An Introduction to Algol 68 through Problems

A Learner and
A J Powell

AN INTRODUCTION TO ALGOL 68 THROUGH PROBLEMS

A. LEARNER
Department of Pure Mathematics
Queen Mary College
A. J. POWELL
Department of Computer Science
Queen Mary College

M

ISBN 978-1-349-02228-1 ISBN 978-1-349-02226-7 (eBook)
DOI 10.1007/978-1-349-02226-7

First published 1974 by
THE MACMILLAN PRESS LTD
London and Basingstoke
Associated companies in New York Dublin
Melbourne Johannesburg and Madras

SBN 333 16620 5

Typeset in Great Britain by
PREFACE LIMITED
Salisbury, Wilts

Contents

Preface

This book is intended to serve the beginner as both an introduction to programming and an introduction to the programming language Algol 68. The contents of the book form the basis of the programming component of a first-year course in computer science at Queen Mary College, University of London. No previous knowledge of programming is required but some acquaintance with elementary mathematical terminology and notation is assumed. Those familiar with other programming languages should also find this text of use as an introduction to Algol 68.

The algorithmic approach to solution of problems has been used both to illustrate program construction and to motivate the introduction of the various features in the language. This has allowed the majority of topics to be introduced and elaborated without too much artificiality. Some minor details have been relegated to exercises, for which complete solutions are given. However, it is not intended as a comprehensive treatment of Algol 68, which can be found in C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to Algol 68*, North-Holland, Amsterdam (1971).

All the programs, sections of programs and solutions to exercises are written to conform to the standard definition of Algol 68 and all of them have been validated by testing with the Algol 68R compiler on an ICL 1904S at Queen Mary College. The programs are written so that only minor modifications are needed for this purpose, and the method of conversion is explained in Appendix III. The language is as defined in the IFIP authorised *Revised Report on the Algorithmic Language ALGOL 68*, Springer-Verlag, New York (1974). The style of programming is deliberately explicit and designed more for student perusal than for efficiency.

BNF (Backus-Naur Form) is used to define the syntax, serves as an adjunct to the text and helps to avoid the description of minutiae of the recipe-book approach. There is no pretension that this is a definition of Algol 68 but it should enable the novice to check definitions and verify the syntax of programs. It is not suggested that everything which is correct with respect to this BNF is necessarily a correct Algol 68 program.

The algorithmic approach and the use of BNF have been used effectively in the teaching of Algol 68 to beginners.

We gratefully acknowledge the helpful criticism of Professor J. S. Rohl, who read draft versions of the book, and also thank the Queen Mary College Computer Centre for their cooperation in running Algol 68 programs, and in particular for the introduction of a 'cafeteria' service with an instant turn-round.

A. LEARNER
A. J. POWELL

London 1974

1 Computers and their Languages

Plotting a satellite orbit, stocktaking in a large company, planning the routes of a fleet of oil tankers, typify practical problems that are too time-consuming, need to be solved too quickly, or are too important, to leave to error-prone human computation. Computers are unimaginably faster and less subject to error and have been used in these and many other tasks.

Computers can also assist in the solution of theoretical problems that humans can do better, such as playing chess, but the methods used may illuminate both the structure of the task and our own methodology.

This book is concerned with solving problems with the aid of a computer. Actually computers are not very good at solving problems – in particular they are poor at solving the problem 'How does one set about solving a problem?' Therefore, we have to specify the process of solution in a very explicit manner that can be communicated to a computer.

The problems in this book are quite simple although the solutions may seem strange to a novice programmer. They are less strange if one understands how to communicate with a computer. This, in turn, benefits from a knowledge of some of its characteristics.

The typical modern computer stores information in its memory or *store* which consists of a large number of storage locations, and the digital pattern in a particular storage location represents the information stored therein, and can be regarded as its value. A whole number (integer), if not too large, can be stored in a single location, but a decimal fraction (real) will probably require more than one.

Apart from numbers, anything that can be mapped into a digital pattern can be stored, for example a person's name and his qualifications, which may occupy several locations. Thus the contents of a storage location may be regarded as different sorts of information depending on the interpretation of its pattern.

A desk calculator can also store numbers in its registers. A digital computer is distinguished from such a machine by the fact that it also stores the actual instructions that it has to obey in crder to transform the contents of its memory, and so do some useful computation. The instructions themselves may be modified or repeated many times.

A *stored program* is a sequence of instructions specifying the steps in the computation. The machine decodes these instructions in turn, and performs one of the basic operations in its repertoire, such as altering the store's contents, simple arithmetic operations on store representing numbers, comparing two values taken from store or taking the next instruction out of its normal sequence. There must also be *input* and *output* instructions that put information into memory and take it out.

This basic set of instructions constitutes the *machine language* and is different for each type of computer. It is a detailed and sometimes lengthy and difficult task to translate the solution of a problem written in English into such a simplified language. Even then the machine-language solution for one machine cannot be used on a different computer without changes. Hence we use the machine to do the translation into its own unique language. However, the machine cannot be expected to translate English, which is both ambiguous, for example, 'I can fish' or 'He has grown another foot', and redundant in vocabulary and grammar. Legal phraseology shows how difficult it is to be precise.

Instead, we first write our own solution in a *programming language* that is sufficiently like English to be readable, but which has a very precise grammar and meaning to its words. This step is also valuable because it forces us to be crystal-clear in specifying exactly what we want the computer to do. Further, writing programs in such a *high-level language* often leads to new ideas for improving the solution and attempting other problems.

Among the international high-level languages the commonest are Fortran, Cobol, Algol 60, PL/1 and Algol 68. It is the last of these, first defined in 1968, as the name suggests, that is used in this book. This is the most recent universal language, and is suitable for a wide variety of problems, both numerical and non-numerical.

An Algol 68 program is translated into machine code by a standard translation program called a *compiler*. Our program is data for this compiler, and the output from the compiler is a machine-language program. To use our program and any other programs written in the same language on a different computer, it is only necessary to have a compiler for the high-level language available.

How do we get our program into the computer, so that it can be compiled and how, in turn, does our program get its own data and exhibit its results? Input devices are not yet able to accept handwritten or typewritten characters or the spoken word, and so information to be *read* into the computer is usually represented by characters punched into cards or paper tape. A pattern of holes on a card (or tape) is produced by a punching machine divorced from the computer, and represents a character in some coded form. The card is read by a *card reader* that converts the pattern into binary form and transmits the result to the central machine.

Output can also be obtained punched on cards or tape, but is usually

produced in a readable form as printed characters on continuous stationery in a *lineprinter*. The input and output devices, called *peripherals*, may also include teletypewriters, graph plotters, or visual-display units. A card reader is relatively slow in transferring information, and so a faster medium, such as magnetic tape, discs or drum is used as a *filestore*. Information that we do not need to read, such as the machine-language translation of our program, will be transferred to the filestore in binary form for subsequent use.

There are, therefore, several stages in getting a computation performed after the program has been written: the instructions forming the program, together with any data it requires, are punched onto cards; the program is compiled, and the result is a machine-level program that is usually put into filestore; this program is *loaded* into the main memory and *run*. The output from the program appears on an output device. Compilation needs no knowledge of the actual data and if the program itself needs any data, then it is not read until run-time.

The above sequence of events occurs if the program is error-free (has no 'bugs'). This is very rarely the case with the first version of a program. While doing the translation, the compiler checks the validity of each sentence in the program, reports any errors and prints a diagnosis on the lineprinter. Even if the program is correct Algol 68, so that it can be compiled and loaded, it may still fail to work correctly when run because of logical errors, or mistakes in the data. These *execution errors* are detected by special instructions inserted in the program by the compiler or by the operating system, and run-time diagnostics are produced.

Debugging is an important part of programming. Programs should be designed so that they can be split into sections and these sections separately tested using sample data. Also, the first version of a program should produce intermediate results so that the route the solution takes may be followed and verified.

In addition to translation, the compiler also has access to certain predefined constants, such as π, and also to standard functions like 'square root' or 'cos', that are beyond the basic arithmetic operations of the machine. In Algol 68, this information is called the *standard prelude*, and relevant parts will be included in the compiled program before it is run.

Early computers could only manage one program at a time, but nowadays several different programs are usually in store simultaneously. The control and allocation of resources to the various programs is performed by the computer itself, via a resident program called its *operating system*. To write a program one does not need to know the operating system, but to get the program compiled and run it is necessary to specify the compiler required and to request that various peripherals be connected. A program and its data is called a *job*, and to get a job run it will usually be necessary to issue commands to the

operating system to process the job in the required fashion and also to identify the constituents of the job. These commands are interpreted by the system, and will ensure that the correct sequence of compiling, loading and running is maintained. This process of putting a job into the computer differs from one computer installation to another and will not be expounded further.

Algol 68 has been precisely defined in an IFIP report, and we shall use this definition as a standard. However, different compilers may not include the full scope of the language, or may give different interpretations of minor points. Each compiler is an *implementation* of the language (implementations are discussed in Appendix III).

2 Summing a Series: Iteration

A common problem in applied computing is to find the sum of a series. If this is a once-only calculation, for example to find $1 + 1/8 + 1/27 + \ldots$ correct to 10 decimal places, then it is usually not worth using a computer. As a more realistic problem consider a series containing a variable y. Let

$$S = y - y^3/3 + y^5/3 \,.\, 5 - y^7/3 \,.\, 5 \,.\, 7 + \ldots$$

S is an infinite sum, but as the time available for the calculation is finite, we add only a limited number of terms, say 4, as a first approximation.

We have now posed the problem, so we may attempt to write a possible solution as a sequence of steps.

1. Read in the data, which is the value of y.
2. Calculate the sum = $y - y^3/3 + y^5/3 \,.\, 5 - y^7/3 \,.\, 5 \,.\, 7$
3. Print the sum.

Such a systematic method for solving a problem, is called an *algorithm*. Our algorithm is trivial. However, *finding and refining the algorithm* are key steps in computing, and even for the simplest problem one should write the steps in reasonable detail.

Exercise 2.1

Write algorithms for (a) constructing a table of squares of 1,2, . . . 10; (b) making a telephone call to the Gas Board.

We now translate the algorithm into Algol 68, a very explicit language that can be processed by a computer. It is not necessary to use only the single letter names for variables usual in mathematics, so we choose 'y' and 'sum'.

Locations must be reserved to store the values of y and sum. This is done by 'declaring' their names, and specifying to what sort of objects they refer. '**real** y' is such a *declaration*, and declares 'y' to be the name of a place where a real number may be stored in the subsequent program. We may visualise the declarations of 'y' and 'sum' as in figure 2.1.

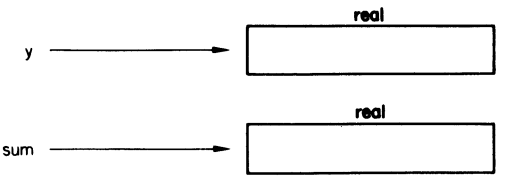

Figure 2.1

'y' and 'sum' both refer to places that can hold real values. The actual calculation of step 2 involves assigning values to fill these places. We do this by an *assignment statement*, using the symbol :=, read as *becomes*, which although written as a colon followed by an equals sign, is regarded as a single symbol. The assignment statement 'y := 2.3', for instance, puts 2.3 in the place pointed to by 'y'. We say that y now *refers to* the value 2.3. If this is followed by 'y := y + 4.2', this has the effect of taking the current value of of y, 2.3, adding this to 4.2, and making the result 6.5, the new value of y. These are not equations, but instructions to obtain the value of the right-hand side of the assignment and put it in the place referred to by name on the left-hand side.

A program or section of program, starts with **'begin'** and finishes with **'end'**. These words, in boldface type when printed, or underlined when written, are single keywords that the compiler recognises. **'real'** is another such word. We can now write a program for our simple algorithm.

```
'begin
        real y;
        read (y);
        real sum;
        sum:= y – y↑3/3 + y↑5/(3 * 5) – y↑7/(3 * 5 * 7);
        print (sum)
end'
```

The semi-colons between statements are *separators*, and may be thought of as an order to go on to the next statement. Putting a semi-colon between 'print (sum)' and **'end'** would be a mistake as **end** is not a statement, but acts like a closing bracket.

After declaring y, 'read (y)' reads the next number from the data supplied with the program and stores its value in the place to which the name y refers. 'read', and also 'print', are functions built into the language, that is they are defined in the standard prelude to all programs. Similarly, other useful functions like 'sin' or 'sqrt' are the common property of all programs, and can be written as functions in the conventional way, for example: 'sqrt (4.3)', 'sin(x)', where 4.3 and x are called their arguments. A full list of *standard functions* appears at the end of Appendix I.

The next step in the program is to declare 'sum' for which any other name, such as z or b could be substituted, but it is preferable to choose names that suggest the quantities to which they refer.

The required *arithmetic expression* is then assigned to 'sum'. It is written using ↑, /, *, +, – which are the symbols for exponentiation (taking powers), division, multiplication, addition and subtraction, respectively. These operators and their operands appear on a single line in the expression, so that one cannot use positions above and below to indicate, for instance, the numerator and denominator of a division. The order in which these operations are performed is important. For example in a + b * 2 the multiplication precedes the addition but in a * b ↑ 2 forming the square of b takes precedence over the multiplication. In general ↑ has precedence over * and /, but the latter two have equal precedence, being read left to right if they occur together. Parentheses can override these rules. Thus y↑3/(3 * 5) is $y^3/15$. but y↑3/3 * 5 is $5y^3/3$. + and – are also of equal precedence, lower than the other operators. Hence 'a + b/3 * 2 – 7 * c/5' is a + 2b/3 – 7c/5 in more familiar notation.

Multiplication signs must not be omitted, since 'ab' would be regarded as a single name (undeclared) and not as the product 'a * b'.

The expression in the program fits on to one printed line, and would also fit on to a single card if the program were punched on cards, but lines may be broken at any place. The semi-colon is the separator, and spaces and new lines are ignored inside statements.

Exercise 2.2

Write the following expressions in Algol 68
(a) $b^2 - 4ac$, $(1/2)\sin(ax)$
(b) Evaluate 3/4 + 2 ↑ 2 * 5

Each statement is called a 'unitary clause' or *unit* and a sequence of statements and declarations, separated by semi-colons, of which the last is a unit, is called a *serial clause*. Thus 'read (y)' and '**real** y; read (y)' are both serial clauses, the former being a unit. The whole program is a *closed clause* because it is a serial clause surrounded by **begin**, **end**. One can use parentheses instead of **begin** and **end**. A closed clause is itself a unit, possibly contained in a larger serial clause.

Even in our simple program there are unnecessary computations. Each term of the series can easily be computed from the previous one, so that we may refine our algorithm

1. Read y.
2. Set sum and term to y.
3. Multiply term by $-y^2/3$ and add to sum.
4. Multiply term by $-y^2/5$ and add to sum.

5. Multiply term by $-y^2/7$ and add to sum.
6. Print sum.

In Algol 68 this becomes

```
'begin
        real y; read (y);
        real sum; real term;
        term := y; sum := term;
        term := -term* y↑2/3;
        sum := sum + term;
        term := -term*y↑2/5;
        sum := sum + term;
        term := -term*y↑2/7;
        sum := sum + term;
        print (sum)
end'
```

A number of immediate improvements can be made in this program. We can collect declarations of two or more objects of the same type, for example '**real** sum, term'. This declares that both 'sum' and 'term' refer to reals, without specifying the order in which this is done, unlike '**real** sum; **real** term', where first 'sum' and then 'term' are declared. There is a significant difference between commas and semi-colons, the former separating elements in a list that are to be collected together, and the latter separating sequential statements.

Instead of separately assigning to 'sum' and 'term', we can write 'sum := term := y'. This is performed from right to left, so that it could be written 'sum := (term := y)', and works as follows. The unit 'term := y' assigns the value of 'y' to 'term' as usual, but the unit also has a *result*, namely 'term'. This result is then assigned to 'sum', and the net effect is that both 'sum' and 'term' now refer to the value of y. We can follow the action of the double assignment by the numbered arrows in figure 2.2

It is important to appreciate that a unit may both perform operations and supply a result.

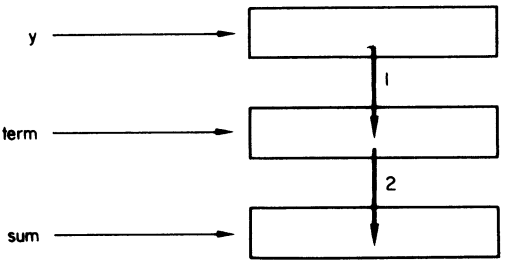

Figure 2.2

After making the above improvements the program is still repetitious. Let us consider its structure. We can visualise the sequence of operations as a sequence of boxes labelled with the actions to be taken, with arrows pointing from one to the next. For instance our first program may be represented by figure 2.3.

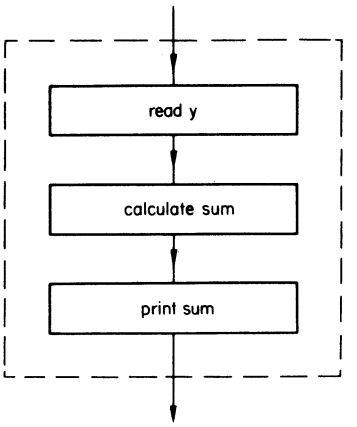

Figure 2.3

In fact, such a simple algorithm could be placed in a single box, as indicated by the broken lines. The second program if drawn in the same way would contain three pairs of boxes all performing the similar operations of multiplying the term by a factor and adding it to the sum. Figure 2.4 shows how we can replace this by a single box. The arrow leading back from the larger sub-box to the smaller indicates repetition of the action in the larger. We leave the whole box enclosed by broken lines after three loops around, controlled by n.

Repetitions like this are translated into a *do clause* in Algol 68.

```
'for n to 3 do
        term := -term*y↑2/(2*n + 1);
        sum := sum + term
        od'
```

The clause following **do**, closed by **'od'**, which is a reversed **'do'**, is performed three times with n, a counting variable with only whole-number values, taking the values 1, 2, 3 successively. n does not have to be declared like 'sum' or 'term', since it follows **'for'**, indicating that it is a control counter.

'to 3' specifies that the upper bound of counting is 3. In a do clause the upper bound may be any unit that yields an integral value. The lower bound is 1 by default when it is not mentioned, as here. We could

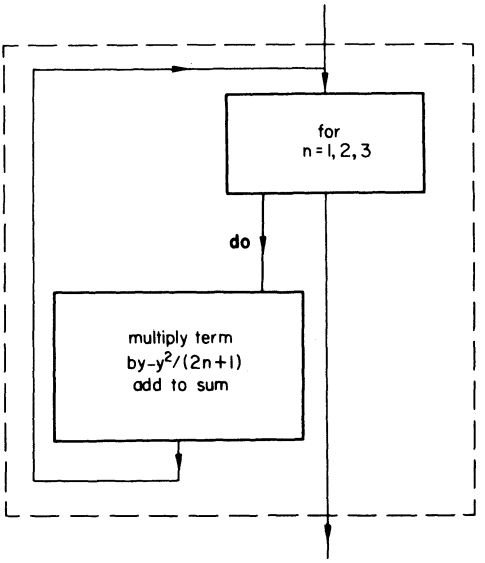

Figure 2.4

write **'for** n **from** 1 **to** 3 **do'** with the same effect, or even

```
'for n from 2 to 4 do
        term := –term*y↑2/(2*n – 1);
        sum := sum + term
                        od'
```

n then corresponds to the number of the term in the series as it takes the values 2,3, 4.

Let us write a new program, but change it to add 8 terms instead of 4, a trivial alteration using the do clause

```
'begin
        real y; read (y);
        real sum, term; sum := term := y;
        for n from 2 to 8 do
                        term := –term*y↑2/(2*n – 1);
                        sum plusab term od;
        print (sum)
end'
```

'plusab' can be read as 'plus and becomes'. 'sum **plusab** term' is a short way of writing 'sum := sum + term'. We could similarly replace the assignment to 'term' by 'term **timesab** –(y↑2)/(2*n – 1)'. The various arithmetical-assignment abbreviations are listed in Appendix I.

n occurs inside the do clause only in the expression (2*n – 1), which

takes the values 3, 5, 7, . . . 15. This small calculation could be avoided by writing

```
'for n from 3 by 2 to 15 do
      term timesab -(y↑2)/n; sum plusab term od'
```

The '**by** 2' specifies that steps of 2 are taken, so that n has the values 3, 3 + 2 = 5, 5 + 2 = 7, . . . 15, inclusively. When the **by** part is omitted, steps of 1 are assumed by default. However, this form of the do clause hides the series construction.

Exercise 2.3

Write a program to print out a table of squares and cubes of the integers 1, 2, . . . 10.

Instead of reading in a single value of y, let y take the values 0.3, 0.6, 0.9, 1.2 and calculate the sum both of 8 terms and of 12 terms of the series.

With y = 0.3, figure 2.5, similarly to the previous case, represents

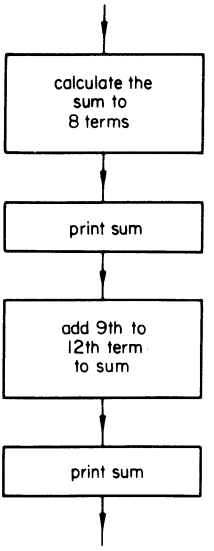

Figure 2.5

the algorithm. However, we want the whole of the above to be repeated for y = 0.3, 0.6, 0.9, 1.2 and so we draw a master diagram, as figure 2.6.

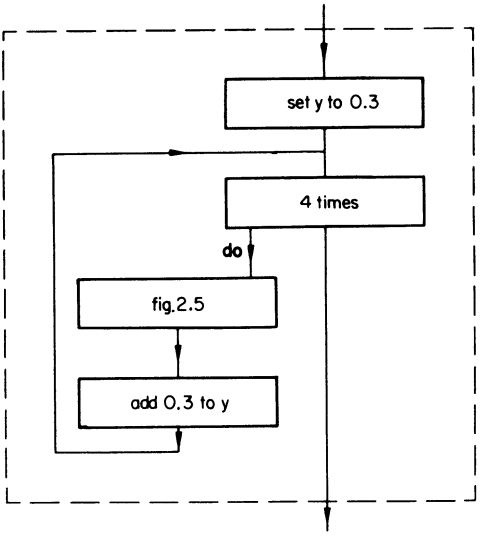

Figure 2.6

The corresponding program is

```
'begin
        real y := 0.3, sum, term;
        to 4 do
                        sum := term := y;
                        for n from 2 to 8 do
                        term timesab -(y↑2)/(2*n - 1);
                        sum plusab term od;
                        print (sum);
                        for n from 9 to 12 do
                        term timesab -(y↑2)/(2*n - 1);
                        sum plusab term od;
                        print (sum);
                        print (newline);
                        y plusab 0.3
        od
end'
```

The outer loop has no counting variable since it does not enter into the calculations inside the do clause. **'to 4 do'** is equivalent to **'for** counter **from 1 by 1 to 4 do'**. The **od** and **end** at the end of the program are needed to match the **do** and **begin**. The program has been indented to correspond to the structure of figure 2.6. This is unnecessary but does assist in identifying the sections and also in matching **begin end** or parentheses correctly.

The declaration of the real variable y *initialises* it to 0.3. Any

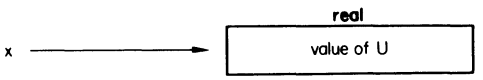

Figure 2.7

variable x can be both declared and initialised to the value of any unit U by '**real** x := U', provided that U can be evaluated to a real when this declaration is met. We picture this declaration in figure 2.7. '**real** y, sum; sum := y' would be wrong because y has no value when it is assigned. Initialising and non-initialising declarations may be mixed as in the program.

'print (newline)' sets the output position on the lineprinter to the beginning of the next line. However the program output is poor, since a bald number is not enough. The results of programs, and also the programs themselves, should be readable even months later when the original details have been forgotten.

A proper display of the results of the above program would be

y	8-term sum	12-term sum
0.3	0.29115993780	0.29115993780
0.6	0.53292771893	0.53292771894
0.9	0.69219235100	0.69219235564
1.2	0.76057935308	0.76057995126

We are assuming that real numbers are printed to 11 significant figures. Real numbers are only held to a certain accuracy in a computer, so that the last significant figure is somewhat meaningless. Nevertheless, for y = 0.3 the sum of 8 terms is the same as of 12 terms, smaller values of y giving smaller terms, whereas for y = 1.2 the two sums differ markedly. For accurate results, the number of terms to be added should depend on the value of y.

We decide to add terms until the last one added has absolute value less than epsilon, where epsilon is some suitably small number, say 10^{-11}. (The absolute value of x, supplied by the standard-prelude operator **abs** x, is x if $x \geqslant 0$ and $-x$ otherwise). A control on the repetitions of a calculation is needed that does not stop after a fixed number of times as in previous do clauses, but after a certain condition has been satisfied.

There is a **while** option in do clauses that allows for this:

```
'real x := 0;
while x < 6 do x plusab 1.8 od;
print(x)'
```

The second statement is an instruction repeatedly to add 1.8 to x until 'x < 6' becomes false (that is until $x \geqslant 6$). Clearly 7.2 is printed.

'**while**' is not different in principle to the previous do clauses, and has a similar picture in the form of two boxes, one to control the looping through the other. It is more flexible because the condition is

tested each time around the loop, whereas the lower bound, step and upper bounds (where present) of a counting variable are calculated once and for all at the start of the clause.

while can be combined with the other options, for instance, '**for** n **from** 0 **by** −1 **while** c > 0 **do** S **od**' where S stands for some serial clause. If c ⩽ 0 when this do clause is reached, then S is never performed. Otherwise the clause S is repeated with n = 0, −1, −2, . . as long as c is positive. If c is not ultimately made ⩽ 0 by the execution of S, the program will continue for ever, or at least until some external action is taken to terminate it.

Exercise 2.4

Each of the following clauses is preceded by '**real** x := 1'. To what value does x refer after their execution?

(a) **for** m **by** 3 **while** x < 7.5 **do** x **plusab** m **od**
(b) **from** 2 **to** 5 **do** x **minusab** 1 **od**
(c) **for** m **from** 0 **by** −1 **while abs** x < 3 **do** x **timesab** (m − 1) **od**

The sum of the series may not be wanted for regular groups of numbers like 0.3, 0.6, 0.9, 1.2, but for various values of y. We arrange for our program to accept as data a sequence of numbers and print out each of them with the corresponding sum. We do not know in advance how many values, but as they are all non-zero we can terminate the data with the value 0.

Here is the new program, followed by some comments on its features.

```
'begin
        real y, sum, term;
        real epsilon = 1 & −11;
        while read (y); abs y > epsilon
        do      print ((newline, "Y = ", y, "SUM OF
                SERIES = "));
                sum := term := y;
                for n while abs term > epsilon
                do term timesab −(y↑2)/(2*n + 1);
                   sum plusab term
                od;
                print (sum)
        od
end'
```

1 & −11 is a *denotation* of the number 10^{-11}, just as '2' is a denotation of the integer 2. '&' means 'times 10 to the power'. Thus, '2.346 & 2' denotes the same real number as '234.6'. This is called

the *floating point* denotation, since altering the exponent following the ampersand moves the point about in the decimal part preceding it.

We choose '1 & −11' as a small value distinguishable from 0, since we are interested in the result to 10 decimal places.

The declaration of 'epsilon' is not like those of 'y', 'sum' and 'term', and cannot be combined with them. It contains an equals symbol which *identifies* the name 'epsilon' with the value 10^{-11}, as figure 2.8

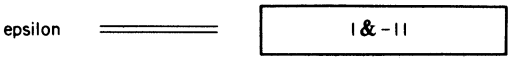

Figure 2.8

indicates. There is no arrow in this diagram because 'epsilon' cannot point to other values. Assignments to it in the program are illegal, since writing 'epsilon' is equivalent to writing '1 & −11'. This declaration with an '=' in it is called an *identity declaration* and must be clearly distinguished from '**real** eps := 1 & −11' which would declare the name 'eps' and initialise it to 10^{-11}. Later, it could be altered to any other real value by assignments.

Declarations of constants like 'epsilon' are used for several reasons. The right-hand side might be a complicated number or even a complete unit that we do not want to rewrite, and an appropriate name replacing this unit is easier to write and subsequently to read. Further, in the case of a unit, unnecessary re-evaluation is avoided. We could apply this elsewhere by declaring '**real** ysq = y↑2' and in the do clause use 'term **timesab** −ysq/(2∗n + 1)' to avoid repeatedly squaring y. The initialising declaration '**real** ysqu := y↑2' while also avoiding re-evaluation, would reserve a place for the name 'ysqu' which is unnecessary since it is never altered in value.

Exercise 2.5

What is wrong with the declarations (a) **real** pi = 3.14, s;
(b) **real** pie = 3.14, pi3 = pie↑3?

The output will now appear in the form

Y = 0.6 & 0 SUM OF SERIES = 0.3292771894 & −1

with a new line for each value of y, and numbers printed in floating-point form.

