. REAL FIELD DESCRIPTOR

The REAL field descriptor is of the form:-

```
        Fw.d
```

where

- w is an integer constant indicating the total width of the number (i.e. the total number of columns it takes up).
- d is an integer constant indicating how many columns are to be interpreted as following an implied decimal point. This 'd' is ignored if there is a decimal point in the number being read.

Consider the following statements:-

```
        READ(5,110)VAL,EWE
  110   FORMAT(F10.3,F7.2)
```

and the data card

```
bbb1234567bb-45892633
```

The first value (to be read into variable VAL) starts in column 1, and is 10 columns wide, the last 3 columns are to be interpreted as being AFTER the decimal point. So, VAL will contain the number 1234.567, as a result of the READ. The READing then continues from this place (column 11) and the next 7 columns are to be used to form the value for variable EWE, the last 2 columns to be interpreted as that part of the number after the decimal point. So, EWE will have the value of -45.89 . Notice that the remaining columns are ignored, as there are no further variables on the input list. Remember, blanks are taken as zeros, and a decimal point appearing in the data will override the 'd' specification, in the Fw.d

For example consider:-

```
      READ(5,123)VICE,SQUAD
123   FORMAT(F10.3,F7.2)
```

and the data card

```
bb47047.9bb-12bbb2345
```

This will set VICE to 47047.9 and SQUAD to -120.0

9.1.3. READING INTEGERS and REALS

You can mix up both integers and reals in the one READ statement, so long as your FORMAT statement reflects the type of variable you are reading into.
It is an error to read in a real number using I format, or to read in an integer using F format, and both of these will cause your program to error, or give incorrect results.

Another common error is to forget to ensure that there is a 1 to 1 correspondence between the type of format descriptor used (I or F), and the type of variable name in the input/output list (integer or real respectively). If this is not the case, then the computer will not tell you of any error, but will not store the number you want in the variable on the input list, or will not write out the correct value of a variable on the output list. This happens because of the way that reals and integers are stored in the computer memory. We will not go into this in any detail, but merely point out that there is a difference, and it matters, at least in input/output.

For example, we wish to read 5 numbers from a card, the first 2 real, the next 2 integer, and the last one real. Now we will have to reflect this in BOTH the READ input list, and in the FORMAT. So let us first choose variable names of:-

GREAT, BLUE, MOVIES, NEVER, FAIL

Notice, the first two are REAL, the next two INTEGER, and the last is REAL. Now before we can go any further, we must specify where on the card the numbers are to go.

columns	width
1 - 7	7
8 - 16	9
17 - 24	8
25 - 30	6
31 - 40	10

Assume that if there is no decimal point in a number that we want the number to be interpreted as having 1 digit after the decimal point (for REALs only!). Then, consider the statements:-

```
       READ(5,211)GREAT,BLUE,MOVIES,NEVER,FAIL
 211   FORMAT(F7.1,F9.1,I8,I6,F10.1)
```

and the data card of

```
bb23.44b-8.23899bbbb2bbbbb-234bbbb981233bbb
```

You should verify that the following is equivalent to the above:-

```
       GREAT = 23.44
       BLUE  = -8.23899
       MOVIES= 2000
       NEVER = -234
       FAIL  = 98123.3
```

9.1.4. Skipping columns on the input line

To skip across (and ignore) columns in the input may be done using a field descriptor of:-

```
       nX
```

Where 'n' is the number of columns to skip over.

For example, suppose we wanted to read a line that contained two integers. The first was in columns 10 to 15, and the second was in columns 55 to 60. So we want to skip over columns 1 to 9, read the first integer, then skip the next 39 columns, and read the second integer. The following would do this for us:-

```
       READ(5,102)MOOD,MUSIC
 102   FORMAT(9X,I6,39X,I6)
```

9.1.5. Skipping lines of input data

As with WRITE statements, the slash may be used in FORMAT statements with READ statements to indicate that the current line is no longer required and that further values should be read from the next line. For example, when the lines of data

```
bb123bbb-12
b-46104b56
```

are read by the statements

```
      READ (5,130) MORE,LIME,KOOPS
  130 FORMAT(I5/I4,I6)
```

the variables MORE, LIME, and KOOPS are set to 123, -46, and 104056 respectively.

9.2. More on FORMAT REPEAT specifications

It has already been shown that the statement

```
  140 FORMAT(1X,F10.3,F10.3,F10.3,I4,I3,I3,F10.3)
```

is equivalent to

```
  140 FORMAT(1X,3F10.3,I4,2I3,F10.3)
```

The facility exists to repeat a group of field descriptors by enclosing the group in parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed group. If no group repeat count is specified, a group repeat count of one is assumed. For example,

```
  150 FORMAT(1X,2(F10.2/1X,I4),2X,I5)
```

is equivalent to

```
  150 FORMAT(1X,F10.2/1X,I4,F10.2/1X,I4,2X,I5)
```

except in the case of the FORMAT being used for a READ or WRITE statement where there are more variables on the I/O list than there are field descriptors in the FORMAT. See below for an explanation.

9.3. Interaction between READ/WRITE and FORMAT statements

The number of values that will be read/written is determined solely by the number of variables in the I/O list in the READ/WRITE statement. The position of these values on the line is determined by the FORMAT statement.

When a READ statement is executed, the next input line is read and thereafter additional lines are read only as the format specification demands (e.g. on encountering / in a FORMAT statement). When a WRITE statement is executed, a new line is started and thereafter additional lines are started only as the format specification demands (e.g. a slash).

Except for the effects of the repeat count, the format specification is interpreted from left to right. To each I and F field descriptor in a FORMAT statement, there corresponds one variable in the I/O list in the READ/WRITE statement. To each literal (string in quotes) or X field descriptor, there is no corresponding variable in the I/O list.

Whenever the format control encounters an I or F field descriptor in a FORMAT statement, it determines if there is a corresponding variable specified in the I/O list. If there is, the value of that variable is either read or written using the conversion specified by the field descriptor. If there is no corresponding variable, the format control terminates. For example, the statements

```
         A = 12.3
         B = 24.6
         C = -56.8
         WRITE (6,160) A,B
   160   FORMAT(1X,'THIS IS A HEADING'/1X,'A = ',F10.2/1X,'B =',
        *       F10.2/1X,'C = ',F10.2)
```

will write the lines

```
THIS IS A HEADING
A =       12.30
B =       24.60
C =
```

If the format control proceeds to the last right parenthesis in a FORMAT statement, a test is made to determine if another variable is specified in the I/O list. If another variable doesn't exist, control terminates. However, if another variable is specified, a new line is started and control reverts to the group repeat specification terminated by the second last right parenthesis (i.e. to the left parenthesis corresponding to the second last right parenthesis) or, if none exists, to the first left parenthesis. In the case of cycling back to a repeat group, the repeat count is used also (i.e. the repeat count that is outside of the left parenthesis that is gone back to). For example, the statements

```
         AVE   = 1.0
         BEST  = 2.0
         COUNT = 3.0
         TOTAL = 4.0
         FIVE  = 5.0
         WRITE (6,180) AVE, BEST, COUNT, TOTAL, FIVE
   180   FORMAT (1X,'FIGHT = ',F6.2,3X,'RING = ',F6.2,3X,'BELL')
```

This writes out:

```
FIGHT =   1.00   RING =   2.00   BELL
FIGHT =   3.00   RING =   4.00   BELL
FIGHT =   5.00   RING =
```

Another example is

```
         WRITE (6,190) AVE, BEST, COUNT, TOTAL, FIVE
   190   FORMAT (1X,'FIGHT'/2(1X,'RING = ',F6.2),3X,'BELL ',F6.2)
```

This would produce

```
 FIGHT
 RING =    1.00 RING =    2.00  BELL   3.00
 RING =   4.00 RING =    5.00  BELL
```

The following output indicates what would be produced if the group repeat were expanded rather than being included as a repeat.

```
         WRITE (6,200) AVE, BEST, COUNT, TOTAL, FIVE
   200   FORMAT (1X,'FIGHT'/1X,'RING = ',F6.2,1X,'RING = ',
        $         F6.2,3X,'BELL ',F6.2)
```

This would produce the following (note the extra line of FIGHT):

```
 FIGHT
 RING =    1.00 RING =    2.00   BELL   3.00
 FIGHT
 RING =   4.00 RING =    5.00   BELL
```

9.4. Exercises

9.4.1. Exercise 9A

Find the errors in the following FORTRAN statements and suggest ways in which they might be corrected.

```
         READ (5,100) I,J,A
    100  FORMAT(3I5)

         READ (5,110) I
         IF(J.GRT.I) WRITE (6,120) J
    110  FORMAT(I5.1)
    120  FORMAT('1',F6.2)

         DO 130 JJK = 1,10
         J = -5
         K = 0
         READ (5,130) A
         L = J/K+A
         IF(L.LT.J) STOP
    130  FORMAT(F6.2)
```

9.4.2. Exercise 9B

How many lines will be read?

```
      READ (5,160) A,I,B,J,CRASH,KREME
  160 FORMAT(F3.1,I5)

      READ (5,170) A,I,B,J,CRASH,KREME
  170 FORMAT(F3.1,I5,F3.1,I5)
```

9.4.3. Exercise 9C

How will the line of data

123456.12798.00012004567800.10

be read by the following statements?

```
      READ (5,180) I,WUNDA,WOT,IT,WILL,BE
  180 FORMAT(I3,F7.0,F4.2,I4,F5.1,F4.2)

      READ (5,190) I,WUNDA,WOT,IT,WILL,BE
  190 FORMAT(I5,F6.2,F3.1,I7,2F5.2)

      READ (5,200) I,WUNDA,WOT,IT,WILL,BE
  200 FORMAT(I1,F3.1,F4.1,I3,F10.6,F3.2)
```

10. CHAPTER 10

10.1. A new dimension, the ARRAY

Consider the problem in which we read in data, being student grades in some subject. The problem is to read in the marks, and calculate the average grade. This may be done for N students by the following algorithm.

```
1. Set the total to zero initially
2. Repeat for each of the N students:
   (a) read in his/her grade
   (b) add it to the total
3. Calculate the average as the total divided by the number of students.
```

A FORTRAN program to do this is then:

```
      READ(5,*)N
      SUM = 0.0
      DO 20 M = 1,N
         READ(5,10)SCORE
10       FORMAT(F6.2)
         SUM = SUM + SCORE
20    CONTINUE
      AVE = SUM/N
```

Now consider a similar problem. We wish to calculate the average mark, and then print out those marks that are GREATER THAN the average. An algorithm (i.e. method) for solving this problem is:

1. Read in the marks, summing them as they are read.

2. Calculate the average

3. Scan ALL the marks, to find and print out all marks greater than the calculated average.

To solve this problem we need to store ALL the exam marks when we read them, so that we will be able to scan through them later, after we have calculated the average.

One solution would be to read all the data twice. This would be inefficient as I/O is very slow compared to the rest of your program, and would also be inconvenient to the user, who would have to enter his data twice. Another solution you may propose is to store each of the marks in a SEPARATE variable name. This approach suffers from two points. Firstly, what would happen if we had a large number of marks (say 9000), we would have THAT MANY variables! Secondly, this method would be very tedious, when it came to writing the 9000 odd IF statements to test if the variable value was greater than the average. This last statement implies that we wish to treat all the numbers in exactly the same way.

The solution to our problem is to introduce a new type in FORTRAN, called an ARRAY.

An array is a means of specifying a number of quantities that are to be stored in the one variable name. In order to use an array, we must specify two things:

(a) The variable name of the array
(b) The size of the array. This is the maximum number of quantities that may be stored in the array.

The statement:

```
      DIMENSION GRADE(125)
```

specifies 125 quantities (of type REAL). These quantities are called GRADE(1), GRADE(2), and so on up to GRADE(125). Note that they each have the same name GRADE, and are distinguished by what we call a SUBSCRIPT - an integer enclosed in brackets. The individual quantities are called ARRAY ELEMENTS, and all 125 of them considered together are called an ARRAY.

ARRAY ELEMENTS can be used anywhere that a variable can, for example, in an arithmetic expression, on the left hand side of an assignment statement, in a FUNCTION reference, and so on.

The real power in the use of arrays lies in the fact that we can refer to array elements, such as GRADE(I). If I has the value 25, then GRADE(I) refers to GRADE(25), and if I has the value 79 then it refers to GRADE(79).

In practice, of course, we arrange for I to take on the series of values in which we are interested, while referring to GRADE(I). And we do this in a loop. For example, to read in N numbers into the array GRADE we write:

```
       DO 20 I = 1,N
          READ(5,10)GRADE(I)
 10       FORMAT(F6.2)
 20    CONTINUE
```

So now we are in a position to write the new program. The complete program is:

```
C
C  PROGRAM TO READ IN STUDENT GRADES, ONE PER CARD, AND
C  TO CALCULATE THE AVERAGE GRADE. THEN TO PRINT OUT ALL
C  THOSE GRADES THAT ARE GREATER THAN THIS AVERAGE.
C
C  THE FIRST CARD CONTAINS THE NUMBER OF GRADES (IN COLS 1-5)
C  THE GRADES APPEAR IN COLUMNS 1-6
C  THERE IS A MAXIMUM OF 125 GRADES
C
      DIMENSION GRADE(125)
      READ (5,10) N
 10   FORMAT(I5)
      SUM = 0.0
```

```
C  READ IN THE GRADES AND SUM THEM

      DO 50 I = 1,N
         READ (5,40) GRADE(I)
 40      FORMAT(F6.2)
         SUM = SUM+GRADE(I)
 50   CONTINUE

C  FIND THE AVERAGE

      AVE = SUM/N

C  SCAN, PRINTING ALL GREATER THAN THE AVERAGE

      DO 70 I = 1,N
         IF (GRADE(I).GT.AVE) WRITE(6,60) GRADE(I)
 60      FORMAT (1X,F7.2)
 70   CONTINUE
      STOP
      END
```

10.1.1. Array sizes

You will notice that although we don't know how many grades there are going to be - it is N - the size of the array GRADE is specified as 125. The rules of FORTRAN insist that the size of an array (as declared in the DIMENSION statement) be a positive integer constant, and 125 was chosen on the grounds that it will always be bigger than the number of grades used. Clearly, some judgement must be applied in choosing suitable sizes for arrays. An array that is declared unnecessarily large will waste memory space (or worse still, may not fit in memory!). An array that is declared too small, will result in errors occurring, as you are not allowed to reference array elements that are OUT OF BOUNDS of the declared array size. In many programs a test is made to check that you never try this. For example, in the above example, we could insert after reading a value for N, the lines:

```
      IF (N.GT.125) THEN
         WRITE (6,*)  ' MAXIMUN OF 125 GRADES ALLOWED '
         STOP
      END IF
```

10.1.2. Forms of subscripts

When referring to an array element, it is possible to use slightly more general subscripts than the simple integer constant or integer variable we have seen so far.

1966 standard Fortran had very strict rules as to what forms of subscripts were allowed. The 1977 standards extended the allowable expressions to be any integer expression, including function references and other subscripted variables.

Univac has extended this further to allow any integer or real expression. Most Fortrans will accept a fairly general form of integer expression as a subscript, but you should try to keep your subscript expression simple for the sake of understandability.

If the version of Fortran that you wish to use has a limited form of subscript, then this is very easy to get around by replacing

```
        TOP = GRADE(A+B)
```

with

```
        IND = A+B
        TOP = GRADE(IND)
```

So as well as referring to GRADE(25) and GRADE(I) we can refer to GRADE(I-10), which, if I has the value 25, refers to GRADE(15) and also GRADE(2*I+3) which refers to GRADE(53).

10.1.3. Examples of Subscripts

Whenever a subscript is used, any variables within the subscript expression must have been given a value at some prior point in the program, as the subscript expression is evaluated first and then this uniquely identifies a particular element within the array. It is the programmer's responsibility to ensure that the subscript expression has a value between 1 and the maximum array size as declared in the DIMENSION statement. Some Fortrans will provide error termination at execution time if this happens, and some do not, but merely give incorrect results.

Provided that

(a) the arrays mentioned have been dimensioned (i.e. appear in a DIMENSION statement)

(b) the variables in the subscript expressions have been given a value at some prior point in the program

(c) the subscript expression has a value between 1 and the dimensioned size of the array (inclusive)

the following are valid array references

```
        FIRST(677)
        HGIELS(LE)
        A(MAX(LAD,10))
        HUNTER(LFS+1)
        XMAS(2*KANBRA)
        NEXT(4*KRAFT-3*KAAS)
        TOOH(19*KAT+MARK(7))
```

Some examples of invalid subscripts are:

Example	Reason
DOGS(BAR)	Real variable not allowed as a subscript
KATZ(I+J+1.0)	Real expression not allowed as a subscript

10.1.4. Example of complex subscript use

Sometimes it is difficult to see why you would want to ever use a subscript that is more complicated than a simple integer variable. One example of this is given as follows.

Suppose that we had an array in which each set of three consecutive locations referred to rainfall figures: the minumum, the maximum, and the average. This means that if we had 25 areas, we would have 25 lots of 3 figures, and so would need an array of size 75.

Suppose that variable LOC indicates the area number that we are interested in. Now if we want the minimum rainfall figures for this area, it would be given by

```
        RAIN ( (LOC-1)*3 + 1 )
```

Value of LOC	Element of RAIN
1	1
2	4
3	7
4	10
etc	etc

10.1.5. Type Declarations for Arrays.

As with integer and real variables, an array name need not have the default type depending on its first letter. To specify array GRADE as integer, use INTEGER GRADE(125) instead of DIMENSION GRADE(125). To specify A(10) as an integer array, and IX(12) and Y(14) as real arrays, use:-

```
  INTEGER A(10)
  REAL    IX(12),Y(14)
```

instead of

```
  DIMENSION A(10),IX(12),Y(14)
```

10.1.6. Summary points about DIMENSION statements

1. Any number of DIMENSION statements may appear in a program.

2. A variable may appear only once in any DIMENSION statement.

3. The size of an array MUST be an integer CONSTANT or integer expression containing constants (not variables).

4. The DIMENSION statement must appear in the program before any executable statements, (i.e. at the top), with REAL and INTEGER type declarations.

5. Any array used in the program must appear in a DIMENSION statement or INTEGER or REAL declaration.

6. 1977 Fortran introduced a mechanism for allowing the lower bound of the dimension of an array to be any value (and consequently the upper bound could be anything greater than the lower bound). This will not be covered in this course. The syntax for such a declaration is

```
        DIMENSION variable(lower-bound : upper-bound)
```

10.2. Reading and writing arrays

Suppose that we have 10 integers on an input line, and that each integer takes 6 columns. So we say the numbers may be read by the format:

```
 20     FORMAT(10I6)
```

We could read them like this:

```
      DIMENSION NUMS(10)
      READ(5,20)NUMS(1), NUMS(2), NUMS(3), ....,NUMS(10)
```

This is tedious, and seems to be made for a DO loop. So can we write this?

```
      DIMENSION NUMS(10)
      DO 30 I = 1,10
         READ(5,20)NUMS(I)
 20      FORMAT(10I6)
 30   CONTINUE
```

The answer of course is NO. The reason is that every READ statement will start reading a new line. The first time around the loop, the first number is read into NUMS(1) correctly. The second time around, a new line is read, and so the other nine values (that were on the first line) are lost.

Clearly, some form of DO-looping is required. Let us look at the correct piece of program to see how it is implemented.

```
      DIMENSION NUMS(10)
      READ(5,20) (NUMS(I), I = 1,10)
 20   FORMAT(10I6)
```

Here, instead of a normal input list, such as A, B, ZAP we have a DO - IMPLIED list. In this case it consists of the array element name, NUMS(I), followed by a comma, followed by I = 1,10 (which clearly reflects the structure of the DO statement), and the whole lot is enclosed in brackets. It can be thought of as 'NUMS(I) for I going from 1 to 10'. There are really 10 variables on the input list, the variables NUMS(1) to NUMS(10). The IMPLIED DO loop is merely a shorthand way of explicitly writing out the ten array elements.

The DO-implied list may be part of a larger input-output list. For example, the statement:

```
      READ(5,60) P, BEAN, (NUMS(I), I = 1,10)
```

causes the first two numbers to be read into P and BEAN and then the next 10 into NUMS(1) to NUMS(10).

```
      READ(5,70) (ZOT(I), I = 1,5), (NUMS(I), I = 1,10)
```

causes the first five numbers to be read into ZOT(1) to ZOT(5), and the next 10 numbers into NUMS(1) to NUMS(10).

```
      READ(5,80) (ZOT(I), ZOWIE(I), I = 1,5)
```

illustrates the fact that there may be a list as the first part of the DO - implied list. The first two numbers are read into ZOT(1) and ZOWIE(1), the next two into ZOT(2) and ZOWIE(2) and so on.

We have shown the initial and final value of the 'control variable' to be both integer constants. In fact they may be integer variables (just as in a real DO loop). Further, we may also specify an increment value, if we want one other than the default value of 1 (also just like a DO loop).

The following example will read in numbers into every second element of array SPRED. Notice that 10 numbers will be read, the first going into SPRED(1), AND THE LAST GOING INTO SPRED(19).

```
      INTEGER SPRED
      DIMENSION SPRED(20)
      READ(5,90) (SPRED(IND), IND = 1,20,2)
 90   FORMAT(10I6)
```

10.3. Common errors with arrays

A variable has been used as a subscripted variable but has not been declared as an array in a DIMENSION statement, or has not been given a dimension in a type declaration.

A declared array name is referenced without specifying its subscript.

The size of an array specified in a DIMENSION statement or type declaration is not an integer constant.

An array element is referenced using a variable, but this variable is either real, or does not currently have a defined value, or has a value less than 1 or greater than the maximum size specified in the declaration.

10.4. Exercises

All exercises involving arrays must include array declarations.

10.4.1. Exercise 10A

Find the mistakes in the following.

```
      DIMENSION J(20)
      DO 14 I=1,100
 14   J(I) = 0

      DIMENSION ARRAY(150)
      DO 22 K=2,47,2
 22   ARRAY = ARRAY(K+3) + K

      DIMENSION I(10),ARRAY(20)
      DO 7 I = 1,20,2
      ARRAY(I) = -3.
  7   ARRAY(I-1) = REAL(I)
```

10.4.2. Exercise 10B

Write a program to calculate and print the average rainfall figures for each of 8 localities.

Data consists of 8 lines of input (one line per locality). Each line of input has 12 numbers on it representing the rainfall for each of 12 months, for a locality.

Use only one (one dimension) array.

10.4.3. Exercise 10C

Identify the following DIMENSION statements as being correct or not.

```
(i)      DIMENSION A(4),K(7)
(ii)     DIMENSION BAD(12),ROTTEN(17+12)
(iii)    DIMENSION GOOD(K)
(iv)     DIMENSION BIG(1000),SMALL(3)
```

10.4.4. Exercise 10D

Consider each pair of the following statements (not necessarily contiguous) to be in separate programs. For each pair, indicate any inconsistencies. This means, any errors that are NECESSARILY errors, not merely potentially errors.

```
      DIMENSION X(5),L(10),A(15)
      Y = X+65.0/A(I)

      W = N(3)+I4*6
      D = W**4

      DELTA = A(I)/X**8
      START = DELTA**Y-A(B)

      DIMENSION A(10),K(15),A(3),T(6)
      T(15) = 2.0*T(3)+K(5)*3.14159
```

10.4.5. Exercise 10E

Write a series of statements to zero all locations in the array B from B(1) to B(100) inclusive.

10.4.6. Exercise 10F

Statistics are being kept on 20 southern Queensland national purple tailed hens. At the end of each month, a card is punched which contains the following information for each chicken.

Columns 1 to 2	contain an integer to be used for identification,
columns 3 to 4	contain the number of eggs laid this month,
columns 5 to 8	contain an integer which is the weight of feed consumed in grams,
columns 9 to 12	contain an integer which is the weight of the bird in grams.

Write a program to read the 20 cards and print out for each hen,

identification number
number of eggs laid and difference from the average,
weight of feed consumed and difference from the average,
weight of the bird and difference from the average.

10.9.4. Exercise 100

Consider each pair of the following statements [illegible] to be in separate programs. For each pair, indicate any inconsistencies. This means, any errors that are NECESSARILY errors, not merely POTENTIALLY errors.

```
      DIMENSION X(5),L(10),A(15)
      Y = A+B5.0/A(1)

      W = X(5)+L**6
      Z = W**A

      DELTA = A(I,J)**2
      START = DELTA**I-A(6)

      DIMENSION A(10),X(15),A(3),T(6)
      T(15) = 2.0*T(3)-X(5)*3.14159
```

10.9.5. Exercise 102

Write a series of statements to zero all locations in the array B from B(1) to B(100) inclusive.

10.9.6. Exercise 103

Statistics are being kept on 20 southern Congealed rational purple tailed ducks. [illegible] card is prepared which contains the following information:

```
Columns 1 to 2     contain an integer to be used for identification.
columns 3 to 4     contain the number of eggs laid [illegible]
columns 5 to 8     contain an integer which is the weight of feed consumed in
                   grams.
columns 9 to 12    contain an integer which is the weight of the bird in
                   [illegible].
```

Write a program to read the 20 cards and print out for each duck:

```
      identification number
      number of eggs laid and difference from the average,
      weight of feed consumed and difference from the average,
      weight of the bird and difference from the average.
```

11. CHAPTER 11

11.1. Arrays with TWO dimensions

The arrays we have considered to date have been one dimensional. We have been able to specify a unique array element by using only one subscript. In mathematical terms these are vectors.

We can also have arrays of two dimensions (and indeed 3 dimensions, but we will not be considering those). These may be considered as equivalent to matrices in mathematics.

Suppose we were analysing the results of a number of students who sat a number of exams. The results might be tabulated (by us) as:

	Chemistry A01	Forestry D31	Zoology A01
Bloggs	66	72	51
Nurke	73	88	60
Eccles	50	71	75
Seagoon	24	12	51
Moriarty	77	79	62

For convenience sake, we give students STUDENT NUMBERS, and also give the particular subjects, SUBJECT NUMBERS. So, our table then becomes:

	1	2	3
1	66	72	51
2	73	88	60
3	50	71	75
4	24	12	51
5	77	79	62

Let us give this table a name, say call it MARKS. We may now refer to the mark obtained by student number 2, in subject number 3, by referring to the SECOND ROW and the THIRD COLUMN of the table MARK, and we obtain the mark 60.

We may represent this table in Fortran by declaring a two dimensional array that has five ROWS (corresponding to the 5 student numbers) and three columns (corresponding to the 3 subject numbers). This is done with the DIMENSION statement:

```
      DIMENSION MARK(5,3)
```

The reason for representing this as a two dimensional array, rather than a one dimensional array, is that all the Chemistry marks are in one column, and all of Eccles marks are in one row.

If we imagine a general version of this array, then to find the total mark for subject 2, we would write the following. (NSTUDS is the number of students, i.e. the number of rows in the table, and it is assumed that MARK has been DIMENSION'ed to an appropriate size):

```
      ITOT = 0
      DO 5 I = 1,NSTUDS
         ITOT = ITOT + MARK(I,2)
  5   CONTINUE
```

Similarly, we could find the total marks for a particular student, say for Eccles, student number 3.

```
      ITOT = 0
      DO 6 J = 1,NSUBS
         ITOT = ITOT + MARK(3,J)
  6   CONTINUE
```

where NSUBS is the number of subjects, i.e. the number of columns of the two dimensional array MARK.

Let us now generalise this a little further. We want a program that will read in the table of marks and store them in the array MARK. Now read data that requests the total marks for either a particular student, or a particular subject. We will indicate that we want a student total by reading a line with a 1 in column 1 and the student number we want in columns 3 - 5. If we want a subject total, then we will have a 0 in column 1, and the subject number in columns 3 - 5.

Before we can attack this problem, we need to know a little more about how we may read in a two dimensional array.

11.1.1. Reading and writing two dimensional arrays

A two dimensional array is composed of a number of rows, each row being made up of a number of columns. One way of thinking about two dimensional arrays is to consider that they are composed of a number of one dimensional arrays (each corresponding to a row for example) repeated for however many rows we have.

The subscripts required to identify an element of the two dimensional array are

(a) a row subscript to identify which row vector to choose
(b) a column subscript to identify which column to choose of the row vector specified by (a).

The reading of a two dimensional array may be expressed in the following algorithm.

```
Repeat for each row of the array:

   Read in numbers into a row vector.
```

Suppose we want to read numbers into a 2 dimensional array, of size 4 rows and

3 columns. There are 4 lines of data, each containing 3 numbers to go into one row of the array and each number taking 5 columns of a data line.

