Ministry of Defence
(Procurement Executive)

ALGOL 68-R
Users Guide

London Her Majesty's Stationery Office £1.25 net

Ministry of Defence
(Procurement Executive)

ALGOL 68-R
Users Guide

By P M Woodward and S G Bond
Division of Computing and Software Research
Royal Radar Establishment

Algol 68-R is the name of the computer programming system in practical use at the Royal Radar Establishment and elsewhere. The system is based on the definition of ALGOL 68 sponsored by the International Federation for Information Processing.

London Her Majesty's Stationery Office

First published 1972
Second edition 1974
Third impression 1979

ISBN 0 11 771600 6

Preface

This new edition of the Users' Guide describes Algol 68 in its most widely used form—the Algol 68-R system for large ICL 1900 computers produced by the Computing and Software Research Division of the Royal Radar Establishment. The Guide reflects the experience gained in several years of teaching and practical use.

In the first implementation of a language, particularly a language on the scale of Algol 68, small departures from the original definition* are inevitable. The revised report on Algol 68 is likely to reduce these differences by coming closer to Algol 68-R; it will also introduce new ones such as the use of OD at the end of a DO-clause. But these are relatively minor considerations. The growing popularity of Algol 68 is due largely to its versatility, and especially its data structuring capability, rather than to grammatical minutiae.

The increasing power and complexity of computer software is reflected in the thickness and technicality of its documentation, so that for the occasional programmer the advances can be self-defeating. With this in mind, every attempt has been made to keep the present Guide brief without departing too far from the use of plain English. However, we prefer to start from the very beginning rather than use some other language such as Fortran, Algol 60 or Coral 66 as a starting point, as the best Algol 68 programmers are those who approach the language with a mind open to new possibilities. Brevity has been achieved by concentrating strictly on how to use the language, omitting such matters as library facilities for which separate documentation is available at user installations. The treatment of input and output has been much enlarged in this edition of the Guide, and now includes formatting. The presentation of the syntax rules in Appendix 3 has been further simplified, and we would particularly like to recommend their use for reference purposes.

PMW, SGB

April 1974

* "Report on the Algorithmic Language ALGOL 68" by A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck and C. H. A. Koster, submitted by the ALGOL Working Group to the General Assembly of the International Federation for Information Processing. The report has been reprinted in Numerische Mathematik, volume 14, pp. 79–218 (1969).

Contents

1

Introduction

A program to print the sum of two and two, complete in every respect, is

```
two and two
BEGIN
    INT two = 2 ;
    print(two + two)
END
FINISH
```

The part which does the actual calculation is 'two + two' and all the rest could be described as a formality. However, it is a mistake to dismiss formalities as mere overheads, for most of the challenge in large scale computing problems rests in the *organization* of the work rather than the arithmetic. When the arithmetic is difficult, the library of standard procedures can come to the rescue with proven methods, already programmed. But when the organization is difficult, the formal structure of the language is what counts. In the present Guide, we concentrate on this formal structure from the very outset, in chapters 1 and 2. The later chapters deal with specific topics, all of them necessary, but all subservient to the main concepts underlying the structure of a program.

In this introductory chapter, it is necessary to concentrate largely on terminology, but in defining terms like modes, identifiers, declarations and clauses, the principles of program design already begin to emerge. The first principle is that every item of data, however small or large, must be classified. This is where the real strength of Algol 68 resides, and we therefore take it as the starting point.

1.1 Modes

Programs describe operations to be carried out on *values*, some of which will probably come from a file of data to be read by the program while it is running, while others—such as numerical constants—will be written into the program itself. Values are classified according to type, known in Algol 68 as their *mode*. A selection of simple modes, with examples of actual values, is

	as denoted in program	*input as data to program*	
INT	12	12	an integer
REAL	3.25	3.25	a real number
BOOL	FALSE	F	a boolean value
CHAR	"H"	H	a character

A complete list is given in Appendix 1, along with the corresponding forms of *denotation* to be used within a program.

More complicated objects than simple values require more complicated modes. It is a special feature of Algol 68 that the programmer can construct such modes for himself to describe the entities handled in the program. For example, the mode for a data-structure begins with the word STRUCT and goes on to include the modes of all the constituent parts of the structure. Arrays also have their own modes. Procedures have modes which start with the word PROC and include a full specification of the modes of parameters and result. The mode of a variable begins with the word REF and continues with further mode information to distinguish one type of variable from another. The scheme is so general that it becomes misleading to use the word 'value' to describe an entity which can be handled in Algol 68. We have chosen the word *object* to mean a simple value, variable, structure, array or procedure. Every object has a mode, and every mode describes a class of object.