Between the **while** and the **do** there may be any serial clause whose value is true or false. Recall that the definition of the serial clause is a sequence of declarations or units, and its value is that of the last in the sequence, which must be a unit. Here we exploit this to read y and then

test if its absolute value is > epsilon. We do not test the equality 'y = 0' or the inequality '**abs** y > 0' because **real** values are stored only to a limited accuracy in the computer, and a real value 0, read in, may not be identical to the real constant 0. In some implementations any test for equality of reals is forbidden.

'print' and 'read' can have more than one argument provided that each argument is separated by a comma and that there is an extra pair of parentheses to collect them into a row. The arguments may be literal strings in quotes (inverted commas) that are printed out as they stand, variables or constants whose value is to be printed, or instructions to begin a new line or a new page or print a space. In this text, the literal strings in programs may be accidentally broken by the ends of lines, and are in capital letters for emphasis, since lineprinters usually print only capitals. The two typefaces are equivalent as far as our programs are concerned.

The new program structure is essentially the same as before – after the initialisation, a calculation is repeated until the value of the next data item is ⩽ epsilon. This calculation, like the inner box in figure 2.6 can be expanded into a sequence of four operations: print information to identify the result; declare and initialise 'sum' and 'term'; do a controlled repetition of the series sum; print the sum. This inner controlled repetition is a do clause with no bounds specified for the counter n, so that it is performed with n = 1, 2, 3, . . (lower bound and step both being 1 by default) until 'term' has absolute value ⩽ epsilon.

For any given y we may be interested in knowing the number of terms summed, which is the value of n + 1 when we leave the loop. However, the counter n is *local* to the do clause, and cannot be referred to once the clause is left. We need some variable to be declared beforehand that will hold the number of terms summed. This variable takes whole number values, which may be stored in the computer in a form different to real numbers (usually one word compared to two for reals). Accordingly, it is declared in a different way using **'int'**, short for *integer*. Integer variables can be used in arithmetic expressions and assignment statements. For example

'**int** k, j := 1; j := j + 2; k := j↑3 – 7*j;'.

After this, k points to a place holding the integer value 6.

Integers can be assigned to reals, for example '**real** x; **int** a := 3; x := a'. The last assignment takes the value of a, the integer 3, and *widens* it, meaning that it is to be regarded as a real number that can be referred to by the **real** variable x. In diagrammatic terms, the place pointed to by 'x' now contains 3 stored as a real.

Assignment the other way, of reals to integers, is not allowed, but there are operators to make a conversion. 'a := **round** x' assigns the nearest whole number to the value of x to the integer a. '**entier** x' is the greatest integer ⩽ x.

Let us rewrite the program to count the number of terms summed, and at the same time remove another weakness. For very large values of y, hundreds of terms may be summed before 'term < epsilon' becomes true. There may even be values of y for which the series diverges (but not for this particular series). Therefore we stop the summation, regardless of the next term, once 100 terms have been summed. We also record the fact that summation was stopped at 100 terms, since one should regard the corresponding sum with suspicion. Hence, after the sum there will be one of two messages depending on whether 100 terms were reached or not.

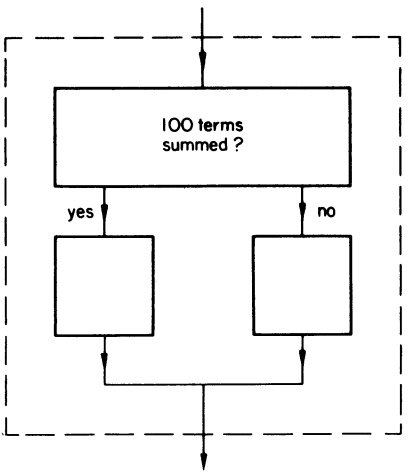

Figure 2.9

This requires a box like figure 2.9 with two possible routes through it. The inner boxes are filled in with the appropriate action in a complete picture.

This box is translated into the clause in the following program that starts with **'if'** and ends with **'fi'** (a reversed **if**), an example of an *if clause.*

```
'begin
        real y, sum, term; real epsilon = 1 & -11;
        while read (y); abs y > epsilon
        do    int no of terms := 1;
              print((newline, "Y =", y, "SUM OF SERIES = "));
              sum := term := y;
              for n while abs term > epsilon and
              noofterms < 100
```

```
                do term timesab -(y↑2)/(2*n + 1);
                sum plusab term;
                no ofterms plusab 1 od;
                print((sum, newline));
                if no of terms = 100
                then print("SUMMATION STOPPED AFTER
                100 TERMS")
                else print(("CONVERGED AFTER",
                noofterms, "TERMS"))
                fi
        od
end'
```

'no of terms', 'no ofterms' and 'noofterms' all refer to the same number since spaces are not significant except in quoted strings or boldface words, the latter being regarded as single symbols.

The inner do clause has two conditions in the **while** part, linked by **and**. Repetition of the controlled clause continues as long as they are both true.

Exercise 2.6

Rewrite the program so that summation stops when the difference of successive terms has absolute value less than epsilon.

The if clause contains **if** ... **then** ... **else** ... **fi**. Between the **if** and the **then** is a serial clause whose value is either true or false; here it is 'no of terms = 100'. If it is true, the serial clause between **then** and **else** is performed, and if it is false the clause between **else** and **fi** is performed.

if, **then** and **fi** are always present in an if clause, but the **else** is optional. For example, we could simply note the cases where 100 terms were summed by '**if** noofterms = 100 **then** print("100 TERMS SUMMED") **fi**'. If 'noofterms' has a value not equal to 100 at this point, then execution of the program passes on to the next statement, as in figure 2.10.

Exercise 2.7

What is the value of c after the execution of the following clause when n is (a) 2; (b) −3; (c) −6? '**if** n > −5 **and odd** n **then** c := (**sign** n) *3 **else** c := −n **fi**' ('**odd** n' is true when n is odd and '**sign** w' is +1, 0, −1 when w > 0, w = 0, w < 0 respectively. **odd** and **sign**, like **round** and **entier** mentioned above, are among the standard operators)

We have introduced many new notions in this chapter, and would like some way of checking whether they are being used correctly in a

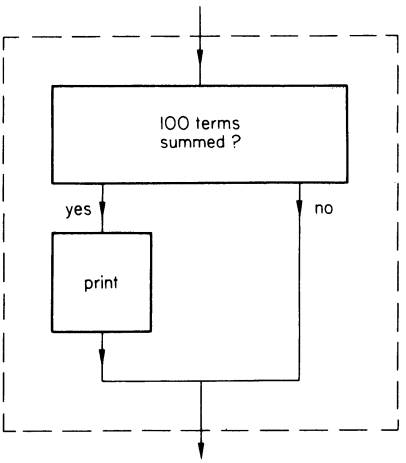

Figure 2.10

program. There is a formal way of defining the syntax (grammar) of a programming language called *Backus-Naur Form* (BNF for short). In this system each construct in the language is placed in a class, and each class defined in terms of other classes or primitive objects like 3 or **begin**. For instance the class of all digits consists of 0 or 1 or 2 or . . . 9. This is concisely expressed in a rule

⟨ digit ⟩ ::= 0|1|2|3|4|5|6|7|8|9

The ⟨ , ⟩ brackets distinguish the class of digits from the string of letters d–i–g–i–t. The ::= symbol means 'consists of', and the vertical bar means 'or'. We could now define

⟨ two digit no ⟩ ::= ⟨ digit ⟩ ⟨ digit ⟩

meaning that a two digit no, that is, an element of the class ⟨two digit no⟩, is a digit followed by a digit. However, we need integers with any number of digits in them, and so we introduce an extension of BNF, illustrated in

⟨ unsigned positive int ⟩ ::= $\langle \text{ digit } \rangle_1^*$

$\langle zz \rangle_1^*$ represents one or more of the elements in the class ⟨zz⟩ following each other. ⟨zz⟩* means zero or more, that is a string of elements each in ⟨zz⟩. The superscripts and subscripts are operations on classes. Thus

⟨ unsigned positive int ⟩ ::= ⟨ digit ⟩ ⟨ digit ⟩*

is equivalent to the above, and shows more clearly that an unsigned positive integer contains at least one digit.

Exercise 2.8

Our definition allows numbers like 0063. Construct a new definition that excludes leading zeros. (*Hint* Split the class of digits into 0 and the rest).

Clearly it is an uninteresting exercise to derive the rules for all possible denotations of reals and integers. A complete BNF, in alphabetical order of classes, is contained in Appendix II (it must be stressed that this is a simplified and modified version of the formal definition of Algol 68 in the IFIP Report).

Apart from constants, we have introduced names, or *identifiers* consisting of one or more letters or digits, of which the first is a letter. Hence

⟨ letter ⟩ ::= a | b | . . . . | z
⟨ identifier ⟩ ::= ⟨letter ⟩⟨ letter ⟩|⟨ digit ⟩⟩*

We have here a double bracket. ⟨⟨classa⟩|⟨classb⟩⟩ stands for an element of classa or of classb. These double brackets can be starred, as in the above definition.

Exercise 2.9

Which of the following are identifiers? (a) baker's dozen; (b) time in athens; (c) 76 trombones; (d) mr. and ms.

One unit that we have met is a do clause, defined by

⟨ do cl ⟩ ::= ⟨ **for** ⟨identifier ⟩⟩[1] ⟨ **from** ⟨ unit ⟩⟩[1]
⟨ **by** ⟨ unit ⟩⟩[1] ⟨ **to** unit ⟩⟩[1]
⟨ **while** ⟨ ser cl ⟩⟩[1] **do** ⟨ ser cl ⟩ **od**

The notation ⟨ ⟩[1] means zero or one occurrence, that is an *optional* occurrence of the object in brackets. As previously remarked, the default values when an option is omitted are **from** 1, **by** 1, **to** infinity, **while true**. Strings like **'from'** or **'do'** that are not in class brackets in a rule stand for themselves.

Exercise 2.10

'for k **by** −2 **to** m **while** k∗m < 40 **do** S **od'**

How many times is S executed when (a) m = −6; (b) m = −10?

The other control clause in the programs was an if clause, and its BNF definition is

⟨ if cl ⟩ ::= **if** ⟨ ser cl ⟩ **then** ⟨ ser cl ⟩ ⟨ **else** ⟨ ser cl ⟩⟩[1] **fi**

On the face of it, the above definitions are improper. An if clause is a possible unit, and the unit may form part of a serial clause, that in turn can be part of an if clause, and so on. However, any given if clause is of finite length, so that when we check it against the BNF, we must ultimately reach a more fundamental unit, like an expression or an assignment statement.

Arithmetic expressions require a more complicated BNF rule. We have the multiplicative and additive operators

⟨ mult op ⟩ ::= * | /
⟨ adding op ⟩ ::= + | −

Assume that we have defined a basic building block called a *primary*. Examples of primaries would be identifiers or number denotations or even any closed clause.

⟨ basic arith exp ⟩ ::= ⟨ primary ⟩ | (⟨ arith exp ⟩)
⟨ factor ⟩ ::= ⟨ sign ⟩[1] ⟨ basic arith exp ⟩ | ⟨ factor ⟩↑⟨ basic arith exp ⟩
⟨ term ⟩ ::= ⟨ factor ⟩ | ⟨ term⟩ ⟨ mult op ⟩ ⟨ factor ⟩
⟨ sign ⟩ ::= + | −
⟨ arith exp ⟩ ::= ⟨ term ⟩ | ⟨ arith exp ⟩ ⟨ adding op ⟩ ⟨ term ⟩

We have here examples of *recursive* definition. For instance, the second choice for term is a term followed by a mult op followed by a factor. Again, the finite length of any term forces us to choose the first possibility, namely a factor, at some stage in our analysis of it. For example 2*3*4*5 is a term. Let us check this. 5 is a primary, so a basic arith exp and so a factor. * is a mult op. Hence it is enough to show 2*3*4 is a term. Repeat the process twice, and we are left with 2, which is a term because it is a factor. The reader may care to show that ((((8)))) is an arith exp in the same way.

The following exercise is difficult, but it illustrates the sort of process going on in the compiler when it examines an Algol 68 program to see whether it is syntactically correct.

Exercise 2.11

Given that y,2,n,1 are all in the class ⟨ primary ⟩, show that (y↑2)/(2*n − 1) is in the class ⟨ arith exp ⟩.

Problems for Chapter 2

1. Write a program to compute the sum $1 - x^2/2 + x^4/(2 \cdot 3 \cdot 4) - \ldots$ to 5 decimal places for $x = 0, \pi/8, \pi/4, 3\pi/4, \pi/2$. (**real** pi = 3.14159 ... is a constant declaration in the standard prelude). Alongside your results print the values of cos(x). ('cos' is also in the standard prelude).

2. Given two weights in stones, pounds and ounces, draw the diagram of an algorithm to compute their sum and difference.

3. Write a program that reads in the real numbers a,b,c and prints out the real solution(s) of $ax^2 + bx + c = 0$, if any exist, or a suitable message otherwise.

4. The bisection method of finding a root of $f(x) = 0$ between a and b, given $f(a) < 0 < f(b)$ is to halve the interval (a,b) successively, choosing the half in which the function changes sign. Use this method to find a root of $x^3 + 2x - 1 = 0$ between 0 and 1 accurate to 10 decimal places.

5. The sequence of Fibonacci numbers 0,1,1,2,3,5,8,13,21, . . . is constructed so that each number is the sum of the preceding two numbers. Write a program that given N, outputs the Nth Fibonnaci number. (Do not choose N too large – the maximum size integer that can be stored in the computer is soon exceeded!)

3 Traversing a Maze: Multiple Values

An important question to answer when constructing an algorithm is how to *represent* the basic objects to be manipulated. Simple variable names like 'y' or 'sum' sufficed for the series summation programs, but other problems may involve referring to a list of 1000 numbers, squares on a chessboard, or a path through a maze.

In these situations, inventing and using a different identifier for each listed number, square on the board, or position in the maze, would be unwieldy. Furthermore, individual names would make it well-nigh impossible to refer to a group of numbers in the list, or all squares on the board collectively.

One way to represent a list is by a *row* of numbers with a single name, and an individual number can be selected by its position in the row. A chessboard can be represented by an 8 x 8 *array* of squares, and the piece positions referred to by their row and column in the array. We need to use a similar representation when we consider the problem of finding a path through the simple maze in figure 3.1.

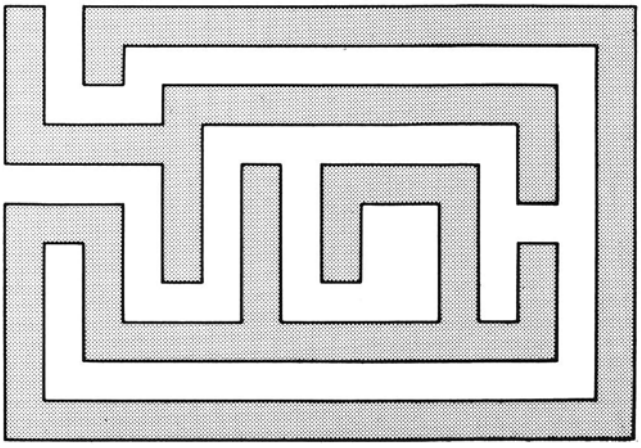

Figure 3.1

We can represent the maze by the rectangular array of figure 3.2, with stars and dots occupying the positions of blocked and free squares respectively.

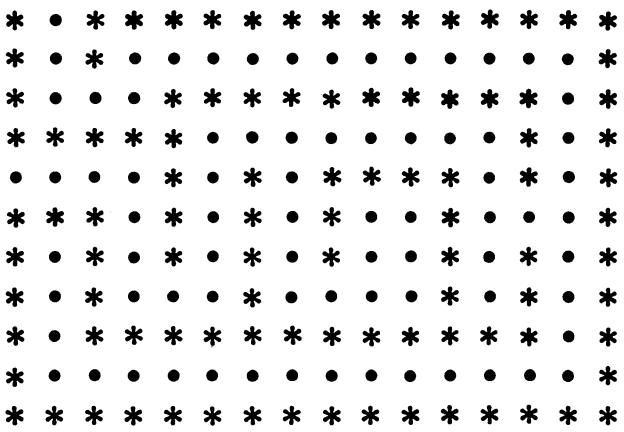

Figure 3.2

Given the positions of the entrance and exit squares, we may attempt to 'solve' the maze in the following way

1. Enter the maze.
2. Move to an adjacent position if it is not occupied.
3. Leave if the new position is the exit, otherwise repeat from step 2.

This is not an algorithm since it does not always find the exit – in fact one could step back and forth forever between adjacent free squares. Nevertheless it is the basis of our final solution.

We need to declare the maze as an array of objects. Algol 68 allows a declaration called a *multiple value* (or multiple for short) of a set of objects all of the same type. For example, '[1:8, 1:8] **int** amaze' declares 'amaze' to be a multiple referring to a two-dimensional array of integers. The limits on its size are specified by the values of the *index bounds* enclosed in [,] and so, in terms of the diagrams of declarations in the previous chapter, we may think of 'amaze' pointing to 8 x 8 = 64 places, arranged in 8 rows and 8 columns, each of which can hold an integer value.

Each *element* in this multiple can be individually referred to by the names 'amaze[1,1]', 'amaze[1,2]', . . . 'amaze[8,8]'. Thus 'amaze[i,j]' points to the place in the ith row and jth column; i and j are called its *indices*. It may be used like a simple variable. For example, 'amaze[4,5] := 2' assigns the value 2 to the element in row 4, column 5, and then 'amaze[2,7] := amaze[4,5] + 3' would assign the value 5 to 'amaze[2,7]'.

The declaration of a multiple value fixes the number of indices, called its *dimension*, and the lower and upper bounds of each index.

'[−10:1000] **real** vector' makes 'vector' the name of a one-dimensional multiple (a vector!) of 1011 real elements with indices ranging from −10 to +1000 including 0. '[1:m, 1:n, 1:p] **int** dim3' declares a three-dimensional multiple of integer type, provided that the values of m,n,p are known at the time of declaration so that the upper bound of each index can be determined.

Possible references to various elements of the above multiples are 'vector[225]', vector[−3]', vector[j − 7]', 'dim3[20,k,1]'. In the last two cases j − 7 and k are evaluated and must lie within the respective bounds, so that $-3 \leqslant j \leqslant 1007$ and $1 \leqslant k \leqslant$ (value of n at declaration of dim3).

We have met two basic types, **real** and **int**, and can now introduce *multiple types*: '[] **int**' refers to a row of integers, '[,] **int**' refers to a row row of integers (but it is easier to use rows and columns), '[,,] **int**' refers to a row row row of integers, and so on. A multiple of any type can be declared and used.

Exercise 3.1

To how many elements do the following multiples refer?
'[−3:4] **real** a; [0:20,0:20] **int** m; [1:2, 2:4, 3:6] **int** cube'.

Returning to the maze problem, we could use an integer array to represent the maze by coding the free and blocked squares by distinct integers. However, this is somewhat unnatural since the array of integers has to be decoded to see the maze and the path through it. To avoid this, we use a dot and a star to represent free and blocked conditions, respectively.

To do this, we introduce another fundamental type. '**char** c' declares 'c' to be a name referring to a *character*, that is, one of the single symbols we use in our programs, and denoted by enclosing in double quotes, for example "W" or "3" or "£" or " " (the blank character). 'c := "T"; c := "+" ' are possible assignments to 'c'. Unlike numbers, there are relatively few operations on characters – one of them is '**abs**'. '**abs** c' yields an integer code for c (the particular code depends on the implementation). Relational operators, like <, when applied to characters, compare their codes.

We can now represent the maze by a multiple of characters by the declaration '[1:8,1:10] **char** maze', assuming it is 8 x 10 say. The positions of the free and blocked squares, represented by the characters "." and "*", could be assigned to the 'maze[i,j]' in the program, but it would be better to allow for different mazes as data by reading into 'maze' a sequence of dots and stars, and also reading the entrance and exit coordinates.

The 'read' and 'print' functions may take **char** objects as arguments. Hence

'**for** i **to** 8 **do for** j **to** 10 **do** read(maze[i,j]) **od od**'

will read in an 8 x 10 multiple row by row. However, the 'read' procedure can accept a multiple variable as argument, and 'read(maze)' is equivalent to the above and more efficient. When reading character data, note that the space " " is also a valid character, so the 80 . s and *s should occur without intervening spaces.

Exercise 3.2

Write a program that prints out the internal code for the characters "1", "2", . . . "9".

Let us return to the 'solution' of the maze problem and try to improve it. Assume that we move only in straight lines north – south or east–west (in either direction), and that we systematically look for an adjacent free square to the east, south, west, north in that order, and move to the first one found. The maze in figure 3.3 still defeats us. We

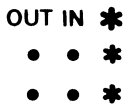

Figure 3.3

can avoid looping paths by marking the squares traversed and never re-entering them. The central steps of this solution are represented by figure 3.4.

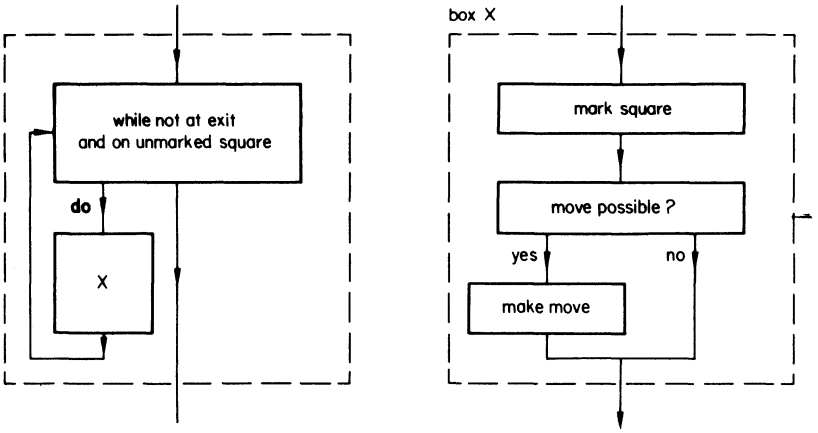

Figure 3.4

In the program translation of this diagram, we explain the structure by *comments*. These are insertions anywhere in the program surrounded by **'comment . . . comment'** or ¢ . . . ¢; however the simpler form **C . . . C** is used in this text. Any text between these symbols is ignored by the compiler and has no effect on the program run. Comments should always be used as aids to the writing, reading and documenting of programs.

```
'begin
        [1:8,1:10] char maze;
        read(maze); comment no spaces between data comment
        print(maze);
        int i,j,p,q; read((i,j,p,q));
        print(("ENTRANCE COORDS =", i, j, "EXIT COORDS
        =", p,q,newline));
        C (i,j) and (p,q) assumed to be within maze C
        while(i ≠ p or j ≠ q) and maze[i,j] ≠ "T"
        do print((i,j,newline)); C print coords. of route taken C
            maze[i,j] := "T";
            if j < 10 and maze[i,j + 1] = "." then j plusab 1
                C move East C
            else if i < 8 and maze[i + 1,j] = "." then i plusab 1
                C South C
            else if j > 1 and maze[i,j – 1] = "." then j minusab 1
                C West C
            else if i > 1 and maze[i – 1,j] = "." then i minusab 1
                C North C
            comment having tried unsuccessfully to move
            E,S,W,N, i,j are unchanged, so the while test fails on
            the next repetition comment fi fi fi fi
        od C now repeat do clause C;
        comment out of do clause so either route found or stuck
        comment
        if i = p and j = q then print("MAZE HAS BEEN TRAV
        ERSED")
                                else print("STUCK IN MAZE") fi
end'
```

This program will solve a simple maze like figure 3.3, but as the last message suggests, it unfortunately does not always work.

Exercise 3.3

Prove the last remark.

Improvements are clearly needed, and we start with some minor points.

The four closing **'fi'** of the central if statement, one to match each **'if'**, can be replaced by a single **'fi'** if each **'else if'** is replaced by the abbreviation **'elif'**.

Only 8 x 10 mazes are accepted, and to allow arbitrary mazes, their size must be part of the data. Thus '**int** m,n; read((m,n)); [1:m,1:n] **char** maze'. This last declaration with values of m and n which may alter when the program is run, uses *dynamic arrays*. The space required for the multiple is not fixed but depends on the values of m and n, unlike the declaration '[1:6,1:4] **int** matrix', say, which only requires static storage for 24 integer values.

Instead of repeatedly testing whether a move strays out of bounds, we plant an extra hedge right around the maze. This means that more space will be required to represent the maze, but time is saved when running the program as the test statements and their repetition are removed. The trade-off between *space* and *time* is frequently encountered in program construction. To plant this hedge, if the maze had bounds 1:m, 1:n then we must declare an enlarged maze with bounds 0:m + 1, 0:n + 1. Rows 0 and m + 1, and columns 0 and n + 1 have to be filled with obstacles. '**for** j **from** 0 **to** n + 1 **do** maze[0,j] :='*' **od**' will seal off the 0th row. We can refer to this whole row as 'maze[0,]' and the assignment 'maze[m + 1,] := maze[0,]' will copy the contents of row 0 into row m + 1 thus blocking it.

We may then write 'maze[,0] := maze[,n + 1] := maze[0,]' to block the eastern and western boundary columns, but this will only work if the maze has the same number of rows and columns, a simplifying assumption that we henceforth make.

Exercise 3.4

Write a do clause that constructs the transpose (rows and columns interchanged) of a 4 x 4 array.

The greatest weakness of the program is that it gets stuck in any dead end, whereas one may find a way out by retracing one's steps and taking an alternative route. As one retreats from a dead end the squares should be marked so that they are not revisited. To do this backtracking, we could keep a list of the squares as they are visited, but it is sufficient to know the previous square by storing the direction of each move.

We can at last write an algorithm.

1. Read in the maze and print it out.
2. Plant extra borders all round the maze.
3. Read in and print out the entrance and exit coordinates. Say the entrance is at maze [i,j].
4. Mark the (i,j) square as traversed.

5. If (i,j) is the exit then stop – maze successfully solved.
6. If maze[i,j + 1] is free, then add 1 to j, store East (as the move direction) and repeat from step 4.
7. If maze[i + 1,j] is free, then add 1 to i, store South and repeat from step 4.
8. If maze[i,j – 1] is free, then subtract 1 from j, store West and repeat from step 4.
9. If maze[i – 1,j] is free, then subtract 1 from i, store North and repeat from step 4.
10. A dead-end square has been reached. If at the start then no path is possible – take a helicopter out; otherwise backtrack by moving in the reverse direction to the last move, changing (i,j) appropriately and repeat from step 4.

We can store the move directions by marking the squares themselves, so that a printout of the final maze would show the path, or by separately storing the path, altering it suitably when backtracking.

Exercise 3.5

Test the maze algorithm on the mazes of figure 3.5, where • marks free squares, * the blocked squares, S is the start and X the exit.

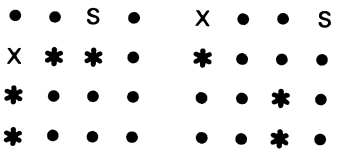

Figure 3.5

It is possible to prove formally by induction that if any path exists through the maze, the algorithm will find it, although it may not be the shortest, and that if there is no path then it will stop at step 10. The proof could be based on the following observations.

The number of free squares diminishes as they are marked, so that step 5 or 10 must be reached. In the latter case there are no free squares accessible from the current square, and if it is the start then every reachable square has been marked without finding the exit.

There are Algol 68 constructs that help in the conversion to a program.

After reading the integer 'size', the declaration '[0:size + 1, 0:size + 1] **int** maze' allows for the extra border hedges. To read the actual maze data only into part of this larger multiple, we could use a do clause but prefer to use *trimmers* '1:size' that allow us to refer only to part of the array. 'read(maze[1:size, 1:size])' reads the data into '**maze**[1,1]', '**maze**[1,2]', . . . '**maze**[size, size]' successively.

Similarly if '[−2:10] **real** vector' has been declared, and followed by the do clause '**for** i **from** −2 **to** 10 **do** vector[i] := i*i **od**' then 'vector[3:9]' refers to seven real numbers in the row vector with values 9,16, . . . 81. This trimmed name is itself indexed with its lower bound automatically set at 1, so that 'vector[3:9] [2]' is the same as 'vector[4]' and refers to the value 16. We can make the subscripts of 'vector[3:9]' run more naturally from 3 to 9 by using '**at**' in a trimmer, as in 'vector[3:9 **at** 3]'. vector[3:9]' is the same as 'vector[3:9 **at** 1]'. 'vector[3:9 **at** 5]' would have lower bound 5 and upper bound 11. 'vector[3:9 **at** 3] [5]' refers to the value 25, as does 'vector[5]'.

Exercise 3.6

The declaration '[1:50,1:50] **int** mat' is followed by a clause that assigns 7*i + 2*j to 'mat[i,j]'. To what values do the following names refer? (a) mat[2,4]; (b) mat[,4] [5]; (c) mat[10:20,4] [5]; (d) mat[10:20, 4:40] [10, 6]; (e) mat[10:20 **at** 5, 4:40 **at** 0] [10,6]

Another way of overcoming the problem of variable sizes for multiples is to use *flexible bounds*. '**flex** [p:q] **real** vec' declares a row called 'vec', initially with subscript bounds p and q, but both of these bounds are flexible. If p > q then the vector is initially the empty row ( ). 'vec := (23.7, 4.2, 7.5)' sets the bounds of the vector at 1 and 3 (they could be set differently by using **at**). The right-hand side of this assignment is a *display* of a row. Once the bounds of a flexible multiple have been set, then indexing must not go beyond these bounds, although the entire multiple may have its bounds changed either by using another display or by assignment of another multiple. For example, after '[−3:2] **real** v; **for** i **from** −3 **to** 2 **do** v[i] :=2*i **od**; vec := v', the new bounds for 'vec' are −3 and 2.