First, look at how to read in any given row of the array, say row 1. This may be done by:

```
     DIMENSION NURGLE(4,3)
     READ(5,110) (NURGLE(1,J),J=1,3)
 110 FORMAT (3I5)
```

To read in all 4 rows we could have

```
     READ(5,110)(NURGLE(1,J),J=1,3)
     READ(5,110)(NURGLE(2,J),J=1,3)
     READ(5,110)(NURGLE(3,J),J=1,3)
     READ(5,110)(NURGLE(4,J),J=1,3)
```

This may be abbreviated further by the use of a DO loop that will vary the row subscript from 1 to 4, and so giving:

```
     DIMENSION NURGLE(4,3)
     DO 5 I = 1,4
        READ(5,110)(NURGLE(I,J),J=1,3)
 5   CONTINUE
 110 FORMAT (3I5)
```

11.1.2. Nested Implied DO loops

Let us go back to reading in a one dimensional array using an IMPLIED DO loop.

```
         DIMENSION A(15)
         READ(5,107) (A(I), I=1,15)
   107   FORMAT(15F5.1)
```

We can see that the syntax of the implied DO loop is:

```
         (list of variables, control = start, finish, increment)
```

The 'list of variables' may also contain a DO-implied list. So, we can write:

```
         READ(5,109)( (B(I,J), J = 1,10), I = 1,5)
```

This may be thought of as being 'B(I,J) with J going from 1 to 10 and I going from 1 to 5'. J is in the INNER loop, and so varies most frequently (i.e. the inner loop completes for each value of the outer loop). So this statement would read B(1,1) then B(1,2) then B(1,3) .... up to B(1,10) then B(2,1) and so on.

We can now use this to read and write two dimensional arrays. For example, to read integers into a two dimensional array, of size 4 rows and 3 columns (supposing all 12 numbers are on one line in I5 format), we need the statements:-

```
        DIMENSION NURGLE(4,3)
        READ(5,101) ( (NURGLE(I,J), J = 1,3), I = 1,4)
  101   FORMAT(12I5)
```

Remember the DO implied list is merely a shorthand form for writing out all the array elements explicitly, so in the above READ statement there are really 12 variables on the I/O list.

Suppose we wish to do the same thing, but now we design the data differently, and say that a line will contain 3 numbers, these corresponding to a row. So there will be 4 lines of data, each with one row to be read into NURGLE. We can do this by FORMAT control, with:

```
        DIMENSION NURGLE(4,3)
        READ(5,109) ( (NURGLE(I,J), J=1,3), I=1,4)
  109   FORMAT(3I5)
```

This works, because when we get to the end of the FORMAT statement (which we will after reading three numbers), we start the FORMAT statement again and start reading another line. So a total of 4 lines is read.

A much better way, because it is more straight forward, is to use an explicit DO loop with an implied DO loop as in:-

```
        DIMENSION NURGLE(4,3)
        DO 5 IROW = 1,4
           READ(5,110) (NURGLE(IROW,J), J=1,3)
  5     CONTINUE
  110   FORMAT(3I5)
```

11.1.3. Example

Let us now return to our exam mark example. We shall define the format of the data as being:

1. Each line of data will contain the marks for a particular student in all subjects. There will be a total of 25 subjects, and the marks will be in I3 format.

2. There will be a maximum of 200 students, and the last line will be a 'dummy' student that has a negative mark, thus signalling the end of the data.

So to read in the marks we need:

```
        DIMENSION    MARK(201,25)
        MAXS = 200
  C
  C READ IN THE STUDENT TABLE
  C
        NROWS  = 1
  10    CONTINUE
```

```
        READ (5,20) (MARK(NROWS,J),J=1,25)
 20     FORMAT(25I3)
C
C       INCREMENT AND TEST ROW POINTER
C
        IF (MARK(NROWS,1).LT.0) THEN
           NROWS = NROWS - 1
           GO TO 40
        ELSE IF (NROWS.LE.MAXS) THEN
           NROWS = NROWS + 1
           GO TO 10
        ELSE
           WRITE (6,30) MAXS
 30        FORMAT (1X,'TOO MANY STUDENTS MAXIMUM OF',I3,' ALLOWED')
           STOP
       END IF
 40    CONTINUE
```

So at this point in the program, we have set up the array of marks in the Fortran array called MARK. Now we need to read in data, which will indicate the type of total we want, and which student, or subject, to total. We have not specified how the number of requests will be indicated. Let us say that we should read requests until a type greater than 1 is read.

```
C
C    READ IN A REQUEST
C
 50    CONTINUE
       READ(5,60)ITYPE,NUMBER
 60    FORMAT(I1,1X,I3)
C
C      BRANCH ON THE VALUE OF ITYPE
C      ITYPE = 0, FORM TOTAL FOR A GIVEN SUBJECT
C            = 1, FORM TOTAL FOR A GIVEN STUDENT
C           GT 1, STOP
C
       IF (ITYPE.GT.1) STOP
       IF (ITYPE.EQ.0) GO TO 90
C
C    FIND THE TOTAL MARKS GAINED BY THE REQUESTED STUDENT
C
       ITOT = 0
       DO 70 J = 1,25
          ITOT = ITOT + MARK(NUMBER,J)
 70    CONTINUE
       WRITE(6,80)NUMBER,ITOT
 80    FORMAT(1X,'THE TOTAL FOR STUDENT NUMBER ',I3,' IS ',I5)
       GO TO 50
C
C      FORM THE TOTAL MARKS SCORED IN THE
C      REQUESTED SUBJECT  (ITYPE=0)
C
 90    CONTINUE
       ITOT = 0
       DO 100 I = 1,NROWS
          ITOT = ITOT + MARK(I,NUMBER)
 100    CONTINUE
        WRITE(6,110)NUMBER,ITOT
 110    FORMAT(1X,' THE TOTAL FOR SUBJECT NUMBER ',I3,' IS ',I5)
        GO TO 50
        END
```

Now let us put all the sections together, and include the necessary commands to execute the program on the UNIVAC computer assuming that no mistakes were made in inputting the program! The data will be set up to read in the array as specified in the example above, and to request totals for student 2, and also for subject 1.

```
 sign-on procedure (userid/password)
 @CAT,P PROG.
 @ASG,AZ   PROG.
 @ED,I   PROG.STUD
 C
 C  AUTHOR:       L. LANDAU
 C  DATE:         12 MAY 1976
 C  MODS:         FEB 1981
 C                L.LANDAU
 C                TO RENUMBER STATEMENT NUMBERS
 C  LANGUAGE:     UNIVAC ASCII FORTRAN LEVEL 9R1
 C  COMPUTER:     UNIVAC 1100/82
 C  LOCATION:     AUSTRALIAN NATIONAL UNIVERSITY
 C
 C  PROGRAM DESCRIPTION:
 C  --------------------
 C
 C
 C  THIS PROGRAM WILL READ IN A TABLE OF STUDENT MARKS
 C  OBTAINED IN EACH OF 25 EXAMS.
 C
 C  THE PROGRAM WILL THEN READ IN REQUESTS FOR TOTALS OF EITHER
 C  (0)   TOTAL MARKS GAINED IN A PARTICULAR SUBJECT
 C  (1)   TOTAL MARKS GAINED BY A PARTICULAR STUDENT
 C  (>1)  STOP PROGRAM
 C
 C
 C  DATA DESCRIPTION:
 C  -----------------
 C
 C  THE STUDENT/SUBJECT TABLE COMES FIRST IN THE DATA,
 C  WITH EACH STUDENT TAKING ONE LINE,
 C  AND EACH SUBJECT TAKING 3 COLUMNS. IF ANY MARKS ARE
 C  OMITTED, THEN THEY WILL BE TREATED AS ZERO.
 C  THERE WILL BE A MAXIMUM OF 200 STUDENTS.
 C  THE END OF THE STUDENT DATA
 C  IS SIGNIFIED BY A NEGATIVE FIRST SUBJECT MARK.
 C
 C  FOLLOWING THE TABLE OF MARKS, THERE ARE REQUESTS FOR TOTALS
 C  OF EITHER STUDENT MARKS IN A PARTICULAR SUBJECT, OR
 C  THE TOTAL MARKS FOR A PARTICULAR STUDENT. THESE REQUESTS
 C  TAKE THE FORM:
 C
```

```
C       COLUMN        MEANING
C       ------        -------
C         1           TYPE OF REQUEST
C                     IF 0 THEN TOTAL THE STUDENT MARKS IN THE
C                     GIVEN SUBJECT
C                     IF 1 THEN TOTAL THE SUBJECT MARKS FOR THE
C                     GIVEN STUDENT
C                     IF GREATER THAN 1 THEN STOP
C
C        3-5          THE STUDENT NUMBER, OR SUBJECT NUMBER.
C
      DIMENSION    MARK(201,25)
      MAXS = 200
      MAXSUB = 25
C
C READ IN THE STUDENT TABLE
      NROWS = 1
 10   CONTINUE
      READ (5,20) (MARK(NROWS,J),J=1,MAXSUB)
 20   FORMAT(25I3)
C
C     INCREMENT AND TEST ROW POINTER
C
      IF (MARK(NROWS,1).LT.0) THEN
         NROWS = NROWS-1
         GO TO 40
      ELSE IF (NROWS.LE.MAXS) THEN
         NROWS = NROWS + 1
         GO TO 10
      ELSE
         WRITE(6,30) MAXS
 30      FORMAT(1X, 'TOO MANY STUDENTS MAXIMUM OF ',I3,' ALLOWED')
         STOP
      END IF
C
C   COME HERE WHEN END OF STUDENT TABLE FOUND
C
 40   CONTINUE
C
C   READ IN A REQUEST
C
      READ(5,50)ITYPE,NUMBER
 50   FORMAT(I1,1X,I3)
C
C   VALIDATE THE REQUEST IS IN RANGE
C
      IF(ITYPE.GT.1) STOP
      IF((ITYPE.EQ.0.AND.NUMBER.GT.MAXSUB) .OR.
     $   (ITYPE.EQ.1.AND.NUMBER.GT.NROWS)) THEN
         WRITE(6,60)ITYPE,NUMBER
 60      FORMAT(1X,'FOR A TYPE ',I1,' REQUEST, NUMBER ',I3,
     $          ' IS OUT OF RANGE....IGNORED')
         GO TO 40
      END IF
C
```

```
C     BRANCH ON THE VALUE OF ITYPE
C     ITYPE = 0, FORM TOTAL FOR A GIVEN SUBJECT
C           = 1, FORM TOTAL FOR A GIVEN STUDENT
C
       IF (ITYPE.EQ.0) GO TO 90
C
C   FIND THE TOTAL MARKS GAINED BY THE REQUESTED STUDENT
C
       ITOT = 0
       DO 70 J = 1,MAXSUB
          ITOT = ITOT + MARK(NUMBER,J)
 70    CONTINUE
       WRITE(6,80)NUMBER,ITOT
 80    FORMAT(1X,'THE TOTAL FOR STUDENT NUMBER ',I3,' IS ',I5)
       GO TO 50
C
C     FORM THE TOTAL MARKS SCORED IN THE
C     REQUESTED SUBJECT  (ITYPE=0)
C
C
 90    CONTINUE
       ITOT = 0
       DO 100 I = 1,NROWS
       ITOT = ITOT + MARK(I,NUMBER)
 100   CONTINUE
       WRITE(6,110)NUMBER,ITOT
 110   FORMAT(1X,' THE TOTAL FOR SUBJECT NUMBER ',I3,' IS ',I5)
       GO TO 50
       END
@EOF
@ED,I  PROG.STUD/DATA
 66 72 51
 73 88 60
 50 71 75
 24 12 51
 77 79 62
 -1
1   2
0   1
3
@EOF
@FTN,CS   PROG.STUD
@EOF
@ADD,E PROG.STUD/DATA
```

This would print out:

```
THE TOTAL FOR STUDENT NUMBER   2 IS   221
THE TOTAL FOR SUBJECT NUMBER   1 IS   290
```

This program is still very primitive.

For example, it cannot handle the situation of students not sitting for exams. It provides only elementary output. What may be more interesting is to provide figures on who got more than 50%, or who the top student was. How would you make these changes? How much program re-design would it require?

11.1.4. Rainfall example

Suppose that we had a table of rainfall figures for different areas over different years starting from 1950. So the table may look like:

	Qld	NSW	Vic	Tas	NT	ACT	SA	WA
1950	103.5	99.3	89.6	121.6	41.2	88.8	84.5	77.4
1951	104.7	98.4	90.6	125.8	38.5	86.3	78.3	68.3
1952	109.8	101.6	92.4	132.6	39.5	85.4	79.5	72.1
1953	101.6	101.9	99.9	119.1	32.1	90.5	81.6	85.6

and so on for other entries, one row per year
until the last one in (say) 1979:-

1979	117.7	103.2	97.1	128.8	40.0	91.4	86.3	88.7

We could write a program to read these rainfall figures into an array and then extract rainfall figures for different areas over different years. For example, the Fortran to read the figures in could be:

```
C
C     MAXIMUM OF 40 YEARS  (STARTING AT 1950)
C     MAXIMUM OF 10 AREAS
C
      DIMENSION RAIN(41,10)
      MAXARA = 10
      MAXYRS = 40
C
C    READ IN HOW MANY AREAS THERE ARE
C
      READ (5,20) NAREAS
 20   FORMAT (I2)
      IF (NAREAS.LE.MAXARA) GO TO 40
      WRITE (6,30) MAXARA
 30   FORMAT(1X, 'TOO MANY AREAS, CAN ONLY HANDLE ',I3)
      STOP
C
C     READ IN RAINFALL FIGURES, THE END IS INDICATED BY
C     A NEGATIVE RAINFALL FOR THE FIRST AREA
C
 40   CONTINUE
      IYR = 1
 50   CONTINUE
      READ (5,60) (RAIN(IYR,LOC), LOC = 1,NAREAS)
 60   FORMAT (16F5.0)
```

```
      IF (RAIN(IYR,1).LT.0) GO TO 80
      IYR = IYR + 1
      IF (IYR .LE. MAXYRS+1) GO TO 50
      WRITE (6,70) MAXYRS
 70   FORMAT(1X,'TOO MANY YEARS, CAN ONLY HANDLE ',I3)
      STOP
C
C   COME HERE WHEN ALL RAIN DATA IS READ
C
 80   CONTINUE
      IYR = IYR - 1
```

Now the problem is to produce rainfall figures. Suppose that we want the average rainfall in Tasmania betwen 1960 and 1969 (inclusive).

```
      TOTAL = 0.0
      DO 90 IYEAR = 1960,1969
      TOTAL = TOTAL + RAIN (IYEAR-1949,4)
 90   CONTINUE
      AVRAIN = TOTAL/10
```

Generalising this a little, we want to solve the following problem:
To be able to read in 3 numbers indicating:

(a) the start year (for example 1960)
(b) the number of years to cover
(c) the area number that we are interested in

and from this find the average rainfall in that area over the requested years.

```
C
C   READ IN THE INPUT PARAMETERS: START YEAR
C                                 NUMBER OF YEARS
C                                 LOCATION CODE
C
C
      IBEGIN =  1950
      READ (5,100) IST, NUMYR, LOCAT
 100  FORMAT (I4,1X,I2,1X,I2)
C
C  CHECK VALIDITY OF REQUESTS
C
      LAST = IBEGIN + IYR - 1
      IF( IST.GE.IBEGIN .AND. IST.LE.LAST) GO TO 120
      WRITE (6,110) IBEGIN, LAST
 110  FORMAT(1X,'THE FIRST RECORDING YEAR IS ',I4,' AND THE',
     $          ' LAST IS ',I4)
      STOP
 120  CONTINUE
      IF (NUMYR+IST-1 .LE. LAST) GO TO 130
      WRITE (6,110) IBEGIN, LAST
      STOP
```

```
  130 CONTINUE
      IF (LOCAT .LE. NAREAS) GO TO 150
      WRITE (6,140) LOCAT
  140 FORMAT(1X,'THERE ARE NO FIGURES FOR AREA ',I3)
      STOP
C
C   NOW WE HAVE VALIDATED THE INPUT SO DO THE CALCULATION
C
  150 CONTINUE
      LAST   = IST + NUMYR - 1
      TOTAL  = 0.0
      DO 160 IYEAR = IST,LAST
         INDEX = IYEAR - IBEGIN + 1
         TOTAL = TOTAL + RAIN (INDEX,LOCAT)
  160 CONTINUE
      AVRAIN = TOTAL/NUMYR
      WRITE (6,170) LOCAT, AVRAIN
  170 FORMAT(1X,'THE AVERAGE RAINFALL IN AREA ',I3,
     $       ' IS ',F8.2,' CENTIMETRES')
      STOP
```

As an exercise, how could you find out the yearly difference in rainfall between any two areas between any two years?

11.1.5. More FORMAT Descriptors: E format

The UNIVAC computer is able to represent very large, and very small REAL numbers (in the range 10**(-38) to 10**38), and using the F format descriptor is not convenient for writing out these numbers.

The E format descriptor may be used with REAL variables to read or write numbers, so that they appear with an EXPONENT. The form of this format descriptor is:

```
        Ew.d
```

The 'w' and the 'd' have the same meaning as for F format.

If E format is used to output a number the form of output will be:

```
        .nnnnn+eee
```

The 'nnnnn' are the digits after the decimal place. Plus or minus 'eee' is the exponent for the decimal number, and is so arranged that the first 'n' of the 'nnnnn' is non-zero. There will NEVER be any digits to the left of the decimal point.

The number of places after the decimal point is specified by 'd'. Note that a space must be left for a possible '-' sign to the left of the number, so there are a possible 6 extra places taken up other than the d decimal places and so 'w' must be at least six greater than 'd'. The statements:

```
      FIRST = 2.3756
      SEC   = -677.32E-12
      THIRD = 3444.55E18
      WRITE(6,101) FIRST,SEC,THIRD
  101 FORMAT(1X,E12.3,E20.6,E14.7)
```

will print the line:

```
bbbb.238+001bbbbbbbb-.677320-009bb.3444550+022
```

On input, if the decimal point is on the data card it overrides the 'd' specification of the E format descriptor. The exponent need not be three digits, and the number must be right-justified in the field (otherwise the blanks at the right end of the field are read as zeros, and included in the exponent).

11.1.6. L format

Logical values (true and false) may be read or written with the L format, which has the form:

```
Lw
```

where w is an integer representing the width of the field.

On input, blanks and/or a decimal point are allowed to precede a T or F, and the rest of the field is ignored. So with a format descriptor of L7,

```
b.TRUE.  ,  bbbTbbb  and  THAT'Sb  are all read as TRUE, and
.FALSE.  ,  Fbbbbbb   and   bbbbFAT   are FALSE.
```

On output, w-1 blanks are written, followed by a T or F.

11.2. Exercises

11.2.1. Exercise 11A

Show how the given data values would be printed under the control of the format descriptor E10.3

(a) 0.0	(d) 10.0	(g) 0.00027
(b) 323.33	(e) -42.1E-5	(h) -663.2544
(c) 44.3E10	(f) 0.001002	(i) 0.0000003E-17

11.2.2. Exercise 11B

rite a program to generate the numbers 1 to 30 in a one dimensional array and then print this array so that there is a heading before the numbers, and then the numbers on one line. Print the numbers again, six per line, with a line number on each line. The output should be

```
THE NUMBERS ARE

 1  2  3  4  5  6  7  8  9  . . .  30

   1    1    2    3    4    5    6
   2    7    8    9   10   11   12
   3   13   14   15   16   17   18
   4   19   20   21   22   23   24
   5   25   26   27   28   29   30
```

11.2.3. Exercise 11C

Repeat exercise 11B, but using a two dimensional array with 10 rows and 3 columns. The output should be exactly the same. Fill up the 10 by 3 array by rows rather than by columns.

11.2.4. Exercise 11D

Write a series of statements to add the corresponding elements of two m x n arrays A and B and store the result as an m x n array C. Assume that the maximum values of m and n will be 14 and 10 respectively.

11.2.5. Exercise 11E

The coordinates of a point in an n dimensional space are punched on cards. The value of n is punched on a separate card which is placed first. Write a program to read these values and find the distance d of the point from the origin.

$$d = (x1^2 + x2^2 + \ldots + xn^2)^{1/2}$$

Assume that n will not be greater than 25.

11.2.6. Exercise 11F

Re-write your answer to exercise 10B, this time also printing out the entire rainfall figures for the month with the highest average. Use only one two-dimensional array.

12. CHAPTER 12

12.1. Subprograms

A program is a sequence of Fortran statements that is executed by a computer. The programs we have seen to date are referred to as MAIN programs.

When a MAIN program is executed, the computer starts by executing the first line and then proceeds sequentially according to the rules we have learned. A subprogram is also a sequence of Fortran statements. Execution of a subprogram is initiated by the main program, when the main program refers to the name of the subprogram, ie. when the main program CALLS the subprogram. When a subprogram finishes, control goes back to the next statement in the main program and so execution of the main program is continued.

There are two types of subprograms in Fortran, known as SUBROUTINEs and FUNCTIONs. The main difference between the two is in the manner by which they are started.

An example of a subprogram reference is the use of the standard generic FUNCTION covered in a previous chapter. The statement:

```
      VAL = MAX(QUANT,ZENA)
```

is an example of a program referring to the subprogram (in this case FUNCTION) called MAX. As well as referring to the subprogram by name, it may be required to supply a parameter list for the subprogram to use. This is done in the above example by enclosing the parameters in brackets after the function name, in the reference to the function.

The parameters provide a means of passing information to and from the subprogram. i.e. they are the communication between the two program parts.

The referencing of the function causes execution of the Fortran statements that comprise the body of the function. So far we have not been able to see the lines of Fortran involved, as the system keeps track of it all for the standard Fortran functions. Now we shall see ways in which to create our own subprograms for which the lines of Fortran that make up the body of the subprogram (i.e. what it will do) must be supplied by us.

12.2. Subroutines

12.2.1. Referencing subroutines

When a subroutine is referenced we say that a CALL has been made to the subroutine.

Subroutines have names that are from 1 to 6 characters in length composed of alphabetic and/or numeric characters with the first one being alphabetic (these are the same rules as for variable names). Some examples of valid subroutine names are below.

```
        ADDER
        F47A62
        J
        READ
        MOVE
```

A subroutine is called by using the CALL statement, which has the form:

```
        CALL  name (parameter list)
```

The parameter list (called a list of ACTUAL parameters) is composed of constants, variables, arrays, function names and expressions separated by commas. As is the case with FUNCTIONs used so far, these ACTUAL parameters supply the subroutine with variables and values that it actually requires to operate on.

The subroutine may reference the parameters (use their values) or in the case of an actual parameter being a variable name, the subroutine may assign it some other value.

The actual parameters are referenced within the subroutine by their association with a corresponding list of parameters that the subroutine is aware of. These parameters are called DUMMY parameters (sometimes referred to as FORMAL parameters). The list of DUMMY parameters is kept in the first line of the subroutine, called the subroutine header.

The ACTUAL and DUMMY parameters represent the same corresponding physical location in computer memory. The main program may call them one set of names (ACTUAL parameters), and the subprogram another set of names (DUMMY parameters). However, both refer to the same physical object. The winner of the 1977 Melbourne Cup could be referred to as No. 2, or "Gold and Black", but it is the same horse.

If there are no parameters then the brackets may be omitted.

12.2.2. SUBROUTINE statement

The form of the subroutine header is :

```
        SUBROUTINE  name (DUMMY parameter list)
```

The DUMMY parameters MUST be variable or array names which are associated with a list of ACTUAL parameters when the subroutine is called. If the parameter list is empty, the brackets may be omitted.

The association between ACTUAL and DUMMY parameters is in the order in which the parameters appear. The first ACTUAL parameter is associated with the first DUMMY parameter, the second parameter of each list associated with the second parameter of the other, etc.

To make the association possible, both lists must correspond in:

(a) the number of parameters appearing in each
(b) the TYPE of the parameters (INTEGER or REAL)

The lines of Fortran within the subroutine (called the BODY of the subroutine) involve the use of the DUMMY parameters. When the subroutine is called, references within the subroutine to DUMMY parameters become references to the corresponding ACTUAL parameters.

The DUMMY parameters are so called because they do not exist as separate variables. They are merely used as "alias" names in the subprogram for the ACTUAL parameters, which have physical locations associated with them.
When a subroutine references one of its DUMMY parameters, it is actually referencing the physical location of the corresponding ACTUAL parameter.

For Example

```
     TAX = 1.6              SUBROUTINE SUB(CASH)
     CALL SUB(TAX)                   .
             .                       .
             .                 XYZ = CASH/2.0
             .                 WRITE(6,*)XYZ
             .                       .
             .                       .
```

would have exactly the same effect as :-

```
     TAX = 1.6
     PQR = TAX/2.0
     WRITE(6,*)PQR
```

CASH is just a name used by the SUBROUTINE to reference a variable, which in this case happens to be known by another name in the main program.
Since a DUMMY and ACTUAL parameter are the same location, any alteration in the value of a DUMMY parameter in a subprogram will also alter the value of the ACTUAL parameter in the main program.

12.2.3. A subroutine for getting into a car

Suppose that it is possible to write a subroutine in FORGLISH, which is a mixture of FORTRAN and ENGLISH. The following may then be a subroutine for getting into a car.

```
        SUBROUTINE  ENTER(CAR,DOOR)
          1. open the DOOR of the CAR
          2. move through the DOOR into the CAR
          3. close the DOOR of the CAR
        END
```

This FORGLISH subroutine would work equally well on any type of CAR. If it were called by:

```
        CALL ENTER(HOLDEN,DOOR)
```

then the CAR referred to in the subroutine would actually be HOLDEN, as the actual parameter HOLDEN is associated with the dummy parameter CAR. In the call:

```
        CALL ENTER(VW,DOOR)
```

the CAR referred to in the subroutine would be VW. In the call:

```
        CALL ENTER(DATSUN,BOOT)
```

the subroutine would now be used to get into the BOOT of DATSUN.
In the call:

```
        CALL ENTER(F111,CANOPY)
```

the subroutine would now be used to get into an F111.
What would happen in the call:

```
          CALL ENTER(GATE,HOUSE)
```

Following through the subroutine, and associating the parameters together, we get the following:

```
       GATE    is associated with CAR
       HOUSE   is associated with DOOR
```

and now the subroutine would be trying to:

1. open the HOUSE of the GATE
2. move through the HOUSE into the GATE
3. close the HOUSE of the GATE

What has gone wrong?

The associations are not what was intended. What was meant was to have the associations:

```
       HOUSE   is associated with CAR
       GATE    is associated with DOOR
```

In order to do this we would need the call:

```
         CALL ENTER(HOUSE,GATE)
```

The order of the ACTUAL parameters is VERY important.The subroutine cannot tell if the order is 'correct' or not, it merely sets up the associations according to the order in which the parameters appear.

12.2.4. Execution of the BODY of a subroutine

When a CALL is made to a subroutine the following events occur:

1. The association between ACTUAL and DUMMY parameters is made
2. The statements in the BODY of the subroutine are executed.

The subroutine must return control of execution back to the MAIN program that called it, so that the MAIN program may proceed with the lines of Fortran that follow the CALL.
This is done by executing a RETURN statement in the subroutine, causing the execution of the MAIN program to proceed from the line immediately following the CALL statement that caused us to get into the subroutine in the first place. There must be at least one executable RETURN statement in the subroutine. The need for conditional RETURN's may result in there being more than one RETURN statement in a subroutine. For example

```
      IF(NAME.EQ.LEIGH) RETURN
      READ(5,100) MESAGE
100   FORMAT(I5)
      WRITE(6,101)MESAGE
101   FORMAT(1X,' FOUND MESAGE OF ',I5)
      RETURN
```

This part of the subroutine body says to return to the calling program in one of two ways. If the value of NAME is equal to the value of LEIGH then return straight away. But if they are not equal, then read in a value for MESAGE and write it out before returning.

12.2.5. Execution paths using subroutines

The use of subroutines causes the execution path through the program to deviate at each CALL, to execute the body of the subroutine which causes a RETURN to the caller (eventually) which then proceeds on. This sequence of events may be shown by following the numbers on the arrows of the diagram below. (The arrows and numbers will be inserted in the lectures.)

```
   Main Program
  .....................          ..........................
  .                   .          . SUBROUTINE ADDER(....) .
  .                   .          .                        .
  .                   .          .                        .
  . CALL ADDER(....)  .          .                        .
  .                   .          .                        .
  .                   .          . RETURN                 .
  .                   .          . END                    .
  .                   .          ..........................
  .                   .
  .                   .
  .                   .          ..........................
  .                   .          . SUBROUTINE TOT(....)   .
  .                   .          .                        .
  . CALL TOT(....)    .          .                        .
  .                   .          .                        .
  .                   .          . RETURN                 .
  .                   .          . END                    .
  . STOP              .          ..........................
  . END               .
  .....................
```

12.2.6. Program structure

The subroutines are set up so that they follow the MAIN program. The main program and each of the subroutines will have their own END statements.

This structure is one way of organising the main program and subroutines and is not necessarily the way that it is done by all Fortrans, even on the same computer.

12.3. Example of a subroutine

Write a subroutine that will determine if a parcel may be posted, given the dimensions of the parcel and its weight.

The criteria for acceptance are

```
        weight < 10kg
        length + 2*(width+depth) < 100cm
```

We define the length as the longest side, depth as the smallest side, and width as the remaining side.

It should be up to the program to sort out which side is which, so that all that the program user has to do is to input the three dimensions in any order, and also the weight of course.

12.3.1. Choosing parameters

We need to have parameters to pass to the subroutine and also a parameter that will be returned from the subroutine, so that we will know if we can post the article or not. So we have the following:

Name	Type	Description
SIDE1	Real	One of the dimensions of the parcel
SIDE2	Real	One of the dimensions of the parcel
SIDE3	Real	One of the dimensions of the parcel
WEIGHT	Real	The weight of the parcel in kg.
POST	Logical	Will be set by the subroutine to .TRUE. if the parcel can be posted .FALSE. if the parcel cannot be posted

12.3.2. Subroutine header and call

The subroutine header line will be

```
      SUBROUTINE SEND (SIDE1,SIDE2,SIDE3,WEIGHT,POST)
      LOGICAL POST
```

Note that we must declare the type of POST explicitly. The corresponding actual parameter in the main program will have to be declared as a logical variable in the main program also.

The subroutine call will have 5 parameters, the first 3 will be REAL and represent the dimensions of the parcel, the next is REAL and represents the weight of the parcel, the last is LOGICAL and is an indicator that we can test to see if the parcel passed our tests.

So, an example of the subroutine use will be

```
      LOGICAL OK
      READ (5,*) S1, S2, S3, WEIGHT
      CALL SEND (S1,S2,S3,WEIGHT,OK)
      IF (.NOT. OK) WRITE(6,*)S1,S2,S3,WEIGHT,' IS NOT ACCEPTABLE'
```

12.3.3. Subroutine body

The subroutine has to first work out which dimension is which and then see if both criteria are satisfied.

```
      SUBROUTINE SEND (SIDE1,SIDE2,SIDE3,WEIGHT,POST)
      LOGICAL POST
C
C   WORK OUT WHICH SIDE IS WHICH,
C   LENGTH IS THE LONGEST
C   DEPTH IS THE SHORTEST
```

```
C   WIDTH IS THE MIDDLE ONE
C
       AL = MAX (SIDE1,SIDE2,SIDE3)
       AD = MIN (SIDE1,SIDE2,SIDE3)
       AW = SIDE1
       IF (SIDE2 .GT. SIDE1 .AND. SIDE2 .LT. SIDE3) AW = SIDE2
       IF (SIDE3 .GT. SIDE1 .AND. SIDE3 .LT. SIDE2) AW = SIDE3
       DIM = AL + 2.0*(AW + AD)
       POST = (WEIGHT .LT. 10.0) .AND. (DIM .LT. 100.0)
       RETURN
       END
```

12.3.4. Alternate approach

One disadvantage of the above subroutine is that we don't know exactly what is wrong with the rejected parcel. We could overcome this in a number of ways:

(a) Write out a message in the subroutine if something is wrong with the parcel, and still return the value of POST as above, so that we would know how to process that parcel further (if needed).

(b) Return one of a number of values (rather than just TRUE or FALSE) depending on what was wrong with the parcel. Then we'd have to test this value in the main program and write out an appropriate message.

As an exercise, re-write the subroutine adopting approach (a) above.

12.4. Simulation exercise

Appendix 4 contains a much fuller example of the stages in writing a subroutine. It is also a simulation exercise, where two walkers approach each other, one step at a time until they meet. The exercise is to find out where they meet, and how far each one walked. It is worth while working your way through this example.

12.5. Using SUBROUTINES

Subroutines may be referenced from other subroutines as well as from the main program. Care must be taken to ensure that a subroutine does not call itself, either directly or indirectly. An example of calling oneself indirectly would be

  Main calls A., A calls B., B calls C., then C calls A again.

The significant characteristic of a subroutine is that it is quite independent. It can be written in isolation from the main program. The main program need know nothing about how it works. It needs to know only its SPECIFICATION (i.e. WHAT it does and how to call it). This means that it is possible to set up LIBRARIES of subroutines, so that once a subroutine has been written for, say, solving linear simultaneous equations or calculating standard deviations or correlation coefficients, it may be made available to other interested programmers, to be used by them as larger building blocks, or simply to save them the trouble of writing their own versions.

12.6. Calculate the area of a triangle

Problem:

To calculate the area of a triangle given its sides, and also to determine if the triangle is reasonable. A reasonable triangle is one where the sum of any two sides is greater than the third. The steps involved in writing a subroutine to do this are:

(a) Choose a name for the subroutine, say, TRI, and decide on the DUMMY parameters. Here the dummy parameters will be:

A, B and C	the three sides
AREA	the calculated area
OK	a LOGICAL variable, set to TRUE if the triangle is reasonable and FALSE otherwise

So we arrive at the heading line:

```
      SUBROUTINE TRI ( A,B,C,AREA,OK )
```

(b) Next we write the body of the subroutine, to perform the desired operations on the dummy parameters.

The entire subprogram is then:

```
      SUBROUTINE TRI ( A,B,C,AREA,OK )
C
C    AUTHOR:   L. LANDAU
C    DATE:     SEPT 1977
C              MODIFIED BY L. LANDAU OCT 1979
C
C    PARAMETER DESCRIPTION
C    ---------------------
C
C     PARAMETERS WITH AN * NEXT TO THEM ARE ALTERED BY THIS
C     SUBROUTINE
C
C       A      ONE SIDE OF THE TRIANGLE              (REAL)
C       B      ONE SIDE OF THE TRIANGLE              (REAL)
C       C      ONE SIDE OF THE TRIANGLE              (REAL)
C  *    AREA   THE CALCULATED AREA OF THE TRIANGLE   (REAL)
C  *    OK     .TRUE. IF THE TRIANGLE IS REASONABLE  (LOGICAL)
C              .FALSE. IF THE TRIANGLE IS NOT REASONABLE
C
C     IF THE TRIANGLE IS NOT REASONABLE AN AREA OF 0.0 WILL
C     BE RETURNED
C
```

```
C    PURPOSE:
C
C       TO CALCULATE THE AREA OF A TRIANGLE GIVEN ITS SIDES
C       AND TO TEST TO SEE IF THE TRIANGLE IS REASONABLE BY
C       ENSURING THAT THE SUM OF ANY TWO SIDES EXCEEDS THE THIRD
C
      LOGICAL OK
      IF( A+B .LE. C) GO TO 5
      IF( A+C .LE. B) GO TO 5
      IF( B+C .LE. A) GO TO 5
C
C   TRIANGLE IS REASONABLE
C
      OK = .TRUE.
      S  = 0.5 * (A + B + C)
      AREA = SQRT( S*(S-A)*(S-B)*(S-C) )
      RETURN
C
C   TRIANGLE IS UNREASONABLE
C
 5    CONTINUE
      OK = .FALSE.
      AREA = 0.0
      RETURN
      END
```

This subroutine can now be used by any program to calculate the AREA of any given triangle specified by the program. So when writing such a program, the programmer need not worry about finding areas of triangles, but can simply write

```
     CALL TRI( SIDE1,SIDE2,BASE,AREA,LEGAL )
```

where SIDE1, SIDE2, BASE have been given values in the MAIN program at some point prior to the call, and LEGAL has been declared as type LOGICAL.

12.7. Matching ACTUAL and DUMMY parameters

It is important to note that there MUST be a one to one correspondence between ACTUAL parameters and DUMMY parameters. They must agree in three things:

1. NUMBER OF PARAMETERS.

There must be the same number of actual parameters as there are dummy parameters, for ANY call of the subroutine.

2. TYPE OF PARAMETERS.

There may be a mixture of REAL and INTEGER parameters, but the respective types of the actual parameters must agree with the respective types of the dummy parameters. For example, if the subroutine has a first line of:

```
        SUBROUTINE ZOT(I,ADDER,LAST,AVER,MANY,ZZ,CORR)
```

then the actual parameters would have to be of type INTEGER, REAL, INTEGER, REAL, INTEGER, REAL, REAL, respectively.

3. DIMENSIONALITY.

If an array name is used as a parameter, then it must be dimensioned BOTH in the calling program, and also in the subroutine.

12.8. Use of arrays as parameters: ADJUSTABLE DIMENSIONS

Problem:

To write a subroutine that will calculate the sum and average of the elements of a one dimensional array.

(a) First choose a name, say, CALSUM, and decide on the dummy parameters. Here they are the number of elements of the array, say N, which is an INTEGER. Also we need the name of the array, say A, which we will say is of type REAL, and then the sum, say SUM, which again is REAL, and lastly the average AVE which is also REAL. This produces the subroutine header line of:-

```
      SUBROUTINE CALSUM ( A,N,SUM,AVE )
```

(b) Second, write the body of the subroutine to perform the desired operations on the dummy parameters:

```
      SUBROUTINE CALSUM ( A,N,SUM,AVE )
C
C   AUTHOR:   L. LANDAU
C   DATE:     SEPT 1977
C   PURPOSE:
C      TO CALCULATE THE SUM AND THE AVERAGE OF THE FIRST N
C      ELEMENTS OF AN ARRAY
C
C   PARAMETER DESCRIPTION
C   ---------------------
C
C    ANY PARAMETERS WITH AN * TO THEIR LEFT WILL BE ALTERED
C    IN THE SUBROUTINE
C
C     A        THE NAME OF THE ARRAY             (REAL)
C     N        THE SIZE OF THE ARRAY 'A'         (INTEGER)
C  *  SUM      THE TOTAL OF 'N' ELTS OF 'A'      (REAL)
C     AVE      THE AVERAGE OF THE FIRST 'N'      (REAL)
C              ELEMENTS OF THE ARRAY 'A'
      DIMENSION A(N)
      SUM = 0.0
      DO 5 I = 1,N
         SUM = SUM + A(I)
 5    CONTINUE
      AVE = SUM/N
      RETURN
      END
```

The body of the subroutine is quite straight forward, except for the DIMENSION of A. Clearly we must DIMENSION it, as rule 3 above states that we must, but we are not sure of the dimensioned size of the corresponding actual parameter in the main program.

A special facility exists to cater for this, as shown above. We can make it an ADJUSTABLE ARRAY A(N). This says that the size of the array is unknown and may even vary from one call to the next. However at each call its size will be known, it will be given by N. As A is called an ADJUSTABLE ARRAY, so N is called an ADJUSTABLE DIMENSON.

BOTH ADJUSTABLE ARRAY AND ADJUSTABLE DIMENSION MUST BE DUMMY PARAMETERS.

12.9. Example of SUBROUTINE use

Problem:

We have a record and tape collection and wish to maintain a register of items, so that it may be easily updated, listed and maybe in the future, resorted. At the moment, we can only process numerical data, so that the record title etc will have to be coded. The data to be recorded may be such things (coded as integers) as:

- type of medium (record or tape)
- record/tape number
- location code (shelf number, borrowed, missing)
- music type
- condition of record/tape
- date obtained
- playing time (minutes and seconds)
- type of tape (low noise, chrome)
- Dolby indicator
- source of tape (pre-recorded, tapecopy)

The program should:

1. Read in all the data.
2. Edit the data, and print out any formatting errors.
3. List the data, printing 50 entries per line
   and a heading and page number on the top of each page.

The main program may look something like this:

```
C
      comments describing the codes used and input data etc.
C
C     SUBROUTINES USED
C
C     REED     THIS READS IN DATA FOR ONE TAPE OR RECORD AND RETURNS
C              INFORMATION IN AN ARRAY
C
C     EDIT     THIS CHECKS THE INFORMATION READ IN
C              AND RETURNS A PARAMETER TO INDICATE
```

```
C             IF THERE WAS AN ERROR.  IT IS
C             RESPONSIBLE FOR PRINTING ERROR
C             MESSAGES TOO.
C
C    HEAD     PRINTS A HEADING ON THE TOP OF A PAGE.
C             ALSO INCREMENTS A PAGE COUNTER, AND RESETS
C             THE LINE COUNT TO ZERO.
C
C    OUTPUT   WRITES OUT A LINE, INCREMENTS LINE COUNT
C
C
     DIMENSION MUSIC (10)
     LOGICAL OK
     LINCNT = 0
     NGOOD = 0
     IPAGE = 0
     CALL HEAD (IPAGE,LINCNT)
C
C    READ IN A MUSIC RECORD
C    END OF FILE IS INDICATED BY A
C    NEGATIVE TAPE/RECORD IDENTIFIER
C
  5  CONTINUE
     CALL REED(MUSIC,10)
     IF(MUSIC(1).LT.0)GO TO 99
C
C    CHECK VALIDITY OF THE ENTRY
C    AND COUNT THE NUMBER OF INVALIDS
C
     CALL EDIT(MUSIC,10,OK,LINCNT)
     IF( OK )NGOOD=NGOOD+1
     CALL OUTPUT(MUSIC,10,LINCNT)
     IF (LINCNT.GT.40)CALL HEAD (IPAGE,LINCNT)
     GO TO 5
C
C    END OF FILE FOUND
C
 99  CONTINUE
     WRITE (6,100) NGOOD
100  FORMAT('1','THERE WERE ',I3,' GOOD INPUTS')
     STOP
     END
```

Each subroutine would be described by a block of comments at its head saying:

(a) what parameters it uses
(b) what values are returned
(c) what the subroutine does

The main program does not do very much but call a number of subroutines, but it is easy to follow and provides an ideal starting point for understanding or changing the program.

12.10. Exercises

12.10.1. Exercise 12A

Given a SUBROUTINE line, and declaratives of:

```
    SUBROUTINE JEDDA(DG,MAX,LOG,STRAD)
    REAL LOG
    DIMENSION DG(MAX,MAX),LOG(MAX,10)
```

which of the following calls are legal, given the following declarations in the calling program.

```
    DIMENSION CT(44,44),SUNNY(44,44),KASH(10,10)

  (a) CALL JEDDA(CT,18,SUNNY,14.8)
  (b) ISZ = 30
      BROK= 98.4
      CALL JEDDA(SUNNY,ISZ+3,CT,BROK+ISZ)
  (c) CALL JEDDA(KASH,8,CT,9.1)
  (d) CALL JEDDA(CT,6,SUNNY)
  (e) CALL JEDDA(14,CT,SUNNY,18.9)
  (f) CALL JEDDA(CT(13,4),10,SUNNY(3,3),CT(2,2))
  (g) CALL JEDDA(CT,45,SUNNY,CT(2,3))
```

12.10.2. Exercise 12B

JA is a one dimensional array with 50 elements. Write a SUBROUTINE subprogram to compute the average of the first N elements and a count of the number of these elements that are zero. Call the subprogram AVERNZ(JA,N,AVER,NZ).

12.10.3. Exercise 12C

Write a SUBROUTINE that reads in a two dimensional array, using adjustable dimensions.

12.10.4. Exercise 12D

Write a SUBROUTINE which, given an angle in degrees, e.g. 37.278, calculates degrees, minutes, and seconds as integers. Take the seconds to the nearest integer. There are 60 seconds in a minute and 60 minutes in a degree.

12.10.5. Exercise 12E

Write a complete program, consisting of a main program and three SUBROUTINE subprograms, which will sort a list of numbers as follows. The number of numbers will be read by the main program which then calls the first SUBROUTINE to read the numbers to be sorted. The main program will then call the second SUBROUTINE to sort the numbers into descending order. The third SUBROUTINE will be called to print the numbers in their original order, and will then be called again to print out the numbers in their sorted order. All input should be in fixed field format and all output should be in a presentable form with suitable headings.

12.10.6. Exercise 12F

Set a variable to the value 1.0. Multiply the variable by 0.1 and then this result by 10.0 and set this result back in the variable. Repeat this multiplication procedure for a total of 10000 times and print out the final result. Why isn't it 1.0?

Be careful that your program is correct before you run it on the computer. Otherwise, you may attempt to print out 10000 lines of output, and then you'll be in trouble for wasting paper!

12.10.9. Exercise 12E.

Write a complete program, consisting of a main program and three SUBROUTINE subprograms, which will sort a list of numbers. The number of numbers will be read by the main program which then calls the first SUBROUTINE to read the numbers to be sorted. The main program will then call the second SUBROUTINE to sort the numbers into descending order. The third SUBROUTINE will be called to print the numbers in their original order, and will then be called again to print out the numbers in their sorted order. All input should be in fixed field format and all output should be in a presentable form with suitable headings.

12.10.10. Exercise 12F

Set a variable to the value 1.0. Multiply this variable by 10.0 and then divide this result by 10.0 and set this result back in the variable. Repeat this multiplication procedure for a total of 10000 times and print out the final results. Why isn't it 1.0?

Be careful that your program is correct before you run it, particularly the output. Otherwise, you may attempt to print out 10000 lines of output and then you could be in trouble for wasting paper!

13. CHAPTER 13

13.1. FUNCTIONS

Functions have already been introduced in a previous chapter. So far the functions that have been used have been standard Fortran functions such as MOD, ABS, SIN etc. Each function has a specific name, a list of actual parameters and a type of result (i.e. integer or real).
Functions are used directly within arithmetic expressions, unlike subroutines which require a CALL statement to invoke them.

13.2. Writing your own FUNCTIONS

A function is a type of subprogram and it is composed of a number of lines of Fortran (just as a subroutine is).
The form of a function is:

```
        FUNCTION  name  (dummy parameters)

        body of the function

        END
```

13.2.1. FUNCTION header

The first line is the function header, which identifies the lines that follow as being a FUNCTION. The function header is composed of

(a) name
  This is the name of the function (some standard Fortran function names are MOD, ABS etc) and it follows the rules for variable names. The TYPE of the name (i.e. integer or real) identifies the TYPE of result that the function will return. So a function whose name is of type integer will return an integer result, similarly a real name would mean that the function will return a real result. The result of a function is that value which is assigned to the name of the function from within the body of the function. That is, the function name is treated as a variable, which is assigned a value within the function, and this is the value which is returned to the calling program.

(b) dummy parameters
  These are the same as specified in subroutines. When the function is used, it will have corresponding actual parameters. If there are no parameters then the brackets may be omitted from the function header, but they should be

included in the function reference (so that the compiler can distinguish it from a variable reference).

13.2.2. Last line of a FUNCTION specification

The last line of function must be

```
END
```

as is the case with a subroutine.

13.2.3. Body of a FUNCTION

The body of a function contains

(a) At least one RETURN statement.
(b) At least one assignment of a value to the name of the function.

The RETURN statement is used in the same way as it is in a subroutine: it specifies that execution of the calling program is to continue on from the point at which the function was referenced in the calling program (the calling program may be a MAIN program, a subroutine or a function).

The assignment of a value to the name is done so that a value may be returned from the function. This assignment must be by either

(a) The function name appearing on the left hand side of an assignment statement

(b) The function name appearing in a READ statement

13.3. Example of a FUNCTION

Suppose that we wanted to find the average of the first N numbers in a real array. The FUNCTION AVER will return this answer. This average is used to write out a message. If the average is greater than 50.0 then write out that the average is good; otherwise write out that the average is low.

```
DIMENSION   VALUES (100)

1. read in values into this array
2. read in a value for N
3. test to ensure that N is less than or equal to 100
```

```
      IF (AVER(VALUES,N) .GT. 50) THEN
          WRITE (6,10)
      ELSE
          WRITE (6,20)
      END IF
10    FORMAT(1X,'THE AVERAGE IS GOOD')
20    FORMAT(1X,'THE AVERAGE IS LOW ')
      STOP
      END

      FUNCTION AVER(ARR,NUM)
      DIMENSION ARR(NUM)
C
C  CALCULATE THE AVERAGE OF THE FIRST NUM ELEMENTS
C  OF THE ARRAY ARR
      TOTAL = 0.0
      DO 12 I = 1,NUM
         TOTAL = TOTAL+ARR(I)
12    CONTINUE
      AVER = TOTAL/NUM
      RETURN
      END
```

13.4. Explicit Type Declarations for Functions

The default type of the result of a function is given by the type of its name. Just as the default type of a variable name can be overridden by an explicit TYPE statement (i.e. INTEGER or REAL) so we may do the same with a function, by specifying its explicit type on the function header line as

```
      type FUNCTION name (dummy parameters)
```

For example

```
       REAL FUNCTION NS(A,I)
```

would return a REAL number even though the name NS is an integer name. It is as if the name of the function (in this case NS) appears in a type statement. The function type must be the same in the calling program and the function, so if this form of typing of a function is used then the function name MUST also appear in a type statement in each program (or subprogram) where it is referenced. In the above example, the calling program would need the type statement

```
              REAL NS
```

13.4.1. Example of a LOGICAL function

It is very useful to have type LOGICAL functions, but you MUST ensure that you explicitly TYPE the function name in the program (or subprogram) that references the function. For example, suppose that we had a program that read in an array of numbers that represented the salaries of people. Further suppose that these salaries are sorted into ascending order, but just to ensure this we have a LOGICAL function called EDIT to test for this, and also to test that no salary is less than 100.0 nor greater than 100,000.0

The data description is:

Salaries appear one per line, and the last one is a negative salary. The values are entered in columns 1-10.

```
C
C   AUTHOR:   L. LANDAU
C   DATE:     OCTOBER 1979
C   INPUT DESCRIPTION:
C        SALARIES ARE ENTERED ONE PER LINE AS REAL NUMBERS, IN
C        COLUMNS 1-10.
C
C        END OF DATA IS SIGNALLED BY A SALARY LESS THAN ZERO
C
C   SUBPROGRAMS USED:
C
C       EDIT     A LOGICAL FUNCTION THAT TESTS THE VALIDITY
C                OF THE DATA
C
      LOGICAL EDIT
      DIMENSION SALARY(1001)
      MAXNUM = 1000
      IROW = 1
C
C   READ IN SALARIES
C
 10   CONTINUE
      READ(5,20) SALARY(IROW)
 20   FORMAT(F10.0)
      IF (SALARY(IROW) .LT. 0.0) GO TO 40
      IROW = IROW + 1
      IF (IROW .LE. MAXNUM+1) GO TO 10
      WRITE(6,30)MAXNUM
 30   FORMAT(1X,'TOO MANY SALARIES, CAN ONLY HANDLE ',I4)
      STOP
C
C   COME HERE WHEN END OF DATA IS FOUND
C
 40   CONTINUE
      IROW = IROW - 1
      IF (EDIT(SALARY,IROW)) GO TO 60
      WRITE(6,50)
 50   FORMAT(1X,'NOT SORTED SALARIES OR OUT OF RANGE')
      STOP
 60   CONTINUE

the rest of the main program would come here

      END
      LOGICAL FUNCTION EDIT(SALARY,N)
C
C   AUTHOR:  L. LANDAU
C   DATE:    OCT 1979
C   PURPOSE:
C     TO CHECK THAT THE ARRAY IS SORTED INTO ASCENDING ORDER AND
C     THAT ALL THE NUMBERS LIE IN THE RANGE OF 100.0 TO 100,000.0
C
C   PARAMETER DESCRIPTION:
C
C       SALARY      ....  A REAL ARRAY OF SALARIES
C       N           ....  THE NUMBER OF SALARIES IN THE ARRAY
C
C   VALUE RETURNED BY THE FUNCTION:
C
C      IF THE DATA IS OK  THEN RETURN .TRUE.
C      IF THE DATA IS NBG THEN RETURN .FALSE.
C
      DIMENION SALARY(N)
      IF( SALARY(1).LT.100.0 )    GO TO 90
      IF( SALARY(N).GT.100 000.0) GO TO 90
      DO 5 I = 2,N
      IF( SALARY(I-1).GT.SALARY(I) ) GO TO 90
 5    CONTINUE
C
C    ALL OK SO RETURN TRUE
C
      EDIT = .TRUE.
      RETURN
C
C   COME HERE IF ERRORS FOUND
C
 90   CONTINUE
      EDIT = .FALSE.
      RETURN
      END
```

13.4.2. Writing FUNCTIONS

Using functions will pose no problems, as we have been using the basic ones all along. Any problems that arise will do so in the writing of the functions themselves. Clearly, we do not first write a subroutine and then convert it to a function, as we did above: we write it directly. Let us see this by writing a function to calculate the area of a triangle from its sides. The technique used in writing the function is the same as for the subroutine:

(a) Choose a NAME and TYPE for the function (say REAL FUNCTION AREA) and decide on its dummy parameters (the REAL quantities A, B, and C representing the sides). Thus we arrive at the heading:

```
      REAL FUNCTION AREA(A, B, C)
```

(b) Write the body to calculate the appropriate value from the dummy parameters, and assign that value to the name of the function. This is quite simply done here by calculating S, and then AREA and then returning.

```
      REAL FUNCTION AREA(A, B, C)
      S = 0.5*(A + B + C)
      AREA = SQRT( S*(S-A)*(S-B)*(S-C) )
      RETURN
      END
```

13.5. Why use subprograms?

Subroutines and functions may be used as self-contained building blocks to write a program. If a problem can be decomposed into sub-problems that may easily be solved, so a Fortran program may be designed in a similar way. This is the divide and conquer strategy of solving problems. Further, since a subprogram may exist in isolation from a main program, it may be tested independently also, and when it has been proved to work, it may be combined with other routines in the overall programming system.

If a subprogram is general, it may be used in a variety of situations, thus saving much repetitive effort on the part of the programmer.

Programs that use subprograms are easier to follow, and easier to maintain, two of the desirable goals that make a program 'better'.

Most computer installations maintain a library of subprograms which may be used to solve common problems such as sorting, solving sets of linear equations, returning the date and time, statistical analysis subroutines, and many more.

13.5.1. Common errors and points to note

There must be the same number of parameters in the use of a subprogram as there are in the definition.

There must be a one to one correspondence between variable types in both parameter lists i.e. if a parameter is an integer in the definition, then an integer must be supplied as the actual parameter.

If an array name is used as a parameter, then it must be dimensioned in the calling program and the corresponding parameter in the definition must be dimensioned also. Both of these dimensions must be identical, unless adjustable dimensions are being used. In this case, the array must be defined in the calling program with a fixed dimension, and the values of the DIMENSION statement must be supplied as parameters to the subprogram. Only arrays in the parameter list can have adjustable dimensions.

The name of a FUNCTION subprogram must appear at least once in the definition subprogram on the left hand side of an assignment statement or in a READ statement.

The name of a SUBROUTINE subprogram must not appear in any statement in the defined subprogram, except as the name of the SUBROUTINE in the SUBROUTINE statement itself.

If the definition of a subprogram changes the value of a variable in the parameter list, then a constant cannot be used as the corresponding actual parameter, or the value of the constant may be changed.

A RETURN statement or a logical IF statement containing a RETURN statement must not be the terminal statement of a DO loop.

A CALL statement or a logical IF statement containing a CALL statement must not be the terminal statement of a DO loop.

13.6. Exercises

13.6.1. Exercise 13A

Write a REAL function to calculate the area of a circle of radius r.

$$\text{area} = 3.14159265r^2$$

13.6.2. Exercise 13B

Define an REAL function to compute

$$f(x) = x^2 + (1 + 2x + 3x^2)^{1/2}$$

Then use the function to compute

$$a = \frac{6.9 + y}{y^2 + (1 + 2y + 3y^2)^{1/2}}$$

$$b = \frac{2.1z + z^4}{z^2 + (1 + 2z + 3z^2)^{1/2}}$$

$$c = \frac{\sin y}{y^4 + (1 + 2y^2 + 3y^4)^{1/2}}$$

$$d = \frac{1}{\sin^2 y + (1 + 2\sin y + 3\sin^2 y)^{1/2}}$$

13.6.3. Exercise 13C

Write a FUNCTION subprogram, with two parameters r and p, that calculates the area of

an equilateral triangle of side r when p = 1,

a square of side r when p = 2,

a circle of radius r when p = 3.