The current upper and lower bounds of 'vec' at any point in the program are given by '**upb** vec' and '**lwb** vec' where **upb** and **lwb** are standard operators. For a higher-dimensional flexible multiple, these operators give the bounds of the first index, and 2 **upb**, 2 **lwb**, etc. give the bounds for the second and higher indices. The use of flexible bounds requires more elaborate storage space when the program is run and one must weigh this additional overhead against convenience.

We decide to store the move directions in a list using the letters N,E,S,W. Then a successful path, omitting dead ends, can be printed out in this form.

In our algorithm we therefore need a multiple value of characters such as '[1:5] **char** path', and then 'path[1]', . . 'path[5]' each refer to a single character. The assignment 'path := ("N","E","S","S","W")' causes 'path' to refer to this multiple display, and 'path[3]', for instance, points to the character "S". A display of characters may be

abbreviated, 'path := "NESSW" ' would assign to 'path' the same row as before.

A declaration of a constant set of characters is possible by an identity declaration (compare '**real** pie = 3.14'), but in this case the bounds must be omitted as they are determined by the number of characters on the right-hand side. After '[ ] **char** title = "AMAZING" ', the lower bound of 'title' is fixed at 1 and the upper bound fixed at 7, and 'title[5]', say, refers to "I". 'title' cannot be changed but 'path' can, although any assignment of less than 5 characters to 'path' will have to be padded out with blank characters and any longer path will require a new declaration. Flexible multiples come to the rescue.

'**flex** [1:0] **char** path' allows 'path' to become a *string* of any number $\geqslant 0$ of characters. This is so frequently used, that the abbreviation '**string**' is defined, synonymous with '**flex** [1:0] **char**'.

'**string** rope := "HOW LONG IS A PIECE OF STRING" ' sets the bounds of 'rope' initially at 1:29. 'rope[13]' is "A" and '**upb** rope' is currently 29. After 'rope := "A SHORTER ROPE" ', '**upb** rope' is 14 and 'rope[13]' is "P". The declaration '**string** heading = "NOOSE" ' fixes 'heading' identically as a string of 5 characters that cannot be altered.

Strings can be compared like characters and can also be concatenated by +, so 'rope := title + "MAZE" ' makes 'rope' refer to the string "AMAZINGMAZE".

Exercise 3.7

Given a string, write a program to print its characters in reverse order, and the number of vowels in the string.

We can give a BNF definition of the multiples that we have met.

⟨ basic type ⟩ ::= **real** | **int** | **char**
⟨ bounds ⟩ ::= ⟨ unit ⟩ : ⟨ unit ⟩
⟨ flexible ⟩ ::= **flex**
⟨ type ⟩ ::= ⟨⟨ flexible ⟩[1] [⟨ bounds ⟩ ⟨,⟨ bounds ⟩⟩*] ⟩ * ⟨ non-

We see that the bounds can be unitary clauses delivering integral values, so that '[0:(read(m); size := m)] **int** column' is permitted.

Just as for basic types, one declaration can include several variables of the same multiple type, thus

'[2:7,2:7] **int** matrix, table, square'

A non-multiple type may be a basic type or other types considered later, such as procedures and structures.

Figure 3.6 is a diagrammatic representation of the repetitive steps of the maze algorithm. Box Z is a refinement of box X in figure 3.4, and

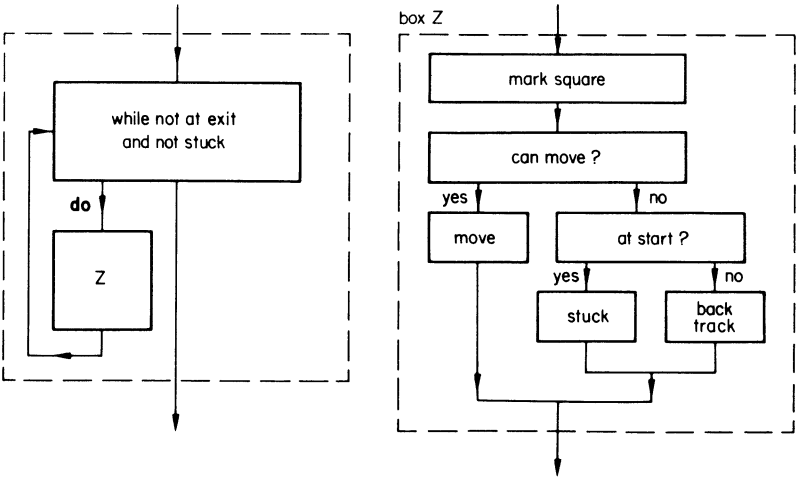

Figure 3.6

this is not surprising since the algorithm itself is a refinement of the earlier unsuccessful solutions.

We have failed and must leave the do loop if the start is traversed again. This can be achieved by using a variable, called 'stuck' say, in the while condition that only takes the values true or false.

Such variables are called *boolean* after the pioneering English logician George Boole, and Algol 68 has a corresponding basic type, **bool**. A declaration '**bool** flag' makes 'flag' a boolean variable to which can be assigned the values true or false, which are denoted by **true** or **false**.

The *logical operators* **or**, **and**, **not** can be applied to booleans with the results given by the following table

P	Q	not P	P and Q	P or Q
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Exercise 3.8

'A eq B' means that A and B have the same truth values.
(a) Show that **not** (P **or** Q) eq ((**not** P) **and** (**not** Q))
P **and** (Q **or** R) eq (P **and** Q) **or** (P **and** R)

whatever the values of the boolean variables P,Q,R.
(b) Under what conditions does

P **or** (Q **and** R) eq (P **or** Q) **and** R?

Initialising declarations like '**bool** flag := **false**', followed by the assignment 'flag := **not** flag' in a later clause, and a subsequent test of the form '**if** flag **then** . . .', are useful to control the execution or non-execution of a group of orders.

not, **and**, **or** have binding power in this order, so that we can omit some parentheses. Thus 'P **and not not** Q **or** R' has the same value as '(P **and** (**not** (**not** Q))) **or** R'. Such combinations of boolean variables using logical operators, and with the values true or false, are called *boolean expressions*.

Arithmetic expressions can also be included in boolean expressions by using the *relational operators* $<, \leqslant, =, \neq, >, \geqslant$, as in, 'x > 26 **or odd** n **and** (**if** y < 6.5 **then** a **else** b **fi**) ≠ c'. The BNF for ⟨ bool exp ⟩ is rather like that for ⟨ arith exp ⟩ and can be found in Appendix II.

We now program the algorithm using a boolean variable, and continuing to represent the maze as a two-dimensional array of characters with "." in free squares and "*" in blocked squares. The output is to include a successful path, if any, as a string made up of the letters N,E,S,W, and any subsequent adventurer could easily follow this path. The string cannot be printed directly as we move since dead ends must be lopped off. Hence we store it in a **string** called 'path'.

```
'begin
        int size; read(size) C of the maze C;
        [0:size + 1, 0:size + 1] char maze;
        read(maze[1:size, 1:size]);
        comment the array is trimmed to read in the actual maze
          comment
        print(("SQUARE MAZE OF SIZE", size, newline,
          newline));
        for i to size do for j to size do print((maze[i,j],space)) od;
                         print(newline) od;
        C we print each row of the maze on a new line C
        to 4 do print(newline) od C to tidy up the output C;
        for k from 0 to size + 1 do maze[0,k] := "*" od;
        maze[ ,0] := maze[ ,size + 1] := maze[size + 1, ]
          := maze[0, ];
        comment we have planted the surrounding hedge comment
        int i,j,k,m; read((i,j,k,m));
        print(("ENTRANCE COORDS = ",i,j,
        "EXIT COORDS = ",k,m,newline,newline));
        bool stuck := false;
        if i < 1 or j < 1 or k < 1 or m < 1 or i > size or j > size or
          k > size or m > size
          then print (("ERROR IN ENTRANCE/EXIT DATA"
          ,newline)); stuck := true fi;
        string path := " "; C set path to space initially C
```

```
        while (maze[i,j] := "T"; C current square traversed C
                not stuck and (i ≠ k or j ≠ m) C not at exit C)
        do if maze[i,j + 1] = "." then j plusab 1; path plusab "E"
                C this means path:= path + "E" C
           elif maze[i + 1,j] = "." then i plusab 1; path plusab
              "S"
           elif maze[i,j − 1] = "." then j minusab 1; path plusab
              "W"
           elif maze[i − 1,j] = "." then i minusab 1; path plusab
              "N"
           elif path = " " then stuck := true C entrance is a dead
              end C
           else int p = upb path; C now backtrack and prune
              path C
                char lastmove = path[p];
                if lastmove = "E" then j minusab 1
                elif lastmove = "S" then i minusab 1
                elif lastmove = "W" then j plusab 1
                elif lastmove = "N" then i plusab 1 fi;
                path := path[1:p − 1]
           fi
        od;
        if stuck then print("NO PATH POSSIBLE")
        else print (("MAZE SOLVED BY FOLLOWING THE
           PATH",newline,path))
        fi
end'
```

There is a problem if the path has length greater than the width of the lineprinter page, since the print routine abandons the output of a string when the end of the line is reached. Later we shall discuss 'format' for the attractive arrangement of output.

This is the output of the above program for a particular maze

```
SQUARE MAZE OF SIZE  3

. . .
. * .
. . *
ENTRANCE COORDS = 1  3 EXIT COORDS = 3  2

MAZE SOLVED BY FOLLOWING THE PATH
WWSSE
```

When developing the program, additional information should be printed to help debugging, such as the coordinates of every move made and of the dead ends pruned. The special 'print' instructions can be removed when the program works correctly. (The question of how one can know that a program is working 'correctly' is a very difficult one.)

The algorithm should also be tested on data where there is no route out, on illegal data and on a maze where several dead ends are encountered before a path is found.

To improve the efficiency and readability of this program drastically, requires the use of procedures and this is our next topic. We also show how to define operators like **'plusab'** between strings if this is not implemented.

Problems for Chapter 3

1. Write a program whose input is a sequence of integers, all less than 400, and whose output is their equivalent in Roman numerals.

2. Given a square grid of heights above sea level, including a mountain top, find a possible path for a river from the mountain top to the lowest point on the grid. (Water does not run uphill but may run diagonally relative to the grid. Assume a path does exist).

3. Write a program to form the matrix product of two matrices, one mxp and the other pxn. (Definition in any textbook on linear algebra).

4. Given a list of numbers in any order, write a program that prints them out in ascending order.

5. If a game of noughts and crosses is represented by a 3 x 3 array, write a program to read in a state in a game and print out the state after the next move, which should complete a line of crosses if possible or else put a cross in the 'best' position.

4 Calendars: Procedures

To answer questions like 'what day was 4 July 1776?', or 'how many days are there to 1 Jan 2005?', we need to study the calendar. The Julian calendar of mediaeval Europe became out of phase with the astronomical year, so that the Pope Gregory XIII promulgated a new calendar in 1582. This was slowly adopted throughout the Western world – by Britain in 1752, but not by Russia until 1917.

The present international calendar decrees that year n A.D. is a leap year when n is divisible by 400 but not by 4000, or if not divisible by 400, when divisible by 4 but not by 100. For example, 1900, 4000, 1977 are not but 2000, 3096, 1976, are, leap years.

We need this information to construct algorithms about dates, for instance to compute the day of the week of a given date. The following steps could be the basis for such an algorithm.

1. If y is the year of the date, find the weekday of 1 Jan y, using the fact that 1 Jan 1973 was a Monday.
2. Find the number of days from 1 Jan y to the given date.
3. From steps 1 and 2 calculate the weekday of date.

Such calculations are made easier by coding the weekdays Sunday to Saturday by the integers 0,1, . . . 6. The calculations of days can then be made discarding multiples of 7. This is called arithmetic *'mod 7'* and the results are always between 0 and 6. For example, 8 mod 7 = 1, (10 + 4) mod 7 = 0, (6 – 38) mod 7 = 3.

There is a corresponding standard Algol 68 operator on two integers, written **'mod'**. It is closely related to another operator called *integer divide* and written ÷. 'n ÷ t' is defined only when n and t are both integers, and has the value q where n = qt + r, $0 \leqslant r < \textbf{abs}\ t$. Another way of looking at n ÷ t is that if $t > 0$, and this is the only situation in which it will be used, it equals **entier**(n/t).

We see that n = (n ÷ t) * t + (n **mod** t), and that n is exactly divisible by t if and only if n = (n ÷ t) * t, or equivalently, n **mod** t = 0.

365 mod 7 is 1 so that the weekday of 1 Jan y advances one day each ordinary year and two in a leap year. From the definition of leap year already given, the formula for the number of days that 1 Jan advances between the base year 1973 and year y, assuming $y \geqslant 1973$, is

(y – 1973) **C** one day forward for each year **C**
+ (y – 1973) ÷ 4 **C** one extra day every 4 years **C**
– (y – 1901) ÷ 100 **C** less one day every 100 years **C**

+ (y − 1601) ÷ 400 **C** plus one day every 400 years years **C**
− (y − 1) ÷ 4000 **C** less one day every 4000 years **C**

Let us call this integer 'advance days(y)', so making clear that it depends on y.

Using the above, we can put more detail into our algorithm.

1. Read in the day, month and year of the date in question, assuming that the year y ⩾ 1973.
2. Calculate advance days(y).
3. Calculate the number b of days from 1 Jan y to the given date, noting that if y is a leap year, Feb has 29 days.
4. 1 Jan 1973 was a Monday, code number 1, so print the day whose code is (1 + advance days(y) + b) mod 7.

Exercise 4.1

Perform this algorithm for the date 29 Feb 2000.

Apart from the input and output, the algorithm comprises two logically distinct calculations, and their isolation as separate steps makes it easier to follow and check.

A self-contained piece of program that takes some data like 'y', or even no data, and defines some operation possibly producing a result, like 'advance days(y)', can be made into a *procedure*. There are standard procedures like 'sin' or 'sqrt' that can be used in any program by writing 'sin(x)' or 'sqrt(x)' where 'x' is real. Similarly, we want to write 'advance days(y)' and get the integer calculated above.

This is achieved by declaring

```
'proc advance days = (int n) int:
   ((n − 1973) + (n − 1973) ÷ 4 − (n − 1901) ÷ 100
         + (n − 1601) ÷ 400 − (n − 1) ÷ 4000)'
```

Let us examine this declaration in detail. '**proc** advance days =' specifies that this is an identity declaration of a constant of *procedure type* named 'advance days' (or 'advancedays' since spaces are not significant). This compares with any other constant declaration such as '**int** dozen = 12', and like them it is placed somewhere in the program before the name is used.

Next follows '(**int** n)'. This is a *parameter list* with just one *formal parameter* called 'n', that is a dummy variable replaced by the actual argument when the procedure is used (see below). 'n' is of type **int**, and '**int** n' in the parameter list looks like a declaration, but is called a *formal declarer*.

Then comes '**int** :'. **int** is the type of result expected, and the colon plays an important role – it indicates that the integral result is to be obtained by evaluating the *procedure body* lying between the brackets or **begin end** that follow the colon. The body is a unit, but in general it may be any serial clause, the value of which, it will be remembered, is that of its final unit. The whole right-hand side is a *routine text*.

After the declaration, we can write for instance 'm := (advance days(y) + 1) **mod** 7' anywhere in the program. This finds the weekday code of 1 Jan y and assigns it to 'm'. 'advancedays(y)' is a *call* of the procedure and produces the appropriate result.

If U stands for any unit having an integral value, then writing 'advancedays(U)' has the same effect as writing '(**int** n = U; ((n – 1973) + . . . – (n – 1) ÷ 4000))'. Thus the formal parameter 'n' is identified with the *actual parameter* U, and the procedure body is then performed, all made into a closed clause. Closing the clause avoids any possible clash of the name 'n' with another 'n' defined elsewhere in the program.

The 'sin' function, and other standard functions, are defined as procedures in the standard prelude and can be used in any program. Another difference from our procedure is that 'sin' takes a **real** parameter and produces a **real** result. In physical calculations, we may want to declare a modification of the sin function

'**proc** newsin = (**real** x, **int** n) **real** : (sin(2*pi*x/n))'

This has two parameters, one real, one integral, and a real result. 'newsin(y,m)', 'newsin(y + t, 36)' are possible calls of this procedure. Many other procedures appear in this and later chapters.

For example, to test if the year y A.D. is a leap year, we can use a procedure

'**proc** leap = (**int** m) **bool** : (**if** m **mod** 4 ≠ 0 **then false elif**
m **mod** 100 ≠ 0 **then true elif** m **mod** 400 ≠0 **then false else**
m **mod** 4000 ≠ 0 **fi**)'

After this declaration, 'leap(y)' has the value **true** when y is a leap year and **false** otherwise.

The procedure body of 'leap' could be replaced by

(n **mod** 4 = 0 **and** n **mod** 100 ≠ 0 **or** n **mod** 400 = 0 **and** n **mod** 4000 ≠0)',

which is also exactly **true** when n is a leap year. In the previous case, with the if clause as procedure body, if m is not divisible by 4 (which happens 3/4 of the time), then m **mod** 4 ≠ 0 and the result **false** is obtained without further calculation. The alternative boolean expression will be evaluated completely for any given year (unless the compiler optimises boolean expressions). This is a serious disadvantage if such an expression is to be evaluated millions of times in a program.

Exercise 4.2

Write procedures with one integral parameter and the names (a) is even'; (b) 'has 2 factors less than 100', and with the results suggested by these names.

The use of procedures to structure a problem logically is the essence of good programming. A step of an algorithm can often be performed by a procedure, so that procedures are another type of box in the diagrammatic representation of an algorithm. The whole algorithm, or large parts of it, may be regarded as a procedure, so it can be a superbox enclosing several others. More commonly, one of the steps may call a procedure. For example to represent the repeated performance of a calculation Z while m is a leap year we have figure 4.1

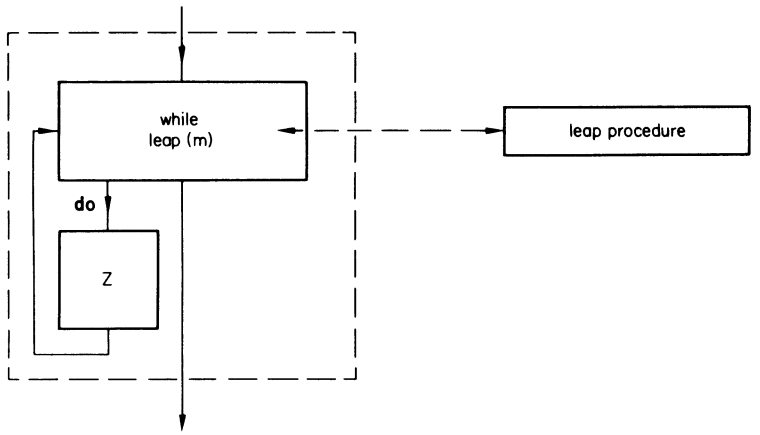

Figure 4.1

If the problem has been analysed carefully, the same procedure may be used at different stages. Apart from the declarations, including procedure declarations, the program translation of the diagram may be quite short and consist of procedure calls and their interfacing through the choice of parameters. In addition, each procedure may be separately tested and debugged using sample data.

When a large task is to be tackled by a team of programmers they often break it down into subtasks, and write individual procedures for these.

In calendar problems we have to handle dates. A date like 3 Mar 2034 is a *structure* with three parts – the day, the month and the year. Algol 68 allows the declaration of such a structure as follows

'**struct** (**int** day, [1:3] **char** month, **int** year) date'

'date' has three *fields*, or parts, which can be selected by writing 'day **of** date', 'month **of** date', 'year **of** date'. Each of these is a variable of

the appropriate type, for instance 'day **of** date' refers to an integer as seen in figure 4.2.

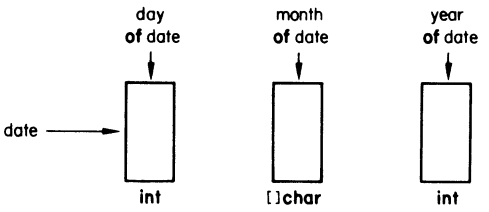

Figure 4.2

However, the main point of the declaration is that structures, like multiples, can be treated as single objects. 'date := (12, "JUL", 1984)' assigns to 'date' the display on the right-hand side, which is a row of denotations of values of the types and in the order that the structure declaration specified. Then

'day **of** date **plusab** 13; month **of** date := "DEC" '

makes 'date' now refer to Christmas day 1984.

The fields of a structure may be of any type, including other structures, and there may be any number ⩾1 of fields. We extend our BNF definition to cater for structures

⟨ non-multiple type ⟩ ::= **struct** (⟨ field ⟩ ⟨,⟨ field ⟩⟩*)
⟨ field ⟩ ::= ⟨ type ⟩ ⟨ identifier ⟩ ⟨,⟨ identifier ⟩⟩*

From this definition we see that adjacent fields of the same type can be collected, as in

'**struct** (**real** bust,waist,hips) pinup'

This has a similar effect to '[1:3] **real** vitalstats', but in the structure case the fields are selected by **of**

'**if** bust **of** pinup < hips **of** pinup **then** print ("PEAR SHAPE") **fi**

whereas in the multiple, the elements are selected by indexing

'**if** vitalstats[1] > vitalstats[3] **and** vitalstats[1] > vitalstats[2]
then print("WINE GLASS") **fi**'.

Exercise 4.3

Given the declaration '**struct** (**string** nationality, **int** age, **struct**(**int** day,[1:3]**char** month, **int** year)birthdate) john', write assignment statements to make John British and born on Christmas day 1940.

We now use the procedural approach in writing a program for the calendar problem, including the new language features. The input is a

date, like 3 MAY 1999 . 'read(date)' will read this row of denotations into their respective fields. The output is to be of the form "3 MAY 1999 IS A MONDAY". The calculation of the advance of the weekday of 1 Jan is modified to allow for years earlier than 1973 (but still A.D.) and is incorporated in a procedure for finding the weekday of any first of January.

To deal with the different number of days in each month, we introduce a convenient device called the *case clause.* It can be represented by figure 4.3. The route taken depends on the value of m. If $1 \leqslant m \leqslant t$, the inner box marked with the label 'm' is performed. If m is not in this range, then the box marked 'other' is performed. This box may be omitted and then if $m > t$ or $m < 1$ the result is undefined. t can be any integer $\geqslant 1$.

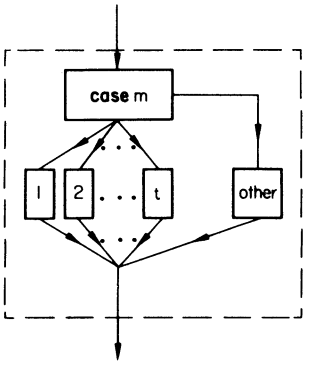

Figure 4.3

We see that this control box is analogous to the if clause where one of two routes is taken.

The case clause is written in Algol 68 as follows, taking three cases as an example, '**case** m **in** U1,U2,U3 **out** S **esac**', where U1,U2,U3 are units and S is a serial clause. One of the clauses is performed depending on the value of m. If $1 \leqslant m \leqslant 3$, then Um is chosen from the list between '**in**' and '**out**', otherwise S is performed. '**out** S' may be omitted if it is certain that $1 \leqslant m \leqslant 3$. 'm' may be replaced by any unit that has an integral value.

It may be necessary for certain actions to be taken if m = 1 or m = 3, but nothing is done if m = 2. This can be arranged by '**case** m **in** U1, **skip**, U3 **esac**'. '**skip**' is a dummy unit that causes no action when the program is executed. It is useful in other situations, as later programs will demonstrate.

The BNF for case is

⟨ case cl ⟩ ::= **case** ⟨ unit ⟩ **in** ⟨ unit ⟩ ⟨,⟨ unit ⟩⟩$_1^*$ ⟨ **out** ⟨ ser cl ⟩⟩^{1} **esac**

case, **in**, **out**, **esac** can be abbreviated to (| |) as can **if**, **then**, **else**, **fi**.

Here is the program

```
'begin
        struct (int day, [1:3] char month, int year) date;
        print((newline, (read(date); date), "IS A    "));
        comment 'read' delivers no result, so the unit '(read(date);
        date)' is used to read into the structure and supply it as a
        parameter for the 'print' procedure, which prints its fields
        in order comment
        proc leap = (int n) bool: (if n mod 4 ≠ 0 then false
                elif n mod 100 ≠ 0 then true elif n mod 400 ≠ 0
                then false else n mod 4000 ≠ 0 fi);
        C leap(y) is true exactly when y is a leap year C
        proc firstday = (int year) int:
        begin int base1 = if year < 1973 then 1976 else 1973 fi,
           base2 = if year < 1973 then 2000 else 1901 fi,
           base3 = if year < 1973 then 2000 else 1601 fi;
        (1 + (year – 1973) + (year – base1) ÷ 4
        – (year – base 2) ÷ 100 + (year – base3) ÷ 400
        – (year – 1) ÷ 4000) mod 7
        end; comment firstday(year) is the integer coding of the
           weekday of 1 Jan year. 1 Jan 1973 was a Monday, coded 1
           comment
        proc elapsed days = (int day,month,year) int:C there are 3
        integer parameters for this procedure and the parameter list
        is equivalent to (int day, int month, int year) C
        ((case month in 0, 31, 59, 90, 120, 151, 181, 212, 243,
                273, 304, 334 out skip esac) +
        if month > 2 then abs leap(year) else 0 fi + (day – 1));
        comment the result is the number of days from 1 Jan year
        to the parameter date. 'out skip' is explained after the
        program. 'abs' applied to a boolean yields 1 for true and 0
        for false. For instance, year y has (365 + abs leap(y)) days
        in it comment
        C to use the last procedure, the length 3 character string
        denoting the month has to be converted into the corre-
        sponding integer. This is done by comparing it with the
        entries in the constant multiple following C
        [,] char monthchecker = (("JAN"),("FEB"),("MAR"),
             ("APR"),("MAY"),("JUN"),("JUL"),("AUG"),
                ("SEP"),("OCT"),("NOV"),("DEC"));
        comment this is an identity declaration and the bounds
        must not be specified when declaring multiple constants.
        The comma in the [] shows that 'monthchecker' is
        two-dimensional, and the display itself shows that it has 12
        rows each of 3 characters. The inner brackets indicate the
        rows, and as usual, one shortens ("J","A","N") to ("JAN")
```

```
        etc. The use of multiples is clumsy, and we suggest
        alternatives after the program comment
        int k := 1; C to be the month number C
        for j to 12 while month of date ≠ monthchecker[j, ] do k
          plusab 1 od;
        comment this sets k to month number. Variables referring
        to a row of char are compared. If this is not implemented
        we need an inner loop to compare each character, (see
        exercise 4.9b) comment
        if k > 12 then print ("ERROR IN MONTH OF DATE")
        else
          int y = year of date, d = day of date;
          print((case (elapsed days(d,k,y) + firstday(y)) mod 7 + 1
            in "SUNDAY","MONDAY","TUESDAY","WEDNES
            DAY","THURSDAY","FRIDAY","SATURDAY"
            out "MISTAKE" esac))
        fi
end'
```

When a case clause has to deliver a value, as in the arithmetic expression of the 'elapsed days' procedure, the **out** option must be present. **'out skip'** is a grammatical formality, since if month has value <1 or >12, then this option is taken with the undefined value **skip**. However, this will be an execution error, whereas the program will not compile without the **out** part. Similarly, when an if clause is part of an arithmetic expression, then the **else** must be present.

'skip' as the last unit of a serial clause *voids* its value. Do clauses are automatically voided. A unit like **'if** x > 0 **then** 1/x **fi'** with no **else** part has a void result.

Exercise 4.4

Use **case** in a procedure with parameters a,b,c to print the real roots of $ax^2 + bx + c = 0$, or to print a message if the roots are not real.

To avoid the two-dimensional multiple 'monthchecker' in the program we could, if the implementation allows, use a multiple of multiples, '[1:12] [1:3] **char** monthcheck' or, better, '[1:12] **string** monthch'. An alternative is to use **bytes**, a basic type of character-string explained in chapter 7.

Another solution introduces one of the most powerful and useful properties of Algol 68. New types can be defined in terms of standard types or those already defined. The mechanism is a *mode declaration* and an example is **'mode integer = int'**. This identifies the new type **'integer'** with **int** and then **'integer** a' has the same effect as **'int** a'.

More realistically, in a program that handles many 3 x 3 real

matrices, we may declare

'**mode matrix** = [1:3, 1:3] **real**' and later in the program perhaps '**matrix** a,b,c'.

In our program we could declare a new type linking the month name and number

```
'mode month = struct ([1:3] char name, int number);
   [ ] month checker = (("JAN",1), ("FEB",2), . . . ("DEC",12));
comment 'checker' is a one-dimensional multiple comment
for j while month of date ≠ name of checker[j] do count := j od;
   k := number of checker[count + 1]'
```

of binds weakly, so that 'name **of** checker[j]' is the same as '(name **of** checker[j])'. Notice the two sorts of multiple selection in '(name **of** checker[2])[1]'. This refers to "F" because 'name **of** checker[2]' points to "FEB", the name field of the second element of the multiple 'checker', which is again a multiple of 3 characters.

We can use structures with just one field, like '**mode seasons = struct** ([1:6] **char** name)'. However, displays of structures with only one field are not permitted. To create an array of four seasons requires something like

```
'[1:4] seasons annual;
   name of annual[1] := "WINTER";
   name of annual[2] := "SPRING" ' etc.
```

unlike the display used to assign to 'checker'.

Some useful modes are in the standard prelude.

'**mode string = flex** [1:0] **char**'
'**mode compl = struct** (**real** re, im)'.

Thus '**compl** z' declares z to be a complex variable with real and imaginary parts 're **of** z' and 'im **of** z', each of which refers to a **real**. 'z := (2.4, 5.2)' gives 'z' the value 2.4 + i5.2, but there is also a special denotation with a binary operator 'i', so 'z := 2.4 **i** 5.2' has the same effect.

The standard arithmetic operators apply to complex numbers, and in suitable contexts real numbers will be widened to complex, just as integers are widened to reals. In addition, the operator '**conj**' gives the complex conjugate, and '**arg**' the principal argument.

Exercise 4.5

Improve your solution to exercise 4.4 by also printing any complex roots.

We extend the BNF to include mode declarations

⟨ identity decl ⟩ ::= **mode** ⟨ modename ⟩
= ⟨ type ⟩ ⟨,⟨ modename ⟩ = ⟨ type ⟩⟩*
⟨ modename ⟩ ::= ⟨ new keyword ⟩

where the latter is an identifier in boldface type (or underlined) that is not already in use. We add modename to the class of non-multiple types, since although it may, like **matrix**, refer to a multiple, the name itself does not start with '[', the multiple bracket.

In the program, it was remarked that identity declarations of multiples must have no bounds, as in '[] **int** small primes = (2,3,5,7)'. This is put into the syntax by

⟨ formal type ⟩ ::= ⟨[⟨,⟩*]⟩*⟨non-multiple type ⟩
⟨ identity decl ⟩ ::= ⟨ formal type ⟩ ⟨ identifier ⟩
= ⟨ unit ⟩ ⟨,⟨ identifier ⟩ = ⟨ unit ⟩⟩*

Any declaration is associated with a certain region of program called its *range*. This extends from the opening-context symbol most closely preceding the declaration to the matching closing-context symbol – examples are from **begin** to **end**, **if** to **fi**, **then** to **else** (or to **fi** if no **else** is present), **else** to **fi**, **case** to **esac**, **in** to comma, comma (separating cases) to comma or **out**, **out** to **esac**, and similarly for the abbreviations or combinations of these symbols, like (to) or **elif** to **fi**.

Ranges within ranges are said to be at different *levels*. Two variables with the same name can only be declared in the same range if they are at different levels – for instance

'**begin real** y := 3.8; (**real** y := 0; print(y)); print(y) **end**'

will print 0 and then 3.8. The inner variable 'y' is quite distinct from the outer 'y' which goes into suspended animation while the inner lives. The *scope* of the outer 'y' is its range with the inner range excised, since another 'y' has been declared.

Exercise 4.6

(a) What is printed on execution of the following?

'**begin int** b, c := 2; c := 3∗c; (**int** c := 2; b := c + 1);
print((b,c)) **end**'

(b) Determine the scopes of the variables in the following (useless) program.

'**begin real** y; read(y); **real** x; **if** y > 3 **then real** z := 8 **else** x := 5 **fi**;
(**real** z,x; z := y; (**real** t; z > 2 | **real** w; y **plusab** 4; **real** z;
z := y | x := z)); x := y **end**'

1974

JANUARY

SUN	MON	TUE	WED	THU	FRI	SAT
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		etc.

Figure 4.4

We can use procedures to print a calendar for one year in the form of figure 4.4.

1. Read in 'year' and print year heading.
2. Set 'month' to 1, for January.
3. Set 'first' to the weekday code of 1 Jan year. (0 to 6 representing Sunday to Saturday.)
4. Print month and weekday headings.
5. Print dates for each week line by line.
6. If 'month' $<$ 12, increase 'first' by the number of days in 'month' mod 7, add 1 to 'month' and repeat from step 4.

The printing requires a careful layout of the results, so we use

```
'proc space = (int t) void: (to t do print(" ") od)'
```

'void' is a special type of result used when the procedure does not produce a result. It indicates that the procedure body is voided.

We can now print the year in the middle of the lineprinter page of 120 columns by

```
'proc middleprint = (int year) void: (space(55); print((year, newline,
   newline)))'.
```

We assume that printed integers occupy 9 places and that leading zeros and + signs are not printed.

Step 3 of the algorithm can be translated into 'first := firstday(year)', using a procedure defined earlier in the chapter.

In step 4, the weekday headings are the same for each month so our procedure has no parameters at all

```
'proc weekdaysprint = void:
        begin space(1); C to move day names over dates C
        for n to 7 do space(13); C since each date will occupy 16
```

```
spaces in procedure 'weekprint' below C
print((n | "SUN","MON","TUE","WED","THU","FRI",
   "SAT" | skip))
od; print(newline)
end'
```

Exercise 4.7

Write a procedure, with the month number as parameter, to print the month heading for the calendar.

Step 5 of the algorithm requires consideration. If, for example, the first of the month is 2, representing Tuesday, we want to print blanks below SUN and MON, and the dates 1 to 5 below TUE to SAT. This can be done by starting the week with the negative date $1-2=-1$ and not printing dates <1. Similarly, for the last week in the month we suppress the printing of dates $>$ number of days in month. Hence

```
'proc weekprint = (int d,n) void : C d is the first date in the week and
   n is the no. of days in month C
(for i from d to d + 6 do
space(7); if i < 1 or i > n then space(9) else print(i) fi
                    od);
C recall that integers occupy 9 spaces (in our implementation) C
proc monthprint = (int month, first) void :
C first is the weekday code of the first day of month C
(int k = nodays in(month, year);
for j from(1 – first)by 7 while j ⩽ k do
weekprint(j,k); print(newline) od)'
```

Exercise 4.8

Write the procedure 'nodays in', with parameters month,year

The test $j \leqslant k$ is made each time around the loop, and the constant k was declared to avoid recalculating 'nodays in(month, year)'. 'year' is a *global variable* in this procedure, not being a parameter or a local variable declared in its body. Such globals should be avoided in general – they can too easily be out of scope or inadvertently altered by the procedure.

All our procedure declarations have been identity declarations of the form **proc** name = a routine text. This adds the following to the BNF

```
⟨ identity decl ⟩ ::= proc ⟨ identifier ⟩ = ⟨ routine text ⟩
                      ⟨,⟨ identifier ⟩ = ⟨ routine text ⟩⟩*
⟨ routine text ⟩ ::= ⟨ void routine ⟩ | ⟨ non-void routine ⟩
⟨ void routine ⟩ ::= ⟨(⟨ formaldec ⟩⟨,⟨ formaldec ⟩⟩*)⟩[1] void:
                      ⟨ closed cl ⟩
```

⟨ non-void routine ⟩ has a similar definition with ⟨ formal type ⟩ replacing **void**. Notice the formal decs as parameters. If a parameter is a multiple, we omit its bounds since they are supplied by the actual parameter, but retain the brackets and any commas between them. Thus

⟨ formaldec ⟩ ::= ⟨ formal type ⟩ ⟨ identifier ⟩ ⟨,⟨ identifier ⟩⟩*

The name of a procedure can be declared without attaching it to a particular routine – compare '**real** x' and '**real** x = 3.6'. The declaration must include the types of parameter and the result expected, for example '**proc (real) real** trig'. In the course of the program we may assign to 'trig' any routine text with one **real** parameter and a **real** result, or the name of another procedure of this type

'trig := **(real** w**) real**: (w * tan(w))' or
'trig := **if** angle < pi/2 **then** sin **else** cos **fi**'

When we call 'trig(x)', it is the routine defined by the latest assignment to 'trig', a variable of procedure type, that is used. Using the same name for two different procedures in this way is rather unusual.

The above form of declaration for **proc** type names shows that the proper identity declaration of 'leap', for instance, is

'**proc (int)bool** leap = **(int** n**) bool**: (. . .)'

(where . . . stands for the procedure body). '**proc** leap = **(int** n**) bool**: (. . .)' is an allowed abbreviation for this, since the display on the right already specifies the types of parameter and result.

A procedure defines an *operation* on its parameters. For example, if '**mod**' were not defined in the standard prelude, we could easily declare a '**proc (int, int) int** modulus' such that 'modulus(a,b)' has the value 'a **mod** b'. However, we can use a routine text directly to *define new operators* that are infixed (written between their operands) in the usual manner, as follows

'**op mod** = **(int** a,b**) int**: (a – (a ÷ b) * b)'

After this definition '9 **mod** 7' is 2. One must also assign a priority to any defined operator, as discussed in the next chapter. Here we assume that the lowest priority is automatically given to our new operators. (The implementation will probably do this, although unary, that is one-operand, operators are all given the top priority).

Any invented boldface word can be used for an operator. Suppose our program involved taking a lot of reciprocals, with the convention that 1/0 is to be a very large number. Then '**op rec** = **(int** x**) real**: ((x ≠ 0 | 1/x | max real))' allows one to write '**rec** w', where w is int, and obtain 1/w if w ≠ 0, and the largest real that exists in the implementation otherwise. 'max real' is an *environment enquiry*.

Exercise 4.9

(a) Declare a unary (one-operand) operator '**ceil**' such that '**ceil** x' is the smallest integer ⩾ the real number x.
(b) Declare a binary (two-operands) operator '≠' between variables of type **[]char**, assuming ≠ is defined in the standard prelude only between single characters.

Unlike **proc** variables, **op** symbols cannot be declared separately from the routines defining them. They only occur in identity declarations of the form

⟨ identity decl ⟩ ::= **op** ⟨ opsymbol ⟩ = ⟨ non-void routine ⟩

The ⟨ opsymbol ⟩ allowed depends on the implementation, but unused characters like £ can always be used, and the standard operators can be re-defined or extended to different types.

An example used in chapter 3 is the concatenation of a string and a character, for example, "BAT" + "H" gives "BATH".

```
'op + = (string s, char c) string:
   (int m = upb s; flex[1:m + 1] char new;
   if m ≠ 0 comment string s non-empty comment
   then new[1:m] := s fi;
   new[m + 1] := c;
   new)'
```

In this definition we could not simply assign 's[m + 1] := c' because s currently has the bounds 1 to m which, although flexible, cannot be stretched in this way.

Exercise 4.10

Define a concatenation operator between strings.

We could not have assigned to s in the procedure for a much more important reason. Consider a simpler example of '**proc** double = (**int** n) **void**: (n := 2*n)'. A call 'double(t)' is theoretically replaced by '(**int** n = t; n := 2*n)' and we see that the procedure must be wrong since n has been identified with the value of t and so we cannot assign to it. A parameter specified '(**int** n)' is a true integer and not a name referring to an integer.

To write a doubling procedure, we want a *name* and not a value as a parameter. This is achieved by placing '**ref**' before the formal parameter specifier

'**proc** double = (**ref int** n) **void**: (n := 2*n)'

'n' is a *reference* to an integer, that is a name pointing to a place where an integer can be stored. This is not a new idea – the first diagrams in chapter 2 show that when '**real** y', for instance, is declared, then 'y' is a reference to a real.

The actual parameter of a call of 'double' must also be a name. Thus '**int** t := 6; double (t)' is equivalent to '**int** t := 6;(**ref int** n = t; (n := 2*n))'. The identity declaration here states that 'n' and 't' are the same names. The name 'n' on the left-hand side of the assignment 'n := 2*n' therefore, also refers to the same value as 't'. On the right-hand side, 2*n is evaluated to the integer 12, and both 'n' and 't' are made to refer to it. The effect is to double the value that 't' refers to, and the local name 'n' 'dies' when we leave the closed clause where it is declared. 'double(6)' would be an incorrect call of this procedure, since 6 is not a name.

An example more relevant to this chapter would be a procedure to advance the date to the next working day, assuming a 5-day week and ignoring public holidays.

```
'proc advance date = ([] char day, ref int date,month,year)void:
   (int n = nodays in(month, year);
   date plusab (day = "FRI" |3| : day = "SAT" |2 | 1);
   C |: is an abbreviation of elif C
   if date > n then date minusab n;
                  if month = 12 then month := 1 ; year plusab 1
                                 else month plusab 1 fi fi)'
```

The parameters 'month', 'year' and 'date' are all **ref int** because they may be altered by the procedure. However, the procedure 'nodays in' expects a pair of integers and not their names as parameters, so that when 'month' and 'year' are used as actual parameters it is their value that is taken.

One important use of **ref** is for multiple value parameters. It is quicker to pass the name of an array, rather than a complete copy, as a parameter to a procedure. Thus, to define a matrix multiplication operator we could write

```
'op * = (ref [,] real a,b) [,] real:
comment As already stated, multiples as parameters omit their
bounds. We assume the lower index bounds are 1 and that 2 upb
a = 1 upb b, making matrix multiplication possible. A more complete
procedure would check these facts and reject unsuitable parameters.
comment
begin int m = 1 upb a, C this is the same as upb a C
   p = 2 upb a, n = 2 upb b;
   [1:m, 1:n] real d;
   for i to m do for j to n do
   ref real dij = d[i, j] ; C dij refers to the same real as d[i, j] but its
```

```
    use involves no further index evaluation and so is more efficient C
    dij := 0;
    for k to p do dij plusab a[i, k] *b[k, j] od od od;
    d
end'
```

In '[1:4, 1:4] **real** x, y, z; read((x, y)); z := x *y' only the names x and y are passed to the routine defining *. Unfortunately, the result of the procedure is a 4 x 4 matrix and not its name. Hence '(x*y)*x' would not work. It is true that the last unit in the routine is the name d, but one can only access its *value* since d is declared locally and is out of range when we leave the procedure. To obtain a **ref** type variable as the result needs a more advanced feature as explained in the final chapter.

Exercise 4.11

Write a procedure with **ref** parameters to calculate the inner product Σ a[i] *b[i] of two vectors a, b.

A procedure may have **proc** type variables as parameters. An earlier problem gives an example: To sum a series involving a variable until the terms are smaller than a prescribed epsilon, it is sufficient to know epsilon, the first term, the value of the variable, and a procedure that, given n, supplies a multiplying factor to obtain the (n + 1)th term from the nth. The series may diverge, and so we should also specify the maximum number of terms to be summed. If this number is reached, we trap the error by calling another procedure. Hence the declaration is

```
'proc series sum = (real firstterm, epsilon, variable,
                    int maxno terms, ref proc (int,real)real mult,
                    proc void error exit) real:
   (real sum, term;
   sum := term := firstterm;
   for n while abs term > epsilon
   do term timesab mult(n, variable);
      sum plusab term;
      if n > maxno terms then error exit fi
   od;
sum)'
```

The 'mult' parameter is specified as a **ref proc** to avoid copying the whole procedure when 'series sum' is called. To sum the series of chapter 2, the actual parameter would be 'multiplier' declared by

```
'proc (int, real) real multiplier: = (int n, real x) real:(-(x↑2)/(2*n + 1)
```

What form should the 'error exit' procedure take? A call of it

indicates that something has gone wrong, and we want to leave the natural sequence of our program and take special action. This is done by transferring to some statement elsewhere in the program.

Any unit occurring after all the declarations of a serial clause can be *labelled* by putting an identifier and colon in front of it such as 'divergent: print("DIVERGENT")'. One does not need to declare the *label* 'divergent'.

To use this device, the *goto clause* '**goto** divergent' written in the program causes the next statement executed to be that following the label. Labels have a scope similar to that of identifiers, that is as regards the range in which they occur. Goto clauses should be avoided because they disturb the natural sequential development of a program, and make it hard to read and check.

Sometimes we reach a situation where we want to leave the whole program without further ado. This can be achieved by '**goto** finish', and inserting before the final '**end**' the dummy labelled-statement 'finish: **skip**'.

We can now construct a call of the procedure 'series sum' that deals with the series of chapter 2, assigning the value to 'sum' as follows

'sum := series sum((read(y);y), 1 & −11, y, 100, multiplier **C** defined above **C**, **void**:(**goto** divergent))'

At the end of the program we could then have

```
goto finish;
   divergent: print("100 TERMS SUMMED AND TERMS STILL >
                                    1 & −11");
   finish: skip
end'
```

It is possible to write '**goto** divergent' for the parameter of type **proc void**, omitting '**void**:'. This is automatically made into a procedure, that is treated as if it really were the routine '**void**: (**goto** divergent)'.

This may seem an unduly complicated way of summing a simple series, but different assignments to 'multiplier' and different values of the other parameters allow any series to be summed by the same procedure.

Exercise 4.12

Construct a suitable call of the above procedure to sum the exponential series $1 + x + x^2/2 + x^3/6 + \ldots$

Parameters of type **proc** can be used rather subtly

```
'proc sum = (proc int a, ref int i, int n) int:
   (int s := 0;
   for k to n do i := k; s plusab a od; s)'
```

If j, m are integer variables, 'sum(**int**:(j*j), j, m)' has the value $1^2 + 2^2 + \ldots + m^2$.

Even more strikingly

'sum (**int**:(sum (**int**:(j*j + m*m), j, m)), m, 20)'

yields the double sum

$$\sum_{m=1}^{20} \sum_{j=1}^{m} (j^2 + m^2).$$

We see that assigning a reference-type parameter in a procedure body may have effects on other parameters that are not obvious.

Exercise 4.13

Complete the declaration '**proc** length = (**proc real** xi, **ref int** i, **int** n) **real** :' so that 'length (**real**: (a[m]), m, k)' is the length of 'a' (sqrt of sum of squares of its components).

The 'series sum' procedure is *iterative*, calculating the sum by adding term by term. Another method is to use the sum to (n – 1) terms to calculate the sum to n terms.

```
'proc recsum = (int n, ref proc (int) real f) real:
                comment f(n) is the nth term of the series comment
                  (if n = 1 then f(1) else recsum (n – 1, f) + f(n) fi)'
```

This procedure is *recursive* since its body contains a call on itself. Consider 'recsum(5, f)', f suitably declared. This calls recsum(4, f), ... recsum(1, f) successively, and the latter evaluates to f(1), which is substituted in the previous call recsum(2, f), evaluates this to f(1) + f(2) and so on, giving eventually the sum of five terms.

A procedure declaration may call itself, since in general an Algol 68 definition can refer to any object already defined or in the *process of being defined.*

When a recursive procedure is executed, one visualises each call of the procedure as a different *incarnation* of it, with the values of any variables local to the procedure body being held in suspense while the new call is performed.

Necessary conditions for a recursive procedure not to loop indefinitely, are that at least one parameter be altered on each call, and that there should be a route through the procedure avoiding any recursive call. The following example shows that these conditions are not sufficient to make a sensible procedure

```
'proc factorial = (int n) real: (if n = 1 then 1 else factorial(n + 1)/n
  fi)'
```

Exercise 4.14

Write a correct recursive definition of factorial, and also an iterative definition.

Recursion is an important concept in computing. Many problems are solved by assuming we can solve an easier case, and this leads to a recursive procedure for the solution. Complicated objects often have a recursive definition, for example a list can be thought of as its head followed by another list.

The next chapter contains several illustrations of the above points.

Problems for Chapter 4

1. Write a procedure to print a calendar for a given year with three months printed side by side across a page

2. Construct a procedure that accepts as input the Morse-code representation of a digit and produces the decimal digit as output. Use this procedure to convert and print strings of Morse-code digits as integers. (The digits n = 0 to 5 have a Morse code of n dots followed by (5 – n) dashes – use a minus sign – and the digits n = 6 to 9 have a code of (n – 5) dashes followed by (10 – n) dots. Dots and dashes to be separated by a single space and digit strings by a comma).

3. Write a procedure to verbalise numbers of up to 7 digits and to print the result, for example

```
12          TWELVE
123         ONE HUNDRED AND TWENTY THREE
7654321     SEVEN MILLION SIX HUNDRED AND FIFTY
            FOUR THOUSAND THREE HUNDRED AND
              TWENTY ONE
```

4. Construct a procedure that computes the number of days from any given date to your next birthday.

5. Write a recursive procedure to reverse the order of characters in a string. Use the procedure in a program to test if a given sentence is a palindrome (the same in forward and reverse order, as "MADAM IM ADAM").

5 Growing a Tree: References

The objects represented in previous programs have not varied greatly after their initial appearance. The size of the maze was fixed by the data, although the output path waxed and waned by one character at a time. Similarly in the calendar problem, a structure was useful, but its shape remained static.

This chapter examines a problem in which an object may alter its size as it is being manipulated. Consider the evaluation of the arithmetic expression 'a + b*c – d/e'. The steps are (i) multiply b and c, (ii) add a to result, (iii) divide d by e and subtract. This order of evaluation can be made more automatic by transforming the expression into *reverse Polish* or *postfix* form, 'abc* + de/–'. Here each of the binary operators *, +, /, – occurs immediately after the two operands that it acts upon, and evaluation is a mechanical process of applying them from left to right. As another example, '2*t↑2 – t*(y + z)' transforms into '2t2↑*tyz + *–' and no parentheses are necessary in this new form.

Exercise 5.1

Write down the postfix forms of 'a↑b*c/d', 'a/b*c↑d', 'a/(b*c)↑d'.

To transform expressions in the above way, one has to know that multiplication has higher *priority* than addition, that is, it binds more tightly or has greater precedence. This was briefly mentioned in chapter 2.

Algol 68 associates a priority between 1 and 9 with each of its standard binary operators, * and + having priorities 7 and 6 respectively. It also allows us to associate priorities with our own operators defined by an **op** declaration

```
'prio min = 9;
op min = (int i,j) int: ((i < j | i | j))'
```

'**min**' now has priority 9, greater than that of +, so 'a **min** b + c' is the smaller of a and b added to c. In most implementations, an operator declared without a priority has the default priority 1, weakest of all. If **min** had priority 1 then 'a **min** b + c' would be the smaller of a and b + c.

All unary (one-operand) operators, such as **not** and unary – have priority 10; for example $-3\uparrow 2$ is $(-3)^2 = 9$, whereas $0 - 3\uparrow 2$ equals $-3^2 = -9$. Priorities are included in the list of standard operators in

Appendix I. The BNF of priority declarations is

⟨ identity decl ⟩ ::= **prio** ⟨ opsymbol ⟩ = ⟨ nonzero digit ⟩

We now construct an algorithm to transform simple unbracketed arithmetic expressions into postfix form. The input is a string made up of single letters or digits as operands, operators +,–,*,/,↑, spaces to be ignored, and £ as terminating symbol, and is mathematically well-formed. The output is to be the postfix form of the expression defined by the string.

The operands appear in the same order on output as on input, and it is only the operators that have to be moved behind the operands, taking care to place the ones with higher priority first. An analogy is the railway shunting yard of figure 5.1, where the truck last into the yard must be removed first before any other trucks lower down can be taken out. We may regard the operators as trucks to be shunted depending on their priority. Thus for 'a + b – c£', we successively print a; shunt +, print b; compare – with the + in yard, it has equal priority, so remove and print + and put – into the yard; print c; end of string, so empty yard printing –.

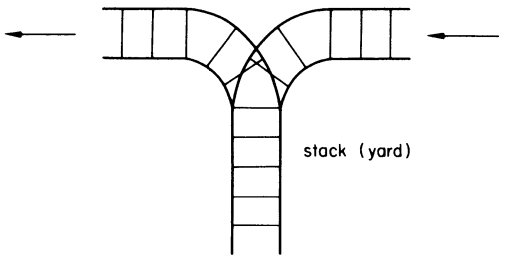

Figure 5.1

The only symbol ever needed from the yard is the last put in, and the only actions required are to add a new symbol or delete one from the top. We think of it as a list with its elements piled one on top of the other and refer to it as a *pushdown list* or *stack*. The actions are then called 'pushing' a symbol on to, or 'popping' a symbol off, the stack.

For 'a*b – c£', the algorithm gives: print a; push *; print b; compare – with the top of stack (*), pop and print *; push –; print c; pop and print stack (–). However, for 'a + b*c↑d – e£' we see that +,*,↑ must all be pushed on to the stack before comparison with – causes them to be popped.

The key step is thus to compare the priority of an incoming operator with that last stacked. We can make this apply to the first operator as well, by introducing a dummy operator of artificially small priority for comparison. The above ideas lead to the following algorithm.

1. Put a dummy operator, ? say, of priority −1 on a stack of operators called 'oplist'.
2. Read next character into the variable c.
3. If c = " " then repeat step 2.
4. If c = "£" then print out oplist in reverse order (last in first out), except for "?" and stop.
5. If c is an operand, print it and repeat step 2.
6. c is an operator. Compare its priority, p say, with that of the last operator in oplist, q say.
7. If $p > q$, add c to oplist and repeat step 2.
8. If $p \leqslant q$, print the last character in oplist, delete it, and repeat step 6, using new last operator in oplist.

Exercise 5.2

Using a pushdown list, write an algorithm that evaluates a postfix form. For example '234+*' would first become '27*' and then '14'.

Before writing a program for the above algorithm, we complicate matters by allowing parentheses in an expression. These modify priority of evaluation by distinguishing, for instance, 'a + b*c' from '(a + b)*c'. The corresponding postfix forms 'abc*+' and 'ab + c*' are unambiguous without parentheses.

When an opening parenthesis occurs, this indicates the start of a subexpression, and we do not want operators within the parentheses to be compared with those outside. Hence the symbol "(" on the input string should initiate some special action. This continues until a matching ")" closing the subexpression, when all operators on the stack down to the "(" must be popped off.

The algorithm requires us to test if a character is one of ?,£,space,(,),+,−,*,/,↑, as all others are assumed to be operands. The following procedure does this checking.

```
'proc matchar = (char c, string s, ref int i) bool:
   (bool match := false;
   for k to upb s while not match
   do if c = s[k] then i := k; match := true fi
      comment otherwise match still false and repeat until end of
      string is reached comment od;
   match)'
```

The call 'matchar(ch, "?£ ()+−*/↑", numb)' yields **true** if 'ch' is one of the special characters in the string – note the space character between £ and (– and sets 'numb' to its position in the string; otherwise it yields **false** and 'numb' is unchanged. The formal parameter i is specified as **ref int** since a successful match causes an assignment to i within the

procedure. There is a standard-prelude procedure 'char in string' defined similarly to 'matchar'.

Returning to the algorithm, it is not necessary to store the actual operators in a stack. Instead, we construct a fixed string of operators and stack integers that point to the position of operators in this string. The same integer can also point to the priority of the operator in a fixed multiple of priorities. It is simpler to stack integers rather than operators and their priorities, although it causes some subscript calculations.

The structured figure 5.2 for the algorithm involves a nesting of repetitions.

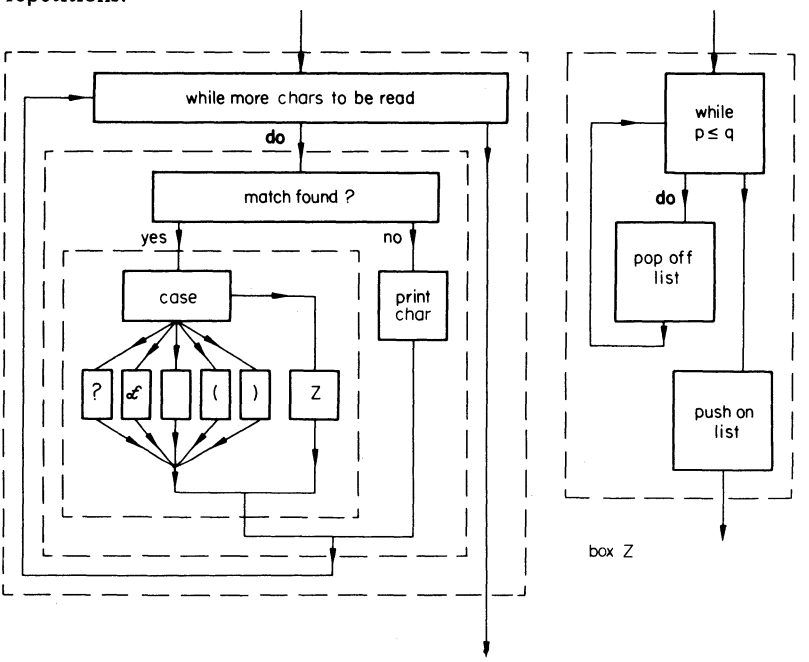

Figure 5.2

As an Algol 68 program this becomes

```
'begin
        string operators = "?£ ()+-*/↑";
        [ ] int priorities = (-1,0,0,1,2,6,6,7,7,8);
        comment no bounds in an identity row declaration.
        Algol 68 priorities have been given to standard operators,
        and priority -1 to the dummy operator "?". "(" has
        priority 1, less than any proper operator to ensure
        subexpression evaluation is started. The priority of ")" is
        not used comment
```

```
        [1:20] int stack; C may need flex for longer expressions C
        char ch; bool morechars := true; C expect at least one char
        C
        int position, top of stack := 1;
        stack[1] := 1; C pointing to the dummy operator C
        while morechars
        do read(ch);
            if not matchar(ch, operators, position)
            then print(ch) C operand C else
            case position
            in C ? found(expression ill-formed) C
                morechars := false,
                (C £ so empty stack C
                for i from top of stack by -1 to 2 do
                        print(operators[stack[i]]) od;
                morechars := false C to leave do loop C),
                C space found C skip,
                C open bracket, stack its position = 4 C
                stack[top of stack plusab 1] := 4,
                (C closing bracket C for i from topofstack by
                -1 to 2
                while stack[i] ≠ 4 do
                    print(operators[stack[i]]);
                    topofstack minusab 1 od;
                C now delete opening bracket C
                topofstack minusab 1)
            out C must be arithmetic operator C
                while priorities[position] ≤ priorities[stack
                [topofstack]]
                do C popoff and print top operator and
                    compare current operator with next top
                    operator C
                    print(operators[stack[topofstack]]);
                    top of stack minusab 1 od;
                C new op has greater priority, so push on C
                stack[topofstack plusab 1] := position
            esac fi C repeat for next character C
        od C ? or £ found since do clause was left C
    end'
```

Exercise 5.3

Rewrite the above program: (a) to accept a sequence of expressions terminated by "%"; (b) to put the postfix form into a string called 'result' instead of printing it.