13.6.4. Exercise 13D

Write a FUNCTION subprogram for which the parameter list contains A, M, and N, where A is an array name, and M and N are the numbers of rows and columns respectively. The function value is to be the sum of the absolute values of all the elements in the array. The dimensions are to be adjustable.

13.6.5. Exercise 13E

Write a FUNCTION subprogram that searches a one dimensional array and returns the largest value.

13.6.6. Exercise 13F

Write a function that will count the number of zeros in a two dimensional integer array.

14. CHAPTER 14

14.1. Character manipulation

1977 standard Fortran introduced major changes in the way characters (text) are handled. 1966 standard Fortran had no special data type for character handling and no character handling operations possible (such as concatenation and substringing). Because of these major changes, both the 1966 and 1977 versions of character handling are iscluded.

The 1966 standards are presented in Appendix 5.

14.2. Declaration

In 1977 a new data type was introduced, called type CHARACTER, which is used to store a number of characters. On the Univac the limit is 511. The form of declaration of character variables is

```
        CHARACTER*1  variable list
```

The 1 indicates the length, or number of characters that may be stored in the variables on the list. In addition to this, variables on the list may be followed by an asterisk and a length specification.

For example:

```
 1      CHARACTER*20 NAME, ADDR
 2      CHARACTER NAME*20, ADDR*20
 3      CHARACTER*5 AXLE*10, MINE, ROB*100
```

Line 1 indicates that the two variables NAME and ADDR are character variables, and can each store 20 characters.

Line 2 has the same effect as line 1, but does it by including individual length specifications.

Line 3 declares that any variables that do not have their own length specifications will be able to store 5 characters each. So, AXLE can store 10 characters, MINE can store 5 and ROB can store 100.

14.3. Character arrays

Character arrays are similar to integer or real arrays except that they contain a number of characters in each element of the array.

Character arrays can be dimensioned in a dimension statement or within the character declaration. For example, the following declarations are identical, and they each declare TRIAL to be a character array of size 100 and X to be a character array of size 10. Each element of TRIAL can store 48 characters, and each element of X can store 2 characters.

```
        CHARACTER      TRIAL*48,X*2
        DIMENSION      TRIAL(100),X(10)

        CHARACTER*48   TRIAL(100),X(10)*2

        CHARACTER      TRIAL*48(100),X*2(10)
```

14.4. Character constants

A character constant, sometimes called a literal or text constant, is any string of characters enclosed within single quotes. If a single quote appears within the character constant, then it must be immediately followed by another single quote. So far, we have been using character constants in headings in WRITE statements. Now they can also appear within character expressions, analagous to integers and reals appearing within arithmetic expressions.

If a character constant appears on the right hand side of an assignment statement, then its value is placed in the character variable on the left, so that the constant is truncated if it is too long, or is placed left justifed in the variable, with blanks filling the rightmost characters if the constant is too short.

For example:

```
        CHARACTER*8 MAN, FUSS*12
        MAN  = 'ABCD'
        FUSS = 'THE BOY''S JOB IS DIFFICULT'
```

```
The value stored in MAN  is:  ABCDbbbb
The value stored in FUSS is:  THE BOY'S JO
```

A b indicates a blank.

14.5. Substrings

Reference is made to substrings by

```
        VAR(e1:e2)    or     ARRAY(subscripts)(e1:e2)
```

where VAR is a character variable, ARRAY(subscripts) is an element of an array of type character, and e1 and e2 are integer expressions.

The value of e1 specifies the leftmost character position of the substring.
The value of e2 specifies the rightmost character position of the substring.

If len is the length of the character variable, then

```
        1<= e1 <= e2 <= len
```

where <= means less than or equal to.

If e1 is omitted, a value of 1 is implied.
If e2 is omitted, a value of len is implied.

14.6. Examples of substrings

```
(a)     CHARACTER*20 STATE
        CHARACTER*10 NAME
        STATE = 'WESTERN AUSTRALIA'
        NAME  = STATE (9:17)
        STATE(1:7) = STATE (9:15)
```

After this,

```
NAME is AUSTRALIA
STATE is AUSTRAL AUSTRALIA
```

(b) Assuming the original declarations above,

```
        STATE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        NAME  = STATE(5:20)
```

After this the values are

```
STATE ........ ABCDEFGHIJKLMNOPQRST
NAME  ........ EFGHIJKLMN
```

(c) Find all the blanks in a character string LINE:

```
        CHARACTER*80 LINE
        READ (5,*) LINE
        DO 10 I = 1,80
           IF (LINE(I:I) .EQ. ' ') WRITE(6,*)'BLANK IN POSITION ',I
   10   CONTINUE
```

```
(d)     CHARACTER*4 C(2,2),C1
        C1 = 'abcd'
        C(1,1)(3:4) = C1(1:2)
        C(1,1)(1:2) = C1(3:4)
```

puts 'cdab' into C(1,1).

14.7. Reading and writing characters

14.7.1. Free format

Character variables may be read and written using free format. In writing, the resultant output takes as many columns as the size of the character variable.

In reading, the data is presented as a character constant, namely, a string of characters enclosed in single quotes.

14.7.2. Fixed format

Input and output of characters is handled by the A field descriptor, which has the form

Aw

where w indicates the width of the input or output field.

If the w is omitted then the declared length of the character variable (that is being read/written) is assumed.

If w is specified and is different to the length of the I/O item then

(i) On input,

(a) If w < length then w characters are placed left justified with blank fill, in the I/O list item.
(b) If w > length then the rightmost 'length' characters are taken from the input field.

(ii) On output,

(a) If w < length then the leftmost w characters are output

(b) If w > length then 'length' characters are written, right justified in a field of length w.

14.7.3. Example of reading and writing

Consider the following data and program

```
      CHARACTER*10 CODE, LOCAT
      CHARACTER*4  MED, LOST
      READ (5,10) CODE, LOCAT, MED, LOST
10    FORMAT (A5,A12,A3,A)
      WRITE (6,20) CODE, LOCAT, MED, LOST
20    FORMAT (1X,A,A,2A6)
```

The input data is

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The result is

Variable	Value
CODE	ABCDEbbbbb
LOCAT	HIJKLMNOPQ
MED	RSTb
LOST	UVWX

and the output is

```
ABCDEbbbbbHIJKLMNOPQbbRSTbbbUVWX
```

14.8. Character operators

The only character operator is the concatenation operator (//)

```
        expr1 // expr2
```

where expr1 and expr2 are character expressions. The value of the above expression is a character value that is the first expression immediately followed by the second.

For example

(a)
```
        CHARACTER A*4,B*8
        A = 'abcd'
        B = A // 'efgh'
```

results in B containing 'abcdefgh'

(b)
```
        CHARACTER*25 T1, T2, T3*15
        T1 = 'GONE SHOPPING'
        T2 = 'BLUE JUMPERS ARE IN THERE'
        T3 = T1(1:2) // ' ' // T2(6:9) // T2(17:23) // ' LAKE'
```

What is stored in T3?

See Hierarchy of Operators (Appendix 3).

14.9. Comparing character expressions

This is done with a logical IF statement where both sides of the relational expression are character expressions.

1977 Fortran, as well as introducing character variables, made it illegal to compare a character expression (for example something enclosed in quotes) with anything other than another character expression. This may cause some problems to people who have 1966 version programs that do character comparisons as there will be comparisons between integers or reals with character expressions (because in 1966 Fortran characters were stored in integer or real variables).

Greater than and less than operators are satisfied by an Ascii collating sequence (see Appendix 3 for Ascii codes).

If two character expressions of unequal length are compared, then the shorter one is considered to be extended by blanks.

14.10. Supplied Functions

Ascii Fortran supplies functions to operate on character variables, array elements and expressions.
The following abbreviations are used:

I	=	integer
C	=	character
L	=	logical
param	=	parameter

Func name	No. of params	Type of param	Type of result	Description
ICHAR	1	C*1	I	Position of the param in the table of ASCII codes (Appendix 3), starting at position 0.
CHAR	1	I	C*1	Character in the ASCII code table indexed by the parameter.
LEN	1	C	I	Length of param, ie. number of characters.
INDEX	2	C	I	Starting position of param 2 within param 1 (= 0 if string not found).
LGE	2	C	L	.TRUE. if param 1 is lexically greater than or equal to param 2 , otherwise .FALSE.
LGT	2	C	L	Lexically greater than.
LLE	2	C	L	Lexically less than or equal.
LLT	2	C	L	Lexically less than.

14.11. Functions and subroutines

A character function is a function whose type is character (the value returned by the function is a character string). The function is declared to be of type character, and the function name must also be declared in the calling program. The form is

```
        CHARACTER*length FUNCTION name (dummy parameters)
```

14.11.1. Passing character parameters

Passing characters can be a painful process, because you need to know the length of the character variable in order to declare it within the function or subroutine. Fortran has a mechanism for allowing an 'adjustable' character length for parameters, so that the length used will be the length of the actual parameter for each subprogram reference. This is done by declaring (in the subprogram) the dummy parameter as

```
        CHARACTER*(*) list of dummy parameters
```

For example

```
        SUBROUTINE X (LINE, WORD)
        CHARACTER*(*) LINE, WORD(10)
```

declares LINE as a character variable and WORD as a character array.

14.12. Sample program

The following program will read a list of names and addresses and print them out in a presentable form. The name on the input line occupies the first 36 columns and the address takes columns 37 to 80 inclusive. The end of the data is indicated by a blank name field.

```
C     AUTHOR:  L. LANDAU
C     DATE:    OCTOBER 1979
C     MODS:    JANUARY 1981 to update to 1977 Fortran
C              Done by Les Landau
C
C     INPUT DESCRIPTION:
C     -----------------
C
C          EACH LINE CONTAINS NAMES AND ADDRESSES. THE LAST LINE
C          CONTAINS BLANKS IN THE NAME FIELD.
C
C          COLUMNS        MEANING
C           1 - 36        NAME OF PERSON
C          37 - 80        ADDRESS OF PERSON
C
C     PURPOSE:
C     --------
C
C          TO READ IN AND LIST PEOPLES NAMES AND ADDRESSES
C          HAVING 50 PER PAGE, WITH PAGE COUNTS AT THE TOP
C
C
      CHARACTER*44 ADDR
      CHARACTER*36 NAME
C
C
      IPAGE = 1
      LINCNT = 0
C
C     WRITE OUT A HEADING
C
      WRITE(6,10) IPAGE
 10   FORMAT('1NAME',46X,'ADDRESS',20X,'PAGE: ',I4/
     $       1X,'----',46X,'-------'//)
C
C     READ THE NEXT NAME AND ADDRESS
C
 20   CONTINUE
      READ (5,30) NAME, ADDR
 30   FORMAT(A36,A44)
C
C     TEST FOR END OF DATA
C
      IF (NAME .EQ. ' ') STOP
C
C     CHECK IF A HEADING IS REQUIRED, THEN WRITE OUT NAME
C     AND ADDRESS AND THEN GO BACK FOR MORE
C
      IF (MOD(LINCNT,50) .NE. 0) GO TO 40
      IPAGE = IPAGE + 1
      WRITE (6,10) IPAGE
 40   CONTINUE
      WRITE (6,50) NAME, ADDR
      LINCNT = LINCNT + 1
 50   FORMAT(1X,A,14X,A)
      GO TO 20
      END
```

As an exercise suggest how you could modify this program to write out the address immediately following the name, ignoring the trailing blanks in the name field. So the output would be

```
 NIT ALLEN, 23 WALDORF GRADE HOLLYOAK DRAIN
```

instead of

```
 NIT ALLEN                                    23 WALDORF GRADE HOLLYOAK DRAIN
```

14.13. Exercises

14.13.1. Exercise 14A

Write a program that will read in a line of text, and then will write out

```
  HAPPY BIRTHDAY text
```

Input is terminated by an end of file. Use the program to print out

```
 HAPPY BIRTHDAY to you
 HAPPY BIRTHDAY to you
 HAPPY BIRTHDAY dear Erin
 HAPPY BIRTHDAY to you
```

14.13.2. Exercise 14B

Write a subroutine that will return the first word in a line of text that is passed to it as a parameter. A word may be delimited by a blank, a comma or a full stop.

You can assume that the line is a maximum of 80 characters. If no delimiter is on the line, then just return the value of the line itself.

The answers present a more general version of this subroutine, and its calling program. When you have answered the question, look carefully at the presented answer.

14.13.3. Exercise 14C

Given an integer variable I, with I not less than 1 and not greater than 12, set up a program that will print in three printing positions one of the abbreviations JAN, FEB, MAR, APR, etc., depending on the value of I. This program can and should be done without the use of GO TO statements.

14.13.4. Exercise 14D

Given an integer variable J with J not less than 1 and not greater than 7, set up a program to print one of the words MONDAY, TUESDAY, etc., depending on the value of J.

14.13.5. Exercise 14E

A company manufactures n products where n is not greater than 50. Each product has a 5-character code and a 20-character description. Write the following main program and SUBROUTINE.

The main program reads the product information into two one-dimensional arrays of length 50, with codes in the first array and the corresponding descriptions in the second array. It then should read in a product code and call the SUBROUTINE which searches the first array, finds the product, and prints out the code and description. The main program should be capable of reading any number of product codes.

14.13.6. Exercise 14F

Write a program to read an integer which represents the day in the year 1981. Print out the the corresponding date in the form

day of week   day of month   month   year

For example, if the input were

    328

the output should be

```
TUESDAY   24TH NOVEMBER  1981
```

14.13.7. Exercise 14G

Write a program that will read with the A field descriptor a line of data containing integers in free field format. The data may contain the ten decimal digits, plus and minus signs, and commas. The program is to convert the characters into the corresponding integer numbers, and print out the original data and the computed integer values.

15. APPENDIX 1 - Notes on doing assignments

The assignments are designed to try to teach you some aspects of Fortran programming. Although it is important for your program to produce the correct answer, we are not interested in JUST this. We already know the answers, and know how to do the problems! It is HOW you arrive at the answers (i.e. the structure of the program, and to some extent, the method used in the program) that is important. The notes below are intended to let you know the kinds of things that will be looked for in marking your assignments. The comments below apply only to those questions that require the writing of a Fortran program.

1. The program must correctly solve the question asked.
   You should check your results. Some results are more easily checked than others. You should look very carefully at your answers and satisfy yourself that they are correct. If there is insufficient information printed out then you should change your program to print out sufficient information, before handing the assignment in.

2. The program should be written in such a way that it can be easily understood by you in a year's time, and also by anyone else (who has not read the assignment sheet) who understands FORTRAN, most importantly the person marking your assignment!

   This may be achieved by:

   (a) Good program design
   (b) Choice of meaningful variable names
   (c) Well structured COMMENTs

3. At the beginning of the program you should use comments to explain:

   (a) What the program is designed to do.

   (b) Who wrote it.

   (c) When it was written.

   (d) What general method the program employs.

   (e) How the data is designed (if any), and how to use the program.

   (f) Any limitations the program has.

In order to help you to remember to put in this information you should include the following titles within your comments, even if they do not seem to apply:

```
C       AUTHOR:
C       DATE:
C       INPUT DESCRIPTION:
C       PURPOSE:
```

All assignments handed in MUST have those titles, at least, or they will be rejected. Of course you must also fill out the titles with appropriate information.

Whenever helpful or necessary throughout the program you should use comments to explain:

(a) What a variable is used to represent (unless obvious).

(b) Why a particular operation is being performed.

(c) Any special 'tricks' you use.

4. Headings on output
All the output from the program should be clearly headed so that even if you didn't have a program listing to accompany the output, it would be clear from your headings what the results are and how they are to be interpreted.

5. Program Generality
There are various degrees of program generality, and a balance needs to be found. Minimally, the program should be able to be run again, using different (although the same quantity) of data, WITHOUT ANY CHANGE TO THE PROGRAM WHATSOEVER! Sometimes program changes may be necessary, in order that the program can run with an increased amount, or different type of data. In this case, the fewer changes required (generally) the more general the program.

Remember that the suggested data for any program is only sample, and the program must be able to handle other data sets as well.

6. Elimination of unnecessary program statements.

This is NOT to be taken to an extreme! A program with fewer statements may in fact be a much worse program both in the sense of understandability and efficency.

An example of what will be looked for in this area is:

We shall see how it is possible to repeat a series of instructions in Fortran by the use of a DO statement (as one means). Another means is to write out the statements (explicitly) the number of times that you wish them to be repeated. The latter is far worse.

If you have any queries as to what is expected in assignments, ask about it in tutorials, and the point will be clarified.

In spite of what you may think, the assignments are not meant to TEST you, but to help you LEARN.

16. APPENDIX 2 - Control commands required by the UNIVAC

Computers use many programs and compilers other than the Fortran compiler so it is necessary to indicate which compilers are to be used and where the data is. These commands to the computer as known as CONTROL STATEMENTS. These were introduced and explained in Chapter 1.

The control statements required to run FORTRAN exercises are as follows. Please ensure that you type them EXACTLY as they appear below, with no extra or fewer blanks than is indicated.

16.1. Log-on procedure

Log onto a Uniscope terminal by typing its site-id, or into a network terminal by typing CONTROL-V U, and then enter the userid/password that has been given to you. Enter a runid, up to 6 characters long, beginning with your three initials. Line printer output will be filed under the third character of your runid, unless you are using the bag service.

16.2. File creation

Use only one file to contain your programs and also your data. The filename that you use could be anything, for example, PROGS. To create the file use:

```
        @CAT,P  PROGS.
```

16.3. Element creation for programs

You will use the editor (@ED) to create your program, choosing element names that reflect your assignment. For example to create the program for assignment 2 question 3b

```
        @ED,IQ  PROGS.A2Q3B
        ...
        ...   type in the text of your program
        ...
        EXIT
```

16.4. Element creation for data

You can type data directly at the terminal, but you should always create an element to put the data in, and then use the @ADD command to introduce your data to the program. This means that you only have to type the data in once, and errors can be corrected. Some consistent element naming is required, and one suggestion is to use version names. So, the element that contains the data for assignment 2 question 3b would be created by

```
        @ED,IQ PROGS.A2Q3B/DATA
        ...
        ...  enter your data
        ...
        EXIT
```

16.5. Program execution

Assuming your program is in the element

```
    PROG.A1Q2
```

then the following commands will compile and run the program:

```
    @FTN,CS PROG.A1Q2
    @EOF
```

You will then get a message appearing at the terminal saying:

```
    ENTERING USER PROGRAM
```

At this point, you should enter your data, by using an @ADD command. So, leaving out the computer's responses you would have

```
        @FTN,CS PROGS.A1Q2
        @EOF
        @ADD PROGS.A1Q2/DATA
```

The S option on the @FTN command will cause a program listing to appear at the terminal. The C option is to run the program after compiling it.

16.6. Obtaining a program listing

To obtain a listing of your program (without doing an execution) do

```
        @SUSPEND
        @FTN,S  PROGS.A2Q3B
        @EOF
        @RESUME,E

            now examine the listing of the program with
            @ED commands, to see if you want to print it.
            If you do want to print it then:

        @RESUME PR      will print it and delete the printfile afterwards
        or
        @RESUME,D       will delete the printfile
```

To obtain a listing of your program complete with an execution (for handing in) do

```
        @SUSPEND
        @FTN,CS PROGS.A1Q3B
        @EOF
        @ADD PROGS.A1Q3B/DATA
        @RESUME,E
```

Then proceed as above to examine and print or delete the printfile.

17. APPENDIX 3 - Ascii Codes, and Hierarchy of Operators

17.1. ASCII Codes and Symbols

A S C I I   C O D E S   A N D   S Y M B O L S
(all codes are expressed in OCTAL)

CODE	SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE	SYMBOL
000		040		100	@	140	`
001		041	!	101	A	141	a
002		042	"	102	B	142	b
003		043	#	103	C	143	c
004		044	$	104	D	144	d
005		045	%	105	E	145	e
006		046	&	106	F	146	f
007		047	'	107	G	147	g
010		050	(	110	H	150	h
011		051	)	111	I	151	i
012		052	*	112	J	152	j
013		053	+	113	K	153	k
014		054	,	114	L	154	l
015		055	-	115	M	155	m
016		056	.	116	N	156	n
017		057	/	117	O	157	o
020		060	0	120	P	160	p
021		061	1	121	Q	161	q
022		062	2	122	R	162	r
023		063	3	123	S	163	s
024		064	4	124	T	164	t
025		065	5	125	U	165	u
026		066	6	126	V	166	v
027		067	7	127	W	167	w
030		070	8	130	X	170	x
031		071	9	131	Y	171	y
032		072	:	132	Z	172	z
033		073	;	133	[	173	{
034		074	<	134	\	174	¦
035		075	=	135	]	175	}
036		076	>	136	^	176	~
037		077	?	137	_	177	

000-037 and 177 are control characters (e.g. carriage return, delete). 040 is the code for a blank.

17.2. Hierarchy of Operators

The following hierarchy is used to determine the order of evaluation of expressions :

Rank	Kind	Operation
1	expressions in parentheses	all
2	functions	all
3	arithmetic	** (exponentiation)
4		*,/ (multiplication, division)
5		+,- (addition, subtraction)
6	character	// (concatenation)
7	logical	.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
8		.NOT.
9		.AND.
10		.OR.
11		.EQV.,.NEQV.

18. APPENDIX 4 - Simulation of two walkers

Suppose we wish to simulate two people walking toward one another and will use a main program to provide us with the input data and results listing - and a subroutine that will be used for each of the two walkers in order that they can take a step.

Two people (let's call them Dale and Erin) leave their respective homes and walk (in a straight line) towards each other. How long will it take before they meet? They each take a step alternately and if one walker is within (their) stepsize of the other, the two are said to have met.

Each person's step size is different and is calculated in the following way.

1. Every 100th step the person must rest (i.e. no distance is travelled in that step.
2. The length of one step is determined by multiplying an initial stepsize by a fitness factor for that person.
3. If the people are within 200 steps of each other, this excitement enables them to take a 10 percent longer step than usual.

18.1.1. Input Data

The input data for each person is described below.

Columns	Type	Meaning
1-10	Real	The distance the home is from an origin point.
11-15	Real	The initial step size (metres)
16-20	Real	The percentage fitness factor.

18.1.2. Diagram

The problem may be represented below, with simple data values shown.

```
   o      0->->->->->->->->->->->X<-<-<-<-<-<-<-<-0
origin    Dale               Meeting              Erin
                             Point

   Distance = 44 metres                 Distance = 2347 metres
   Stepsize = 1.0 metres                Stepsize = 1.2 metres
   Fitness  = 0.9                       Fitness  = 0.8
```

Using this data, the two will meet, with Dale walking 1,114.9 metres taking 1243 steps and Erin walking 1,191.1 metres taking 1243 steps.

18.1.3. Algorithm outline

```
1. Read in data for Dale
2. Read in data for Erin
3. Repeat the following until they have met

    3.1 Dale takes a step
        3.1.1 If the number of steps is exactly divisible by 100 then no
              distance is to be travelled in this step.
        3.1.2 If Dale is within 200 steps of Erin then the stepsize is to
              be increased by 10 percent.
        3.1.3 Calculate Dale's new position
        3.1.4 Add to the number of steps made by Dale so far.

    3.2 Erin takes a step
        3.2.1 If the number of steps is exactly divisible by 100 then no
              distance is to be travelled in this step.
        3.2.2 If Erin is within 200 steps of Dale then the stepsize is to
              be increased by 10 percent.
        3.2.3 Calculate Erin's new position
        3.2.4 Add to the number of steps made by Erin so far.

4. Report the number of steps and distance travelled.
```

18.1.4. Taking a step

Since each person must take a step, let us write some Fortran which will

(a) Calculate the step size
(b) Take the step, which alters the distance from the origin and number of steps taken.

In order to do this, the following quantities are required.

(a) current distance from the origin of the walker	WALKD
(b) current distance from the origin of the other person	OTHERD
(c) the stepsize (as altered by the fitness factor)	SIZE
(d) the number of steps taken so far	NSTEPS
(e) the direction of walking (positive for Dale, negative for Erin).	DIRECT

The Fortran to do this is then

```
C
C   INITIALLY THE STEPSIZE IS THE GIVEN ONE
C
      STEP = SIZE
C
C   IF THE NUMBER OF STEPS IS EXACTLY DIVISIBLE BY 100 THEN
C   NO DISTANCE IS TO BE TRAVELLED
C
      IF (MOD(NSTEPS,100).EQ.0)GO TO 10
```

```
C
C    IF THE WALKERS ARE WITHIN 200 STEPS OF EACH OTHER THEN
C    THE STEP SIZE IS INCREASED BY 10 PERCENT
C
      IF (ABS(WALKD-OTHERD).LT.SIZE*200)STEP = SIZE*1.1
C
C   CALCULATE THE NEW POSITION
C
      WALKD  = WALKD + STEP*DIRECT
 10   CONTINUE
      NSTEPS = NSTEPS + 1
```

These lines of Fortran are a subprogram that is to be referenced from the MAIN program on two occasions.

1. When Dale takes a step
2. When Erin takes a step

When Dale takes a step, then we want to supply the subprogram with his/her parameters (or attributes) and similarly for Erin.

Erin's Parameters	Dale's Parameters	Corresponding Subprogram Parameters	Meaning
EDIST	DDIST	WALKD	The current distance from the origin of the walker.
DDIST	EDIST	OTHERD	The current distance from the origin of the other person.
ESIZE	DSIZE	SIZE	The step size.
NERIN	NDALE	NSTEPS	The number of steps taken so far, by the walker.
-1.0	+1.0	DIRECT	The direction of walking, +1.0 indicates walking left to right, -1.0 indicates walking right to left.

18.1.5. Subroutine Description

Placing the SUBROUTINE line at the head of the walking subprogram and adding the other required garnishes (i.e. at least one RETURN statement and an END as the final statement) we get the complete subroutine below.

```
      SUBROUTINE  MOVE(WALKD,OTHERD,SIZE,NSTEPS,DIRECT)
C
C   INITIALLY THE STEPSIZE IS THE GIVEN ONE
C
      STEP = SIZE
C
C   IF THE NUMBER OF STEPS IS EXACTLY DIVISIBLE BY 100 THEN
```

```
C   NO DISTANCE IS TO BE TRAVELLED
C
      IF (MOD(NSTEPS,100).EQ.0)GO TO 10
C
C    IF THE WALKERS ARE WITHIN 200 STEPS OF EACH OTHER THEN
C    THE STEP SIZE IS INCREASED BY 10 PERCENT
C
      IF (ABS(WALKD-OTHERD).LT.SIZE*200)STEP = SIZE*1.1
C
C   CALCULATE THE NEW POSITION
C
      WALKD  = WALKD + STEP*DIRECT
 10   CONTINUE
      NSTEPS = NSTEPS + 1
C
C    FINISHED WALKING, SO RETURN TO THE CALLING PROGRAM
C
      RETURN
      END
```

18.1.6. Calling the subroutine

The subroutine calls will differ for each of the two walkers, because of the different parameters that each have.