Some manipulations of arithmetic expressions are difficult when they are in string form. For example to replace 'a' by 'e + f' in 'a + b*c – d/a', extra space has to be created. A more vivid way of representing expressions is by a *tree diagram*. (A) and (B) in figure 5.3 are the diagrams of 'a + b*c' and '(a + b)*c', respectively.

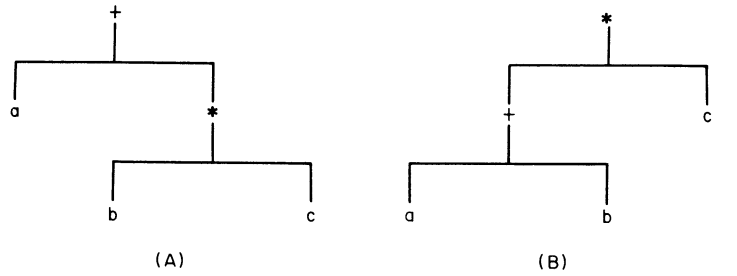

Figure 5.3

This representation also makes the substitution of 'a' by 'e + f' easy. Each 'a' is replaced by '+' with branches 'e' and 'f' hanging from it.

A *binary tree*, like those in the above figure, is a collection of *nodes*, and each node either has two branches leading to other nodes, or no branches at all when it is called *terminal*. All branches point downwards (an Australian tree?) and unless the tree is empty (no nodes or branches) there is a unique topmost node called its *root*, and all the other nodes have one branch coming in from above.

Exercise 5.4

The length of a path is the number of nodes through which it passes. (a) What is the maximum number of nodes in a tree, all of whose downward paths have length $\leqslant$ N? (b) What is the length of the longest possible downward path in a tree with $\leqslant$ M nodes?

The nodes in this sort of tree can be represented in Algol 68 by a structure, by specifying the operator or operand labelling it, and if it is not terminal, the nodes attached to its left and right branches. We number the nodes, and since the tree may contain any number, use a flexible multiple

'**flex**[1:0] **struct** (**int** left, **char** operator, **int** right) nodes'

'nodes[1]' is always the root of the tree. If 'nodes[m]' is terminal, we set its 'left' and 'right' fields to 0, otherwise they are the integer indices of the nodes attached to it. Statements to construct the tree for 'a + b*c' would be

'nodes[1] := (2, "+", 3);

```
nodes[2] := (0, "a", 0);
nodes[3] := (4, "*", 5);
nodes[4] := (0, "b", 0);
nodes[5] := (0, "c", 0)'
```

This construction can be represented by figure 5.4.

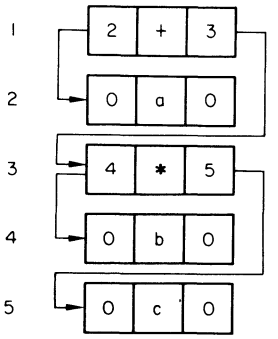

Figure 5.4

The above formation of 'nodes' would not work unless assignments had been made earlier in the program so that '**upb** nodes' is at least 5. We can avoid this problem in setting up another multiple to represent '(a + b)*c' by assigning a display

```
'flex[1:0] struct(int left, char operator, int right) newnodes :=
((2,"*",5), (3,"+",4), (0,"a",0), (0,"b",0), (0,"c",0))'.
```

We can select from these multiples. For instance "+", the first operator in the left branch of root, is 'operator **of** newnodes[left **of** newnodes[1]]'. The leftmost operand in the first tree, which is "a", can be found by

```
'int i := 1; char leftmost;
 while left of nodes[i] ≠ 0 do i := left of nodes[i] od;
 leftmost := operator of nodes[i]'
```

Exercise 5.5

Given an expression in the above tree form, write a program to find the first operator in its postfix form.

To declare several different trees without repeating their full structure, a new mode can be used.

```
'mode trees = flex [1:0] struct (int left, char operator, int right);
 trees oak, ash, elm'
```

The names 'oak', 'ash', 'elm' now refer to a multiple of nodes, each node a structure of the desired sort.

Our representation is still unsatisfactory in two ways. First the nodes are numbered in an arbitrary way that disguises the tree structure and second, manipulations can be tedious. To 'add' two trees would involve setting up a new tree with root labelled "+", and copying the trees into the left and right branches after adjusting the numbering of nodes.

Exercise 5.6

Write a procedure to add two trees in this way.

These difficulties can be avoided by referring to the whole subtree hanging from each branch and not just the next node. Thus

'**mode tree** = **struct** (**ref tree** left, **char** operator, **ref tree** right)'

The object being declared appears within itself, so that this definition is recursive, like the procedures at the end of the last chapter. If we had declared a field '**tree** left', this would be a vicious circle – to decide what a tree is would involve deciding what a tree is, and so on indefinitely. However, the left and right fields are of type **ref tree**, and so are occupied by the names of trees, not complete copies of the tree structure.

Adding two expressions in tree form is now easy. Suppose '**tree** expra, exprb, sumexpr' have been declared, followed by statements to construct 'expra' and 'exprb'. Then the assignment 'sumexpr := (expra, "+", exprb)' sets up the sum tree without copying or moving subtrees.

Terminal nodes can be represented by '(**nil**, "a", **nil**)', where '**nil**' is a *reference to no value*. It may be thought of as a denotation for a **ref void** variable.

Trees are now collections of subtrees directly referring to one another, instead of collections of integer-subscripted nodes. To con-

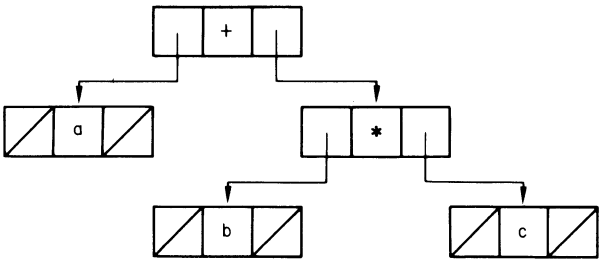

Figure 5.5

struct the tree for 'a + b*c' we may write

```
'tree anode, bnode, cnode, subexpr, expr;
   anode := (nil, "a", nil); bnode := (nil, "b", nil);
   cnode := (nil, "c", nil); subexpr := (bnode, "*", cnode);
   expr := (anode, "+", subexpr)'
```

Figure 5.5 is a picture of this tree.

Exercise 5.7

Write a declaration of '[1:7] **tree** gum' so that it represents a*b/c + d and gum[1] points to the root.

Having found an elegant way to represent trees in Algol 68, we use it to solve some problems.

The following algorithm turns the postfix form of an arithmetic expression into its tree form.

1. If input string has length 0, then stop.
2. If last character in string is an operand, make it into a tree, delete it and stop.
3. Set operator of tree equal to last character in string and delete it from string.
4. Apply the whole algorithm to the remaining string to obtain the 'right' branch.
5. Repeat step 4 to construct the 'left' branch.

For example, "ab*" has an operator as its last character, so its tree has operator field "*". Step 4 acting on "ab" makes the right field refer to '(**nil**, "b", **nil**)' and leaves the string "a". Step 5 then constructs '(**nil**, "a", **nil**)' in the left field. The trimming of the string always occurs at the correct place if the expression is well-formed.

This algorithm is a clear case for a recursive procedure. However, we must first declare a multiple of **tree** outside the procedure in order to store the subtrees as formed, finishing with the root.

```
'[1:20] tree evergreen;
int treetop := 0; C this points to root of tree being created
C
proc trimstring = (ref string str) void:
      (str := (upb str = 1 | "" | str[1:upb str - 1]));
proc construct = (ref string s) ref tree:
      begin
            int n; tree resultree;
            if upb s = 0 then resultree := nil C empty tree C
            elif not matchar(s[upb s], "+-*/↑", n)
```

```
            then C single character tree C
                 resultree := (nil, s[upb s], nil);
                 trim string(s)
            else operator of resultree := s[upb s];
                 trimstring(s);
                 right of resultree := construct(s);
                 left of resultree := construct(s)
            fi; comment evergreen is necessary since result-
            tree is a local tree, and the ref tree result of the
            procedure would be a reference to a place
            which disappears as soon as the serial clause is
            left. We therefore copy the structure into the
            global 'evergreen' and yield a reference to it.
            See chapter 7 for alleviation of this difficulty
            comment
             evergreen[treetop plusab 1] := resultree
       end;
       string exp := "ab + cd+*"; construct(exp) C the value
       of the procedure is ignored here C'.
```

This is a rather difficult program, and in chapter 7 we introduce concepts to avoid using extra local variables.

It is easier to coalesce our two algorithms, and produce one that accepts the original expression in its standard form and transforms it into a tree representation. For this two stacks are needed – 'randstack' to contain the operands, which are references to subexpression trees (possibly **nil**), and 'opstack' to contain the operators, kept in the form of integers pointing to multiples as in the first program. Operators of greater priority than that on the top of the 'opstack' are pushed on to it as encountered. Those of lesser or equal priority cause a subtree to be formed from the top two elements of 'randstack' and the top operator on 'opstack', and a reference to this subtree pushed on to 'randstack'. Hence the objects on the latter stack are *pointers* of type **ref tree** and we discuss this topic later in the chapter. We again need a row of trees as a permanent store for the subtrees as formed.

Here is the program complete except for the 'matchar' procedure declaration, to be found earlier.

```
'begin
       mode tree = struct (ref tree left, char operator, ref tree
       right);
       string operators = "?£ ()+–*/↑";
       [] int priorities = (–1,0,0,1,2,6,6,7,7,8);
       [1:20] int opstack;
       [1:20] ref tree randstack; C pointers to the subtrees C
       [1:20] tree evergreen; C global row for subtrees C
       char ch; bool morechars := true;
```

```
int position, optop := 1, randtop := 0, treetop := 0;
opstack[1] := 1; C "?" put on opstack C
proc maketree = void: comment combines top 2 elements
of randstack with top of opstack, puts result on randstack
and stores it in evergreen comment
(evergreen[treetop plusab 1] := (randstack[randtop – 1],
      operators[opstack[optop]], randstack[randtop]);
      optop minusab 1;
      randstack[randtop minusab 1] :=
      evergreen[treetop] );
while morechars
do read(ch);
      if not matchar (ch, operators, position) then
      C operand found so stack it C
      randstack[randtop plusab 1] :=
      evergreen[treetop plusab 1] := (nil, ch, nil)
      else case position in
      C ? found C morechars := false,
      (C £ found so empty stacks C for i from optop by –1
      to 2 do maketree od; morechars := false),
      C space found C skip,
      C (found C opstack[optop plusab 1] := 4,
      (C) found C for i from optop by –1 to 2
      while opstack[i] ≠ 4 do maketree od; optop minusab 1
      C to delete matching opening bracket C)
      out C operator found C
      while priorities[position] ⩽ priorities[opstack
      [optop]]
            do maketree od;
      opstack[optop plusab 1] := position
      esac fi C get next character C
od C no more characters so do repetition terminates C
end'
```

At the end of the program, 'randstack[1]' refers to the complete tree of the expression, as does 'evergreen[treetop]', and all the subtrees are in 'evergreen[1]' upwards.

The program does not detect ill-formed expressions, and this makes it unacceptable as it stands, since minor punching errors could cause it to go into an infinite loop.

Exercise 5.8

Construct data that will make the above program fail.

The converse problem of converting a tree representation into its reverse Polish string equivalent can be solved by a short recursive algorithm.

1. If tree is a terminal node, print it and stop.
2. Apply algorithm to the left field.
3. Apply algorithm to the right field.
4. Print operator field.

Here is the program for this

```
'proc posttree = (tree expr) void:
   (int n; char ch = operator of expr;
   if not matchar(ch, "+-*/↑",n) then print(ch)
   else posttree(left of expr); posttree(right of expr);
      print(ch) fi)'
```

A modification of this algorithm prints a fully parenthesised form of the original expression

```
'proc printtree = (tree expr) void:
   (char ch = operator of expr; int n;
   print ("(");
   if not matchar (ch, "+-*/↑", n) then print(ch)
   else printtree(left of expr); print(ch);
      printtree(right of expr) fi;
   print(")") )'
```

If 'expra', 'exprb' are respectively the trees for 'a + b*c' and '(a + b)*c', then

posttree (expra)	produces	'abc*+'
printtree (expra)	produces	'((a) + ((b)*(c))'
posttree (exprb)	produces	'ab + c*'
printtree (exprb)	produces	'(((a) + (b))*(c))'

Exercise 5.9

Modify 'printtree' so as to omit parentheses around single characters.

We have met several occurrences of **ref**. To explain this concept in detail, it is necessary to examine critically our classification of objects into 'types'.

After the declarations '**int** b := 3, c; **int** d = 22', 'b,c,d' are all of type **int**, in that they are related to whole number values, but as explained in chapter 2, 'b' and 'c' are declared as *names* of places where an integer is stored, whereas 'd' is identified with the integer *value* 22.

Algol 68 defines 'b,c' as being of *mode* **ref int** since they refer to an integer, and 'd' as of mode **int**. The difference is important since 'd' can

never be altered in the program, but one can assign to variables like 'b' and 'c', changing the integer values to which they refer.

In the assignments 'b := 7; c := d – 3' the left-hand side is a reference and the right-hand side is a value, so that the action is clear. However, in 'c := b + 2', the 'b' on the right-hand side has to be *dereferenced* to its current value, namely 7, this added to 2, and 'c' made to refer to their sum. There has been a *coercion* on the right-hand side. Provided the left-hand side is of **ref** mode and the right-hand side can be coerced to a value of this mode, an assignment will take place. One common coercion called 'widening', has been mentioned – in '**real** x := 8' the integer on the right-hand side is widened to a real value.

It is unfortunate, but in line with other languages, that declarations like '**int** c; **real** x; **bool** flag', create respectively **ref int**, **ref real** and **ref bool** objects. It is doubly unfortunate that exceptions to this rule are the formal declarers in a formal parameter list. Thus in '**proc** f = (**int** m) . . .', the formal parameter 'm' is of mode **int** and one cannot assign to it. A call 'f(c)' is theoretically replaced by '(**int** m = c; . . .)', and we see that 'c' is dereferenced to its value, which must be integral, and 'm' identified with that value.

In contrast, after the declaration '**proc** g = (**ref int** q) . . .', a call 'g(b)' is formally replaced by '(**ref int** q = b; . . .)', and now 'q' is identified with the name 'b', and not its value. There can be assignments to 'q' within the procedure body.

It is frequently necessary, as was seen in the reverse Polish algorithms, to manipulate **ref** objects. The declarations '**int** m := 3; **ref int** p := m' define 'p' to be of mode **ref ref int**, that is, it refers to a place holding 'm', which itself refers to a place that holds the integer value 3, as visualised in figure 5.6.

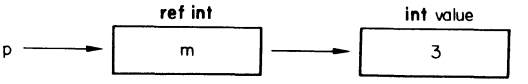

Figure 5.6

'p' may be considered as a *pointer* to a place itself referring to another place. Thus '[1:20] **ref tree** randstack' declared a row of pointers to objects of mode **ref tree**, and the algorithm operated on these pointers avoiding copying of tree structures. Objects of **ref ref . . . ref** mode may also be declared.

In algorithms using references it is often desirable to test for the equality of names as opposed to values. For example, the expression '(x + y) + b*(x + y)' may have its tree generated by

```
'tree x,y,b,factor,bprod,example;
x := (nil,"x",nil); y := (nil,"y",nil); factor := (x,"+",y);
```

```
b := (nil,"b",nil); bprod := (b,"*",factor);
example := (factor, "+", bprod)'
```

If subsequently we want to find the common factor in this expression, we could test the substructures for branches with the same fields, but it would be better to test for the equality of the two occurrences of the whole structure for '(x + y)'. In this situation we cannot use the equality operator, were it defined for trees, as it dereferences its operands.

Algol 68 permits the comparison of names of the same mode with the *identity relators* **is**' and **isnt**' (abbreviations :=: and :≠:). The use of these relators is shown in the following illuminating statements:

```
'char me := "I"; C mode ref char C
ref char myshadow = me; C 'myshadow' is also of mode ref char
                identified with 'me' C
ref char doppel, ganger; C of modes ref ref char C
char my copy; C of mode ref char C
if my shadow is me then print("SUN IS SHINING") fi;
comment both 'me' and 'my shadow' possess the same name because
of the identity declaration, so printing will occur comment
if my copy isnt me then print("NO BRAVE NEW WORLD YET")
fi;
comment 'mycopy' is a different name from 'me' and therefore isnt
relation yields true and printing occurs comment
doppel := ganger := me;
if doppel is me then print("YES") fi; C yes printed since 'doppel' is
dereferenced to yield the name 'me' C
if me is ganger then print("YES AGAIN") fi; C yes again printed C
if doppel is ganger then print("IMPOSSIBLE") fi; C no printing C'
```

The last comparison yields **false** since objects of mode **ref ref char** are being compared and one is 'doppel' and the other 'ganger' which is different. If necessary, dereferencing is applied to one side of the identity relation, but not to both.

We can forcibly dereference names by using a *cast* which coerces to the mode specified. A simple example of such coercion is given by mixing **real** and **int** objects, noting that the BNF of cast is

⟨ cast ⟩ ::= ⟨ formal type ⟩(⟨ unit ⟩).

```
'op newop = (real a, real b) real: (a*b + a/b);
real x := 5.6; int a := 3;
comment 'x newop a' is illegal since the right-hand side must be a
real and widening does not apply to parameters of an op routine
comment
x := x newop (real(a)); C real(a) is a cast forcing a to real C
x := x + a; C is allowed as + is defined between real and int (and
between many other modes) C
a := round x; a := entier x; C both work since int yielded C'
```

We can use the cast for dereferencing in

'**if** (**ref** **char**(doppel)) is (**ref** **char**(ganger)) **then** print("OK NOW") **fi**'

when there will be printing because of the coercions. However, dereferencing can be automatically applied to one side, so that only one cast is required.

'**if** (**ref** **char**(doppel)) **is** ganger **then** print("OK TOO") **fi**'

will print ok too since 'ganger' is dereferenced to mode **ref char**.

In the expression 'example' we can determine if there is a common factor by

```
if (ref tree(left of example)) is factor then
   print("COMMON SUBEXPRESSION") fi'
```

This will print the message because the two names compared are the same.

A similar problem occurs when searching for a terminal node of a tree, or even testing for an empty tree, which is indicated by the presence of **nil** in the left or right fields (we have previously tested for an operand in the operator field with 'matchar', but this is rather clumsy).

nil is a reference that can be coerced to any **ref** mode by a cast. When **nil** is used in structures, and elsewhere, it is automatically coerced to the correct mode.

To find the leftmost node in 'expression tree' we may write

```
'ref tree leftmost := expression tree;
while (left of leftmost) isnt (ref tree(nil)) do
   C the cast could equally well be applied to the left side C
   leftmost := left of leftmost od C a reference to a subtree C'
```

In this particular do clause 'leftmost' changes to point to successive left branches. We could still keep a pointer to the original value of 'leftmost' by making an initial assignment to a **reftree** 'original'. Pointers enable a place in a structure to be accessed without changing the structure.

Exercise 5.10

Rewrite the procedures 'posttree' and 'printtree' to test for **nil** in the left field of terminal nodes.

Each of the terminal nodes of a tree occupies unnecessary space with its dummy left and right branches. We show how to avoid this in chapter 7.

Problems for Chapter 5

1. Write a recursive procedure to check whether two trees represent the same expression.

2. Rewrite 'posttree' as an iterative procedure. (A stack is necessary.)

3. Define an unambiguous prefix form for expressions, and construct a program to convert standard arithmetic expressions into this form.

4. Add checks to the program for converting string expressions into tree form, that reject illegal expressions containing two successive operators, like 'a*/b', or unmatched parentheses like '((a + b)'

5. Write a recursive procedure to evaluate the tree form of an expression given that its operands are single digits. (**abs** applied to characters gives an integer code value. The digit characters "0", "1", . . "9" have adjacent codes, so that d = **abs** "d" – **abs** "0" is true for any digit.)

6. Which programs in this chapter still work if +,– can be unary as well as binary operators? Modify those that do not work. (Consider '+a*((–3) + a)' as an example.)

6 Sorting a File: Transput

Problems of transput are considered in this chapter. Earlier programs used 'read' and 'print' to get information into and out of the computer. These are standard prelude functions that accept a single parameter of almost any type including a bracketed row of objects. The data is regarded as a stream of characters representing the values of variables, whether **int**, **real**, **char**, **bool** or any other mode, inside the program. Names cannot be transput, only values.

An implementation may allow useful control of the appearance of data as it is read or printed, for example the number of places occupied by a printed number or the presence or absence of signs. However, certain general rules apply to transput, and we illustrate these by attempting a data-processing problem.

The interest rate on house mortgages has been increased, and a building society has to send notices to all its clients, each notice to be of the following form, so that it can be easily folded for direct mailing in a window envelope:

MR. Y. ZACHARIAS
234 HIGH ST.
DULLSVILLE
NE2 4PQ

– TO PAY OFF LOAN AT PRESENT REPAYMENT OF £35.99 TAKES 24 YEARS (A)
– AT NEW INTEREST RATE THIS PERIOD BECOMES 28 YEARS
– INCREASE MONTHLY REPAYMENT TO £37.37 TO PAY OFF IN PERIOD (A) ABOVE
REFERENCE NUMBER 5987

(Actual mortgage companies are more polite, but the ideas are the same.)

To construct these notices the society will have to scan its files. It keeps the following information about each of its mortgagees – name, address, code number, original loan, original repayment period, date mortgage commenced, balance still owing, remaining repayment period, interest paid this financial year, current monthly repayment.

This information consists of strings of characters, and is coded and stored on a physical device such as a magnetic tape or a deck of punched cards. It is divided into *lines*, each line occupying a certain length of tape or a single card. We allocate one line for the name, three

for the address, and one line for each of the other items. This makes up a *record* of 12 lines for each client.

The whole file consists of thousands of such records, separated by one or more blank lines, and terminated by a marker line, say a row of Z's.

The messages are to be printed on a lineprinter, with the name and address in the middle of the page, followed by four blank lines, and then the other statements each on a single line and separated by blank lines. Each notice occupies one page, which is rather wasteful since a typical lineprinter has 120 characters to a line and 60 lines to the page.

The algorithm is simple.

1. Find the next non-blank line. If it starts with "ZZ" then stop. (Presumably nobody's name starts with ZZ.)
2. Read the name and address. Take a new page and print them in the required format.
3. Read the other data from the record.
4. Calculate the new mortgage period at the old monthly rate, and the new monthly payment needed to keep the same mortgage period.
5. Print the notice and repeat from step 1.

To avoid irrelevant calculations in step 4, we assume that we are given functions 'newtime', 'newpay' such that if b is the balance owing in pounds, r the remaining repayment period in years, and p the current monthly payment in pounds, then 'newtime (b, r, p)' is the new period in years needed to pay off the loan at the old rate of payment, and 'newpay (b, r, p)' is the new monthly-payment in pounds to pay off over the old period. To simplify matters further, we assume that repayments are between £1 and £100 a month.

We program the algorithm into a procedure to read records, thus making it portable into other programs.

```
'begin
        proc read record = (proc void finished) void:
        comment if there are no more records to read, the
        procedure 'finished' is called comment
        begin
          char a := " "
          while a = " " do read(a) od
          comment this skips blank spaces. The 'read' routine will
          take the next character in the input as the value of a,
          automatically going to a new line if necessary.
          comment
          if a = "Z" then if char b; read(b); b = "Z" then finished
          else read(backspace) fi fi;
          read(backspace);
```

C to make the procedure self-contained, we use a **proc** parameter 'finished' instead of **goto** a label. 'backspace' moves the input pointer back one space, but not beyond the start of a line. It is now at the first non-blank character on the record **C**
[ ] **char** spacer = " ",
C 20 blank spaces since the name and address are to be printed in the 21st column onwards **C**
print(newpage);
to 10 **do** print(newline) **od**; **C** so the output is not at the very top of the page **C**
string s;
to 4 **do** print((spacer, (read((s, newline)); s), newline)) **od**;
comment we read in the 4 lines occupied by the name and address, each as a string, and print them out.
When reading or printing strings the action is terminated on reaching the end of a line. This is necessary since otherwise all input characters would be read into a monster string. However, multiples with fixed bounds are read beyond the end of a line if the item has not yet been found. All multiples are *straightened* as they are transput – that is, elements taken in the lexicographic order of indices (see chapter 3 for an example). Structures are also straightened field by field.

A parameter of 'print' or 'read' can be any expression, including a closed clause as in the program. Such clauses are executed before transput, so that for instance '**int** m; print(((read(m); m), (read(m); m)))' would not print the next two input integers, but print the second one twice
comment
to 5 **do** print(newline) **od**;
comment 4 blank lines have been printed. The other lines of data are now to be read, skipping those not wanted. When reading into a **real**, **int** or **bool** variable, the input stream is scanned for the first non-blank character, taking newlines if necessary. This has to be of the expected type – for example, the + or – sign preceding a number, or the first digit of that number, or T standing for **true**. Once reading of the value has commenced it is terminated by an unexpected character, or a blank, or the end of line. Hence '[1 : 3] **int** d; [1 : 3] **char** c;
for i **to** 3 **do** read((d[i], c[i]))**od**' correctly reads "4B7W2F". Similarly on output using 'print', if there is insufficient room to accommodate the next item, then a new line, and if necessary a new page, is taken. However, output of a flexible multiple of **char** is stopped by the end of the line. **comment**

```
int code, period;
real balance, monthpay;
read((code, newline, newline, newline, newline, balance,
newline, period, newline, newline, monthpay));
int newperiod = newtime(balance, period, monthpay);
real newmonthpay = newpay(balance, period, monthpay);
comment These functions were defined above. Suitable
'proc(real,int,real)int newtime' and 'proc(real,int,real) real
newpay' must be declared in the program. We are now in
difficulties with money! 'print(monthpay)' would print this
real number in floating-point form complete with a large
number of decimal places and a sign. A figure like 35.99,
accurate to the nearest penny and not preceded by a space
or sign, is what is wanted. In our next program we show
how to force the output into this format. Meanwhile, we
must convert the number into a string of characters.
comment
proc convert = (real money) string:
C converts money, £1 < money < £100 into a string of
type dd.pp, suppressing any leading zero.
There is a standard prelude procedure called 'fixed' that
outputs decimals as strings, but we prefer to construct our
own to show what is involved. C
begin int z := 1, C to start loop C y := round(100*money);
[1:4] char result := "    "; C 4 spaces C
for i from 4 by -1 to 1 while z ≠ 0 do
z := y ÷ 10; result[i] := repr(y - z*10); y := z od;
(result[1:2] + ".") + result[3:4] end;
comment of convert routine. + is here string concatenation.
'repr m' is the character whose internal code is the integer
m. repr is the inverse function to abs on characters (see
problems to chapter 5), so repr abs "Y" = "Y" and abs repr
m = m. The codes for characters are unspecified in Algol
68, but one can discover them by test. For simplicity we
assume that the characters "0" to "9" have codes 0 to 9 (as
they do in at least one implementation). comment
string oldpay = convert(monthpay), newpay = convert(new
monthpay);
C we collect the printing details into a procedure C
proc print account = (string oldp,newp, int
oldy,newy,code) void:
(print(("- TO PAY OFF LOAN AT PRESENT REPAY
MENTS OF £", oldp, "TAKES", oldy, "YEARS (A)",
newline, newline,
"- AT NEW INTEREST RATE THIS PERIOD
BECOMES", newy, "YEARS, newline, newline,
```

```
            "– INCREASE MONTHLY REPAYMENTS TO £",
            newp, "TO PAY OFF IN PERIOD (A) ABOVE", newline,
            newline, "REFERENCE NUMBER      ", code))  );
            C The integers will be printed with unnecessary spaces and
            perhaps a plus sign as well. We could convert these also to
            character strings, but prefer to suppose that some imple-
            mentation-dependent action has been taken to suppress
            signs and spaces C
            print account (oldpay, newpay, period, newperiod, code)
        end C of main procedure C;
        do read record(void:(goto end prog)) od;
        endprog: skip
    end'
```

Exercise 6.1

Rewrite 'convert' to deal with any amount of money, using a procedure to print positive integers without spaces, sign or leading zeros.