1. The call for Dale:
   `CALL   MOVE(DDIST,EDIST,DSIZE,NDALE,1.0)`
2. The call for Erin:
   `CALL   MOVE(EDIST,DDIST,ESIZE,NERIN,-1.0)`

18.1.7. The complete program

Putting this all together and using the algorithm described above, the following program results.

```
C   AUTHOR:  L. LANDAU
C   DATE:    DECEMBER 1977
C            MODIFIED BY L. LANDAU, OCTOBER 1979
C   INPUT DESCRIPTION:
C
C     TWO LINES, ONE FOR EACH WALKER GIVING THE WALKER'S
C     STATISTICS. BOTH ARE IN THE SAME FORMAT.
C     THE FIRST LINE DESCRIBES THE WALKER ON THE LEFT WHO IS
C     WALKING TOWARDS THE WALKER ON THE RIGHT (WHO IS DESCRIBED
C     BY THE SECOND LINE).
C     THE FORMATS ARE:
C
C       TYPE       COLUMNS      MEANING
C       ----       -------      -------
C
C       REAL       1 - 10       STARTING POSITION, RELATIVE TO
```

```
C                                  SOME ORIGIN.
C        REAL      11 - 14         PACE LENGTH (METRES)
C        REAL      15 - 19         FITNESS FACTOR IN THE RANGE
C                                  0.0 TO 1.0
C
C          THE FIRST WALKER READ IN WILL TAKE THE FIRST STEP
C
C    PURPOSE:
C
C      TO SIMULATE TWO WALKERS WHO ARE WALKING TOWARDS EACH OTHER.
C      TO FIND OUT WHERE THEY MEET AND HOW MANY STEPS ARE TAKEN.
C      THE WALKING RULES ARE:
C      1. EACH TAKES A STEP ALTERNATELY
C      2. EACH HAS A REST EVERY 100 STEPS
C      3. IF WITHIN 200 STEPS OF EACH OTHER, THE NORMAL STEPSIZE
C         INCREASES BY 10%, FOR THAT STEP.
C
C      VARIABLES USED
C
C          DDIST     DALE'S CURRENT DISTANCE FROM THE ORIGIN
C          DFIT      DALE'S FITNESS FACTOR
C          DPACE     DALE'S NORMAL PACE SIZE
C          DSIZE     DALE'S FITNESS MODIFIED STEPSIZE
C          DSTART    DALE'S START POSITION
C          EDIST     ERIN'S CURRENT DISTANCE FROM ORIGIN
C          EFIT      ERIN'S FITNESS FACTOR
C          EPACE     ERIN'S NORMAL PACE SIZE
C          ESIZE     ERIN'S FITNESS MODIFIED STEPSIZE
C          ESTART    ERIN'S START POSITION
C          NDALE     NUMBER OF STEPS TAKEN BY DALE
C          NERIN     NUMBER OF STEPS TAKEN BY ERIN
C
C    SUBROUTINES USED:
C
C      MOVE     TO SIMULATE TAKING A STEP BY EITHER WALKER
C
C      INITIALLY, NEITHER HAS TAKEN ANY STEPS
C
      NDALE = 0
      NERIN = 0
C
C    READ IN DATA FOR EACH PERSON
C
      READ(5,100)DSTART,DPACE,DFIT
      READ(5,100)ESTART,EPACE,EFIT
  100 FORMAT(F10.0,2F5.0)
C
C    SAVE THEIR RESPECTIVE START POSITIONS
C
      DDIST = DSTART
      EDIST = ESTART
C
C     CALCULATE THEIR STEP SIZES ACCORDING TO THEIR FITNESS
C
      DSIZE = DPACE*DFIT
      ESIZE = EPACE*EFIT
C
C   WALK TOWARDS EACH OTHER UNTIL MEETING OCCURS
C
 5    CONTINUE
      CALL MOVE(DDIST,EDIST,DSIZE,NDALE,1.0)
      IF(DDIST.GE.EDIST)GO TO 50
C
C  NOW THE SECOND PERSON TAKES A STEP
C
      CALL MOVE(EDIST,DDIST,ESIZE,NERIN,-1.0)
      IF(EDIST.LE.DDIST)GO TO 50
      GO TO 5
C
C    COME HERE WHEN THEY HAVE MET
C
 50   CONTINUE
      DTRIP = DDIST-DSTART
      ETRIP = ESTART-EDIST
      WRITE(6,101)NDALE,DTRIP,NERIN,ETRIP
 101  FORMAT(' HELLO!',/
     $     1X,'DALE TOOK ',I10,' STEPS TO WALK ',F10.2,' METRES'/
     $     1X,'ERIN TOOK ',I10,' STEPS TO WALK ',F10.2,' METRES'/)
      STOP
      END

      SUBROUTINE  MOVE(WALKD,OTHERD,SIZE,NSTEPS,DIRECT)
C
C    AUTHOR:  L. LANDAU
C    DATE:    DECEMBER 1977
C             MODIFIED BY L. LANDAU IN OCTOBER 1979
C    PURPOSE:
C       TO SIMULATE THE TAKING OF A STEP EITHER IN THE POSITIVE
C       DIRECTION (TO THE RIGHT) OR THE NEGATIVE DIRECTION (TO
C       THE LEFT).
C
C   PARAMETER DESCRIPTION:
C
C     PARAMETERS THAT ARE ALTERED BY THE SUBROUTINE ARE
C     INDICATED BY AN *
C
C  *    WALKD     THE CURRENT DISTANCE FROM THE ORIGIN
C       OTHERD    THE DISTANCE FROM THE ORIGIN OF
C                 THE OTHER WALKER
C       SIZE      THE STEP SIZE FOR THIS WALKER
C       DIRECT    THE DIRECTION OF WALKING.
C                 +1.0 MEANS TO THE RIGHT
C                 -1.0 MEANS TO THE LEFT
C
C
C   INITIALLY THE STEPSIZE IS THE GIVEN ONE
C
      STEP = SIZE
```

```
C
C   IF THE NUMBER OF STEPS IS EXACTLY DIVISIBLE BY 100 THEN
C   NO DISTANCE IS TO BE TRAVELLED
C
      IF (MOD(NSTEPS,100).EQ.0)GO TO 10
C
C    IF THE WALKERS ARE WITHIN 200 STEPS OF EACH OTHER THEN
C    THE STEP SIZE IS INCREASED BY 10 PERCENT
C
      IF (ABS(WALKD-OTHERD).LT.SIZE*200)STEP = SIZE*1.1
C
C   CALCULATE THE NEW POSITION
C
      WALKD  = WALKD + STEP*DIRECT
 10   CONTINUE
      NSTEPS = NSTEPS + 1
C
C    FINISHED WALKING, SO RETURN TO THE CALLING PROGRAM
C
      RETURN
      END
      ENTER = [illegible]
C
C  MAKE [illegible]
C
    5 CONTINUE
      CALL MOVE(DDIST,EDIST,DSIZE,NDALE,1.0)
      IF(DDIST.GE.EDIST)GO TO 50
C
C  NOW THE SECOND PERSON TAKES A STEP
C
      CALL MOVE(EDIST,DDIST,ESIZE,NERIN,-1.0)
      IF(EDIST.LE.DDIST)GO TO 50
      GO TO 5
C
C  COME HERE WHEN THEY HAVE MET
C
   50 CONTINUE
      DTRIP = DDIST-DSTART
      ETRIP = ESTART-EDIST
      WRITE(6,100)NDALE,DTRIP,NERIN,ETRIP
  100 FORMAT(' HELLO!',/
     $       1X,'DALE TOOK ',I10,' STEPS TO WALK ',F10.2,' METRES'/
     $       1X,'ERIN TOOK ',I10,' STEPS TO WALK ',F10.2,' METRES')
      STOP
      END

      SUBROUTINE  MOVE(WALKD,OTHERD,SIZE,NSTEPS,DIRECT)
C
C   AUTHOR:  L. LASKAR
C   DATE:    DECEMBER 1977
C            MODIFIED BY L. LASKAR IN OCTOBER 1978
C   PURPOSE:
C     TO SIMULATE THE TAKING OF A STEP EITHER IN THE POSITIVE
C     DIRECTION (TO THE RIGHT) OR THE NEGATIVE DIRECTION (TO
C     THE LEFT).
C
C   PARAMETER DESCRIPTION:
C
C     PARAMETERS THAT ARE ALTERED BY THE SUBROUTINE ARE
C     INDICATED BY AN *
C
C  *  WALKD     THE CURRENT DISTANCE FROM THE ORIGIN
C     OTHERD    THE DISTANCE FROM THE ORIGIN OF
C               THE OTHER WALKER
C     SIZE      THE STEP SIZE FOR THIS WALKER
C     DIRECT    THE DIRECTION OF WALKING
C               +1.0 MEANS TO THE RIGHT
C               -1.0 MEANS TO THE LEFT
C
C
C   INITIALLY THE STEPSIZE IS THE GIVEN ONE
C
      STEP = SIZE
```

19. APPENDIX 5 - 1966 standards

19.1. DO loops

A DO statement is used to define a loop. The action following the execution of a DO statement is described in the following steps.

(i) The DO-variable is assigned the value of the initial parameter.

(ii) The range of the DO (i.e., the statements following the DO, up to and including the terminal statement) is executed.

(iii) The DO-variable is incremented by the value of the incrementation parameter.

(iv) If the value of the DO-variable is now less than or equal to the value of the terminal parameter, then the action starting at step (ii) is repeated. If the value of the DO-variable is greater than the value of the terminal parameter, the statement following the terminal statement is executed.

So, effectively a DO statement will repeat all the statements from the one immediately after the DO line, up to and including the terminal statement. This will be done a number of times, determined by the interactions of m1, m2, and m3.

The DO-variable becomes undefined after normal termination of the loop, but keeps its value if you jump out of the loop.

19.2. Nested DO loops

It is possible to have a DO loop wholly contained within another DO loop. This is known as 'nesting DO loops.' For example,

```
        DO 42 I = 1,15         -----------------------+
                                                      !
        . . . code A . . .                            !
                                                      !
        DO 5 J = 3,30,3        -------+               !
                                      !               !
        . . .       . . .             !    Inner      !
                                      !               !
        READ(5,*) L                   >     DO        !    Outer
                                      !               !
        . . .       . . .             !    LOOP       >     DO
                                      !               !
    5   CONTINUE               -------+               !    LOOP
                                                      !
        . . . code B . . .                            !
                                                      !
   42   CONTINUE               -----------------------+

        . . . code C . . .
```

The INNER loop (down to statement number 5) is said to be nested within the OUTER loop (which ranges down to statement number 42). The operation of this example is as follows.

(i) Set I to its initial value of 1.

(ii) Execute code A.

(iii) Set J to its initial value of 3.

(iv) Execute the inner DO loop.

(v) At label 5, increment J by 3 and test if it is greater than 30.
If TRUE, go on to execute code B.
If FALSE, go back to the statement AFTER the inner DO statement, and go back to do step (iv) above.

(vi) When execution arrives at label 42, increment the outer loop DO-variable, I, by 1, and test if it is greater than 15.
If TRUE, go on to execute code C.
If FALSE, go back to the statement AFTER the outer DO statement and repeat step (ii) above.

QUESTION: How many lines will be read in the above example?

19.3. Character handling

As well as numbers, characters may be stored in computer locations. These characters are stored using an integer code for each character, and on the UNIVAC computer, four characters are stored per word. If there are less than four characters to be stored in a word, those characters are left justified in the word and blanks are inserted to make up a total of four characters. Note that the number of characters per word is dependent on the computer being used. If a different computer is being used, these details should be determined before use. The following remarks relate to the UNIVAC computer. A character constant is written as a string of up to four characters enclosed by quote characters, e.g. 'FRED', 'I AM'.

Character variables do not exist as such in standard (pre-1977) Fortran but are stored in integer, real, or double precision variables. Integer and real variables may hold up to four characters while double precision variables may hold up to eight characters. Suppose that there are some characters in the integer variable MONEY and that these characters are to be placed in the real variable CHARS. The statement

```
        CHARS = MONEY
```

would normally do this. However, this statement is really indicating that the integer number in MONEY should be stored in the real variable CHARS. This, of course, involves an integer to real conversion before storing, which plays havoc with the characters that have actually been stored in MONEY. Hence, all variables that are to contain characters should be of the same type, generally integer. The same is true in using variables that contain text, in IF statements, namely that both variables in the relational expression must be the same type.
Arithmetic operations on variables containing characters are generally not meaningful.

Input and output of characters is handled with the A field descriptor, which is of the form

```
        Aw
```

where w is the width of the field.

The field descriptor causes w characters to be read into, or written from, the associated list element. The characters may be any of the allowable Fortran characters, including the blank character. The following description of the operation of the A field descriptor assumes that the characters are stored in integer or real variables.

On output, if w is greater than four, w-4 blanks followed by the four characters in the variable are printed in the field. If w is less than or equal to four, the leftmost w characters in the variable are printed.

On input, if w is greater than or equal to four, the rightmost four characters will be taken from the field of width w and stored in the variable. If w is less than four, the w characters will appear left justified (at the extreme left) in the variable, and blanks are inserted to make up a total of four characters.

Consider the statements

```
        READ (5,100) FIRST,SECOND,THIRD,FOURTH
    100 FORMAT(A8,A3,A4,A1)
        WRITE (6,110) FIRST,SECOND,THIRD,FOURTH
    110 FORMAT(1X,A8,A2,A6,A8)
```

and the input data card

THEbDATAbIS134.86

The READ statement will give the variables the following values.

```
        FIRST  = 'DATA'
        SECOND = ' IS '
        THIRD  = '134.'
        FOURTH = '8   '
```

The WRITE statement will print the line

bbbbDATAbIbb134.bbbb8

where 'b' indicates a blank

The following program will read a list of names and addresses and print them out in a presentable form. The name on the input line occupies the first 36 columns and the address takes columns 37 to 80 inclusive. The end of the data is indicated by a blank name field.

```
C
C     AUTHOR:  L. LANDAU
C
C     DATE:    OCTOBER 1979
C
C     INPUT DESCRIPTION:
C     -----------------
C
C         EACH LINE CONTAINS NAMES AND ADDRESSES. THE LAST LINE
C         CONTAINS BLANKS IN THE FIRST FOUR COLUMNS
C
C         COLUMNS         MEANING
C          1 - 36         NAME OF PERSON
C         37 - 80         ADDRESS OF PERSON
C
C     PURPOSE:
C     --------
C
C         TO READ IN AND LIST PEOPLES NAMES AND ADDRESSES
C         HAVING 50 PER PAGE, WITH PAGE COUNTS AT THE TOP
C
C
      DIMENSION NAME(9),ADDR(11)
C
      IBL = ' '
C
      IPAGE = 1
      LINCNT = 0
C
C     WRITE OUT A HEADING
C
      WRITE(6,100) IPAGE
```

```
 100   FORMAT('1NAME',46X,'ADDRESS',20X,'PAGE: ',I4/
      +        1X,'----',46X,'-------'//)
 C
 C     READ THE NEXT NAME AND ADDRESS
 C
 110   CONTINUE
       READ (5,120) (NAME(I), I=1,9), (ADDR(I), I=1,11)
 120   FORMAT(9A4,11A4)
 C
 C     TEST FOR END OF DATA
 C
       IF(NAME(1).EQ.IBL) STOP
 C
 C    CHECK IF A HEADING IS REQUIRED, THEN WRITE OUT NAME
 C    AND ADDRESS AND THEN GO BACK FOR MORE
 C
       IF( MOD(LINCNT,50) .NE. 0 )GO TO 125
       IPAGE = IPAGE + 1
       WRITE(6,100) IPAGE
  125  CONTINUE
       WRITE (6,130) (NAME(I), I=1,9), (ADDR(I), I=1,11)
       LINCNT = LINCNT + 1
  130  FORMAT(1X,9A4,14X,11A4)
       GO TO 110
       END
```

20. APPENDIX 6 - Summary of Fortran commands covered

It is stressed that this summary is only of those commands covered in this text and is NOT a complete summary of all the commands or of all the capabilities and extensions of the commands that are covered. A complete list could be obtained from a FORTRAN programmers reference manual that may be borrowed from the computer centre.

20.1. A command list

20.1.1. Executable statements

- Assignment
- Block-IF
- CALL
- CONTINUE
- DO
- END
- GO TO
- IF
- READ
  - Formatted
  - Free Format
- RETURN
- STOP
- WRITE
  - Formatted
  - Free Format

20.1.2. Non executable statements

- C ... comment ...
- CHARACTER
- DIMENSION
- FORMAT
- FUNCTION
- INTEGER
- LOGICAL
- REAL
- SUBROUTINE
- type declaration

20.2. Individual statement formats

20.2.1. Assignment

```
        v = e
```

v	a variable name or array element
e	arithmetic, character or logical expression

20.2.2. Block-IF

```
        IF (logical expression) THEN
           ... block 1 ...
        ELSE IF (logical expression) THEN
           ... block 2 ...
        ELSE
           ... block 3 ...
        END IF
```

An ELSE IF block is optional, and there may be several ELSE IF blocks. The ELSE block is optional, there can only be one and it must appear after all ELSE IF blocks.

20.2.3. CALL

```
        CALL n(a,a,a,....)
```

n	subprogram name
a	actual parameters (or arguments) which may be:

1. variable or array names.
2. arithmetic, character or logical expressions.
3. function or subroutine names.

If there are no parameters, then the brackets may be omitted.

20.2.4. CHARACTER

```
        CHARACTER*length list
```

Variables on the list without a length specification following them have a length as specified on the CHARACTER declaration. Form of variables on the list is

```
      variable*len1, variable*len2, variable, .... etc
```

20.2.5. Comment

```
      C ... comment ...
```

A line with a C in column 1 is a comment, and the rest of that line is ignored by the Fortran compiler. A blank line is also treated as a comment.

20.2.6. CONTINUE

```
  x      CONTINUE
```

x an optional statement label

20.2.7. DIMENSION

```
      DIMENSION a(d),a(d),.....
```

a variable names being specified as arrays
d a list of integer constants or formal parameters (in functions or subroutines) which specify the dimension bounds for the array. Up to 7 dimensions may be specified.

20.2.8. DO

```
      DO n i = m1,m2,m3
```

n a statement label indicating the last statement in the range of the DO.

i an integer or real variable (not an array element) called the control variable.

m1 an integer or real expression indicating the initial value for the control variable.

m2 an integer or real expression indicating the finishing or terminal value for the control variable.

m3 optional (if omitted the preceding comma should also be omitted. If omitted it is assumed to be 1). When specified, it is an integer or real expression indicating the increment for the control variable.

DO loops may be executed zero times. The number of iterations is given by:

```
      MAX (INT ((m2-m1+m3)/m3), 0)
```

20.2.9. END

```
END
```

20.2.10. FORMAT

```
x FORMAT(format - specifications)
```

x is a statement label

format-specifications may be nested to a maximum of 4 levels on the Univac. They are field descriptors separated by commas, and grouped in brackets. A slash (/) indicates that a record (either input or output) is to be skipped.

Field descriptors	Meaning
Iw	Integer
Aw	Alphanumeric
wX	Blank (or skip)
'text'	Character string (heading)
Fw.d	Real
Ew.d	Real with exponent

Where:

w	field width is an integer constant
d	decimal point designator

Print control characters (for the first character of output line image):

blank	advance one line
0	advance two lines
1	go to first line of next page.
+	no advance. Print from column 1 of same line.

20.2.11. FUNCTION

```
type FUNCTION n (a,a,a,....)
```

type may be blank or a variable TYPE

n	function name
a	dummy parameters which may be variable or array names. If there are no dummy parameters then the brackets may be omitted.

20.2.12. GO TO

```
GO TO x
```

x a statement label

20.2.13. IF

```
IF(l) s
```

l a logical expression
s an executable statement except a DO, another logical IF or any block-IF command.

20.2.14. READ

20.2.14.1. Formatted

```
READ (5,f,END=n) iolist
```

f	statement number of a FORMAT
n	statement number of an executable statement. END = n clause is optional.
iolist	the input list, variables separated by commas.

20.2.14.2. Free Format

```
READ(5,*,END=n) iolist
```

n	statement number of an executable statement. END = n clause is optional.
iolist	the input list, variables separated by commas.

20.2.15. RETURN

```
RETURN
```

20.2.16. STOP

```
STOP
```

20.2.17. SUBROUTINE

```
SUBROUTINE n (a,a,a,...)
```

n a SUBROUTINE name

a dummy parameters which may be variable names or array names. If there are no dummy parameters then the brackets may be omitted.

20.2.18. Type Declaration

```
t n,n,n,.....
```

t is a data type and may be one of INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER or DOUBLE PRECISION.

n are symbolic names

20.2.19. WRITE

20.2.19.1. Formatted

```
WRITE(6,f) iolist
```

f statement number of a FORMAT

iolist the output list, variables separated by commas. The iolist may be omitted.

20.2.19.2. Free Format

```
WRITE(6,*) iolist
```

iolist the output list, variables separated by commas. The iolist may include text enclosed in quotes.

20.3. Constants

20.3.1. Integer constants

Form: +d or -d

(a) d is a string of digits from the set 0 through 9

(b) the + sign may be omitted
(c) the range of values is -2**35+1 to +2**35-1 (which is 34,359,738,367)

20.3.2. Real Constants

Form1: + or - d.d
Form2: + or - d.dE + or - x

(a) d is a string of digits from the set 0 through 9
(b) x is an integer
(c) in Form2 the decimal point may be omitted
(d) + signs may be omitted
(e) the range of values is (approximately) + or - (1.46936794*10**-39, 1.70141182*10**38)

20.3.3. Character constants

Form: 'cccc....'

c... is a string of ASCII characters. A quote mark in the string is represented by two consecutive quote marks.

20.4. Logical Operators

Operator	Usage	Explanation
.AND.	e1.AND.e2	True if both e1 and e2 are true and false if at least on of e1 or e2 is false.
.OR.	e1.OR.e2	True if at least one of e1 or e2 is true and false if both e1 and e2 are false.
.NOT.	.NOT.e1	True if e1 is false, and false if e1 is true.

20.5. Relational operators

Operator	Usage	Explanation
.GT.	e1.GT.e2	True if the value of e1 is greater than that of e2.
.GE.	e1.GE.e2	True if the value of e1 is greater than or equal to that of e2.
.LT.	e1.LT.e2	True if the value of e1 is less than that of e2.
.LE.	e1.LE.e2	True if the value of e1 is less than or equal to that of e2.
.EQ.	e1.EQ.c2	True if the value of e1 is equal to that of e2.
.NE.	e1.NE.c2	True if the value of e1 is not equal to that of e2.

20.6. Some Fortran Mathematical Functions

Abbreviations : DP = double precision
INT = integer
param = parameter

Name	No. of Params	Type of Params	Type of Result	Description
LOG	1	REAL or DP	REAL or DP	Natural logarithm
LOG10	1	REAL or DP	REAL or DP	Log base 10
EXP	1	REAL or DP	REAL or DP	Exponential
SQRT	1	REAL or DP	REAL or DP	Square root
ASIN	1	REAL or DP	REAL or DP	Arc sine (radians)
ACOS	1	REAL or DP	REAL or DP	Arc cos (radians)
ATAN	1	REAL or DP	REAL or DP	Arc tan (radians)
SIN	1	REAL or DP	REAL or DP	Sine (radians)
COS	1	REAL or DP	REAL or DP	Cosine (radians)
TAN	1	REAL or DP	REAL or DP	Tangent (radians)
COTAN	1	REAL or DP	REAL or DP	Cotangent (radians)
SINH	1	REAL or DP	REAL or DP	Hyperbolic sine
COSH	1	REAL or DP	REAL or DP	Hyperbolic cosine
TANH	1	REAL or DP	REAL or DP	Hyperbolic tangent
ABS	1	REAL, DP or INT	REAL, DP or INT	Absolute value
MAX	>1	REAL, DP or INT	REAL, DP or INT	Maximum value
MIN	>1	REAL, DP or INT	REAL, DP or INT	Minimum value
MOD	2	REAL, DP or INT	REAL, DP or INT	Remainder on dividing the first param by the second
REAL	1	INT or DP	REAL	Convert INT or DP to real
INT	1	REAL or DP	INT	Convert real or DP to INT

21. APPENDIX 7 - Fortran not covered in the course

This appendix introduces those areas of Fortran that have not been covered in the lecture course. The topics here are considered to be of secondary importance to the new programmer, but they are very useful, and should probably be learned at a later stage.

A subprogram is a subroutine or function. A program unit is a main program or a subprogram.

21.1. FURTHER DATA TYPES

There are two other data types that exist in Fortran.

21.1.1. DOUBLE PRECISION

The data type DOUBLE PRECISION may be used if the size of a variable will exceed the maximum limit allowed in single precision or will be too small to be represented in single precision (ie. underflow). On the UNIVAC computer the maximum size of a double precision variable is about 10**308, and the minimum size is 10**(-308).

More often it is the case that rather than the size of the real variable causing problems, it is the maximum number of significant figures (depleted by truncation and rounding errors) that motivates the move to double precision, which (on the Univac) keeps up to 18 significant figures. In fact on some computers the size of number that may be represented in single or double precision is the same, the only reason then is to be able to have a greater number of significant figures.

The disadvantages of using DOUBLE PRECISION are:

(a) That each double precision variable requires TWO memory cells of storage as compared with the ONE required by ordinary REALs.

(b) That the time taken for double precision arithmetic is appreciably longer than the time for the corresponding operation in REALs.

21.1.1.1. DOUBLE PRECISION CONSTANTS

These are very similar in form to the exponent form of REAL constants, except that there is a D instead of the E. The general form is:

nnnn.nnnnD+eee The 'n' are digits comprising the base value of the number.
The 'eee' is the exponent part of the number.

For example the number 1.23456 would be represented as 1.23456D0 as a double precision constant.

Other examples are:
    21.34D-10, 123456789.987654D-127, 1.0D0

21.1.1.2. DOUBLE PRECISION VARIABLES

A double precision variable is formed in the normal way and is declared to be double precision by the type statement

```
        DOUBLE PRECISION variable-list
```

Arithmetic expressions are formed in double precision according to the same rules that apply to real expressions. The arithmetic operators (+,-,*,/,**) are the same. The combination of double precision with real and integer constants and variables in an arithmetic expression is explicitly permitted. In both cases, the result is always a double precision value.

It is permissible to have an integer, real, or double precision variable on the left side of an arithmetic statement and an expression of some other type on the right side. All arithmetic is done according to the type of the expression on the right, and the result is converted according to the variable on the left. Acceptable uses of double precision quantities are

```
        DOUBLE PRECISION D1,D2,D3,D4,D5,D6
        D1 = D2*D3+(D4-8756.7865432D0)/D5
        D1 = 4.0*D2-D3/1.1D0
        D1 = R1+D1+R2
        R1 = (D1*D2-D3*D4)/(D1*D5-D3*D6)
        D1 = R1+2.0
        D1 = (I1-8)*I2
        I1 = R1+D1
        D1 = D2**2
```

21.1.1.3. DOUBLE PRECISION FUNCTIONS

Generic functions, described in chapter 6, may have double precision parameters and/or return double precision results. Some of these functions are:

ABS(double)	returns the absolute value of a double precision number
MOD(dbl1,dbl2)	same as MOD but uses double precision numbers
MAX(dbl1,dbl2,...)	returns largest of double precision numbers
MIN(dbl1,dbl2,...)	returns smallest of double precision numbers
DBLE(real or int.)	returns double precision equivalent of real or integer number
INT(double)	truncates double precision numbers to integer
REAL(double)	returns most significant part of double precision number as a real number
SIN(double)	returns sine of double precision number expressed in

	radians
COS(double)	returns cosine of double precision number expressed in radians
EXP(double)	returns exponential of double precision number
SQRT(double)	returns square root of double precision number

21.1.1.4. INPUT/OUTPUT

Input and output of double precision quantities is handled with the D field descriptor which is similar to the E field descriptor except that

(i) the list variable associated with this field descriptor must be double precision,
(ii) there may be more digits, and
(iii) D is used for the exponent indicator rather than E.

21.1.2. COMPLEX

This data type is used to represent a complex number (in mathematical terms, comprising a real and an imaginary part).

The representation is as a pair of REAL numbers.

COMPLEX constants are represented as a pair of real numbers separated by a comma and enclosed in brackets.

COMPLEX variables must be declared in a type COMPLEX declaration statement.

Input/output is accomplished by using two real field descriptors for each complex value written or read.

For example consider the statements:

```
      COMPLEX FREUD, FRASER, QUINCY
      READ(5,100)FREUD
100   FORMAT(2F10.0)
      QUINCY = (1.2,-3.4679)
      FRASER = QUINCY*FREUD + 2.5
```

Mixed mode (with the exception of exponentiation) is allowed between type COMPLEX and REAL or DOUBLE PRECISION, and the result will be COMPLEX.

A type COMPLEX variable or constant may only be assigned to a type COMPLEX variable.

21.1.2.1. COMPLEX FUNCTIONS

The generic functions for COMPLEX data types are:

REAL(complex)	returns the REAL part of a COMPLEX number.
AIMAG(complex)	returns the imaginary part of a COMPLEX number.
CMPLX(a,b)	express two real numbers or two integers in complex form.
CONJG(complex)	obtain the complex conjugate of a complex number.
INT(complex)	returns the real part, truncated to an integer.
ABS(ar,ai)	returns the real result of SQRT(ar**2 + ai**2), where ar is the real part and ai is the imaginary part of a complex number.

SQRT, EXP, LOG, SIN or COS(complex) as for other data types

21.2. DATA STATEMENT

The DATA statement is used to assign initial values to variables. This assignment is done at the time of compilation and not at the time of execution of the compiled program. The DATA statement is not an executable statement. The form of the statement is

```
        DATA data-list
```

where the data-list is a list of the form

```
        variable-list / value-list /
```

The variables in the variable-list are assigned the corresponding values of the constants in the value-list. There must be a one to one correspondence between the variables and the values, and generally the type of a variable and its value must be the same. If the types do not match then the value is converted to the variable type if possible (eg integer to real), or if conversion is not possible, the initialization causes an error (eg integer to logical or character).

The statement

```
        DATA A,B,C /14.7,62.1,1.5E-20/
```

assigns the value 14.7 to A, 62.1 to B, and 1.5E-20 to C.

The two statements

```
        DATA A /67.87/, B /54.72/, C /5.0/
        DATA A,B,C /67.87,54.72,5.0/
```

have the same effect, the choice being a personal preference.

It is possible to repeat a value a number of times. For example, the statement

```
        DATA R,S,T,U,V,W /6*21.7/
```

assigns the value 21.7 to all six variables.

As the DATA statement assigns initial values to variables, it is legal to redefine the values of these variables later in the program, but, having done so, it is NOT possible to 're-execute' the DATA statement to return the variables to their initial values.

Initial values may be assigned to variable lists in any program unit. Initial values may not be specified for dummy parameters, and may not be specified for any variable more than once.

A variable list on a DATA statement may include variables, arrays, array elements, substring names, and implied-DO groups, separated by commas. The format of an implied-DO group is

```
    (variable-list, index = start, end, increment)
```

where start, end and increment are integer constant expressions. The increment is optional and must be positive. Implied-DO groups may be nested.

Array subscripts must be integer expressions using only constants, parameter variables (see below) and implied-DO index variables. Substring expressions must be integer constant expressions.

If an entire array is initialized (with no implied-DO), then the elements are initialized with the first subscript changing fastest, etc. (eg. down the columns in a 2D array).

If the value list is too short, the last elements of the variable list are not initialized and if it is too long, the last values are ignored.

A DATA statement is placed after type and dimension declarations of variables which appear in the DATA statement.

Example :

```
       REAL A(10), B(10)
       CHARACTER*4 C(2)
       DATA A / 10*0.0 /
       DATA ( B(I), I = 1,5 ) / 5*1E5 /, (B(I), I = 6,10) / 5*2E10 /
       DATA C(1)(1:2), C(2) /'ab', 'cdef'/
```

21.3. IMPLICIT

The IMPLICIT statement assigns a data type to a name depending on the initial letter of the name. It has the form :

```
    IMPLICIT  type (letters), type (letters)
```

where 'type' may include a length (eg. CHARACTER*4) and the letters are single letters or letter ranges separated by commas. A letter range consists of two letters separated by a hyphen. For example, B-F means B,C,D,E,F.

IMPLICIT overrides the default association of particular letters with data types. The letters in brackets become associated with the specified type. Letters which are not included on an IMPLICIT statement retain their default types.

Names affected are all variables, arrays, parameter constants, functions and statement functions within the program unit.

The IMPLICIT statement is placed before type declarations and DATA statements. It may appear after a PARAMETER statement, in which case default types apply to the parameters.

Example :

```
          IMPLICIT LOGICAL (L)
          IMPLICIT CHARACTER*4 (C-E),  COMPLEX (F-H,X-Z)
```

After this, A,B,O-W still default to real numbers and I-K,M,N to integers.

The usual method of converting a program to use double precision instead of real numbers is to use:

```
              IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

21.4. PARAMETER

The PARAMETER statement allows constants in a program unit to be referenced by names, in order to make a program easier to alter. It has the form :

```
     PARAMETER ( n1 = e1, n2 = e2, ... )
```

where n1, n2, ... are variable names and e1, e2, ... are constant expressions.

The PARAMETER statement is not executable, and is placed in the program before the parameter variables are referenced. If a parameter variable is not to have a default type, then its type must be declared (with IMPLICIT or an explicit type declaration) before the PARAMETER statement.

The constant expression is evaluated at compile time, and may consist of constants, parameter variables defined on previous PARAMETER statements, or Fortran-supplied functions.

A parameter variable cannot be redefined in the program. Each reference to a parameter variable in the program unit is replaced by its constant value. It is usually used in declaring array sizes in the main program. It cannot be used as a constant in a FORMAT statement.

Example :

```
         PARAMETER ( N = 100, L = 80 )
         REAL ARRAY(N)
```

```
        CHARACTER*L  LINE
        DATA ARRAY / N * 0.0 /   LEN / L /
```

21.5. COMMON

COMMON refers to an area of memory that Fortran dedicates for the storage of variables. Normally variables are stored somewhere in the computer's memory and it is of no concern to us where this is, as we may always refer to a variable by its name and thus retrieve its value.

The form of a COMMON declaration statement is:

```
        COMMON  variable list
```

For example:

```
        COMMON FRED,I,KID,MUCH,ZAP(26),NUM(18,3)
```

The above line declares that the variables indicated be stored in the COMMON area in the order that they appear on the COMMON line. The arrays indicated (i.e. ZAP and NUM) have their dimensions appearing in the COMMON declaration, rather than on a DIMENSION statement. Dimensions may be declared in type declarations or DIMENSION statements and then the array names (without the dimensions) are listed in the COMMON statement.

A COMMON declaration may occur in main programs and subprograms. The purpose of a COMMON declaration is so that a main program may share the same storage as a subprogram for particular variables. So this then is another way of communicating between subprograms, in addition to the parameter list.

The main thing to remember about COMMON blocks is that storage for variables is allocated in strict order of occurrence on the COMMON declaration. Many program errors result from a 'mismatching' of COMMON blocks across subprograms.

For example:

To write a program that will find the average of the number of integers stored in the array BLAM, of size 25:

```
        COMMON BLAM(25),ANS
        READ(5,*)(BLAM(I),I=1,25)
        CALL AVER
        WRITE(6,10)ANS
    10  FORMAT(1X,'AVERAGE OF 25 NUMBERS IS ',F10.2)
        .     .     .     .     .
        .     .     .     .     .
        .     .     .     .     .
        .     .     .     .     .
        END
        SUBROUTINE AVER
        COMMON ARRAY(25),VALUE
        TOT = 0.0
```

```
      DO 20 I = 1,25
   20 TOT = TOT+ARRAY(I)
      VALUE = TOT/25
      RETURN
      END
```

In addition to having COMMON blocks as above, it is possible to have a number of different COMMON areas in the one program. In this case the COMMON blocks must be distinguished by giving them each a different name (following the same rules as variable names). The type of COMMON block above is called BLANK COMMON as distinct from NAMED COMMON (which may also be called LABELLED COMMON).

In order to declare a named COMMON block, enclose the name in slashes following the word COMMON. For example:

```
      COMMON FRED,JIM,HENRY,BLOGGS
      COMMON /XERXES/NEDDY,NAV(27),DUM
      COMMON /UFO/SPUD,F111(10),TEACUP
```

FRED, JIM, HENRY and BLOGGS occupy the first 4 locations of BLANK COMMON.

NEDDY, NAV and DUM occupy the first 29 locations of the NAMED COMMON block called XERXES.

SPUD, F111 and TEACUP occupy the first 12 locations of the NAMED COMMON block UFO.

It does not matter in which order the COMMON statements appear, as long as each refers to different COMMON blocks. If more than one refers to the same named (or blank) COMMON block, then each successive COMMON declaration is taken to be a continuation of the previous one.

COMMON blocks make the passing of many values between program units a little easier, but less flexible. It is also less obvious which variables are needed and/or changed by the program unit.

21.6. BLOCK DATA

A third type of subprogram is BLOCK DATA. It contains no executable code and is used solely to assign initial values to variables in COMMON blocks. A BLOCK DATA subprogram is not called in the program, so on the Univac, it must be specifically mapped into a program.

The first statement of a BLOCK DATA subprogram is

```
    BLOCK DATA        or        BLOCK DATA name
```

where 'name' is an optional name for the subprogram, and only needs to be used if there is more than one BLOCK DATA subprogram in a program. The name follows the rules for variable names and must not be the same as that of another external subprogram, common block or local name within the subprogram.

The subprogram contains data specification statements and comments, and finishes with an END statement.

21.6.1. Examples

```
(a)      BLOCK DATA BLKA
    C    initializes common block A
         INTEGER K(10)
         COMMON /A/ K
         DATA K /10*0/
         END

(b)      BLOCK DATA
         CHARACTER*8 LINE(10)
         COMMON A,B,C /OUT/ LINE
         DATA LINE /10*'12345678'/, B, C /1.0, 2.0/
         END
```

Note that no initial value is assigned to A, but it is included in the COMMON statement to keep B and C in the correct positions.

21.7. EXTERNAL

EXTERNAL is a specification statement, and must precede all executable statements. It indicates that a name is a subroutine or function name, and not the name of a variable or Fortran-supplied function.

It has the form

```
    EXTERNAL names
```

where 'names' are subroutine or function names separated by commas.

This statement is needed when the first reference to a subroutine or function does not have an explicit actual parameter list. For example, a subroutine or function which is not called from a program unit, but is passed to a subroutine or function as an actual parameter, is only referenced by its name, not with its parameter list, so the program unit needs to be able to distinguish it from a variable that has not been explicitly declared.

Note that a function with no parameters need not have brackets on its FUNCTION statement, but when it is called it must have empty brackets (as shown in the example below).

If a name is the same as a Fortran-supplied function, then the user-supplied function will be used instead of the Fortran-supplied function (eg. to write your own SIN routine).

Example :

```
      COMMON A,B,C
      EXTERNAL FUN
      READ (5,*) A,B,C
      CALL SUB(FUN)            calls SUB with FUN as an actual parameter
      WRITE (6,*) A,B,C
      STOP
      END
C
      FUNCTION FUN
      COMMON A,B,C
      FUN = B**2 - 4*A*C
      RETURN
      END
C
      SUBROUTINE SUB(F)        Function F is a dummy parameter.
      COMMON X,Y,Z
      Z = SQRT( F())/ (2*X)    Call to function F needs empty
      X = SIN(Z)               parameter list
      Y = COS(Z)
      RETURN
      END
```

EXTERNAL tells the main program that FUN is a function or subroutine, and not a real variable like X, Y and Z.

21.8. EQUIVALENCE

This statement specifies sharing of storage by data within a program unit. Note that COMMON specifies sharing of data between program units.
The form is :

```
      EQUIVALENCE (n1, n2, n3, ... ), (n4, n5, ...)
```

where n1 to n5 are names of variables, arrays, array elements or character substrings.
They may not be dummy parameter names. Data in COMMON cannot be equivalenced.

The data in one set of brackets share some or all of the same storage, with the first word of n1 being given the same storage location as the first word of each of n2, n3, etc., and consecutive locations of n1, n2, n3 etc. are also shared. The order of items in the brackets is not important.

For example,

```
      REAL A(10), B(10)
      EQUIVALENCE (A, B(5))
```

defines A(1) to have the same storage location as B(5). Because arrays are always given consecutive locations, A(2) is equivalenced to B(6), ... A(6) to B(10). A(7) to A(10) follow in consecutive locations. Consecutive locations of memory may be depicted with their contents as follows :

```
   1   2   3   4   5   6   7   8   9   10   11   12   13   14
```

```
    B(1) . . . . . B(5) B(6) . . . . . B(10)
                   A(1) A(2) . . . . .  A(6) . . . . . . . A(10)
```

Data of different types and different lengths may be equivalenced. The types are not changed, and no conversion is done. This can be used to reduce storage requirements when some data is used in one part of the program unit and other data is used in another part of the program unit. It is your responsibility to make sure that the program does not destroy required information.

Variables may appear in more than one list, thus effectively combining the lists.

Examples :

```
(a)          CHARACTER*4  A, B, C*8
             EQUIVALENCE  (A, C), (B, C(5:5))
```

equivalences A to the first 4 characters and B to the last 4 characters of C.

(b) A subroutine to use real or integer data depending on whether L is true or false:

```
          SUBROUTINE SWITCH ( L )
          LOGICAL  L
          INTEGER  N(1000)
          REAL     A(1000)
          EQUIVALENCE  ( N, A )
    C
          IF ( L ) GO TO 100
    C     data is integer
          READ (5,*) N
            .....
          RETURN
    C     data is real
     100  READ (5,*) A
            .....
          RETURN
          END
```

Here, either array N or array A is used, but they must be declared separately because they are different types, so equivalencing them saves 1000 words of memory.

21.9. ARITHMETIC STATEMENT FUNCTIONS

It often happens that a programmer will find some relatively simple computation recurring through his program, making it desirable to be able to set up a one-line function to carry out the computation. This function is defined in a program unit and then used whenever desired in that program unit. It is not defined for any other program unit.

An arithmetic statement function is defined by writing a single statement of the form

```
        a = b
```

where	a	is the name of the function, and
	b	is an arithmetic expression.

The name, which is invented by the programmer, is formed according to the same rules that apply to a variable name, including the use of type statements. This name must not be the same as that of any supplied function. The name of the function is followed by brackets enclosing the parameter(s) which must be separated by commas if there is more than one. The parameters in the definition must not be subscripted variables. The right hand side of the definition statement may be any arithmetic expression not involving subscripted variables. It may use variables not specified as parameters and it may use other functions (except itself). All function definitions must appear after other specification statements and before the first executable statement of the program. If the right hand side of the arithmetic statement function uses another arithmetic statement function, the other function definition must have appeared earlier in the program unit.

As an example, suppose that in a certain program, it is frequently necessary to determine the distance between two points in a two dimensional space. An arithmetic statement function can be defined to carry out this computation by writing

```
        DIST(X1,Y1,X2,Y2) = SQRT((X2-X1)**2 + (Y2-Y1)**2)
```

This is only the DEFINITION of the function, and does not cause computation to take place. The variable names used as parameters are only dummies and may be the same as variable names appearing elsewhere in the program. The parameter names are only important to distinguish between integer, real and double precision.

An arithmetic statement function is USED by writing its name whenever the function value is desired and substituting appropriate expressions for its parameters. 'Appropriate' here means, in particular, that if a variable in the definition is real, then the expression substituted for that variable must also be real, and similarly for the other types of variables. The values of these expressions will be substituted into the function definition and the value of the function computed. The actual parameters may be subscripted if desired.

Suppose that it is now desired to use the DIST function to find the distance between the two points (16.9,R-S) and (T+6.9,-22.4), and that then this value is to be added to the cosine of X and the sum is to be stored in the variable ANS. All this can be done with the statement

```
        ANS = DIST(16.9,R-S,T+6.9,-22.4) + COS(X)
```

Suppose that later in the program it is necessary to compute the cube of the distance between the two points (DATA(I),DATA(I+1)) and (0..087,DATA(10)) and store the result in TEMP. The statement required is

```
        TEMP = DIST(DATA(I),DATA(I+1),0.087,DATA(10)) **3
```

It must be emphasized that the variables X1, Y1, X2, and Y2 in the function definition have no relation to any variables of the same names that may appear elsewhere in the program. To illustrate, suppose that the distance between the two points (X1,T+3.4) and (Y2,X2) is required. This value may be found by writing

```
        VALUE = DIST(X1,T+3.4,Y2,X2)
```

The X1, X2, and Y2 that appear here in the USE of the function are completely unrelated to the X1, X2, and Y2 in the DEFINITION of the function.

Some arithmetic statement functions are:

```
        LOG2(X) = LOG(X)/LOG(2.0)
```

to find log of X in base 2, and

```
        SIND(X) = SIN(0.01745329*X)
        COSD(X) = COS(0.01745329*X)
```

to find the sine and cosine of angles in degrees.

21.10. Ordering of statements

1. SUBROUTINE, FUNCTION or BLOCK DATA (for a subprogram)
2. PARAMETER
3. IMPLICIT
4. DIMENSION and type declarations
   (PARAMETER may come after type declarations)
5. EXTERNAL
6. COMMON
7. EQUIVALENCE
8. DATA
9. statement functions
10. executable statements
11. END

FORMATS may appear anywhere between 1 and 10. Some people group them all together at the beginning (after 8) or at the end (before 11).

Comments may appear anywhere before 11.

2 to 7 are called specification statements, and together with 8 set up storage space and initial values at compilation time.

21.11. Arrays

Array subscripts range by default from 1 to N, where N is the number of elements in the array.

In 1977 Fortran, the range of subscripts need not have a lower bound of 1. Subscripts may start at any integer, with the restriction that the upper bound be greater than or equal to the lower bound. Unless the lower bound is 1, both the lower and upper bounds appear on the declaration, separated by a colon.

For example,

```
      REAL NUMBER(0:9)
```

declares an array of 10 real numbers, with legal subscripts 0 to 9. So

```
     NUMBER(0)  = 1.0     is legal, but
     NUMBER(10) = 1.0     is not legal.
```

If the lower bound and colon are omitted from the declaration, the default lower bound of 1 is assumed.

Up to 7 dimensions are allowed. Multi-dimensional arrays may be declared as follows :

```
     REAL X(10,0:9,-5:4), Y(-1:1,10:20)
```

X is a real 10 by 10 by 10 array. Legal array elements are X(1,0,-5), X(4,4,4), X(10,9,-1).

Y is a real 3 by 11 array. Legal array elements are Y(-1,10), Y(0,20), Y(1,15).

Illegal elements are X(1,10,1), X(0,0,0), Y(1,1), Y(-1,5).

There are two ways of avoiding the declaration of fixed-size arrays in subroutines and functions when the arrays are parameters.

(a) An adjustable-size array is declared to have variable length, where the variable is a parameter.

(b) An assumed-size array has * as its LAST dimension. For example,

```
     SUBROUTINE SUB(ARRAY,I)
     REAL ARRAY(I,*)
```

defines a 2-dimensional array with an adjustable second dimension. Since ARRAY is a parameter, no storage is actually allocated, so the compiler does not need to know the exact size of the array. However, during program execution, the dummy array must not assume more storage than is allocated to the actual parameter.

22. APPENDIX 8 - Additional features of UNIVAC FORTRAN

The complete differences between Univac FORTRAN V, Univac ASCII FORTRAN and the ANSI standards of 1966 and 1977 are many and varied, depending on which combination you look at. Some of these differences are described in Univac manuals. In particular, the UNIVAC 1100 SERIES FORTRAN V Programmer Reference Manual provides details of the differences between Univac FORTRAN V and the 1966 standard. The UNIVAC 1100 SERIES FORTRAN(ASCII) Programmer Reference Manual provides the differences between the two Univac FORTRANs. The summary below is an attempt at combining all of this into an extension to the 1966 standard (using ASCII FORTRAN level 9R1 as the version of ASCII FORTRAN). Some points are elaborated later in this chapter.

22.1. Summary

1. Internal subprograms are permitted, where main and internal subprograms are part of the same program unit, which requires only one compilation. (See description below).

2. Octal values may be used to preset variables in a DATA statement.

3. Comment lines may precede a continuation line. Blanks lines are interpreted as comments.

4. The END line is not essential. If missing, an RCL command signals the end of the program unit. The END statement is executable, and implies a STOP in a main program or RETURN in a subprogram.

5. The introduction of 'typeless' data, where a word is treated as 36 true/false values, and there are Boolean functions to operate on the data.

6. The use of the quote symbol to form a literal string rather than the use of a Hollerith constant. Univac Ascii Fortran still allows Hollerith constants (see description below).

7. Arrays may have up to seven dimensions (the 1966 standard only allowed three).

8. An array element may have fewer subscript expressions than the dimensions of the array, with the omitted subscripts being assumed to be 1.

9. Generalised forms of subscripts are allowed, including non-integer subscript expressions.

10. The IMPLICIT statement may be used to extend and dynamically modify the default typing rule of ANSI FORTRAN, so that it extends to other data types and/or covers different first characters of the variable names.

11. A formal parameter of a subprogram may be the symbol $ which is used via the RETURN statement as an alternate exit from the subprogram. The actual parameter to be associated with the $ formal parameter is a statement label

preceded by a $. This mechanism is used to return control to the referencing program unit at other than the normal return from the calling sequence.

12. The use of a formal parameter as a FUNCTION or SUBROUTINE name is permissible if the associated actual parameter is the name of or an entry of a function or subroutine, respectively.

13. All combinations of types are allowed in arithmetic expressions with the exceptions that double precision may not be combined with complex, and typeless may not be combined with either double precision or complex.

14. The RETURN statement is extended to allow a form of:

```
      RETURN k
```

where k is a parameter position index into the SUBROUTINE, FUNCTION or ENTRY statement through which the subprogram was entered. Control is returned to the referencing program at the label specified by the kth statement number in the actual parameter list. The quantity appearing on the RETURN statement may be an integer constant, an integer variable, or a PARAMETER variable.

15. An optional comma is allowed in a DO statement. Eg, DO 15,I = 1,9

16. A DO loop may have a negative increment value, and can be executed zero times.

17. An INTRINSIC statement identifies a name as an intrinsic function, so that it can be used as an actual parameter in a subroutine or function call.

18. Colons in a FORMAT specification. When a colon is encountered in a format, the READ/WRITE operation is ended if the last item in the list has already been processed.

19. The following clauses may be added to READ and WRITE statements:

```
   ERR=l1
   END=l2
```

where l1 and l2 represent statement labels. Control will pass to l1 if there is an error in the I/O (ERR clause), or to l2 if the end of file has been read (END clause).

20. Any print control character other than 0, 1, or + will be treated as if it were a blank.

21. COMMON variables may be used as adjustable dimensions of an array in a subprogram.

22. The use of substring names and array names with implied DO loops are allowed in the variable list of a DATA statement.

23. Additional field descriptors are:

```
       Tw ... character positioning
       Iw.d . Same as Iw but at least d digits to be written
       Jw   . Integer, zero filled
```

Rw ... right justified alphanumeric
Ow ... octal

24. A form of FREE FORMAT is allowed for simplified I/O.

25. Additional intrinsic functions are provided, some are:-

DINT	AND	COMPL	MIN	XOR
FLD	MAX	OR	LOC	DDIM
BOOL	ICHAR	CHAR	LEN	INDEX
NINT	LGE	LGT	LLE	LLT
ANINT	DNINT	NINT	IDNINT	DPROD

26. Additional external functions are provided, some are:-

TAN	DASIN	DACOS	CSINH	CCOSH	CBRT
DTAN	ACOS	SINH	COSH	DTANH	DCBRT
CTAN	LOG	DSINH	DCOSH	CTANH	CCBRT
ASIN	LOG10				

27. ENCODE and DECODE (or internal file READ and WRITE) statements may be used to transfer data around in storage using different format control. The internal unit identifier is a character variable, array, array element or substring.

28. The PARAMETER statement enables variables to be preset at compile time, so that (amongst other things) they may be used to specify the size of arrays. IMPLICIT and type declarations may be used to change the default type of parameter constants.

29. Multiple entry points are available in any subprogram via the ENTRY specification line.

30. The ability to use random access files.

31. The BITS function which allows access to the bit level.

32. An expanded character set to handle the full ASCII set of characters.

33. Double precision complex data type is allowed.

34. Single precision COMPLEX may be 2 integers or 2 reals.

35. Character data type is introduced, with character assignments and comparisons also allowed.

36. Concatenation of character strings is allowed.

37. A '$' is allowed in symbolic names, except for the first character.

38. An actual parameter may be a substring. If it is an array element substring then it may be associated with a dummy parameter which is an array.

39. Multiple assignments are implemented, ie. v1, v2, ... = e .

40. Integer expressions are allowed on computed GO TOs.

41. Integer, real and double precision expressions are allowed for DO parameters and the DO-variable may be real or double precision.

42. The STOP and PAUSE statements are extended to allow a message or an identification number to be printed on stopping or pausing. The form is:

```
STOP n
STOP 'message'
PAUSE n
PAUSE 'message'
```

where n = up to 6 digits.

43. Expressions are permitted in an output I/O list.

44. Expressions are permitted in an implied DO (as with DO loops), and implied DO-loops may be done zero times.

45. The EXTERNAL statement is extended to allow linkage to non-Fortran suproutines, and subprograms with the same names as intrinsic functions.

46. A DEBUG facility is provided.

47. Interactive debugging is provided.

48. Statement number variables may be used for the format number in I/O statements.

49. Exponentiation between variables af all arithmetic types and lengths is permitted.

50. Conversion of constants in DATA statements to match the variable type (with a diagnostic warning also supplied).

51. The first statement of a main program may be:

```
PROGRAM name
```

52. A SAVE statement may be used to retain the values of variables and arrays as they were defined before returning from a subprogram. On re-entry into the subprogram, the specified variables have their saved values. Form:

SAVE n,n,...	where n is a named common block (eg /NAME/), a variable or an array name.

53. Logical operators .EQV., .NEQV. are allowed.

54. EQUIVALENCE may contain character substring names, and integer constant expressions for subscript and substring expressions.

55. The name of a statement function may appear in a type declaration or may be typed with IMPLICIT.

56. In a subprogram, where an array is in the parameter list, it may have an 'assumed' size. The upper dimension bound of the last dimension is declared as

an asterisk, and it is assumed that subscipts will not go out of bounds. Example:

```
        SUBROUTINE SUB(X,Y,N)
        REAL X(N,*), Y(0:*)
```

57. A BLOCK DATA subprogram may have a name. This name may appear in an EXTERNAL statement.

58. A comma before variable lists in DATA statements is optional, ie.