In the above program we had to construct spacing constants, define a conversion to character routine, and a 'print account' procedure that is really a mould into which we put variable parameters to be printed.

There are standard output procedures 'whole', 'fixed' and 'float' that would help in the previous program, but for full control of character transput Algol 68, and many other computing languages, allow a *formatting* technique.

We illustrate this by solving the same problem as above, but with more realistic data. Each line of a client's record contains his code number in the first four positions and the other information of the line starts at column 12. For the first four lines this is the name and address, each consisting of at most 40 characters, and for the other lines it is a message clearly stating the contents.

For example, instead of a bald figure 5000 somewhere on the 6th line, it now reads

```
5987    ORIGINAL LOAN = £5000
```

where 5987 is the record reference number.

To input data like this that is in a known format, the 'read' function is replaced by 'readf' (read in format). For instance the above line could be read by

```
'readf(($ 4d $, code, $ 7x "ORIGINAL LOAN = £" 4d $, amount))'
```

'readf' accepts a row as parameter, each item being a variable or row of variables or a *format denotation.* The latter are surrounded by dollar

signs, and specify the expected input-form of the values of the variables that follow it.

The particular format denotations above each consist of a single *picture*. The first is '4d' meaning that four digits are expected. The second is made up of '7x', meaning seven spaces, and a literal string in quotes that must match the input exactly, and '4d' again.

It may be easier to understand format denotations if we examine part of the BNF definitions.

⟨format den⟩ ::= ⟨picture⟩⟨,⟨picture⟩⟩ *

Thus a format denotation is a list of one or more pictures separated by commas. Each picture corresponds to an item to be transput listed in the parameter that follows.

⟨picture⟩ ::= ⟨⟨insertion⟩[1] ⟨pattern⟩⟨insertion⟩[1] ⟩*
⟨replicator⟩ (⟨picture⟩)

A picture is thus a string of one or more *patterns* possibly with insertions among them, and it may be replicated. A *replicator* specifies the number of repeats and its absence means once only. It is usually an integer.

⟨insertion⟩ ::= ⟨literal⟩ | ⟨alignment⟩
⟨literal⟩ ::= ⟨replicator⟩[1] ⟨string den⟩
⟨alignment⟩ ::= ⟨replicator⟩[1] ⟨tabulator⟩

The literal insertions are string denotations, perhaps replicated. "ORIGINAL LOAN = £" in the example above is a literal insertion. The other insertions are tabulation symbols, x, y, l, p, k, meaning respectively space, backspace, newline, newpage, go to position given by replicator value. '7x' is a replicated space symbol, but we could have used '12k' instead to get to the 12th position in the line after reading the reference number.

⟨pattern⟩ ::= ⟨replicator⟩[1] ⟨suppress⟩[1] ⟨frame⟩

The pattern for transput of a value is defined by a *frame*, for example 'd' is the digit frame, and the frame may be replicated and/or suppressed. The suppress symbol 's' is used when a frame is to be ignored, that is not output, or read but not input.

There are many different frames, including the digit frame 'd', character frame 'a', point frame '.'. Rather than list them here, we use them in programs and explain their functions by comments.

For output there is a similar function to 'readf' called 'printf', for which the format denotations determine the style of printing of values.

Exercise 6.2

Which of the following are in the class ⟨format den⟩ according to the

above BNF? (a) d; (b) yd3, d; (c) d "3"; (d) xyd, 2lsd, "THE END"; (e) 6d "THE""END" x4d.

The program that follows prints out the notices from the data specified, and is quite similar to our first program, except that formats make it easier.

```
'begin
        proc read record = void: C we use labels to jump out of this
        procedure C
        begin char a;
        while read(a); a = " " do read(newline) od;
        if a = "Z" then goto out fi;
        C all record cards have a digit in the first column C
        [1:40] char name, add1, add2, add3;
        C name and address strings are at most 40 characters C
        readf(($ 12k 40a 1 $, name,add1,add2,add3)),
        comment 'a' is the char frame. The format expected is 40
        characters starting at column 12, and taking a newline after
        reading. There are four variables following the single
        format denotation. In such cases, the format is in operation
        until a new one appears as a parameter comment
        int code, period;
        real balance, monthpay;
        readf(($ 12k"CODE NO =" x4d $, code, $ 4l 12k
        "BALANCE OWING = £" 4d . 2d $, balance, $ 1 12k
        "REMAINING PERIOD =" x 2d 2l $, period, $ 12k
        "MONTHLY PAYMENT = £" 2d.2d $, monthpay));
        comment The input data must exactly match the format
        and this acts as a check on its accuracy. For example in the
        12th to 25th position in the first line we expect "CODE
        NO = 5987" (or some other 4 digit number). Similarly the
        balance owing has to be in the form 4 digits, decimal point,
        2 digits like 0923.77. Spaces in the format outside of a
        string denotation are ignored. comment
        printf(($ p 4(21k 40a l) $, (name, add1, add2, add3) ));
        comment Here we do not want a new page each time, so we
        have replicated a picture by placing it in parentheses after
        the replicator 4. The name and address appear on a new
        page in 4 lines each starting at the 21st column. comment
        printf(($ 4l "– TO PAY OFF LOAN AT PRESENT
        REPAYMENTS OF £" zd. 2d $, monthpay, $ "TAKES" zd
        "YEARS (A)" $, period, $ 21 "– AT NEW INTEREST
        RATE THIS PERIOD BECOMES" zd "YEARS" $,
        newtime(balance, period, monthpay),
        $ 21 "– INCREASE MONTHLY REPAYMENT TO £"
        zd . 2d
```

```
            "TO PAY OFF IN PERIOD (A) ABOVE" £, newpay
            (balance,period,monthpay), $ 21 "REFERENCE
            NUMBER" 5x 4d $, code))
            comment 'z' is the zero frame used for printing digits with a
            blank space instead of a zero. Thus, 'zd' will print a
            two-digit integer and suppress any leading zero. The real
            value of the new monthly repayment is automatically
            rounded to 2 decimal places by the picture 'zd . 2d' We
            could print money in the form "£35–99" instead of
            "£35.99" by using the picture ' "£"2ds."–"2d'. The point
            frame is suppressed by the 's' and "–" printed instead. The
            same format would accept "£35–99" as an input value of
            35.99. comment
            end C of read record procedure C;
    do read record od;
    out: skip
    end'
```

Exercise 6.3

Modify the program to check that the first four digits on each line of the record are the reference number of that record. If not, print out the aberrant line and the correct code, and skip the rest of this record before re-entering 'read record'.

A few more remarks on the various frames should be made. The character frame 'a' must not be mixed with other frames in the same picture. A real value must not be transput using only digit frames, either a decimal point frame or an exponent frame 'e', or both, must be present. It is possible to revert to formatless transput by using the general frame 'g', when the conventions of 'read' and 'print' take over.

Apart from a positive integer, a replicator can also be of the form 'n(⟨ser cl⟩)'. The serial clause is executed when the transput function is called, with result a positive integer specifying the number of replications. We can thus define a dynamic number of replications depending on the data. For example, to print both "£33.26" and "£7.64" with no spaces between the pound sign and the 7, use

```
printf(($ "£" n(if money < 10 then 1 else 2 fi) d.2d $, money))'
```

Finally, we mention the use of *choice functions* in the format. Suppose the eight lines of the record following the name and address are shuffled. Each line contains enough information to identify it, whatever order they are, and they can be read by

```
'bool items left := true;
```

```
int n, period, code;
real balance, monthpay; while items left
do readf (($ 12k c("CODE NO = ", "ORIGINAL LOAN",
"ORIGINAL PERIOD", "MORTGAGE COMMENCED",
"BALANCE OWING = £",
"REMAINING PERIOD = ", "INTEREST", "MONTHLY PAY
MENT = £",
" ") $, n));
comment 'c' is the choice function. Starting at the 12th place, the
current line is scanned for the first string in the choice function
parameters that matches. If no match is found, then the last choice
" ", being the empty string, is taken. n then is given the value of the
integer corresponding to the position of the matching string in the
list. comment
case n in read(code), skip, skip, skip, read(balance),
read(period), skip, read(monthpay), itemsleft := false esac;
read(newline)
od'                    and the program continues.
```

For the lines to be skipped, enough string for identification purposes is matched. For the others, we match right up to the values to be read. Of course, suitable 'readf' instead of 'read' could have been used for their input.

The choice function can also be used on output, when it prints the string in the position given by the value of the next parameter. For instance

```
'printf (($ "M IS" c("ODD", "EVEN") $, if odd m then 1 else 2 fi))'
```

There is also a boolean choice function, 'b'. This has two string parameters, and its pattern must be for a boolean variable. The values **true** or **false** of this variable choose the first or second string respectively. The above example is then simpler,

```
'printf(($ "M IS" b("ODD", "EVEN") $, odd m))'
```

These choice functions allow words in input data to control the flow of a program.

Exercise 6.4

Design a program to accept input that is a mixture of items of the form "I = ⟨integer⟩", "J = ⟨integer⟩", "RUN", "FINISH" in any order, and to print the sum of the squares of the i-values and of the j-values whenever "RUN" is met, and to stop when "FINISH" is read.

The use of formats can improve the output of earlier programs. The data for the series-summation problem was a sequence of values for the

variable, terminated by 0. The following clause prints the values of the variable and the sum of series in suitably headed columns. We assume '**proc (real) real** sum' has been declared.

```
'printf (($ p 24k a, 19x 3a l $, ("Y", "SUM")));
real y := 999;
while abs y > 1 & -11
do read(y);
printf(($ l 17k 4z + d . 4d $, y, $ 8x + d . 9d e + 2d $, sum(y) )) od'
```

The value of y is printed as a decimal with up to five figures before the decimal point, leading zeros suppressed except for the last digit, and four figures after the point. We meet here the sign frames '+' and '–'. The former is used when a plus sign is to precede positive numbers, and the latter when the plus sign is replaced by a blank space. In both cases negative numbers have a minus sign. Placing the sign frame after the zero frame, as in '4z + d . dd', moves the sign next to the first unsuppressed non-zero digit, so that the value 76.2 is output as " +76.20". If the pattern were '+4zd . dd', it would appear as "+ 76.20".

The sum in the above call of printf is printed as a signed decimal fraction with one place before the point and nine after, followed by an exponent sign (often & on printed output), 'e' being the exponent frame, followed by a signed two-digit exponent.

Exercise 6.5

Assume that '**proc (int)real** g' has been declared. Print a readable table of the values of g(1) to g(1000) to six significant figures, five values to a line, 40 lines to a page with extra space between every 10 lines. Pages should be numbered and lines start with the first argument integer of the line.

The same format may be used several times in a program, and to avoid re-writing it in full we can declare it separately using a new basic type.

'**format** f' declares 'f' to be a *format-type* variable, that is, of mode **ref format**. Like any other type of variable it can be initialised, assigned, act as the parameter of a procedure or field of a structure. Its value is a format denotation, but there are no operators on formats and they cannot be transput.

We use formats in the following procedure to print a calendar with each successive 3 months arranged side by side on a page, and the full 12 months forming 4 rows of 3 months each. It goes on to a page of 60 lines each of 120 characters in length.

The procedures 'firstday' and 'no days in' are from chapter 4. These were defined so that, for example, 'firstday(1985)' is the integer 2, this

being the coding of Tuesday (Sunday to Saturday are coded 0 to 6) and 1 Jan. 1985 is a Tuesday. 'no days in(2, 1985)' is 28, the number of days in Feb.1985.

We also assume that a fixed multiple '[1:12, 1:9] **char** monthhead' has been defined with 'monthhead[1,]' being "JANUARY" etc., all months filled out with spaces if necessary to occupy 9 characters.

```
'proc calprint = (int year) void:
  begin
  format f1 = $ 2l 19k 9a, 2(28x 9a) 2l $,
         f2 = $ 10x 6 (3ax), 3a $,
         f3 = $ 10x 6(x 2z x), x 2z $;
       C These are the formats to print month headings, day
       headings and dates respectively. Spacing has been carefully
       chosen C
       printf(($ p 56k dxdxdxd 4l $, year));
       C This prints eg. "1 9 8 5" in the middle of the page C
       [1:12, 0:6] int dates;
       comment 'dates[i, ]' will be the dates of the current
       calendar row of the ith month. We print across pages and so
       must remember the position left in previous months. If j is
       the weekday code of the first of the month, the first
       calendar row starts with the integer 1 – j, provided we
       suppress the printing of dates < 1 or > (number of days in
       month) by making them zero and using the '2z' pattern in
       format f3. comment
       [1:12] int first;
       for i to 12 do
       first[i] := firstday(year);
       for j to i – 1 do first[i] plusab nodays in(j, year) od;
       dates[i, 6] := –(first[i] mod 7) od;
       comment 'first[i]' taken mod 7 is now the day of the first
       of the month numbered i. We initialise 'dates' in the above
       way to fit in with the do clauses that follow. comment
       for i by 3 to 10 C months 1, 4, 7, 10 lead the rows of
       months C
       do printf(( f1, monthhead[i:i + 2,] ));
       to 3 do printf(( f2, ("SUN", "MON", "TUE", "WED",
       "THU", "FRI", "SAT"))) od; C month and day headings
       done C
       to 6
       C each month has up to 6 rows of dates C
       do print(newline);
       for j from 0 to 2 C 3 months across page C
       do dates[i + j, 0] := 1 + dates[i + j, 6];
          for k to 6 do dates [i + j, k] := 1 + dates[i + j, k – 1] od;
          C 'dates' is now correct for one week of month i + j. We
```

```
            put it in a new multiple, making unwanted dates zero C
            [0:6] int m;
            int nij = nodays in(i + j, year);
            for k from 0 to 6 do
            int dijk = dates[i + j, k];
            m[k] := if dijk <0 or dijk > nij then 0 else dijk fi od;
            printf ((f3, m)) C prints one line of one month C
    od C end of 1 line for each of 3 months across page C
    od C end of 6 lines completing the 3 months C
    od C end of year C
end C of procedure C'
```

Exercise 6.6

Months often need only 5 rows of dates. Modify the main loop of the program so that in these cases the last row is printed as blanks without checking that each date is > number of days in month.

If all output in a section of the program is to be in the same format, say 'form', we can call simply 'printf(form)'. After this 'printf((x, y))', for instance, will output x and y in the format 'form'. If the format is not completely exhausted in printing x and y, perhaps because there are more pictures or replications to come, then the remainder of the format is still in operation when 'printf' is called again. Similar remarks apply to 'readf'.

Exercise 6.7

Write a procedure to output any multiple of integers, all of absolute value $<10^5$, in the form of five-digit signed integers, each separated by a space or newline. The line length is 120 characters and each digit occupies 7 spaces (including its trailing separator), so that there are 17 numbers plus an extra space on each line, except possibly for the last line.

So far, all transput has been on standard *channels*, which are assumed to be a cardreader and a lineprinter. The data is held in standard *files* called 'standin' when it is input, and 'standout' when it is to be output. These standard files are available to all programs, and the corresponding channels are accessible by using 'read' or 'readf' on standin, and 'print' or 'printf' on standout.

One can open other files for transput, and they may be associated with other channels that allow communication with different peripherals, or with discs or tapes.

We demonstrate this by considering an elementary example of file handling.

Two branches of our mortgage company decide to merge. They each have a file of customer records in code-number order, and two files have to be combined into a single file sequenced by the code numbers of the records. For large files, sophisticated methods should be used, but the following rough and ready procedure works.

1. Two input files, called 'in1' and 'in2', are on pre-assigned channels. Open them ready for use, and open an output file called 'out1' on a suitable channel.
2. Read in the integer codes 'code1' and 'code2' from the first record line of the respective files.
3. If code1 < code2, transfer a record from in1 to out1. Read the code number of the next record on in1 into code1, unless the end of the file has been reached, when 10000 (> any code number) is made the value of code1.
4. Dually to step 3 if code2 < code1. (If they are equal, see exercise 6.8)
5. If code1 < 10000 or code2 < 10000 repeat from step 3.
6. Compare the number of records transferred with the final size of the output file. This is a check on correct execution.

Step 1 has to be performed partly by the computer *software* outside of Algol 68. Input channels numbered 1 and 2, say, are associated with magnetic-tape readers and the appropriate decks loaded. An output channel numbered 1 is opened that can be written to. We assume this has been done by special control instructions.

Within the program we declare *file variables* that refer to rather complicated structures defined in the standard prelude. To open these files there is a standard procedure 'open' with 3 parameters which associates the file, whose name is the first parameter, with a channel whose number is the third parameter. (The second parameter is a book name, but we can ignore books by using an empty string.)

All files are divided into characters, lines and pages, and on opening are at position (1,1,1), meaning the character position number, line number and page number. One can find the current position by three **proc(ref file)int** procedures called 'char number', 'line number', 'page number'.

'get' and 'put' are used for transput on a file, and are similar to 'read' and 'print' used with the standard files. They have two parameters, the file name and the variables to be input or values to be output (rowed if more than one). Thus 'read((b, newline))' is equivalent to 'get (standin, (b, newline))'.

It is worth remarking that 'newline', 'newpage', 'space', and 'backspace' are all procedures with one **ref file** parameter. This parameter can be omitted inside 'get', 'put', 'read' and 'print' since it is automatically supplied. For example, 'put(out, newline)', where 'out' is the name of a file opened on an output channel, has the same effect as

'newline(out)'. Hence 'print(space)', 'put(standout, space)' and 'space(standout)' all print a space on the standard-output file, the last being the most direct.

Here is the program to merge the two branch files.

```
'begin
        file in1, in2, out1;
        open(in1, " ", 1);
        open(in2, " ", 2);
        open(out1, " ", 1);
        comment the files are now linked with the channels,
        numbered 1,2,1, which were set up for input, input and
        output respectively by the operating system outside of the
        program. The implementation may allow enquiries about a
        channel, for example how many lines per page are possible,
        but we assume that our channels have properties consistent
        with the data comment
        proc skipblanks = (ref file folio) void:
          (char a;
          while get(folio, a); a = " " do newline(folio) od;
          backspace(folio)  );
        comment this procedure leaves the input pointer at the
        beginning of the first line with a non-blank first character
        comment
        proc end of file = (ref file folio) bool:
          (char a;
          get(folio, (a,backspace));
          a = "Z");
        comment the end marker on the files is a row of Zs as
        before. There is a procedure called 'logical file ended' in the
        prelude but its use is beyond the scope of this book
        comment
        proc transfer record = (ref file from, to) void:
                                (string s; to 12 do get(from, (s,newline));
                                   put(to, (newline,s)) od);
        comment We transfer each record of 12 lines as 12 strings.
        One could use 'getf(from, –)' and 'putf(to, –)' and perhaps
        change the format as a record is transferred. 'getf' and
        'putf' are procedures for formatted transput on files, and
        bear the same relation to 'get' and 'put' as 'readf' and
        'printf' bear to 'read' and 'print'. comment
        int max = 10000;
        int code1, code2, recordcount := 0;
        proc move = (ref file from, to, ref int code) void:
```

```
C this moves one record and sets code, testing for end of
file C
begin transfer record(from, to); recordcount plusab 1;
   newline (to) C to space output C;
   skipblanks(from);
   if end of file(from) then code := max
                        else getf (from, code)
         C when this procedure is called, a format will have
         been associated with the 'from' parameter, for
         reading 'code' C fi
end; C now comes the program proper C
skipblanks(in1); skipblanks(in2);
format w = $ 4d 4y $; C to read the 4-digit code from head
of line, and return to start of line. Rather artificial. C
getf(in1, (w, code1)); getf(in2, (w, code2));
comment Associates the format w with two different files.
If the implementation does not allow this, make a copy of
w. comment
while code 1 < max or code2 < max do
if code 1 < code2 then move(in1, out1, code1)
else move(in2, out1, code2) fi   od;
C both files now completely merged C
int m = line number(out1);
comment 'line number' is a standard procedure supplying
the current line number of the file. m has value the number
of lines transferred plus 1, since it points to the number of
the next line in 'out1'. We expect 12 lines plus 1 blank line
per record, so as a check comment
if record count ≠ m ÷ 13 then
print("COUNT OF RECORDS TRANSFERRED
DISAGREES WITH OUTPUT FILE SIZE") fi;
C 'print' is on the file standout, presumably a lineprinter C
close(in1); close(in2); close(out1);
C The program ending will close these files anyway, but it is
worth doing in case we insert more program and expect to
read from position (1,1,1) again. C
print((newline, "JOB FINISHED"))
C and lots more diagnostics in practice, since transferring
these files does not generate hard copy (printout etc.) to
check correct execution. C
end'
```

Exercise 6.8

The above program assumes that neither file is empty, and that they

have no common record. Modify to overcome these restrictions, and if a common record is detected then transfer it once only to 'outl' and print it on 'standout' with a message.

It may be useful to summarise some of the Algol 68 procedures introduced in this chapter. We do this informally with the argument type set in double quotes.

print("variables" with no formatting on the file standout);
read(into "variables" with no formatting from the file standin);
printf(on standout in "format", output "variables");
readf(on standin in "format", input into "variables");
put(on an output "file", "variables" with no formatting);
get(on an input "file", values into "variables" with no formatting);
putf(on an output "file", in "format", output "variables");
getf(on an input "file", in "format", read into "variables");
open(a "file", identifying a "book", on a "channel");
close(a "file").

We conclude this chapter with a warning. Transput is one area where language designers and language implementers often differ. The conventions of a particular operating system may influence the implementation. The programmer should check, particularly when using files, which of the standard functions are available.

However, since 'real-life' programs handle large amounts of data, there is more elaborate transput available than has been mentioned in this chapter. We have regarded our input and output essentially as strings of characters. There is another form of transput called *binary* that ignores the internal structure of the data, and converts it in some undefined way into binary code. Data output by the binary output function 'put bin' to some backing store can be input later by 'get bin'. This leads to more efficient usage of the filestore.

Problems for Chapter 6

1. Write an output procedure to print the solution of the maze problem of chapter 3. The result should be a picture with the entrance, exit, blocked squares, dead-ends tried and the successful path all clearly marked.

2. Calendars in pocket diaries usually have the days of the week printed vertically alongside the dates with 3 months across the page. Write a program to print such a calendar, and as a refinement, fit every month into 5 columns by transferring excess dates to the beginning of

the month. Thus

	JAN	FEB	MAR
Sunday	31 3 10 17 24	7 14 21 28	7 14 21 28
Monday	4 11 18 25	1 8 15 22	1 8 15 22 29
Tuesday	5 12 19 26	2 9 16 23	2 9 16 23 30
Wednesday	6 13 20 27	3 10 17 24	3 10 17 24 31
Thursday	7 14 21 28	4 11 18 25	4 11 18 25
Friday	1 8 15 22 29	5 12 19 26	5 12 19 26
Saturday	2 9 16 23 30	6 13 20 27	6 13 20 27 etc.

3. Vital information on a group of students is punched on cards in the following format

columns	1–15	last name
	16–20	initials
	21–25	first year of course eg.1979
	30–37	letter grades of up to 8 half-course units taken in first year, for example, AFDBCBCE
	40–47	grades of second year
	50–57	grades of third year

Possible grades are the letters A to F where F represents failure. Write a program whose input is a file of such cards terminated by a blank card. The output is a headed table, with one line for each student, that displays the number of half-course units passed, the number of As and the letter-grade average (rounded down) of his best 9 half-course units.

4. Plot the function cos(t), using lineprinter Xs to draw the curve with the t-axis running vertically down the middle of the page.

5. Write a program to print a difference table for f(x) with x taking values from a to b at intervals of (b – a)/n. The table should have columns

x	f(x)	df	d^2f	d^3f	...	$d^{10}f$
x_0	f_0					
x_1	f_1	g_1	f_1^2			
x_2	f_2	g_2	f_2^2	.	...	
.....						

where $f_i = f(x_i)$, $g_i = f_i - f_{i-1}$, $f_i^2 = g_{i+1} - g_i$ etc. x, f(x) with the differences d^2f, d^4f, . . , and the odd differences df, d^3f, . . should be printed on alternate lines, and the whole table have the triangular shape indicated. Test with functions sin(x) and cos(x) with a = 0, b = pi/2, n = 36.

7 Problems Re-visited: Advanced Features

In the final chapter, we reconsider some of the algorithms used in earlier chapters and improve or vary the solution with the introduction of further language features.

In several of the programs it was necessary to specify in the declarations the maximum amount of storage required for the various data structures. This limit was usually in excess of the actual amount used by our simple test data and could have been reduced by the use of flexible multiples in some cases. However the flexible bounds may still need to be stretched to cater for larger structures and would also prove difficult for accommodating more complex objects, such as a heavily recursive tree structure.

In the maze-traversal program we represented the maze as a square array, and if the actual data for a maze did not fit this shape then it would have to be padded out.

A solution to both these kinds of space-utilisation problems is to generate space as and when it is required, and Algol 68 provides this feature. Two kinds of space can be generated, *local* and *heap* (or *global*).

A *local generator* is introduced by a declaration preceded by **loc** that both reserves storage and yields the name of that space. For example

```
'ref char ch; comment 'ch' is of mode ref ref char comment
ch := loc char := "N" '
```

The generator **'loc char'** provides a storage place for one **char** value and yields the name referring to that place. The character "N" is then assigned to this place, and finally the name assigned to 'ch'.

The scope of a locally generated name is the smallest including range that is either a routine or contains a new declaration, and on leaving that range the space is released. Most of the declarations we have used have been local generators since **'real** r' is strictly an abbreviation of **'ref real** r = **loc real'**. This identity declaration makes the name 'r' refer to a space on a local stack that can hold different real values. Contrast this with an earlier identity declaration **'real** pi = 3.14' that makes 'pi' refer to the constant 3.14 and not to a space that can hold different real values.

An initialising declaration like '**char** t := "N" ' is an abbreviation of '**ref char** t = (**loc char** := "N")' and this form indicates clearly that "N" has been placed on the stack and that 't' refers to it. 't' is of mode **ref char** whereas 'ch' defined above is of mode **ref ref char**, referring to an unspecified **ref char**, that in turn refers to "N".

We may also create anonymous local space such as **loc int** for a local integer and even **loc** [m:n] **int** which creates an anonymous local reference to a row of integers, assuming that m and n have values at this point.

In the original maze algorithm, we assumed that the data always represented a square maze but let us reconsider this problem when the maze has jagged edges as in figure 7.1. The irregularity of the boundary

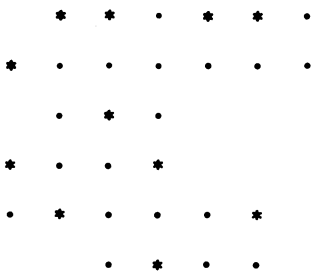

Figure 7.1

could be surmounted by surrounding the maze with an artificial square and completing each row of data with the necessary number of blocked symbols. To avoid this wastage of space (and time) we use local generators.

We need to declare a sequence of rows where each row has its length and starting point as part of the data. The complete data for the maze is arranged so that, first we have the number of rows (the size), and second, for each row we have a pair of integers specifying the starting column for that row and the length of the row, followed by the characters for the actual row, "." for free squares and "*" for blocked.