```
 DATA variable list /constant list/[,] variable list /constant list/
```

59. Substring names are allowed in input/output lists.

60. An empty input/ouput list is allowed on READ/WRITE to skip a record or to write an empty record.

61. OPEN,CLOSE and INQUIRE statements are available for file-handling.

62. For a subroutine with no parameters, empty brackets are optional on the SUBROUTINE statement and CALL. For a function with no parameters, empty brackets are optional on the FUNCTION statement (but must be included on the function reference except when the function name is an actual parameter).

22.2. Internal and External Subprograms

Subroutines, functions and BLOCK DATA are referred to collectively as subprograms. A program consists of a main program and zero or more subprograms.

Subprograms may be internal or external. An external subprogram is either the first subprogram in a file element or follows a previous END statement. If a subprogram is not terminated by END then the following subroutine or function is internal to it.

An internal subprogram is considered to be nested within the previous external subprogram, and has access to its data as well as having its own local data. Data consists of variables, arrays, statement functions, parameter variables and common variables. Any data declarations (eg. REAL, DIMENSION, COMMON) in an internal subprogram create variables local to that subprogram, and any variables used that do not exist in the external subprogram are local. Statement labels are local.

An external subprogram may have many internal subprograms, which may not be referenced by any other external subprogram except as parameters. Internal subprograms can call others within the same external subprogram but cannot reference each other's data except through COMMON blocks and parameters. Data used in an external subprogram cannot be referenced by another external subprogram except through COMMON blocks and parameters.

A main program must appear first in a file element, and the last subprogram in an element must finish with an END statement. (Note : to use @FTN,C all subprograms must be in the same element.)

BLOCK DATA is an external subprogram which cannot contain any internal subprogams. Functions and subroutines which have been specified as EXTERNAL may be internal or external subprograms.

Example :

```
C     main program
         .
         .
      FUNCTION A        (internal to main program)
         .
         .
      SUBROUTINE B      (internal to main program)
         .
         .
      END
      FUNCTION C        (external)
         .
         .
      SUBROUTINE D      (internal to function C)
         .
         .
      END
      SUBROUTINE E      (external)
         .
         .
      END
```

22.3. Hollerith Data Types

1966 Fortran did not have type CHARACTER, and a character constant was not enclosed in quotes. Characters were stored in integer, real, logical and double precision variables, and constants were Hollerith constants. Univac Ascii Fortran has retained this data type for compatibility with 1966 Fortran.

A Hollerith constant has the form:

```
        nHstring
```

where n is the number of characters in the string.
Examples:

```
         3HWOW
         6Hthat's            (equivalent to 'that''s')
        11HUPPER/lower
```

Hollerith constants may be assigned to character variables as well as to integer, real, logical and double precision variables (but type Hollerith is distinct from type CHARACTER). 4 characters fit into an integer, real or logical variable, and 8 into a double precision variable. On input and output, A4 format is required for characters in integer, real and logical variables, and A8 format is required for characters in double precision variables. Example:

```
      DOUBLE PRECISION TITLE
      CHARACTER*3 TAIL
      TITLE = 8HHEADING
      TAIL = 'END'
      WRITE (6,10) TITLE,TAIL
   10 FORMAT (1X,A8/1X,A3)
```

1977 Fortran made it illegal to compare a character expression with anything other than another character expression. In Univac Ascii Fortran, a Hollerith constant may only be compared with another Hollerith constant or an expression of type character (ie character constant, character variable, or combination of these).

23. APPENDIX 9 - Answers to selected exercises

23.1. CHAPTER 2

23.1.1. Exercise 2A

```
(i)    Z = (A+B)/(C+D)

(ii)   Z = A + B/(C+D)

(iii)  Z = (A+B)/C + D

(iv)   Z = A + B/(C+D/E)

(v)    Z = N*(N-1)/2

(vi)   Z = (A-B)*(C-D) / (E*(F+G))

(viii) Y = -2.314 + (5.67*Z - 3.29E-4)*Z**3 + 4.13*Z**7
```

23.1.2. Exercise 2B

(a) REAL

(b) REAL

(c) INTEGER

(d) NEITHER

(e) REAL

(f) REAL

(g) NEITHER

(h) NEITHER

(i) NEITHER

(j) NEITHER

(k) REAL

23.1.3. Exercise 2C

(a) REAL

(b) ILLEGAL

(c) REAL

(d) ILLEGAL

(e) ILLEGAL

(f) INTEGER

(g) ILLEGAL

(h) REAL

(i) ILLEGAL

(j) ILLEGAL

(k) REAL

(l) REAL

23.1.4. Exercise 2D

(i) 3B is an illegal variable name

(ii) There is a mismatch of open and close brackets

(iii) 3.14159 is an illegal variable name

(iv) Adjacent operators ** and -

(v) Only 6 characters are allowed for variable names.

(vi) X+Y is an illegal variable name.

(vii) Missing operator between the two bracketted expressions

23.1.5. Exercise 2E

(i) Z becomes-1.0666667 (approximately)

(ii) Z becomes 4.0 (note that this is NOT 4, but 4.0)

(iii) Z becomes 1.0

(iv) Z becomes 6.0

(v) Z becomes 1.0

(vi) K becomes 3 (note that this is NOT 3.0, but 3)

(vii) K becomes 2

(viii) K becomes 1

(ix) K becomes 6

(x) K becomes -1

(xi) K becomes 0

23.2. CHAPTER 3

23.2.1. Exercise 3A

```
C
C   AUTHOR:   LESLIE LANDAU
C   DATE:     OCTOBER 1977
C   INPUT DESCRIPTION:  FREE FORMAT, 2 REALS 2 INTEGERS
C   PURPOSE:
C     TO READ IN 4 NUMBERS AND TO WRITE THEM OUT IN REVERSE
C
      READ(5,*)VAL1,VAL2,NUM3,NUM4
      WRITE(6,*)NUM4,NUM3,VAL2,VAL1
      STOP
      END
```

23.2.2. Exercise 3B

```
C AUTHOR:     LESLIE LANDAU
C DATE  :     OCTOBER 1977
C INPUT :     TWO INTEGERS IN FREE FORMAT
C PURPOSE:
C           TO ADD, MULTIPLY, DIVIDE THE TWO INTEGERS
C           AND TO RAISE THE FIRST TO THE POWER OF
C           THE SECOND
C
      READ(5,*)I,J
      WRITE(6,*)' ORIGINAL INPUT IS ',I,J
      K = I+J
      L = I*J
      M = I/J
      N = I**J
```

```
C
C  WRITE OUT RESULTS
C
      WRITE(6,*)' ADDITION = ',K,' MULT = ',L,' DIV = ',M,
     $          ' POWER = ',N
      STOP
      END
```

23.2.3. Exercise 3C

```
C  AUTHOR:      L. LANDAU
C  DATE  :      OCT  1977
C  INPUT :      A POSITIVE REAL NUMBER IN FREE FORMAT
C  PURPOSE:
C            TO CALCULATE THE INTEGRAL AND FRACTIONAL PARTS
C            OF THE REAL NUMBER READ IN
C
      READ(5,*)VALUE
      INT = VALUE
      FRACT = VALUE - INT
      WRITE(6,*)' THE INTEGRAL PART OF ',VALUE,' IS ',INT,
     $          ' AND THE FRACTIONAL PART IS ',FRACT
      STOP
      END
```

23.2.4. Exercise 3D

```
C  AUTHOR :     L. LANDAU
C  DATE   :     OCT 77
C  INPUT  :     TWO REALS IN FREE FORMAT
C  PURPOSE:     TO CALCULATE THE DIFFERENCE BETWEEN
C               THE SQUARES ( FIRSTSQ - SECONDSQ)
C
      READ(5,*)VAL1,VAL2
      SQ1 = VAL1*VAL1
      SQ2 = VAL2**2
      DIFF= SQ1 - SQ2
      WRITE(6,*)VAL1,' SQUARED IS ',SQ1
      WRITE(6,*)VAL2,' SQUARED IS ',SQ2
      WRITE(6,*)' DIFFERENCE BETWEEN SQUARES = ',DIFF
      STOP
      END
```

23.3. CHAPTER 4

23.3.1. Exercise 4A

(a) Valid
(b) Valid
(c) Invalid (Illegal adjacent operators)
(d) Valid
(e) Invalid (Missing arithmetic expression)
(f) Valid
(g) Invalid (Illegal operator)
(h) Invalid (Illegal operator)

23.3.2. Exercise 4B

```
C
C  AUTHOR:  LES LANDAU
C  DATE:    9 SEP 79
C  INPUT:   ONE CARD CONTAINING TWO INTEGERS IN FREE FORMAT
C  PURPOSE: TO DETERMINE WHICH INTEGER IS THE LARGER, AND WRITE
C           THAT ONE OUT FIRST
C
      READ(5,*) INT1,INT2
      IF (INT1 .GT. INT2) THEN
          WRITE(6,*)' NUMBERS ARE ',INT1,INT2
      ELSE
          WRITE(6,*)' NUMBERS ARE ',INT2,INT1
      END IF
      STOP
      END
```

23.3.3. Exercise 4C

```
      READ (5,*) MAXIN
      IF (MAXIN .EQ. 10) THEN
          FORGET = 0.0
      ELSE IF (MAXIN .EQ. 16) THEN
          WRITE (6,*) 'Gotit'
          STOP
      ELSE IF (MAXIN .EQ. 19) THEN
          FORGET = 0.5
          WRITE (6,*) ' Found one'
      END IF
```

23.3.4. Exercise 4D

```
    IF (A.GT.-0.00001 .AND. A.LT.0.00001) A = 0.0
```

23.3.5. Exercise 4E

```
      IF (IND .EQ. 16) THEN
          K = 6
          I = 9
      ELSE IF (L .GT. J+4) THEN
          X = 19.6
          READ (5,*) Y
          L = 0
      ELSE IF (MAIN .LT. I) THEN
          COST = 19.0
          TRY  = 14.2
      ELSE
          PAY = 0.0
      END IF
```

23.4. CHAPTER 5

23.4.1. Exercise 5A

(i)	FALSE
(ii)	TRUE
(iii)	FALSE
(iv)	FALSE
(v)	TRUE
(vi)	FALSE
(vii)	FALSE
(viii)	TRUE
(ix)	FALSE
(x)	FALSE

23.4.2. Exercise 5B

(i)	CORRECT	
(ii)	INCORRECT	(statement label is a variable)
(iii)	CORRECT	
(iv)	INCORRECT	(DO-variable cannot be a number)
(v)	CORRECT	
(vi)	CORRECT	(real number is truncated to an integer)
(vii)	CORRECT	

(viii) CORRECT

(ix) CORRECT (1, N4A2 and 2 are converted to real numbers)

(x) CORRECT

(xi) INCORRECT (missing statement label)

(xii) CORRECT (but confusing)

23.4.3. Exercise 5C

(a) 14
(b) 0, 4, 8, 12, 17, 21, 25, 29

23.4.4. Exercise 5D

```
C
C   AUTHOR:     LES LANDAU
C   DATE:       9 SEP 79
C   INPUT:      THERE IS NO INPUT
C   PURPOSE:    TO ADD UP ALL THE EVEN INTEGERS BETWEEN 98 AND 224
C               INCLUSIVELY, AND TO WRITE OUT THE TOTAL.
C
      ITOTAL = 0
      DO 5 NUM = 98,224,2
         ITOTAL = ITOTAL + NUM
    5 CONTINUE
C
C   WRITE OUT THE RESULTANT TOTAL
C
      WRITE(6,*) ' SUM OF EVEN INTEGERS FROM 98 TO 224 = ', ITOTAL
      STOP
      END
```

23.4.5. Exercise 5E

```
C
C AUTHOR:    LES LANDAU
C DATE:      9 SEP 79
C INPUT:     FIRST LINE:
C                THIS CONTAINS AN INTEGER IN FREE FORMAT WHICH
C                INDICATES THE NUMBER OF LINES TO FOLLOW
C            SUBSEQUENT LINES:
C                CONTAIN A REAL NUMBER (FREE FORMAT)
C
C PURPOSE:    TO DETERMINE THE LARGEST AND SMALLEST NUMBER
C
C RESTRICTION:  THERE MUST BE AT LEAST ONE NUMBER
```

```
C
C
C     READ IN HOW MANY NUMBERS THERE ARE
C
      READ(5,*) NUM
      IF (NUM .LT. 1) THEN
          WRITE(6,*)' YOU NEED AT LEAST ONE NUMBER!!'
          STOP
      END IF
C
C     READ IN THE FIRST NUMBER AND SAY ITS BOTH THE BIGGEST AND
C     THE SMALLEST.
C
      READ(5,*) VAL
      BIG = VAL
      SMALL = VAL
C
C    NOW PROCESS THE REMAINING NUM-1 NUMBERS
C
      DO 5 L = 2,NUM
         READ(5,*) VAL
         IF(VAL.GT.BIG) BIG = VAL
         IF(VAL.LT.SMALL) SMALL = VAL
    5 CONTINUE
C
C     WRITE OUT RESULTS
C
      WRITE(6,*) 'THE LARGEST  NUMBER WAS ',BIG
      WRITE(6,*) 'THE SMALLEST NUMBER WAS ',SMALL
      STOP
      END
```

23.4.6. Exercise 5F

(i) 12
(ii) 12
(iii) 9
(iv) 15
(v) 4

23.4.7. Exercise 5G

44 J is incremented to 44, then tested against 38, and control is passed to the WRITE statement.

23.4.8. Exercise 5H

```
C
C   AUTHOR : K. HANDEL
```

```
C   DATE   : 28 JUNE 81
C   INPUT  : (1)  AN INTEGER IN FREE FORMAT INDICATING THE NUMBER
C                 OF VALUES TO FOLLOW
C            (2)  THE VALUES, REAL NUMBERS, ONE PER LINE
C
C   PURPOSE : FIND THE AVERAGE OF ANY NUMBER OF NUMBERS READ IN
C             FROM INPUT
C   RESTRICTION : THERE MUST BE AT LEAST ONE NUMBER
C

      TOTAL = 0

      READ (5,*) NUM
      DO 100 I = 1,NUM
         READ (5,*) VAL
         TOTAL = TOTAL + VAL
  100 CONTINUE

      AVER = TOTAL/NUM
      WRITE (6,*) 'AVERAGE IS  ',AVER
      STOP
      END
```

23.5. CHAPTER 6

23.5.1. Exercise 6A

```
(i)    ANS = SQRT(B*B - 4.0*A*C)
(ii)   ANS = SIN(2.4)
(iii)  IANS= MAX(J,LARG)
(iv)   ANS = MAX(A,BIG)
(v)    ANS = ABS(ECC)
(vi)   IANS= ABS(I)
       JANS= ABS(M)
       ANS = ABS(A)
(vii)  ANS = REAL(KKK)
(viii) ANS = SQRT(REAL(INT))
(ix)   ANS = MAX(REAL(I),X)
```

23.5.2. Exercise 6B

```
C  AUTHOR  :      L.LANDAU
C  DATE    :      7TH OCTOBER 1977
C  INPUT   :      ONE INTEGER IN FREE FORMAT
C  PURPOSE :      TO CALCULATE THE FACTORIAL OF THE INTEGER
C                 READ IN
C  RESTRICTIONS:
C                 THE UNIVAC COMPUTER CAN ONLY REPRESENT
```

```
C                 INTEGERS UP TO 2**35 -1 AND SO THE
C                 MAXIMUM FACTORIAL IS 13!
C
      READ(5,*)NUM
C
C  STOP THE PROGRAM IF NUM IS TOO BIG
C
      IF (NUM .GT. 13) THEN
          WRITE(6,*)NUM,' IS TOO BIG, 13 IS MAX'
          STOP
      END IF
      IFACT = 1
      DO 5 MULT = 1,NUM
         IFACT = IFACT*MULT
 5    CONTINUE
C
C  OUTPUT  RESULTS
C
      WRITE(6,*)' THE FACTORIAL OF ',NUM,' IS ',IFACT
      STOP
      END
```

23.5.3. Exercise 6C

```
(i)   M = MOD(IJ,K)
(ii)  SMALL = MIN(A,B)
(iii) WARM  = REAL(ION)/REAL(KAN)
(iv)  ADAM  = AMOD(VAL,WORLD)
```

23.6. CHAPTER 7

23.6.1. Exercise 7A

```
C
C  AUTHOR:   L. LANDAU
C  DATE:     9 SEP 79
C  INPUT:   THREE REAL NUMBERS AND ONE INTEGER
C           IN FREE FORMAT. INPUT FOR VARIABLES
C           A, B, P, K  (RESP)
C
C  PURPOSE:  TO DO ONE OF THREE CALCULATIONS
C            DEPENDING ON K BEING -VE,0,+VE
C
      READ(5,*)A, B, P, K
      IF(K.LT.0) Y = SQRT(A*A + SIN(P)**2)
      IF(K.GT.0) Y = SQRT(B*B - SIN(P)**2)
      IF(K.EQ.0) Y = SQRT(A*A + B*B)
C
```

```
C  WRITE OUT RESULTS
C
      WRITE(6,*) 'PARAMETERS READ IN : A = ',A,
     $              'B = ',B,'P = ',P,'K = ',K

      WRITE(6,*) 'CALCULATED VALUE, Y = ',Y
      STOP
      END
```

23.6.2. Exercise 7B

```
C
C  AUTHOR:    L. LANDAU
C  DATE:      9 SEP 79
C  INPUT:     ONE REAL NUMBER IN FREE FORMAT
C             CORRESPONDING TO X
C  PURPOSE:   TO EVALUATE  THE SUM OF THE
C             SERIES:
C                        N
C                      COS (X)
C                     ---------
C                        N!
C
C            FOR N = 0 TO 20
C            THIS METHOD SUMS THE SERIES IN
C            A FORWARD DIRECTION
C   NOTE:
C        TYPE REAL ARITHMETIC MUST BE USED
C        FOR FACTORIAL CALCULATION
C
      READ(5,*)X
      FACT = 1.0
      SUM  = 1.0
      CX   = COS(X)
      TERM = 1.0
C
C  SUM THE SERIES
C
      DO 5 NTERMS = 1,20
         FACT = FACT * NTERMS
         TERM = TERM * CX
         SUM  = SUM + TERM/FACT
5     CONTINUE
C
C  WRITE OUT RESULTS
C
      WRITE(6,*) 'SUM OF SERIES USING ',X,' AS VALUE FOR X IS ',SUM
      STOP
      END
```

Note that the factorial becomes large, and for a larger N would overflow. Can you suggest another formula for calulating SUM which does not require FACT to be stored?

23.6.3. Exercise 7C

```
C  AUTHOR:   K.HANDEL
C  DATE:     5 AUGUST 1981
C  INPUT:    NONE
C
C  PURPOSE: CALCULATE E TO THE POWER OF X
C           FOR X RANGING FROM 1 TO 100,
C           USING THE EXP FUNCTION.
C  RESTRICTION: X WILL BECOME TOO LARGE FOR EXP
C               AT SOME STAGE. X COULD BE MADE
C               INTO DOUBLE PRECISION TO EXTEND
C               THE RANGE.

      INTEGER X
      DO 10 X = 1,100
         WRITE (6,*) X,EXP(X)
   10 CONTINUE
      STOP
      END
```

23.7. CHAPTER 8

23.7.1. Exercise 8A

(i) There is no field descriptor in the FORMAT corresponding to the variable K. This may be fixed by putting in a field descriptor for K or by eliminating K from the WRITE statement.

(ii) (a) GATHER requires an F field descriptor
(b) MOSS requires an I field descriptor
(c) The field descriptor F3.5 doesn't make sense.

(iii) (a) There is no terminal statement for the DO

(b) There is no point in setting I to 17 outside the DO as we read in a value for it within the DO.
(c) We cannot read in a value for K within the DO as K is the control variable (and we are no allowed to change it).
(d) As F is not used within the loop, its value will be overwritten each time around the loop.
(e) The writing out of I requires an I field descriptor.
(f) The construction .=. is not allowed and should be .EQ.

23.7.2. Exercise 8B

There will be 4 lines printed (including blank lines).

23.7.3. Exercise 8C

```
      0   10 -58790062*****
     .00  16.77-586.21   5.48 131.10
```

23.7.4. Exercise 8D

```
  (a) 676.7
  (b) 6381
  (c)      132.6
  (d) ****          (need F5.2 to get 12.16)
  (e)      99999
```

23.7.5. Exercise 8E

```
C
C AUTHOR:    LES LANDAU
C DATE:      9 SEP 79
C INPUT:     TWO REAL NUMBERS IN FREE FORMAT
C            THE FIRST IS THE LENGTH OF CUBE A
C            THE SECOND IS THE SMALLEST SIDE OF BLOCK B
C
C PURPOSE:   TO CALCULATE THE SURFACE AREAS OF CUBE A AND
C            BLOCK B, AND ALSO TO FIND THE LARGER ONE AND THE
C            DIFFERENCE IN AREA
C METHOD OF CALCULATION:
C            THE AREA OF A CUBE IS 6 TIMES THE SQUARE OF THE WIDTH
C            THE AREA OF THE BLOCK B IS GIVEN BY:
C            2*2K*3K + 2*2K*K + 2*3K*K
C            WHERE K IS THE LENGTH OF THE SHORTEST SIDE
C
      READ(5,*) H, BK
      AREAA = 6 * H*H
      BKH = BK*2
      BKL = BK*3
      AREAB = 2*BKH*BKL + 2*BKH*BK + 2*BKL*BK
      DIFF  = ABS(AREAA-AREAB)
      BIG   = MAX(AREAA,AREAB)
C
C  NOW WRITE OUT THE RESULTS
C
      WRITE(6,100)
 100  FORMAT('1','LES LANDAU',///
     $       52X,'COMPARISON OF SURFACE AREAS'///)
```

```
      WRITE(6,101) H, BK, BKH, BKL, AREAA, AREAB
 101  FORMAT(1X,'SIDE OF CUBE A:',5X,F10.2,/
     $       1X,'WIDTH OF BLOCK B:',3X,F10.2/
     $       1X,'HEIGHT OF BLOCK B:',2X,F10.2/
     $       1X,'LENGTH OF BLOCK B:',2X,F10.2//
     $       1X,'SURFACE AREA OF A:',2X,F10.2/
     $       1X,'SURFACE AREA OF B:',2X,F10.2/)
      WRITE(6,102) DIFF,BIG
 102  FORMAT(1X,'DIFFERENCE IN SURFACE AREA: ',3X,F10.2//
     +       1X,'LARGER SURFACE AREA: ',3X,F10.2)
      STOP
      END
```

23.7.6. **Exercise 8F**

```
C
C  AUTHOR:  L: LANDAU
C  DATE:    9 SEP 79
C  INPUT:   THERE IS NO INPUT
C  PURPOSE: TO PRINT OUT THE SINE, COSINE
C           TANGENT, SECANT, COSECANT, COTANGENT
C           OF EVERY INTEGRAL ANGLE FROM
C           1 TO 89 DEGREES

      WRITE(6,10)
 10   FORMAT('1','ANGLE',4X,'SINE',6X,'COSINE',
     $       6X,'TANGENT',7X,'SECANT',7X,'COSECANT',
     $       6X,'COTANGENT')

C  CALCULATE THE VALUE OF PI FROM ATAN FUNCTION

      PI = 4.0 * ATAN(1.0)
      DEGRAD = PI/180.0
      DO 15 IDEG = 1,89

C FIND THE TRIG FUNCTIONS

         ANGLE = IDEG * DEGRAD
         S    = SIN(ANGLE)
         CS   = COS(ANGLE)
         T    = TAN(ANGLE)
         SEC  = 1.0/CS
         COSC= 1.0/S
         COT  = 1.0/T

C WRITE  OUT  RESULTS

         WRITE(6,20)IDEG,S,CS,T,SEC,COSC,COT
 15   CONTINUE
 20   FORMAT(1X,3X,I2,3X,F7.4,4X,F7.4,4(4X,F10.4))
      STOP
      END
```

23.8. CHAPTER 9

23.8.1. Exercise 9A

(i) A type F field descriptor is required for reading in a value for A

(ii) There is no such relational operator as .GRT.
A field descriptor of I5.1 does not make sense.
The format for writing out J should contain an I field descriptor.

(iii) Finishing a DO statement on a FORMAT statement is not a good practice.
Cannot divide by zero (J/K is trying to do this)

23.8.2. Exercise 9B

(a) 3
(b) 2

23.8.3. Exercise 9C

```
           I    WUNDA     WOT     IT        WILL    BE

 (a)     123  456.127    98.0     12        45.6  78.0
 (b)   12345   6.1279     8.0  12004       567.8   0.1
 (c)       1     23.4    56.1    279  8.00012004  5.67
```

23.9. CHAPTER 10

23.9.1. Exercise 10A

(a) The array J is dimensioned to size 20 and yet the DO loop will index up to 100
(b) The left hand side of the third line should have a subscript, as the array ARRAY must always appear with a subscript.
(c) - The variable I is being used as both an array AND as a simple variable. This is not allowed.
- ARRAY(I-1) when I is 1 will be referencing ARRAY(0) which is illegal.

23.9.2. Exercise 10B

```
   C  AUTHOR:    K. HANDEL
   C  DATE:      5 AUGUST 1981
   C  INPUT:     Rainfall data for 8 localities, one line per locality.
   C             12 real numbers per line, in free format,
   C             representing the rainfall for each of the 12 months
   C             for a locality.
   C  PURPOSE:   calculate average rainfall for each of 8 localities.

         REAL RAIN(12)
         DO 40 LOCAL = 1,8
            READ (5,*) RAIN
            AVE = 0.0
            DO 20 MONTH = 1,12
               AVE = AVE + RAIN(MONTH)
      20    CONTINUE
            WRITE (6,*) 'Average rainfall for locality',LOCAL,
        $                ' is', AVE/12.0
      40 CONTINUE
         STOP
         END
```

23.9.3. Exercise 10C

(i) correct
(ii) correct
(iii) incorrect
(iv) correct

23.9.4. Exercise 10D

(a) X appears in a DIMENSION statement and so should have a subscript in the second line

(b) no error

(c) Cannot have a REAL variable as a subscript.

(d) A variable may not appear twice on a DIMENSION statement.

The maximum subscript for T is 6

23.9.5. Exercise 10E

```
      DIMENSION B(100)
      DO 5 LOC = 1,100
      B(LOC) = 0.0
5   CONTINUE
```

23.9.6. Exercise 10F

```
C
C  AUTHOR:  L  LANDAU
C  DATE  :  SEPT 1979
C  INPUT DESCRIPTION:
C   FIRST CARD:
C     CONTAINS AN INTEGER (IN FREE FORMAT)
C     WHICH INDICATES THE NUMBER OF CARDS
C     TO FOLLOW.
C
C   FOLLOWING CARDS:
C     COLUMNS      TYPE       MEANING
C       1-2       INTEGER     IDENTIFICATION NUMBER
C       3-4       INTEGER     NUMBER OF EGGS LAID
C                             THIS MONTH
C       5-8       INTEGER     FEED CONSUMED (GRAMS)
C       9-12      INTEGER     WEIGHT OF THE BIRD (GRAMS)
C
C  PURPOSE:
C  --------
C        READ IN HEN DATA AND PRINT OUT FOR EACH HEN
C       (A) IDENT
C       (B) EGGS LAID AND DIFFERENCE FROM AVERAGE
C       (C) FEED EATEN,   DIFFERENCE FROM AVERAGE
C       (D) BIRD WEIGHT,  DIFFERENCE FROM AVERAGE
C
C        THERE IS A LIMIT OF 50 HENS

      DIMENSION IDENT(50),LAYD(50),IFEED(50)
      DIMENSION IWEIGH(50)
C
C  READ IN NUMBER TO PROCESS AND ENSURE < 50
C
      READ(5,*)NUM
      IF(NUM.GT.50) THEN
         WRITE(6,*)' TOO MANY HENS, MAX IS 50. RECOMPILE',
     $             ' PROGRAM WITH LARGER ARRAYS'
         STOP
      END IF
      ITOTEG = 0
      ITOTFD = 0
      ITOTWT = 0
C
C  READ IN HEN DATA AND COMPUTE TOTALS
C
      DO 5 IHEN = 1,NUM
         READ (5,10) IDENT(IHEN),LAYD(IHEN),IFEED(IHEN),
     $               IWEIGH(IHEN)
         ITOTEG = ITOTEG + LAYD(IHEN)
         ITOTFD = ITOTFD + IFEED(IHEN)
         ITOTWT = ITOTWT + IWEIGH(IHEN)
  5   CONTINUE
 10   FORMAT(I2,I2,I4,I4)

C  WRITE OUT RESULTS
C
      WRITE(6,15)
 15   FORMAT('1','HEN ANALYSIS'/
     $        1X,11X,'EGGS    DIFFERENCE   FEED    ',
     $        'DIFFERENCE    BIRD    DIFFERENCE'/
     $        1X,' IDENT',5X,'LAID',4X,'FROM  AVE',
     $        4X,'EATEN   FROM  AVE',5X,'WEIGHT',
     $        3X,'FROM  AVE')
      AVEEG  = REAL(ITOTEG)/NUM
      AVEFD  = REAL(ITOTFD)/NUM
      AVEWT  = REAL(ITOTWT)/NUM
      DO 25 IHEN = 1,NUM
         DIFEG = LAYD(IHEN) - AVEEG
         DIFFD = IFEED(IHEN)- AVEFD
         DIFWT = IWEIGH(IHEN)-AVEWT
         WRITE (6,20) IDENT(IHEN),LAYD(IHEN),DIFEG,
     $                IFEED(IHEN),DIFFD,
     $                IWEIGH(IHEN),DIFWT
 20      FORMAT(2X,I4,5X,I4,4X,F8.1,5X,I4,4X,F8.1,
     $          6X,I4,4X,F8.1)
 25   CONTINUE
C
C  WRITE OUT TOTALS AND AVERAGES
C
      WRITE(6,30)ITOTEG,ITOTFD,ITOTWT
 30   FORMAT(//1X,'TOTALS:  ',I5,16X,I5,17X,I5)
      WRITE(6,35)AVEEG,AVEFD,AVEWT
 35   FORMAT(1X,'AVERAGE:  ',F6.1,15X,F6.1,16X,F6.1)
      STOP
      END
```

23.10. CHAPTER 11

23.10.1. Exercise 11A

(a)	.000+000	(d)	.100+002	(g)	.270-003
(b)	.323+003	(e)	-.421-003	(h)	-.663+003
(c)	.443+012	(f)	.100-002	(i)	.300-023

23.10.2. Exercise 11B

```
C
C  AUTHOR:  L  LANDAU
C  DATE  :  SEPT 1979
C  INPUT DESCRIPTION:
C        THERE IS NO INPUT
C
C  PURPOSE:
C        TO GENERATE THE NUMBERS 1 TO 30 IN AN ARRAY
C        AND TO WRITE OUT THE ARRAY
C        (A) ON ONE LINE
C        (B) SPREAD OVER 5 LINES WITH LINE NUMBERS
C
      DIMENSION NUM(30)
C
C  PUT NUMBERS INTO NUM AND WRITE IT OUT
C
      DO 5 I = 1,30
         NUM(I) = I
  5   CONTINUE
      WRITE(6,10)(NUM(I),I=1,30)
 10   FORMAT(1X,'THE NUMBERS ARE'/
     +       '0',30(I3,1X))
C
C  NOW WRITE OUT THE ARRAY OVER 5 LINES
C PREFIXING EACH LINE WITH A LINE NUMBER
C
      IST = 1
      IFIN= IST+5
      DO 20 I = 1,5
         WRITE(6,15)I,(NUM(K),K = IST,IFIN)
 15      FORMAT(1X,I5,6I5)
         IST = IFIN+1
         IFIN= IFIN+6
 20   CONTINUE
      STOP
      END
```

23.10.3. Exercise 11C

```
 C    AUTHOR:  LES LANDAU
 C    DATE:    25TH MARCH 1981

 C    INPUT DESCRIPTION:
               NO INPUT

 C   PURPOSE:
 C        TO GENERATE NUMBERS IN A TWO DIMENSIONAL ARRAY
 C        OF SIZE 10 BY 3 AND TO WRITE OUT THE ARRAY:
 C          (A) SPREAD OVER ONE LINE
 C          (B) SPREAD OVER 5 LINES, WITH LINE NUMBERS
```

```
      DIMENSION NUM (10,3)
C   PUT NUMBERS INTO THE ARRAY BY ROWS
      KNT = 1
      DO 20 I = 1,10
      DO 10 J = 1,3
         NUM (I,J) = KNT
         KNT = KNT + 1
 10   CONTINUE
 20   CONTINUE
C   NOW WRITE THEM OUT ON ONE LINE
      WRITE (6,30) ((NUM(I,J), J = 1,3), I = 1,10)
 30   FORMAT (1X, 'THE NUMBERS ARE'//
     $        1X, 30(I3,1X))
C   NOW WRITE THEM OUT OVER 5 LINES
      IR = 1
      DO 50 LINE = 1,5
         WRITE (6,40) LINE, ((NUM(I,J),J=1,3),I=IR,IR+1)
 40      FORMAT (1X,I5,6I5)
         IR = IR + 2
 50   CONTINUE
      STOP
      END
```

23.10.4. Exercise 11D

```
C  Add M x N arrays A and B together and store in M x N
C  array C. Max M is 14, max N is 10.
C  This is just a series of statements, not a full program.