To solve this new shaped maze we shall modify the algorithm of chapter 3.

```
'begin
        int size; read(size);
        [0:size + 1] ref [] char maze; comment maze[i] is of mode
        ref ref [] char and will contain references to row i of the
        maze comment
        int col, length; string path := " ";
        for n to size
        do read((col, length, newline)); C col, starting position of
```

```
row, assumed positive, and length are on one line. Maze on
next C
    maze[n] := loc [col – 1, col + length] char;
C generate exact space required for each row plus extra
positions to be blocked at each end C
    for j from col to col + length – 1 do read(maze[n] [j])
                                          od;
C maze[n] selects the row and [j] the element in the row C
    maze[n] [col – 1] := maze[n] [col + length] := "*";
C to block the ends of the nth row. North and South moves
may still step outside the maze and need an extra check.
Now fill out front of jagged edge with blanks C
    for j from –2 to col – 2 do print(" ") od;
    for j from col – 1 to col + length do print(maze[n] [j])od;
    print(newline) C one row on each line C
od;
comment all rows read and printed. Now complete borders
for row 0 and row size + 1 comment
maze[0] := loc [0: size + 1] char;
for j from 0 to size + 1 do maze[0] [j] := "*" od;
C space could be saved by blocking only the exact length of
row 1 C
maze[size + 1] := maze[0];
comment both sides of this assignment are ref ref [] char
and the right-hand side is dereferenced once to ref[] char
and then the assignment made. Thus maze[size + 1] refers
to the same blocked row as maze[0] but only one copy of
the row is needed comment
int i, j, k, m; read((i, j, k, m));
print(("ENTRANCE  COORDS =", i, j, space,
"EXIT COORDS = ", k, m, newline, newline));
bool movepossible := true; C boolean marker C
if i < 1 or j < lwb maze[i] or k < 1 or m < lwb maze[k] or
i > size or j > upb maze[i] or k > size or m > upb maze[k]
then
print ("IMPREGNABLE MAZE?"); movepossible := false
fi;
while (maze[i] [j] := "T" C mark traversed C;
    not (i = k and j = m) and movepossible) do
if maze[i] [j + 1] = "." then j plusab 1; path plusab "E"
elif j > lwb maze[i + 1] and j < upb maze[i + 1] then
if maze[i + 1] [j] = "." then i plusab 1; path plusab "S" fi
elif maze [i] [j – 1] = "." then j minusab 1; path plusab
"W"
elif j > lwb maze[i – 1] and j < upb maze[i – 1] then
if maze[i – 1] [j] = "." then i minusab 1; path plusab "N"
```

```
            fi
            elif path = " " then movepossible := false
            else
            int p = upb path; char lastmove = path[p];
            int q;
            matchar (lastmove, "ESWN", q);
            case q in j minusab 1, i minusab 1, j plusab 1, i plusab 1
            out print("HELP"); movepossible := false esac;
            path := path[1:p - 1] C prune dead-end C
            fi
    od;
    if movepossible then
    print(("MAZE TRAVERSED BY FOLLOWING ROUTE", path,
    newline))
    else print("NO PATH POSSIBLE") fi
end'
```

In this revised version of the program, we have used local generators and replaced a sequence of **if** by a case clause.

Exercise 7.1

Use local generators to create a Pascal triangle of order 15 as a triangular array, and print it using a suitable format. The integers in a Pascal triangle are arranged as follows

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 5 1 etc.
```

In contrast, the other kind of Algol 68 generator creates global space, that is space which is available throughout the range of the entire program. This generator uses heap storage as opposed to the stack-like storage for local generators. We have already used heap storage implicitly with flexible multiples, for example **string**. A *heap generator* is introduced by a declaration preceded by **heap**.

An example to create terminal nodes for a tree could be

```
'mode tree = struct (ref tree left, char operator, ref tree right);
    comment the definition from chapter 5 comment
ref tree node1, node2, node3;
    node1 := heap tree := (nil, "a", nil);
    node2 := heap tree := (nil, "b", nil);
    node3 := heap tree := (nil, "c", nil)'
```

Here the generator **heap tree** creates storage space for the structure, the tree for "a" is then assigned to it and thence assigned to "node1' which has been previously declared, and is of mode **ref ref tree** (The word **heap** is omitted in some implementations.) Similarly for "b" and "c".

To generate a tree for 'a*b' we can now write

'**ref tree** expr2 := **heap tree** := (node1, "*", node2)', and for 'c + a*b', '**ref tree** expr3 := **heap tree** := (node3, "+", expr2)'

However, we may create a tree for a similar expression 'c + a/b' in one step without explicitly naming the subtrees as above.

'**ref tree** cab := **heap tree** := (node3, "+", **heap tree** := (node1, "/", node2))'

Unnamed space is generated to receive 'a/b' but it is pointed to by 'right **of** cab'. Thus heap space may be generated as required and its range is the whole program provided that there is some way of referring to that space. Consider the following

```
'ref tree coniferone,conifertwo,deciduous,a,b,x,y,p,q,r,u,v;
a := heap tree := (nil, "a", nil); C and similarly for the other letters C
   begin ref tree fleetwood;
      coniferone := heap tree := (a, "+", b);
      coniferone := heap tree := (x, "*", y);
      conifertwo := heap tree :=
      (p, "+", heap tree := (q, "*", r));
      fleetwood := conifertwo;
      deciduous := loc tree := (u, "–", v);
      goto autumn
   end;
```

autumn: **comment** at this point in the program 'coniferone' is available and refers to 'x*y', 'conifertwo' refers to 'p + q*r', 'fleetwood' is not known since its declaration was purely local to the serial clause, and although 'deciduous' exists the place to which it refers is undefined. It does not refer to 'u–v' because this was put into locally generated space within the serial clause and so is out of scope. Although space for 'a + b' was generated by the first assignment statement there is no means of access to it after the second assignment to 'coniferone' **comment**'

If this piece of program were embedded in a loop or do clause then heap space would be unnecessarily consumed. If the total available space for **heap** becomes exhausted then any heap space which is no longer accessible may be retrieved by an automatic process known as *garbage collection*. Thus, if garbage collection were initiated in the above example, the space taken for 'a + b' would be reclaimed but that for 'x*y' and 'p + q*r' would be untouched. Space for 'q*r' although created anonymously is accessible via 'right **of** conifertwo'.

We can now improve the algorithm in chapter 5 for reading an expression into a tree by using heap generators in the procedure 'maketree'

```
'proc maketree = void:
begin randstack[randtop−1] :=heap tree := (randstack[randtop−1],
   operators[opstack[optop]], randstack[randtop]);
   optop minusab 1; randtop minusab 1
end'
```

In this version we dispense with the global multiple named 'ever green[treetop]' which previously kept global copies of the latest subtree, replacing its occurrence by **'heap tree'**. All the structures generated are accessible from the root of the tree by using the selectors **'left of'** and **'right of'**.

Exercise 7.2

Improve the 'construct' procedure from chapter 5 in a similar way by using heap generators.

A possible ⟨ unit ⟩ in the BNF is an ⟨ expression ⟩, and this can be a ⟨ primary ⟩. Among the latter

⟨ primary ⟩ ::= **loc** ⟨ type ⟩ | **heap** ⟨ type ⟩

incorporates generators into the grammar.

The operator for matrix multiplication declared in chapter 4 was of limited use because the result was [,]**real** and not the name of a new matrix. Using heap generators we can now provide a more useful definition of the operator yielding a **ref** [,] **real** result.

```
'op * = (ref [,] real a,b) ref [,] real:
(int m = 1 upb a, p = 2 upb a, n = 2 upb b;
comment lower bounds are taken as 1 and the matrices assumed to
   be conformal, that is 1 upb b = 2 upb a comment
ref [,] real c;
c := heap [1:m, 1:n] real; C generate heap space for matrix result C
for i to m do for j to n do
   c[i,j] := 0;
   for k to p do c[i,j] plusab a[i,k] *b[k,j] od od od;
   c)'
```

We may now use this operator to form a matrix product, for example, (a*b)*d. The result of a*b is the name of a matrix, available on the heap, and so can be used as a multiplier with d.

Exercise 7.3

Write a declaration for the operator + on two matrices.

In our representation of trees, the terminal nodes are wasteful of space with their formally **nil** branches. An alternative would be to store the character of the terminal in the previous node in place of the **ref tree** pointer. In this case, the left or right field of a node could be a character instead of a **ref tree**. Algol 68 allows us to *unite* two modes by a declaration of the form '**union (char,int)** chin'.

'chin' may refer to either an integer or a character, but to only one of these types at any one time. The instructions 'chin := "a"; chin := 4' successively make 'chin' refer to a character and an integer, but 'chin' itself remains in the class defined by

⟨ non-multiple type ⟩ ::= **union** (⟨ formal type ⟩ ⟨, ⟨ formal type ⟩⟩i*)

Assignments like '**int** i; i := chin' are illegal, because 'chin' is a variable of united type and does not have **int** values. For the same reason 'print(chin)' is disallowed.

We can get at the current value of a **union** variable by a *case conformity*

'**case** chin **in** (**char** c): print(c), (**int** i): print(i) **esac**'

The effect of this clause is to find the type of value currently referred to by 'chin' (**int** since 'chin := 4' was the last assignment), and assign this value to the variable of matching type, 'i'. Further, the corresponding choice of the second alternative is made in the case clause, so that 'print(i)' is performed.

Exercise 7.4

Given a variable 'u' of type the union of all the basic types so far mentioned except **format** write instructions to print the current value of 'u' and its type.

We may apply united modes in a tree definition where left or right branches can be either a character or a tree name. Thus

'**mode tree = struct (union(char,ref tree)** left, **char** operator, **union(char,ref tree)** right)'

This could be further abbreviated by first declaring '**mode chartree** = union (**char, ref tree**)', and then using this in the definition of tree. However, the declaration of chartree would then contain a reference to a not-yet-defined **tree**. Algol 68 allows such mutually recursive definitions, but some implementations do not. However, the problem can be overcome by deferring the full definition, writing

'**mode chartree**;
mode tree = struct (chartree left, **char** operator, **chartree** right);
mode chartree = union (char, ref tree)'

Here the first **chartree** is an *incompletely defined* new mode that can be used in declarations of other modes, but cannot be used to declare variables until its complete declaration is given.

We may now represent expressions as 'chartrees', constructing the tree for 'a + b*c', say, concisely by

'**chartree** expr := **heap tree** := ("a","+",**heap tree** := ("b","*","c"))'

with the corresponding picture of figure 7.2 (compare with figure 5.5).

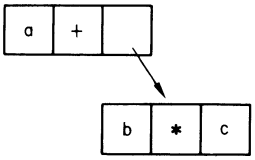

Figure 7.2

Exercise 7.5

A list may be defined as a finite sequence of zero or more cells, where a cell is declared by

'**mode cell = struct (union(char,ref cell)** item, **ref cell** link)'
A list for (a,b,c) could be created by '**ref cell** list :=
heap cell := ("a", **heap cell** := ("b", **heap cell** := ("c", **nil**)))'

Write a procedure to construct a list from a sequence of n characters. Notice that the link field of the final cell in the list cannot be a **char**, and is filled with **nil**.

In order to manipulate expressions represented in the united chartree mode we again need the case conformity. For example, the printtree procedure of chapter 5 becomes

```
'proc printtree = (chartree expr) void:
   begin print("(");
   case expr in (char ch): print(ch),
                (ref tree xpr): (printtree(left of xpr);
                print(operator of xpr);
                printtree(right of xpr)) esac;
   print(")") end'
```

As **chartree** is a united mode, we may not apply selectors such as 'left' or 'right' to it directly, and have to use the case conformity to extract the **char** or **ref tree**.

Although the united mode allows a more compact representation and possibly fewer written statements, it may be slower in operation

since the appropriate mode must be selected from the united variable prior to most manipulations.

Exercise 7.6

Replace the definition of **cell** in exercise 7.4 by

'**mode atom = union (int, char);**
mode cell = struct (union (atom, ref cell)item, **ref cell** link)'

Write a procedure to count the number of atoms in a cell, using this definition.

Heaping and the use of united modes, although useful for some algorithms, are expensive ways of using storage space and time. In contrast we now consider features that conserve space.

A storage location (machine word) can normally hold several characters, although the actual number depends on the wordlength. Characters *packed* into a word are called **bytes** in Algol 68. An identity declaration '[] **char** july = "JUL" ' might place the three characters "J", "U", "L" into different words, but '**bytes** july = bytespack ("JUL")' will pack "J","U","L" into a single word using the standard procedure 'bytespack' that converts strings into bytes.

The mode **bytes** is thus similar to [] **char**, but the number of characters is restricted by the length of the machine word, and can be found by the environment enquiry 'bytes width' which we shall assume has the value 4. 'bytespack' fills in with null characters to the right of the string to make up the bytes width.

Had we used **bytes** earlier, both the calendar and the maze data structures would have occupied less space. It is a basic type and we may declare [] **bytes**, whereas the implementation may not allow [] [] **char**.

Standard operators on bytes include the comparison operators and 'n **elem** b' which selects the nth character of byte b. For example

```
'string midsummer := "JUN";
bytes summer := bytespack("JULY");
print (3 elem summer) C L printed C;
summer := bytespack(midsummer);
for i to 4 do print(i elem summer) od;
comment prints J,U,N,and a final zero, which is printed for the null
    character. comment'
```

We can also pack more than four characters into bytes by using two or even three machine words. Algol 68 has *long* modes such as **long bytes** implying that two words are being used, and so up to eight characters may be stored. **long long bytes** uses three words accommodating up to twelve characters. Similarly **long int** may be implemented, occupying twice the space of an **int** value with a corresponding

increase in the magnitude of integers that can be processed. However, not every mode has a **long** variant and the number of longs available depends on the implementation.

The calendar program from chapter 4 may be changed to use bytes, replacing the declaration of 'monthchecker' by

```
'proc (string)bytes bp := bytespack;
[ ] bytes monthchecker = (bp("JAN"), bp("FEB"), . . .
bp("DEC"))'
```

Thereafter we use 'monthchecker[i]' instead of 'monthchecker[i,]' to refer to the string for the ith month.

Exercise 7.7

Write a procedure to reverse the order of the elements of a **bytes** variable.

Internally, computers hold numbers (and everything else) in the scale of 2, and each word contains a string of binary digits (0 or 1 referred to as *bits*). We can access these by using the mode '**bits**', similar to [ ] **bool** (assuming a **bool** occupies only one bit).

The number of bits per word is supplied by the environment enquiry 'bitswidth', which we shall assume is 24. The standard function 'bitspack' turns a row of **bool** into **bits**, filled out with **false** represented as 0 at the left end of the word. The standard operator **elem** selects a bit and yields the boolean value, **true** if it is 1 or **false** if it is 0, and there may be operators for shifting the bits of a word to the left or right or cyclically.

bits can serve to keep a compact copy of the data for the original maze algorithm of chapter 3. There were only two different values in the data input, one to represent a free square and the other a blocked square. If we assume that the size of the maze is not more than 24 (bitswidth), then we can represent the maze as a row of bits, and therefore use only 'size' words of store instead of the 'size x size' words used previously.

The arrangement of data for bits input is a sequence of 0s and 1s with the input stream searched for the first non-space character, taking new lines or new pages if necessary. We let 1 represent a free square and 0 a blocked square. To input the maze in this compact form we now write

```
'int size; read(size); if size > 24 then goto too big fi;
[1:size] bits maze;
read(maze)'
```

The input is 'size' rows, each row having 'size' (< 24) bits starting on a fresh line. The remainder of the algorithm puts different values into the positions of the maze so that **bits** with only two possible values is inadequate.

Exercise 7.8

Declare a multiple of **char** 'charmaze' and using the **bits** 'maze', filled above, copy the contents of 'maze' into 'charmaze', replacing 0 by "*" and 1 by ".".

The use of logical operations on **bits** values is illustrated by an extension of the mortgage problem from chapter 6. The building society may need to interrogate its file for information about clients, such as those with an outstanding mortgage in excess of £7500, those who have missed a repayment, those with repayments above £50 per month, those with a repayment period above 25 years, those whose mortgage will be fully paid in the current year, and those who live in London.

This could be done by accessing each record and then examining the various fields. A more efficient method would be to add an extra field for each record to store the information. This field would be a **bits** field and arranged so that a bit was 1 (or **true**) when a particular condition was satisfied in that record and 0 (or **false**) otherwise. Thus if the above information was required, a bit pattern of 101100 would indicate a mortgage over £7500 with repayments over £50 and a term of over 25 years. An algorithm to produce a list of all clients in this category would be

1. Read **bits** field of record.
2. Test appropriate bits for value 1.
3. If required bits are 1, read rest of record card and print client's name and number, otherwise skip record.
4. Repeat from step 1 until file exhausted.

To compare bit patterns we are usually only interested in part of the pattern and wish to *mask* (or discard) the rest. This can be done by the logical operators **and**, **or**, **not** which apply bitwise to bits values. For example, **not** changes 0s to 1s and vice versa. Hence to look at the required 6 bits, assuming that they are at the right-hand end of the word, we mask the remaining bits. This can be done by using an **and** operation with the constant 111111(that is 63 in binary, 18 left-end bits all zero). A piece of program for the algorithm is

```
'bits key; getfield(key); comment assume this is a proc. to extract
    the appropriate field from the record comment
if key and (bin 63) = bin 44 then printclients fi
comment bin applied to an integer gives the bits value representing
    that integer. bin 44 = 101100, the bit pattern desired comment'
```

If at a later date a client pays off some of the mortgage capital to make the outstanding sum less than £7500, then the field would also need to be changed by writing

```
'if 19 elem key then key := key and not(bin 32) fi
comment elem has been used to select the bit. If bit 19 is 1, it is to
    be changed to 0 but the remaining bits left, so that only bit 19 is
    zero in the mask comment'
```

Exercise 7.9

Declare a multiple of structures to represent the following personal details: male, year of birth, number of children, number of grandchildren, spouse living, year of graduation, year of retirement, salary.

Write a procedure to list from the multiple all 70-year-old, non-retired, graduate widowers with salary over £1000 p.a.

This chapter has introduced some of the more advanced features of Algol 68 that are relevant to the examples discussed in earlier chapters. It is not intended to be a comprehensive treatment of these features or indeed of any of the topics met earlier. However, it should be sufficient for the solution of a wide range of problems.

Debugging was mentioned in chapter 1 as an important aspect of programming. Wherever feasible when writing programs, as much information as possible should be obtained by judicious insertion of print statements about the course of the program and the intermediate values of variables. These are not in evidence in the examples in this book but were used extensively in verifying the correctness of the programs.

Problems for Chapter 7

1. An expression may include both unary and binary operators. Write a procedure corresponding to 'printtree' of chapter 5 that prints a fully bracketed form of such an expression.

2. Using the definitions of list in exercise 7.5, write procedures to join two lists and to find the first atom in a list.

3. Rewrite the maze program so that once having found a route to the exit, it goes on to find the shortest one.

4. Write procedures to test for the equality of two atoms; to produce a list whose components are the components of a given list in reverse order; to list the atoms common to two given lists.

5. Given an expression represented as a tree, write a program to obtain the derivative of the expression again in tree form and then print the result.

Appendix I Standard Operations and Functions

This list includes the standard operations and functions in this book together with other commonly used additions. The standard prelude of a particular implementation will normally include all of these and other information.

Unary Operators (priority = 10)

Operator	Mode of argument	Mode of result	Remarks
not	**bool**	**bool**	
	bits	**bits**	negation of each bit
+,–	**int**	**int**	
	real	**real**	
bin	**int**	**bits**	conversion to bits
conj	**compl**	**compl**	complex conjugate
arg	**compl**	**real**	principal argument
abs	**int**	**int**	
	real	**real**	
	bool	**int**	0 for **false**,1 for **true**
	bits	**int**	inverse for **bin**
	char	**int**	code for each character
odd	**int**	**bool**	
sign	**int** | **real**	**int**	+1, 0, –1 for positives, zero, negatives respectively
round	**real**	**int**	nearest whole number
entier	**real**	**int**	greatest int below
repr	**int**	**char**	character corresponding to integer code
upb | **lwb**	multiple	**int**	upper and lower bound

Binary operators

Operator	Priority	Mode of first argument	Mode of second argument	Mode of result	Remarks
↑	8	**int** **real** **compl**	**int**	**int** **real** **compl**	taking powers
upb **lwb**	8	**int**	multiple	**int**	bounds of each dimension
÷ **mod**	7	**int**	**int**	**int**	a = (a ÷ b)*b + a **mod** b 0 ⩽ a **mod** b < b
* /	7	**int** **real** **compl** **int** **real** **compl** **int** **real** **compl**	**int** **real** **compl** **real** **int** **int** **compl** **compl** **real**	**int** **real** **compl** **real** **real** **compl** **compl** **compl** **compl**	**int / int** is **real**
elem	7	**int**	**bits** **bytes**	**bool** **char**	selection
+,–	6	**int** **real**	**int** **real**	**int** **real**	

Operator	Priority	Mode of first argument	Mode of second argument	Mode of result	Remarks
		compl	**compl**	**compl**	
		int	**real**	**real**	
		real	**int**	**real**	
		compl	**int**	**compl**	
		int	**compl**	**compl**	
		real	**compl**	**compl**	
		compl	**real**	**compl**	
+	6	**string**	**string**	**string**	concatenation
		char	**string**	**string**	
		string	**char**	**string**	
$<\leqslant$	5	**int**	**int**		
$>\geqslant$		**real**	**real**		
		real	**int**		
		int	**real**		
		char	**char**	**bool**	character by
		string	**string**		character
		string	**char**		comparison
		char	**string**		**true** after first
		bytes	**bytes**		difference

Operator	Priority	Mode of first argument	Mode of second argument	Mode of result	Remarks
= ≠	4	**bool**	**bool**	**bool**	
		int, **real**, **compl** with	**int**, **real**, **compl**		
		bits	**bits**		
		char, **string** with	**char**, **string**		
		bytes	**bytes**		
and	3	**bool**	**bool**	**bool**	
		bits	**bits**	**bits**	
or	2	**bool**	**bool**	**bool**	
		bits	**bits**	**bits**	
timesab	1	**ref int**	**int**	**ref int**	a := a*b
divab		**ref real**	**real**	**ref real**	a := a/b
plusab		**ref compl**	**compl**	**ref compl**	a := a + b
minusab		**ref real**	**int**	**ref real**	a := a – b
		ref compl	**int**	**ref compl**	no **divab**
		ref compl	**real**	**ref compl**	on **int**
plusab	1	**ref string**	**string**	**ref string**	a := a + b
		ref string	**char**	**ref string**	concatenation

Functions

Name	Modes
sqrt	**proc (real) real**
exp	
ln	
cos	
arccos	
sin	
arcsin	
tan	
arctan	
bytespack	**proc (string) bytes**
bitspack	**proc ([ ]bool) bits**

Appendix II A BNF for Algol 68

The following alphabetical grammar in BNF form is intended as an aid to the explanations in the book, and as a guide in checking the syntax of programs.

The notations '⟨ w ⟩' for the class of all w objects, '::=' for consists of '|' for or, '⟨ w ⟩* (respectively ⟨ w ⟩⁺)' for zero (respectively one) or more w objects one after the other, and '⟨ w ⟩[1]' for an optional occurrence of a w object, appeared in chapter 2.

'⟨ w ⟩, ⟨ w ⟩⟩*', that is a list of w objects of length at least one and separated by commas, is so common that the abbreviation '⟨⟨ w ⟩ LIST ⟩' is used for it. For example, '2' and '2,3' and '2,3,2' are all objects in ⟨⟨ digit ⟩ LIST ⟩. The notation is due to F. G. Duncan (*Algol Bulletin* 26, p. 28).

The names of the classes have been abbreviated for convenience, for example 'decl' for 'declaration'. Some classes are implementation-dependent and remain undefined. 'comment' and abbreviations like (|) for **if then fi** have been omitted from the grammar.

It is to be stressed that this is not an attempt to write a formal BNF for Algol 68. Such a treatment would not for example single out ⟨ arith exp ⟩ or ⟨ bool exp ⟩ from the generality of expressions. The classes and productions have been chosen for didactic and pedagogic reasons.

All the programs in this book lie in the class ⟨ program ⟩, but most strings in the latter class constructed by blindly following the BNF will be nonsense. (This would remain true even if we narrowed our broad definitions of classes like ⟨ unit ⟩.)

⟨ adding op ⟩ ::= + | −
⟨ alignment ⟩ ::= ⟨ replicator ⟩[1] ⟨ tabulator ⟩
⟨ arith exp ⟩ ::= ⟨ term ⟩ | ⟨ arith exp ⟩ ⟨ adding op ⟩ ⟨ term ⟩
⟨ arith op ⟩ ::= ⟨ adding op ⟩ | ⟨ mult op ⟩

⟨ basic arith exp ⟩ ::= ⟨ primary ⟩ | (⟨ arith exp ⟩)
⟨ basic bool exp ⟩ ::= ⟨ primary ⟩ | **not** ⟨ basic bool exp ⟩ |
⟨ unit ⟩ ⟨ bool op ⟩ ⟨ unit ⟩ | (⟨ bool exp ⟩)
⟨ basic type ⟩ ::= **real** | **int** | **char** | **bool** | **format** | **bytes** | **bits**
⟨ bool den ⟩ ::= **true** | **false**
⟨ bool exp ⟩ ::= ⟨⟨ bool exp ⟩ **or** ⟩[1] ⟨ bool term ⟩
⟨ bool op ⟩ ::= < | ≤ | = | ≠ | ≥ | >
⟨ bool term ⟩ ::= ⟨⟨ bool term ⟩ **and** ⟩[1] ⟨ basic bool exp ⟩

⟨ case cl ⟩ ::= **case** ⟨ unit ⟩ **in** ⟨ unit ⟩ ⟨, ⟨ unit ⟩⟩‡* ⟨ **out** ⟨ ser cl ⟩⟩¹ **esac** |
case ⟨ identifier ⟩ **in** (⟨ type ⟩ ⟨ identifier ⟩):⟨ unit ⟩
⟨,(⟨ type ⟩ ⟨ identifier ⟩):⟨ unit ⟩⟩‡* ⟨ **out** ⟨ sercl ⟩⟩¹ **esac**
⟨ cast ⟩ ::= ⟨ formal type ⟩(⟨ unit ⟩)
⟨ char ⟩ ::= ⟨ letter ⟩ | ⟨ digit ⟩ | ⟨ other char symbol ⟩
⟨ closed cl ⟩ ::= (⟨ ser cl ⟩) | **begin** ⟨ ser cl ⟩ **end**
⟨ compl den ⟩ ::= ⟨ real den ⟩ **i** ⟨ real den ⟩

⟨ decimal ⟩ ::= ⟨ int den ⟩ ⟨.⟨ int den ⟩⟩¹
⟨ decl ⟩ ::= ⟨ identity decl ⟩ | ⟨ type ⟩ ⟨⟨ identifier ⟩ ⟨ := ⟨ unit ⟩⟩¹ ⟩ LIST ⟩ |
⟨ incomplete mode ⟩
⟨ denotation ⟩ ::= ⟨ int den ⟩ | ⟨ real den ⟩ | ⟨ compl den ⟩ | ⟨ bool den ⟩ |
⟨ string den ⟩ | $ ⟨ format den ⟩ $ | ⟨ row display ⟩ |
⟨ routine text ⟩
⟨ digit ⟩ ::= 0 | ⟨ non-zero digit ⟩
⟨ do cl ⟩ ::= ⟨ **for** ⟨ identifier ⟩⟩¹ ⟨**from** ⟨ unit ⟩⟩¹ ⟨**by** ⟨ unit⟩⟩¹
⟨ **to** ⟨ unit ⟩⟩¹ ⟨**while** ⟨ ser cl ⟩⟩¹ **do** ⟨ ser cl ⟩ **od**

⟨ expression ⟩ ::= ⟨ primary ⟩ | ⟨ arith exp ⟩ | ⟨ bool exp ⟩

⟨ factor ⟩ ::= ⟨ factor ⟩ ↑ ⟨ basic arith exp ⟩ | ⟨ sign ⟩¹ ⟨ basic arith exp⟩
⟨ field ⟩ ::= ⟨ type ⟩ ⟨⟨ identifier ⟩ LIST ⟩
⟨ flexible ⟩ ::= **flex**
⟨ formal dec ⟩ ::= ⟨ formal type ⟩ ⟨⟨ identifier ⟩ LIST ⟩
⟨ formal type ⟩ ::= ⟨ [⟨ , ⟩*] ⟩* ⟨ non-multiple type ⟩ |
struct (⟨⟨ formal dec ⟩ LIST ⟩)
⟨ format den ⟩ ::=
⟨⟨⟨ replicator ⟩ (⟨⟨ picture ⟩ LIST ⟩) | ⟨⟨ picture ⟩ LIST ⟩⟩ LIST ⟩
⟨ frame ⟩ ::= d | e | g | a | z | . | + | – | ⟨ frame choice⟩
⟨ frame choice ⟩ ::= ⟨ c | b ⟩ (⟨⟨ string den ⟩ LIST ⟩)

⟨ goto cl ⟩ ::= **goto** ⟨ label ⟩

⟨ identifier ⟩ ::= ⟨ letter ⟩ ⟨⟨ letter ⟩ | ⟨ digit ⟩⟩*
⟨ identity decl ⟩ ::= ⟨ formal type ⟩ ⟨⟨⟨ identifier ⟩ = ⟨ unit ⟩⟩ LIST ⟩ |
mode ⟨⟨⟨ modename ⟩ = ⟨ type ⟩⟩ LIST ⟩ |
proc ⟨⟨⟨ identifier ⟩ = ⟨ routine text ⟩⟩ LIST ⟩ |
op ⟨⟨⟨ opsymbol ⟩ = ⟨ non- void routine ⟩⟩ LIST ⟩ |
prio ⟨⟨⟨ opsymbol ⟩ = ⟨ non-zero digit ⟩⟩ LIST ⟩
⟨ if cl ⟩ ::= **if** ⟨ ser cl ⟩ **then** ⟨ ser cl ⟩ ⟨ **elif** ⟨ ser cl ⟩ **then** ⟨ ser cl ⟩⟩*
⟨ **else** ⟨ ser cl ⟩⟩¹ **fi**
⟨ incomplete mode ⟩ ::= **mode** ⟨ modename ⟩
⟨ insertion ⟩ ::= ⟨ literal ⟩ | ⟨ alignment ⟩
⟨ int den ⟩ ::= ⟨ digit ⟩‡*
⟨ is cl ⟩ ::= ⟨ expression ⟩ ⟨ **is** | **isnt** ⟩ ⟨ expression ⟩

⟨ label ⟩ ::= ⟨ identifier ⟩

⟨ letter ⟩ ::= A | B | . . . . | Z
⟨ literal ⟩ ::= ⟨ replicator ⟩^{1}⟨ string den ⟩

⟨ mode name ⟩ ::= ⟨ new keyword ⟩
⟨ mult op ⟩ ::= * | / | ÷
⟨ multiple primary ⟩ ::=
⟨ primary ⟩ [⟪⟪ unit ⟩^{1}⟨:⟨ unit ⟩⟫^{1}⟨ **at** ⟨ unit ⟩⟫^{1}⟩ LIST ⟩]

⟨ new keyword ⟩ not defined e.g. matrix, tree
⟨ non-multiple type ⟩ ::= ⟨ basic type ⟩ | ⟨ mode name ⟩ | ⟨ prelude type ⟩ |
struct (⟪ field ⟩ LIST ⟩) | **ref** ⟨ formal type ⟩ |
proc ⟨(⟨ formal type ⟩ **LIST** ⟩)⟫^{1}⟨ **void** | ⟨ formal type ⟩⟫ |
union (⟨formal type ⟩⟨, ⟨ formal type ⟩⟫$^{*}_{1}$)
⟨ non-void routine ⟩ ::=
⟨(⟪ formal dec ⟩ LIST ⟩)⟫^{1}⟨ formal type ⟩ : ⟨ closed cl ⟩
⟨ non-zero digit ⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨ op primary ⟩ ::= ⟨ unit ⟩^{1}⟨ opsymbol ⟩⟨ unit ⟩
⟨ opsymbol ⟩ ::= ⟨ arith op ⟩ | ⟨ bool op ⟩ | ⟨ prelude op ⟩ |
⟨ new keyword ⟩ | ⟨ spare opsymbol ⟩
⟨ other char symbol ⟩ not defined e.g. £, /, ”

⟨ pattern ⟩ ::= ⟨ replicator ⟩^{1}⟨ suppress ⟩^{1}⟨ frame ⟩
⟨ picture ⟩ ::= ⟪ insertion ⟩^{1}⟨ pattern ⟩⟨ insertion ⟩^{1}⟫$^{*}_{1}$
⟨ prelude op ⟩ not defined e.g. **upb**, **plusab**
⟨ prelude type ⟩ not defined e.g. **compl**, **file**, **string**
⟨ primary ⟩ ::= ⟨ identifier ⟩ | ⟨ closed cl ⟩ | ⟨ case cl ⟩ | ⟨ if cl ⟩ |
⟨ denotation ⟩ | ⟨ identifier ⟩ **of** ⟨ primary ⟩ |
loc ⟨ type ⟩ | heap ⟨ type ⟩ | ⟨ proc primary ⟩ |
⟨ op primary ⟩ | ⟨ multiple primary ⟩ | ⟨ cast ⟩
⟨ proc primary ⟩ ::= ⟨ primary ⟩⟨ row display ⟩
⟨ program ⟩ ::= **begin** ⟨ ser cl ⟩ **end**

⟨ real den ⟩ ::= ⟨ decimal ⟩ | ⟨ decimal ⟩^{1} & ⟨ sign ⟩^{1}⟨ int den⟩
⟨ replicator ⟩ ::= ⟨ int den ⟩ | n (⟪ ser cl ⟩)
⟨ routine text ⟩ ::= ⟨ void routine ⟩ | ⟨ non-void routine ⟩
⟨ row display ⟩ ::= (⟪ unit ⟩ LIST ⟩)

⟨ ser cl ⟩ ::= ⟨ ⟪ decl ⟩ | ⟨ unit ⟩⟫⟨;⟪ decl ⟩ | ⟨ unit ⟩⟫⟩* ⟩^{1}
⟪ label ⟩ : ⟩^{1}⟨ unit ⟩⟨; ⟪ label ⟩ : ⟩^{1}⟨ unit ⟩⟫*
⟨ sign ⟩ ::= + | –
⟨ spare opsymbol ⟩ not defined e.g. ?, %
⟨ string den ⟩ ::= “⟨ char ⟩*”
⟨ suppress ⟩ ::= s

⟨ tabulator ⟩ ::= x | y | l | p | k

⟨ term ⟩ ::= ⟨⟨ term ⟩ ⟨ mult op ⟩⟩[1] ⟨factor ⟩
⟨ type ⟩ ::=
  ⟨⟨ flexible ⟩[1] [⟨⟨⟨ unit ⟩ : ⟨ unit ⟩⟩ LIST ⟩]⟩* ⟨non-multiple type ⟩

⟨ unit ⟩ ::= ⟨ expression ⟩ ⟨ := ⟨ unit ⟩⟩[1] | ⟨ do cl ⟩ | ⟨ goto cl ⟩ |
  ⟨ is cl ⟩ | **skip** | **nil**

⟨ void routine ⟩ ::= ⟨(⟨⟨ formal dec ⟩ LIST ⟩)⟩[1] **void** : ⟨ closed cl ⟩

Appendix III Implementations

An implementation of Algol 68 will usually impose certain constraints upon the way in which programs are written. These are introduced for ease and efficiency of compilation and so as to reduce the number of passes (the times that the source or intermediate language is scanned). A one-pass compiler is possible under certain restrictions, typically that an object cannot be mentioned until it is defined or is in the process of being defined.

There is a growing number of compilers available on a variety of computers. Among those known to the authors are Algol 68C implemented at Cambridge University on an IBM 370/165, also available on an ICL 4130; a Modular One implementation from Liverpool University; and the Algol 68R compiler constructed by the Royal Radar Establishment at Malvern for the ICL 1900 range.

In the text no mention has been made of hardware representation, that is, how the language is to be represented on an input or output medium. Card punches have a limited character set and so keywords (boldface or underlined in program text) must be distinguished by enclosing them in quotes like 'BEGIN', or reserving these words entirely so that no variable named begin, etc. may be used. Tape punches have upper and lower case so that the former may be kept for keywords. The machine on which programs are run affects, among other things, the range of values of integers and reals that can be used, the number of bits and bytes per word and the maximum size of multiples.

A compiler may impose conditions that make the language implemented significantly different from Algol 68 as defined in the Revised Report. Transput particularly may diverge from the standard. The peculiarities of any implementation are usually well documented.

All the programs in this book were run on an ICL 1904S computer using the Algol 68R compiler, which occupies 36K 24 bit words. The language is not a sublanguage of Algol 68 and details can be found in *Algol 68R Users Guide*, H.M.S.O., 1972. The following conditions must be observed in an Algol 68R program:

- any closed clause containing a declaration must start with one;
- multiples of multiples, including multiples of strings, are not allowed;
- not all the standard operators are defined in the prelude.

In addition, the textual changes given in the following table are needed to convert our programs into Algol 68R.

Algol 68		Algol 68R
do	replaced by	**do begin**
od		**end**
elif		**elsf**
prio		**priority**
compl		**complex**
flex [p:q]		[p:q **flex**]
plusab		**plus**
minusab		**minus**
timesab		**times**
divab		**div**
real(m) when casting		**real val** m
heap		omit
: in case conformity		omit
finish	must be inserted after final **end** in 68R	

Transput differences include the following:

file folio	replaced by	**charput** folio
open (folio, bookname, channel no)		openc (folio, card reader, channel no) for input; 'lineprinter' replaces 'cardreader' for output
close(folio)		closec(folio)
getf(folio, (format, variable list))		inf(folio, format, variable list)
putf(–, (–, –))		outf(–, –, –)

Just one format and one variable list is allowed in 'inf' or 'outf' and not a mixture as in 'getf' or 'putf'.

readf((–, –))	inf(standin, –, –)
printf((–, –))	outf(standout, –, –)
getf(folio, form) (where 'form is a format)	format(folio, form)

Similarly replace putf, readf, or printf, when used to associate a format with a file, by the procedure 'format' in 68R, and after the association, make the replacements

getf(folio, variable list)	in(folio, variable list)
putf(–, –)	out(–, –)

Solutions to Exercises

Chapter 2

2.1 (a) (i) Set n to 1. (ii) Print n and n^2. (iii) Add 1 to n. (iv) If $n < 11$ then repeat from step (ii).
(b) Left to the reader. An algorithm is no guarantee of success!

2.2 (a) b↑2 – 4*a*c, 0.5*sin(a*x). (b) 20.75.

2.3 **'begin for** n **to** 10 **do** print((n, n↑2, n↑3, newline))**od end'**
The 'print' statement here is equivalent to the 4 statements 'print(n); print(n↑2); print(n↑3); print(newline)'.

2.4 (a) 13.0 (b) –3.0 (c) –6.0

2.5 (a) Constant and non-constant declarations cannot be combined. Correct is **'real** pi = 3.14; **real** s'.
(b) Order of these declarations separated by commas is unspecified, so that 'pie' may not have a value when needed. **'real** pie = 3.14; **real** pi3 = pie↑3' is correct.

2.6
```
'begin real y, sum, term, nextterm, error;
real epsilon = 1 & –11;
while read(y); abs y > epsilon do
sum := term := y;
error := 0.5;
for n while error > epsilon do
nextterm := –term*y↑2/(2*n + 1);
sum plusab nextterm;
error := abs (nextterm + term); C + because terms alternate C
term := nextterm od;
print(("Y IS", y, "SUM IS", sum, newline))
od end'
```

2.7 (a) –2 (b) –3 (c) 6

2.8
```
⟨digit⟩ ::= 0 | ⟨non-zero digit⟩
⟨non-zero digit⟩ ::= 1 | 2 | . . . | 9
⟨no leading 0 integer⟩ ::= 0 | ⟨sign⟩[1] ⟨non-zero digit⟩⟨digit⟩*
⟨sign⟩ ::= + | –
```

2.9 (a) No, contains apostrophe. (b) Yes. (c) No, starts with digit. (d) No, contains full stop.

2.10 (a) 4 times. (b) 3 times.

2.11 Following the ::= through the BNF, one can successively 'generate' the strings
⟨arith exp⟩
⟨term⟩
⟨term⟩ ⟨mult op⟩ ⟨factor⟩
⟨factor⟩ / ⟨basic arith exp⟩
⟨basic arith exp⟩ / (⟨arith exp⟩)
(⟨arith exp⟩) / (⟨arith exp⟩ ⟨adding op⟩ ⟨term⟩)
(⟨term⟩) / (⟨term⟩ – ⟨factor⟩)
(⟨factor⟩) / (⟨term⟩⟨multop⟩⟨factor⟩ – ⟨basic arith exp⟩)
(⟨factor⟩↑⟨basic arith exp⟩)/(⟨factor⟩*⟨basic arith exp⟩ – ⟨primary⟩)
(⟨basic arith exp⟩↑⟨primary⟩)/(⟨basic arith exp⟩*⟨primary⟩ – 1)
(⟨primary⟩↑2)/(⟨primary⟩*n – 1)
(y↑2)/(2*n – 1)

Chapter 3

3.1 8; 441; 24.

3.2 '[1:9] **char** nos := ("1", "2", "3", "4", "5", "6", "7", "8", "9"); **for** i **to** 9 **do** print((**abs** nos[i], newline)) **od**'
('**for** n **to** 9 **do** print(**abs** "N") **od**' would not work because "N" is always the same letter character).

3.3 A counter-example would be a maze containing

```
  *  IN  *  *  *
OUT  ●  *  *  *
  *  ●  *  ●  *
  *  ●  ●  ●  *
  *  *  *  *  *
```

3.4 '**begin** [1:4, 1:4] **int** matrix, newtrix; read(matrix);
for i **to** 4 **do** newtrix[i,] := matrix[,i] **od**;
print(newtrix) **end**'

3.5 Marking free squares on correct route by "R" and dead-ends pruned by "D", the solutions are

```
R  R  S  D        X  R  ●  S
X  *  *  D        *  R  R  R
*  D  D  D        D  D  *  D
*  D  D  D        D  D  *  D
```

3.6 (a) 22; (b) 43; (c) 106; (d) 151; (e) 125.

3.7
```
'begin string s; read(s); C reading of strings is
      terminated by end of line (see chapter 6) C
      int no of vowels := 0;
      for k from upb s by −1 to 1 do
      char c = s[k] ; print(c);
      if c = "A" or c = "E" or c = "I" or c = "O" or c = "U"
      then no of vowels plusab 1 fi od;
      print(no of vowels) end'
```

3.8 (a) One method is an exhaustive test of all 8 possible truth values of P, Q, R. Alternatively, assume **'not P and not Q'** is **true**. Then **'not P'** is **true**, so P is **false**. Similarly Q is **false**. Hence **'P or Q'** is **false**, so **'not (P or Q)'** is **true**. Converse and the second result have similar proofs.
(b) If and only if R is **true**.

Chapter 4

4.1 advancedays(2000) = 27 + 6 − 0 + 0 − 0 = 33
b = (31 + 29 − 1) = 59 (31 days in January)
(1 + 33 + 59) mod 7 = 2, so 29 Feb 2000 is a Tuesday.

4.2
```
'proc iseven = (int n) bool: (n = (n ÷ 2)*2);
proc has2factorslessthan100 = (int n) bool:
      begin int q; bool oncediv := false, twicediv := false;
            C markers for once and twice divisible C
            if iseven(n) then oncediv := true; q := n ÷ 2;
                        if iseven(q) then twicediv := true fi fi;
            int f := 3; C to be first odd factor, if any C
            for j from 3 by 2 to 97 while not oncediv
      do if n = (n ÷ j)*j then oncediv := true; q := n ÷ j; f := j fi
      od;
      if oncediv then for k from f by 2 to 97
            while not twicediv do if q = (q ÷ k)*k then
            twicediv := true fi od fi;
            twicediv
      end'
```

4.3
```
'nationality of john := "BRITISH";
birthdate of john := (25, "DEC", 1940);
age of john := C depends when you are reading this, say C 34'
```

4.4
```
'proc roots = (real a, b, c) void:
begin real d = b↑2 − 4*a*c, eps = 1 & −11;
if abs a < eps then
```

```
if abs b < eps then print("NO EQUATION")
else print(("LINEAR EQUATION, ROOT = ", – c/b)) fi
else case sign d + 2 in
print("NO REAL ROOTS"),
print((" EQUAL ROOTS =", –b/(2*a) )),
(real r = (–b + sqrt(d))/(2*a);
print(("TWO REAL ROOTS =", r, –b/a – r)) )
esac fi
end'
```

4.5
```
'proc allroots = (real a, b, c) void:
begin comment assume a ≠ 0, otherwise see 4.4 comment
real d = b↑2 – 4*a*c;
case sign d + 2 in
(compl z; re of z := –b/(2*a);
im of z := sqrt(–d)/(2*a);
print(("2 COMPLEX ROOTS =", z, conj z)) ),
print(("EQUAL ROOTS =", –b/(2*a))),
(real x = –b + sqrt(d);
print(("2 REAL ROOTS =", x/(2*a), (–2*b – x)/(2*a))) )
esac
end'
```

4.6 (a) 3 6

(b) We rewrite the program with markers and comments to indicate scope.

```
'begin real y C scope here to end C; read(y);
real x C scope here to mark 1 and mark 5 to end C;
if y > 3 then real z := 8 C scope only this declaration C
else x := 5 C mark 1 C fi; (real z C scope here to mark 2 and
mark 3 to mark 4 C, x C scope here to mark 4 C; z := y;
(real t C scope the if clause surrounded by brackets
C; z > 2 C mark 2 C |
real w C scope here to vertical else bar C; y plusab 4;
real z C scope only the next assignment C; z := y |
C mark 3 C x := z C mark 4 C)); C mark 5 C x := y end'
```

4.7
```
'proc monthhead = (int month) void:
(print((newline, newline)); space(56);
print((case month in "JANUARY", ... "DECEMBER" out
"WRONG" esac, newline, newline)) )'
```

4.8
```
'proc nodays in = (int month, year) int:
(case month in 31, 28 + abs leap(year), 31, 30, 31, 30, 31, 31, 30,
31, 30, 31 out skip esac)'
```

4.9 (a) '**op ceil** = (**real** x) **int**:
C tricky, x might be 'exactly' a whole number **C**
(**int** k := 0; **while** k > x **do** k **minusab** 1 **od**;
while k < x **do** k **plusab** 1 **od**;
k)'
(b) '**op** ≠ = ([] **char** a,b) **bool**:
begin bool result := **true**; **int** m;
int ua = **upb** a, la = **lwb** a, ub = **upb** b, lb = **lwb** b;
if ua = ub **and** la = lb **then**
for i **from** la **to** ua **while** a[i] = b[i] **do** m := i **od**;
if m = ua **then** result := **false fi fi**;
result **end**'

4.10 Assume + defined between string and character as in text.
'**op** + = (**string** s,t) **string**:
(**int** m = **upb** t; **string** r := s;
for i **to** m **do** r := r + t[i] **od**; r)'

4.11 '**proc** innerprod = (**ref** [] **real** a,b) **real**:
begin int k = **upb** a; **C** assume also k = **upb** b **C**
real prod := 0;
for i **to** k **do** prod **plusab** a[i] *b[i] **od**;
prod **end**'

4.12 '**proc** (**int**, **real**) **real** expmult :=
(**int** n, **real** y) **real**: (y/n);
C this is necessary since a **ref proc** parameter expected **C**
real x; read(x);
print(series sum(1,1 & −11, x, 50, expmult,
void:(print("MISTAKE"))))'

4.13 '**proc** length = (**proc real** xi, **ref int** i, **int** n) **real**:
(**real** s := 0;
for k **to** n **do** i := k; s **plusab** xi*xi **od**; sqrt(s))'

4.14 '**proc** recfact = (**int** n) **int**:
(**if** n = 1 **then** 1 **else** n*recfact(n − 1));
proc itfact = (**int** n) **int**:
(**int** r := 1; **for** i **from** 2 **to** n **do** r **timesab** i **od**;
r) **C** both procs assume n > 0 **C**'
Iteration is usually more efficient than recursion. A recursive procedure call involves the storage of the return point in the calling program, and the stacking of values. Compilers vary in efficiency in dealing with recursion, but it is an attractive method for human problem-solving.

Chapter 5

5.1 ab↑c*d/ , ab/cd↑*, abc*d↑/.

5.2 1. Read next character, call it c.
2. If c is operand push onto list, repeat step 1.
3. If c is operator then pop off top character from list into a, pop off top character from list into b, do arithmetic operation a **op** b, push result onto list, repeat step 1.
4. If c is terminator, result is only item left on list.
N.B. Care must be taken that the pushdown list is not empty when pop off performed. The expression is assumed correct.

5.3 (a) Replace '**char** ch' by '**char** ch := "X"; while ch ≠ "%" **do**', and penultimate line starting '**od** C? or £'
by '**od** C £ found,end of inner do clause **C**
od C % found,end of outer do clause **C**'
(b) Replace 'print' wherever it occurs (4 places) by 'putinstring' where the declaration is
'**string** result := " ";
proc putin string = (**char** c) **void**: (result **plusab** c);'

5.4 (a) $1 + 2 + 2^2 + \ldots + 2^{N-1} = 2^N - 1$
(b) **entier** ((M + 1)/2). A binary tree has an odd number of nodes.

5.5 '**proc** terminal = (**int** i) **bool**: **C** true if node is terminal **C**
((left **of** node[i] = 0) **and** (right **of** node[i] = 0));
bool terminalfound := terminal(1);
int i := 1 **C** start from root **C**;
while not terminalfound **do**
if not terminal(left **of** node[i]) **then** i := left **of** node[i]
C look at left branch and repeat if not terminal **C**
elif not terminal(right **of** node[i])
then i := right **of** node[i] **C** same for right branch **C**
else print(operator **of** node[i]); terminalfound := **true**
fi od'

5.6 '**mode tree** = **struct**(**int** left, **char** operator, **int** right);
comment the singular of **trees comment**
trees leftree, rightree;
C assignments to leftree, rightree required here **C**
int length = **upb** leftree, rength = **upb** rightree;
flex[1: length + rength + 1] **tree** newtree **C** to be the result of the required size **C**;
proc copytree = (**tree** ambidextrous, **int** n) **tree**:
C copies into tree, updating pointers by n unless a terminal node **C**

```
begin tree sapling; int lint,rint;
lint := left of ambidextrous;
rint := right of ambidextrous;
sapling := (lint + (lint = 0 | 0 | n), operator of ambidextrous,
      rint + (rint = 0 | 0 | n))
C sapling is the procedure result C
end;
for i to length do newtree[i + 1] := copytree(leftree[i],1) od;
C pointers of leftree increased by one for newtree C
for i to rength do
      newtree[i + length + 1] := copytree(rightree[i],length + 1)
      od;
C pointers of rightree increased by length + 1 C
newtree[1] := (2, "+", length + 2) C the root C'
```

5.7
```
[1:7] tree gum; gum[7] := (nil,"a",nil); gum[6] := (nil,"b",nil);
      gum[5] := (nil,"c",nil); gum[4] := (nil,"d",nil);
      gum[3] := (gum[7], "*" gum[6]);
      gum[2] := (gum[3], "/", gum[5]);
      gum[1] := (gum[2], "+", gum[4])'
```

5.8 a + – b£ (a + b))£ etc.

5.9
```
'proc printtree = (tree expr) void:
      (char ch := operator of expr; int n;
      if not matchar(ch, "+–*/↑", n) then print(ch)
      else print("("); printtree(left of expr); print(ch);
            printtree(right of expr); print(")")
      fi)'
```

5.10 Replace 'if not matchar(ch,"+–*/↑",n)' by
'if (left of expr) is (ref tree(nil))'
in each procedure.
To avoid repeated use of 'ref tree(nil)', one may define
'ref tree null = nil' and then use
'if (left of expr) is null'

Chapter 6

6.1
```
'proc compact = (int m) string:
      (string result; int z := m;
      if m = 0 then "0"
      else result := "";
       while z ≠ 0 do result := repr(z mod 10) + result;
                  z := z ÷ 10 od;
            result fi);
```

```
proc conversion = (real money) string:
    (string pounds, pence; int e = entier money;
    if money < 0.01 then "0.00"
    else
    pence := compact(round(100*(money – e)));
    pounds := compact(e);
    (pounds + ".") + pence fi)'
```

6.2 (a) Yes; (b) No, '3' is not an insertion, (c) Yes; (d) No, "THE END" is not a picture; (e) Yes. Note that 'print ("THE""END")' prints THE"END since "" denotes the quote character. To split literal strings in a format, use a replicator such as "THE"1"END"

6.3 At the start of the 'read record' procedure body insert

```
'int master, check;
proc errorfound = (int p) void:
begin [1:70] char line; read(line);
    print((newpage, "ERROR FOUND – MASTER CODE IS",
    master,newline, "FIRST WRONG CODE LINE IS",
    newline, line));
    to p do read(newline) od;
    goto nextrec end'.
```

Before the end of 'read record' insert

```
'nextrec: skip'.
```

The code number on the first input line is the master, and we compare the codes of successive lines. If any do not match, then the 'error found' procedure is called with parameter the number of remaining lines in the record Thus, replace the first call of 'readf' by

```
'readf(($ 1k 4d $, master, $ 12k 40a 1 $, name,$4d 4y$,check));
if check ≠ master then errorfound(11) fi;
readf(($ 12k 40a 1 $, add1, $ 4d 4y $, check));
if check ≠ master then errorfound(10) fi'
```

and so on, reading the code of each line separately, resetting to the start of the line, comparing check with master and, if necessary, calling 'error found' with parameters successively 9,8, ... 1, as there are fewer lines in the rest of the record to be skipped.

6.4 For simplicity, we assume that all input is on a single line.

```
'begin int n, i, j, isum := 0, jsum := 0;
do
    readf(($ c(" ", "I =", "J =", "RUN", "FINISH", " ") $, n));
case n in read(space), read(i); isum plusab i*i,
    read(j); jsum plusab j*j,
    print((newline, "ISUM =", isum, "JSUM = ", jsum)),
```

```
        goto stop,
        print(("ERROR IN DATA")); goto stop
    esac od;
    stop: skip end'
```

6.5
```
'for i to 5 do
        printf (($ p2k "N" 50k "G(N)" 105k "PAGE" d 2l $, i));
    for j from 200*(i – 1) + 1 by 5 to 200*i do
        printf (($ n(if (j – 1) mod 50 = 0 then 2 else 1 fi)l
    2zd 5x, 5( 5x + d . 5de + 2d 5x) $, (j, g(j), g(j + 1),
    g(j + 2), g(j + 3), g(j + 4)))) od od'
```

6.6 The integer at the start of each month row has to be tested.
Replace the piece of program between
'int nij = nodays in (i + j,year)' and 'printf((f3,m))'
by 'if dates[i + j,0] > nij then
for k from 0 to 6 do m[k] := 0 od else
for k from 0 to 6 do . . . as before . . . od fi'.

6.7
```
'proc niceput = (ref[ ] int m) void:
begin int b = lwb m; int k;
    printf($ + 5d x $);
    for j to upb m do
      k := j + b – 1;
    printf (m[k]);
    if k mod 17 = 0 then print(newline) fi od
end'
```

6.8 Replace 'getf(in1,(w,code1)); getf(in2,(w,code2));' by
'getf(in1, w); getf(in2, w); C fixing format C
if endoffile(in2) then print("FILE 2 EMPTY");
code2 := max
else getf(in2, code2) fi'
and similarly for file in1.
To deal with common records, replace the **else** part of the main do clause by
'elif code2 < code1 then move(in2, out1, code2)
else move(in1, out1, code1);
print(("DUPLICATED RECORD FOLLOWS",
newline)); move(in2, standout, code2) fi'

Chapter 7

7.1
```
'[1:15] ref [] int pascal;
pascal[1] := loc [1:1] int := 1;
```

```
for m from 2 to 15 do
      pascal[m] := loc [1:m] int;
      pascal[m] [1] := pascal[m] [m] := 1;
      for i from 2 to m – 1 do
      C do unit is skipped if m < 3 C
      pascal[m] [i] := pascal[m – 1] [i – 1] + pascal[m – 1] [i]
od od; C triangle is in the multiple 'pascal' C
for k to 15 do
      printf(($ 2l 10x n(k)(4zd) $, pascal[k])) od'
```

7.2 Replace the last line of the procedure body by
'**ref tree** newtree;
newtree := **heap tree** := resultree'

7.3
```
'op + = (ref [,] real a,b) ref [,] real:
begin int m = upb a, n = 2upb a;
      C assume a and b have the same index bounds C
      ref [,] real c;
      c := heap [1:m,1:n] real;
      for i to m do for j to n do
      c[i,j] := a[i,j] + b[i,j] od od;
      c
end'
```

7.4
```
'mode omnibus = union(int,real,char,bool);
omnibus u := . . . .some assignment to u . . .;
case u in (int i): print((i,"INT")),
      (real r): print((r, "REAL")), (char c):
      print((c,"CHAR")),(bool b): print((b,"BOOL"))
esac'
```

7.5
```
'proc makelist = ([] char g) ref cell:
      (ref cell list;
      list := heap cell := (g[upb g],nil);
      for i from upb g – 1 by –1 to 1 do
            list := heap cell := (g[i],list) od;
      list)'
```

7.6
```
'proc count = (cell list,ref int noofatoms) void:
C noofatoms is 0 when procedure is called C
      (case item of list
      in (int i):noof atoms plusab 1,
      (char c): noofatoms plusab 1,
      (ref cell ce):
      C not an atom so call proc recursively C
      count(ce, noofatoms)
out print("NIL SHOULD NOT OCCUR IN ITEM FIELD");
```

```
            goto errorfound
        esac;
        if link of list isnt (ref cell(nil))
        then count(link of list,noofatoms) C counting the rest of
            the list C
        fi)'
```

7.7
```
'bytes b,d; C b is assigned a value C
[1:4] char c;
for i to 4 do c[5 – i] := i elem b od;
d := bytespack(c)'
```

7.8
```
'[1:size, 1:size] char charmaze;
for i to size do for j to size do
charmaze[i,j] := (j elem maze[i] |"." |"*") od od'
```

7.9
```
'mode persons = struct (bool male, int birthyear,children,
grandchildren, bool alive, int graduateyear,retireyear,
real salary, bits details); C last field for masking C
[1:100] persons crowd; int year := C this year C;
C assume crowd defined completely now C
for i to 100 do details of crowd[i] :=
C the following row of bools C
bitspack((male of crowd[i],year – birth year of crowd[i] = 70,
        children of crowd[i] ≠ 0, grandchildren of
        crowd[i] ≠ 0, alive of crowd[i], graduateyear of
        crowd[i] ≠ 0, retireyear of crowd[i] > year,
        salary of crowd[i] > 1000))od;
proc pry = (persons gom) void:
        comment mask with 11001111 to obtain required 8 bits,
        then compare with 11000111 comment
        (if((details of gom)and bin 207) = bin 199
        then C print required information C fi);
for i to 100 do pry( crowd[i]) od'
```

Index