      REAL A(14,10), B(14,10), C(14,10)
      DO 100 I = 1,M
      DO 100 J = 1,N
         C(I,J) = A(I,J) + B (I,J)
  100 CONTINUE
```

23.10.5. Exercise 11E

```
C  AUTHOR:    K. HANDEL
C  DATE:      5 AUG 1981
C  INPUT:     CARD1 - an integer N, in free format, representing
C             the number of dimensions in space.
C             CARD2 - co-ordinates of a point in N-dimensional
C                     space, in free format.
```

```
C            Further cards each contain co-ordinates of a point.
C  PURPOSE:  Find the distance from the origin of points in
C            N-dimensional space. N is read from input, then
C            the co-ordinates of the points are read from input.
C  RESTRICTION: N, the number of dimensions in space, is less
C               than or equal to 25.

      REAL POS(25)
      READ (5,*) N
      IF (N.GT.25) THEN
         WRITE (6,*) 'Max number of dimensions is 25. ',
     $               'Number read is',N
         STOP
      END IF
  100 READ (5,*,END=140) (POS(I),I = 1,N)
      DIST = 0
      DO 120 I = 1,N
         DIST = DIST + POS(I)**2
  120 CONTINUE
      WRITE (6,*) 'Distance of (',(POS(I),I = 1,N), ') ',
     $            'from origin is ', SQRT(DIST)
      GO TO 100
  140 STOP
      END
```

23.10.6. Exercise 11F

```
C  AUTHOR:  K. HANDEL
C  DATE:    5 AUGUST 1981
C  INPUT:   Rainfall data for 8 localities, one line per locality.
C           12 real numbers per line, in free format,
C           representing the rainfall for each of the 12 months
C           for a locality.
C  PURPOSE: Calculate average rainfall for each of 8 localities,
C           and also the entire rainfall figures for the month
C           with the highest average.

      REAL RAIN (8,12)

      DO 40 LOCAL = 1,8
         READ (5,*) (RAIN(LOCAL,MONTH),MONTH = 1,12)

C  Find the average rainfall for locality LOCAL.

         AVE = 0.0
         DO 20 MONTH = 1,12
            AVE = AVE + RAIN (LOCAL, MONTH)
   20    CONTINUE
         WRITE (6,*) 'Average rainfall for locality', LOCAL,
     $               ' is', AVE/12.0
   40 CONTINUE

C Find month with highest rainfall (HRAIN).
```

```
      HRAIN = -1
      DO 80 MONTH = 1,12
         TOT = 0.0
         DO 60 LOCAL = 1,8
            TOT = TOT + RAIN (LOCAL,MONTH)
   60    CONTINUE
         IF (TOT .GT. HRAIN) THEN
            HRAIN = TOT
            MAXMTH = MONTH
         END IF
   80 CONTINUE
      WRITE (6,*) ' Rainfall for month ',MAXMTH,', which had the '
     $  'highest average, was', (RAIN(LOCAL,MAXMTH), LOCAL = 1,8)
      STOP
      END
```

23.11. CHAPTER 12

23.11.1. Exercise 12A

A) OK
B) OK
C) ILLEGAL (MISMATCH OF PARAMETER TYPES)
D) ILLEGAL (WRONG NUMBER OF PARAMETERS)
E) ILLEGAL (MISMATCH OF PARAMETER TYPES)
F) THIS WOULD WORK, BUT CONFUSING AND SHOULD NOT BE USED.
G) LEGAL, BUT COULD CAUSE AN ERROR ACCESSING OUT OF BOUNDS
IN ARRAYS CT AND SUNNY.

23.11.2. Exercise 12B

```
      SUBROUTINE AVERNZ(JA,N,AVER,NZ)
C AUTHOR:        K. HANDEL
C DATE:          5 AUG 1981
C INPUT:         None
C PURPOSE:       Find the average of the first N elements of
C                an integer array and the number of those
C                elements which are zero.
C PARAMETERS:
C           INPUT:     JA - array of length N (INTEGER)
C                       N - dimension of JA  (INTEGER)
C          OUTPUT:   AVER - average of first N elements
C                           of JA (REAL)
C                      NZ - number of zeros in first N
C                           elements of JA (INTEGER)

      INTEGER JA(N)
      NZ = 0
      AVER = 0.0
      DO 10 I = 1,N
         AVER = AVER + JA(I)
         IF (JA(I) .EQ. 0) NZ = NZ + 1
   10 CONTINUE
      AVER = AVER/N
      RETURN
      END
```

23.11.3. Exercise 12C

```
      SUBROUTINE READA (A,NROWS,NCOLS)

C  AUTHOR:        K.HANDEL
C  DATE:          5 AUG 1981
C  INPUT:         NROWS lines of data, each containing
C                 NCOLS real numbers in free format.
C  PURPOSE:       read data into array A.
C  PARAMETERS:
C       INPUT:   NROWS = number of rows
C                NCOLS = number of columns
C      OUTPUT:   A     = real array, dimensions NROWS x NCOLS.

      REAL A(NROWS,NCOLS)

      DO 10 IROW = 1,NROWS
         READ (5,*) (A(IROW,ICOL), ICOL = 1,NCOLS)
   10 CONTINUE
      RETURN
      END
```

23.11.4. Exercise 12D

```
      SUBROUTINE TRIG (ANGLE,IDEGS,MINS,ISECS)

C  AUTHOR:        K. HANDEL
C  DATE:          5 AUG 1981
C  INPUT:         None
C  PURPOSE:       Convert an angle in degrees (real number)
C                 into degrees, minutes and seconds (integers).
C                 Seconds are rounded to the nearest integer.
C  PARAMETERS:
C       INPUT:    ANGLE - angle in degrees (REAL)
C      OUTPUT:    IDEGS,MIN,ISECS - integer number of degrees,
C                 minutes and seconds.

      IDEGS = ANGLE

C  After subtracting the whole degrees, convert the remainder
C  to minutes (REAL).
```

```
      REM = (ANGLE - IDEGS) * 60
      MINS = REM

C  After subtracting the whole minutes, convert the remainder
C  to seconds.  0.5 is added before truncating the seconds to
C  an integer, so the effect is rounding to the nearest integer.
      ISECS = (REM - MINS) * 60 + 0.5
      RETURN
      END
```

23.11.5. Exercise 12E

```
C  AUTHOR:         K. HANDEL
C  DATE:           5 AUG 1981
C  INPUT:  (A) the number of numbers to be sorted - an integer,
C              right-justified in columns 1-5.
C          (B) the numbers to be sorted - starting in column 1
C              on a new line, put 5 real numbers per line,
C              10 columns per number. If a number does not
C              have a decimal point, 3 decimal places will be
C              assumed. [ Read in subroutine INPUT ]
C  PURPOSE:    Read a list of real numbers, and sort into
C              descending order.
C  The following subroutines are used:
C      INPUT(ALIST,N) - reads N real numbers into array ALIST.
C     OUTPUT(ALIST,N) - prints the first N elements of ALIST.
C       SORT(ALIST,N) - sorts the first N elements of ALIST
C                       into descending order.
C RESTRICTIONS:  ALIST contains real numbers. To sort a list
C                of integers, change the declarations in the
C                main program and all subroutines.
C                A max of 100 numbers may be sorted. To
C                increase this maximum, change the
C                declaration in the main progam.

      REAL ALIST(100)
      READ (5,10) N
   10 FORMAT(I5)
      CALL INPUT (ALIST,N)
      WRITE (6,20)
   20 FORMAT ('1Original data'///)
      CALL OUTPUT (ALIST,N)
      CALL SORT (ALIST,N)
      WRITE (6,30)
   30 FORMAT('1Sorted data'///)
      CALL OUTPUT (ALIST,N)
      STOP
      END

      SUBROUTINE INPUT (A,N)

C  PURPOSE: Read N real numbers into array A, in 5F10.3 format
C  PARAMETERS: INPUT - N = number of integers (INTEGER)
```

```
C             OUTPUT - A = array of N numbers (REAL)

      REAL A(N)
      READ (5,10) (A(I),I = 1,N)
   10 FORMAT (5F10.3)
      RETURN
      END

      SUBROUTINE OUTPUT (A,N)

C  PURPOSE: Print the first N numbers from array A,
C           double spaced.
C  PARAMETERS:  INPUT - N = number of numbers (INTEGER)
C                       A = array of N numbers (REAL)

      REAL A(N)
      WRITE (6,10) (A(I),I = 1,N)
   10 FORMAT ('0',5F10.3)
      RETURN
      END

      SUBROUTINE SORT (A,N)

C  PURPOSE: Sort the first N elements of array A into descending
C           order, using an Exchange Sort.
C  PARAMETERS: INPUT - N = number of elements to be sorted
C                          (INTEGER).
C                      A = original list (REAL)
C             OUTPUT - A = sorted list

      REAL A(N), ATEMP
      DO 40 I = 1, N-1
         MIN = I
         DO 20 J = I+1, N
            IF (A(J) .LT. A(MIN)) MIN = J
   20    CONTINUE
         IF (MIN .NE. I) THEN
            ATEMP = A (MIN)
            A(MIN) = A(I)
            A(I)  = ATEMP
         END IF
   40 CONTINUE
      RETURN
      END
```

23.11.6. Exercise 12F

```
      X = 1.0
      DO 10 I = 1,10000
         X = (X * 0.1) * 10.0
   10 CONTINUE
      WRITE (6,*) X
      STOP
      END
```

23.12. CHAPTER 13

23.12.1. Exercise 13A

```
      FUNCTION AREA (R)
C  PURPOSE:  to calculate the area of a circle of radius R

C  PARAMETERS:
C    NAME         TYPE     DESCRIPTION
C     R           REAL     radius of the circle

      PI = 4.0*ATAN(1.0)
      AREA = PI * R**2
      RETURN
      END
```

23.12.2. Exercise 13B

```
      FUNCTION FN (X)

C  PARAMETERS:
C     NAME      TYPE  DESCRIPTION
C      X        REAL  A PARAMETER TO THE FUNCTION
C  PURPOSE:
C        TO CALCULATE THE EXPRESSION
C            2                2   1/2
C          X  + (1 + 2X + 3X   )
C
C        A  WARNING MESSAGE WILL BE SENT IF THE SQUARE ROOT
C        CANNOT BE TAKEN (DUE TO NEGATIVE ARGUMENT). IN THIS
C        CASE A VALUE OF ZERO WILL BE RETURNED BY THE FUNCTION.

      VAL = 1 + 2*X +3*X**2
      IF (VAL .LT. 0) THEN
         WRITE (6,*) 'ROOT NEGATIVE FOR VALUE OF X OF ', X,
     $               '  ----ZERO RETURNED----> ERROR!!'
         FN = 0.0
      ELSE
         FN = X**2 + SQRT (VAL)
      END IF
      RETURN
      END
```

USE OF THE FUNCTION WOULD BE

```
 (I)   A = (6.9 + Y) / FN (Y)
 (II)  B = (2.1*Z + Z**4) / FN(Z)
 (III) C = SIN(Y) / FN(Y**2)
 (IV)  D = 1.0 / FN ( SIN(Y) )
```

23.12.3. Exercise 13C

```
      FUNCTION AREA (R,P)
C  AUTHOR:     K. HANDEL
C  DATE        5 AUG 1981
C  INPUT:      None
C  PURPOSE:   If P = 1, calculate area of an equilateral
C                       triangle of side R.
C             If P = 2, calculate area of a square of side R.
C             If P = 3, calculate area of a circle of radius R.
C             Return the result as the function value (REAL).
C  PARAMETERS : INPUT - P = switch to indicate which type of area
C                            is required (INTEGER)
C                         R = length of side or radius (REAL)

      INTEGER P
      REAL R, RSQ
      AREA = 0.0
      RSQ = R * R

C  EQUILATERAL TRIANGLE

      IF (P .EQ. 1) AREA = SQRT (3.0) * RSQ/4.0

C  SQUARE

      IF (P .EQ. 2) AREA = RSQ

C  CIRCLE

      IF (P .EQ. 3) AREA = 3.14159 * RSQ
      RETURN
      END
```

23.12.4. Exercise 13D

```
      FUNCTION  SUM (A,M,N,)

C  AUTHOR:      K. HANDEL
C  DATE:        11 AUGUST 1981
C  INPUT:       None
C  PURPOSE:     Return the sum of the absolute values of all
C               elements in M x N array A.
C  PARAMETERS: A - M x N array (REAL)
C            M,N - integer dimensions of array A  (ROWS,COLUMNS)
C  NO PARAMETER IS ALTERED IN THE FUNCTION.
```

```
      REAL A(M,N)
      SUM = 0.0
      DO 100 I = 1,M
      DO 100 J = 1,N
         SUM = SUM + ABS(A(I,J))
  100 CONTINUE
      RETURN
      END
```

23.12.5. Exercise 13E

```
      FUNCTION  AMAX (A,N)
C  AUTHOR:     K. HANDEL
C  DATE:       11 AUG 1981
C  INPUT:      None
C  PURPOSE:    Return the max value in array A.
C  PARAMETERS:  A - 1-dimensional real array of length N.
C               N - integer dimension of A.
C  NO PARAMETER IS ALTERED IN THE FUNCTION.

      REAL  A(N)
      AMAX = A(1)
      DO 100 I = 2,N
         IF (A(I) .GT. AMAX) AMAX = A(I)
  100 CONTINUE
      RETURN
      END
```

23.12.6. Exercise 13F

```
      FUNCTION  NZ (IARRAY,M,N)

C  AUTHOR:     K. HANDEL
C  DATE:       11 AUGUST 1981
C  INPUT:      None
C  PURPOSE:    Return a count of the number of zeros
C              in M x N array IARRAY.
C  PARAMETERS:  IARRAY - integer array with dimensions M x N.
C                 M    - number of rows in IARRAY.
C                 N    - number of columns in IARRAY.
C  NO PARAMETER IS ALTERED IN THE FUNCTION.

      INTEGER IARRAY(M,N)
      NZ = 0
      DO 100 I = 1,M
      DO 100 J = 1,N
         IF (IARRAY(I,J) .EQ. 0) NZ = NZ + 1
  100 CONTINUE
      RETURN
      END
```

23.13. CHAPTER 14

23.13.1. Exercise 14A

```
C AUTHOR:      K.  HANDEL
C DATE:        11 AUG 1981
C INPUT:       Lines of text. No special end-of-file marker.
C PURPOSE:     For each line of text read in, print HAPPY BIRTHDAY
C              followed by the text.

      CHARACTER TEXT*80, HB*15
      HB = 'HAPPY BIRTHDAY'
   10 READ (5,20,END = 40) TEXT
   20 FORMAT (A)
      WRITE (6,30) HB,TEXT
   30 FORMAT (2(1X,A))
      GO TO 10
   40 STOP
      END
```

23.13.2. Exercise 14B

```
C    AUTHOR:  LES LANDAU

C      DATE:  5TH JULY 1981

C  LANGUAGE:  UNIVAC ASCII FORTRAN LEVEL 9R1

C   PURPOSE:  READ IN LINES OF TEXT AND SEPARATE EACH LINE
C             INTO WORDS, WHERE THE DELIMITERS BLANK, COMMA
C             AND FULL STOP ARE WORD DELIMITERS.
C             TWO DELIMITERS IN A ROW IMPLY THE END OF THE
C             INPUT LINE, AND THEN A NEW INPUT LINE WILL
C             BE SOLICITED.

C INPUT DESCRIPTION:
C             INPUT IS FROM UNIT 5 AND CONSISTS OF LINES OF DATA
C             TERMINATED BY AN END-OF-FILE.

      CHARACTER*80 LINE,WORD

C   READ IN LINES OF DATA AND BREAK EACH INTO WORDS

 50   CONTINUE
      ISTART = 1
      READ (5,60,END=200)LINE
 60   FORMAT (A)

C   COME HERE TO FIND THE NEXT WORD (STARTING FROM ISTART)
```

```
 65   CONTINUE
      CALL GETWD (WORD,LINE,ISTART,LEN)
      IF (LEN .EQ. 0) THEN
         WRITE (6,*) 'LENGTH OF ZERO RETURNED, NEXT IMAGE PLEASE'
         GO TO 50
      ELSE
         WRITE (6,70) WORD (1:LEN)
 70      FORMAT (1X,A)
      END IF
      ISTART = ISTART + LEN + 1
      GO TO 65

C   COME HERE WHEN END OF INPUT HAS BEEN FOUND

 200  CONTINUE
      WRITE (6,*) 'END OF INPUT FOUND'
      STOP

      SUBROUTINE GETWD (WORD, LINE, ISTART, LEN)
      CHARACTER*1 DELIM(3)
      CHARACTER*80 LINE, WORD

C  PARAMETERS:
C   NAME     TYPE  DESCRIPTION
C   WORD      CH   THE WORD FOUND ON THE LINE
C                  INCLUDING THE FOLLOWING DELIMITER
C   LINE      CH   THE LINE TO SEARCH FOR WORDS ON
C   ISTART   INT   THE START POSITION ON THE LINE TO
C                  START SEARCHING FOR A WORD
C   LEN      INT   THE LENGTH OF THE WORD FOUND
C                  (NOT INCLUDING THE FOLLOWING DELIMITER)

      LEN = 0
      DELIM (1) = '.'
      DELIM (2) = ','
      DELIM (3) = ' '
      NDEL      = 3
      IF (ISTART .GT. 80 .OR. ISTART .LT. 1) THEN
          LEN = 0
          RETURN
      END IF

C   LOOK FOR A WORD STARTING FROM ISTART, JUMP OUT OF
C   THE LOOP WHEN A DELIMITER HAS BEEN FOUND

      DO 50 LOOK = ISTART,80
      DO 40 J    = 1,NDEL
         IF (LINE (LOOK:LOOK) .EQ. DELIM(J)) GO TO 60
 40   CONTINUE
 50   CONTINUE

C  COME HERE IF NO DELIMITER HAS BEEN FOUND
```

```
      WORD = LINE (ISTART:)
      LEN = 80 - ISTART + 1
      RETURN

C   COME HERE WHEN A DELIMITER HAS BEEN FOUND

 60   CONTINUE
      WORD = LINE(ISTART:LOOK)
      LEN = LOOK - ISTART
      RETURN
      END
```

23.13.3. Exercise 14C

```
C  AUTHOR:    K. HANDEL
C  DATE:      11 AUG 1981
C  INPUT:     An integer in the range 1-12, repesenting a month.
C  PURPOSE:   Given a month represented by an integer, print the
C             name of the month (abbreviated to 3 characters).
      CHARACTER*3 MONTH(12)

      DATA MONTH/'JAN','FEB','MAR','APR','MAY','JUN',
     $           'JUL','AUG','SEP','OCT','NOV','DEC'/

      READ (5,*) I
      IF (I .GE. 1 .AND. I .LE. 12) WRITE (6,10) MONTH(I)
   10 FORMAT (1X,A3)
      STOP
      END
```

Note: Appendix 7 describes the DATA statement, which is the best way of setting up MONTH.

23.13.4. Exercise 14D

```
C  AUTHOR:      K. HANDEL
C  DATE:        11 AUG 1981
C  INPUT:       An integer in the range 1-7, representing a day
C               of the week.
C  PURPOSE:     Given a day represented by an integer, print
C               the day (in characters).

      CHARACTER*9  DAY(7)

      DATA DAY/'MONDAY   ','TUESDAY  ','WEDNESDAY','THURSDAY ',
     $         'FRIDAY   ','SATURDAY ','SUNDAY   '/

      READ (5,*) J
      IF (J .GE. 1 .AND. J .LE. 7)  WRITE (6,10) DAY (I)
   10 FORMAT (1X,A)
```

```
      STOP
      END
```

Note: The DATA statement is described in Appendix 7.

23.13.5. Exercise 14E

```
C  AUTHOR:     K. HANDEL
C  DATE:       11 AUG 1981
C  INPUT: (A)  Product information for a company.
C              The first line contains the number of products
C              (INTEGER in free format).
C              Then put information on each product on a separate
C              line as follows:-
C              columns 1-5  product code (5 characters)
C              columns 11-3  product description (20 characters)

C              Quotes should not be put around codes or
C              descriptions.

C          (B) Product codes for which descriptions are required.
C              Any number of product codes may be read, one per line
C              in columns 1-5. Input is terminated by end-of-file.
C  PURPOSE:    Read all product information, then for given
C              product codes, print the corresponding descriptions.

C  SUBROUTINES CALLED:  DSPLAY - prints code and description for
C                                a given code.

      CHARACTER CODE*5(50), DESCR*20(50), ACODE*5

C  Read the number of products (N) then read codes and
C  descriptions.

      READ (5,*) N
      IF (N .GT. 50) THEN
         WRITE (6,*) 'No more than 50 products are allowed. N = ',N
         STOP
      END IF

      DO 20 I = 1,N
         READ (5,10) CODE(I), DESCR(I)
   10    FORMAT (A5,5X,A20)
   20 CONTINUE

C  Print heading for next section
      WRITE (6,25)
   25 FORMAT ('1CODE      DESCRIPTION'//)

C  Read product codes for which descriptions are required.
C  Call DSPLAY to print descriptions.

   30 READ(5,40,END = 50) ACODE
   40 FORMAT (A5)
      CALL DSPLAY (ACODE,CODE,DESCR,N)
      GO TO 30
   50 STOP
      END

      SUBROUTINE DSPLAY (ACODE,CODE,DESCR,N)

  C  PARAMETERS:  ACODE - a 5-character product code
  C                CODE - a 1-dimensional array of length N,
  C                       containing 5-character product codes.
  C               DESCR - a 1-dimensional array of length N,
  C               containing 20-character product descriptions.
  C                   N - integer dimension of arrays CODE and DESCR.
  C            NO PARAMETERS ARE ALTERED IN THE SUBROUTINE.

  C  PURPOSE:  Search for ACODE in the list of codes in CODE,
  C            then print it and its corresponding description.
  C            A linear search is used.

      CHARACTER*5 ACODE, CODE(N), DESCR*20 (N)

      DO 10 I = 1,N
         IF (ACODE .EQ. CODE(I)) GO TO 20
   10 CONTINUE
      WRITE (6,*) 'CODE ',ACODE, ' NOT FOUND'
      RETURN
   20 WRITE (6,30) ACODE, DESCR (I)
   30 FORMAT (1X,A5,5X,A20)
      RETURN
      END
```

23.13.6. Exercise 14F

```
  C  AUTHOR:   K. HANDEL
  C  DATE:     11 AUGUST 1981
  C  INPUT:    Integers repesenting days of the year, one per line,
  C            in free format. Input is terminated by end-of-file.
  C  PURPOSE:  Print the day and date in full for each day of 1981
  C            read in. Eg. if input is 328, output is
  C            TUESDAY   24th NOVEMBER   1981

  C  RESTRICTIONS:  only works for 1981 because it uses the facts
  C                 that Jan 1, 1981  is a Thursday, and 1981 is
  C                 not a leap year.

      CHARACTER  MONTH*9(12), DAY*9(7), TH*2(4), YEAR*4
      INTEGER MTHEND(12)

      DATA  MONTH/'JANUARY  ','FEBRUARY ','MARCH    ','APRIL    ',
     $            'MAY      ','JUNE     ','JULY     ','AUGUST   ',
     $            'SEPTEMBER','OCTOBER  ','NOVEMBER ','DECEMBER '/
```

```
      DATA  DAY/'MONDAY   ','TUESDAY  ','WEDNESDAY','THURSDAY ',
     $          'FRIDAY   ','SATURDAY ','SUNDAY   '/

      DATA  TH/'st','nd','rd','th'/, YEAR/'1981'/

C  Day of year of last day in each month, for a non-leap year.

      DATA  MTHEND/31,59,90,120,151,181,212,243,273,304,334,365/

C  Read a day of the year.

   10 READ (5,*,END = 100) IDATE
      IF (IDATE .LE. 0 .OR. IDATE .GT. 365) THEN
         WRITE (6,*) IDATE, ' is an illegal date. Ignored.'
         GO TO 10
      END IF

C  Find the month, M.

      DO 10 M = 1,12
         IF (IDATE .LE. MTHEND(M)) GO TO 20
   10 CONTINUE

C  Find the day of month, MDAY, and appropriate suffix, TH(ITH).

   20 MDAY = IDATE - MTHEND(M - 1)
      ITH = MDAY
      IF (ITH .GE. 4) ITH = 4

C  Find the day of week, IDAY.
C  Jan 1, 1981  is Thursday, ie. 4th day, so add 3 to IDATE.

      IDAY = MOD (IDATE + 3,7)
      IF (IDAY .EQ. 0) IDAY = 7

C  Print the lot and read another date.

      WRITE (6,30) DAY(IDAY), MDAY, TH(ITH), MONTH(M), YEAR
   30 FORMAT (1X,A9,1X,I2,A2,1X,A9,1X,A4)
      GO TO 10
  100 STOP
      END
```

Note: The DATA statement is described in Appendix 7.

23.13.7. Exercise 14G

```
C  AUTHOR:   K.HANDEL
C  DATE:     11 AUG 1981
C  INPUT:    Lines of text containing only digits, commas,
C            plus and minus. Input is terminated by end-of-file.
C  PURPOSE:  Decode the text into integers, and print both the
C            lines of text and the integer values in the text.
```

```
C  FUNCTION CALLED:
C      LENGTH - finds the length of TEXT, excluding
C                blanks at end of line.

      CHARACTER  TEXT*80, CH*1, SIGN*1

   10 READ (5,20,END = 50) TEXT
   20 FORMAT (A)
      WRITE (6,30) TEXT
   30 FORMAT (//1X,A)
      L = LENGTH (TEXT)
      IF (L .EQ. 0) GO TO 10

C  initialize

      SIGN = '+'
      NUM = 0
      DO 40 I = 1,L
         CH = TEXT (I:I)
         IF (CH .EQ. '+' .OR. CH .EQ. '-') THEN
            SIGN = CH

         ELSE IF (CH .GE. '0' .AND. CH .LE. '9') THEN
            NUM = NUM*10 + ICHAR(CH) - ICHAR('0')

         ELSE IF (CH .EQ. ',') THEN
C           correct sign, print and reset number
            IF (SIGN .EQ. '-') NUM = -NUM
            WRITE (6,*) NUM
            NUM = 0
            SIGN = '+'
         ELSE
            WRITE (6,*) CH, ' is an illegal character. Ignored.'
         END IF
   40 CONTINUE

C  Print last number in TEXT unless it was terminated by a comma.

      IF (CH .NE. ',') THEN
         IF (SIGN .EQ. '-') NUM = -NUM
         WRITE (6,*) NUM
      END IF
      GO TO 10
   50 STOP
      END

      FUNCTION LENGTH (TEXT)

C  PURPOSE:  Return length of TEXT when blanks at end are
C            excluded.

      CHARACTER*80 TEXT
      DO 60 I = 80,1,-1
         IF (TEXT(I:I) .NE. ' ') GO TO 70
   60 CONTINUE
   70 LENGTH = I
      RETURN
      END
```

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Index

Index

Index

Index

Index

L.S.
QA76.73.F25
L3
1981

G/UG

1032703

+1032703

Index