1.2 Identifiers and declarations

The only values which can be written as denotations (12, "H", etc) are those which are known when the program is written. Other values, and objects such as variables which do not have denotations, must be identified symbolically. Identifiers are constructed arbitrarily by the programmer as any sequence of lower-case letters and digits, starting with a letter. Spaces can be included, but are ignored. Typical identifiers are x, birthday, and algol 68r. (By contrast, modes are always written in upper case, so that INT and int would not be confused.) No identifier can be used until it has been introduced by a *declaration*, either in the user's program or in the system library available to it. The declaration defines the object identified and specifies its mode.

The simplest declarations are those which cause an identifier to stand for an ordinary value with a mode like REAL or INT. For example,

```
REAL h = 0.1 * random
```

introduces the identifier h, with mode REAL, to stand for 0.1 times 'random', which is the identifier for a library procedure which serves up a random number between 0 and 1. It is important to understand that when the program is running, the declaration is *obeyed when it is encountered.* In this instance, a random number is generated, multiplied by 0.1, and the identifier h is defined to be synonymous with the REAL result obtained. Nothing we do with h afterwards can alter the object it stands for. So, in this case, h can be used only as a numerical constant.

Other examples of declarations are

```
REAL grams in 1 ounce = 28.35 ;
REAL four pi = 4.0 * pi ;
CHAR comma = ","
```

These are cases where we know the values we want while we are writing the program, but it may be helpful to use identifiers all the same, so as to avoid writing the actual values every time they are used.

1.3 Variables and assignment

Long calculations are always carried out in stages, starting from initial values and proceeding through intermediate results until final answers are obtained. We need identifiers for intermediate results, and we need a ready method of altering the values to which such identifiers refer. The declarations shown in the previous section are inadequate, because objects of modes such as REAL or INT are constants which cannot change. In fact, no objects can change; they can only be created or destroyed. This leads to the conundrum, 'what type of object is it, which remains constant but can refer to one value at one time and another at another?' The answer is a *working space in the store*. Without itself changing, it can hold different values at different times. In high-level languages, working spaces are given identifiers and are described as *variables*. A consciousness of the fact that variables are receptacles is even more important in Algol 68 than in other languages, as variables are objects in their own right and have their own modes. For example, an integer variable has mode REF INT and is said to 'refer' to the INT object held in it. (The word *refer* is always used in this technical sense.) We must now consider how references are created and their identifiers declared. Just as we can write expressions like 4.0 * pi which deliver numerical values, there are expressions which generate space in store. Space for a REAL, for example, is generated by the expression

```
LOC REAL
```

and similarly for other modes. The word LOC means 'local' and implies that the space is created for local use in the portion of the program we are in at the time (see 2.3). To declare x as a real variable, it should now be evident that we can write

```
REF REAL x = LOC REAL
```

At any time after this declaration, a real value can be *assigned* to x by writing

```
x := 1.5
```

or whatever expression we like on the right-hand side. The assignment can be read as 'x receives 1.5' or 'x now refers to 1.5'. When a new assignment is made, the old value, 1.5, is destroyed.

The object expressed by the right-hand side of an assignment clause must be of the correct mode, which is determined by the mode of the left-hand side. There must be one less REF for the value on the right, ie a REF REAL can only receive a REAL, and similarly for all other modes. For example, a REF REF REAL, which is a receptacle for a reference, can only receive a REF REAL.

1.4 Unitary and serial clauses

The body of a program is composed of steps which are either declarations (chapter 3) or *unitary clauses* (chapter 4). An assignment is one type of unitary clause; other particular examples are

```
read(x)
```

where x is any variable, and

```
print(formula)
```

where formula is any expression which delivers a printable result.

The various steps in a program are separated by semi-colons, and the whole sequence is known as a *serial clause*. A complete program is a serial clause enclosed in brackets (round brackets or BEGIN and END), with an identifier at the top for a title and the word FINISH at the very end, as shown in Example 1.

EXAMPLE 1

```
title
BEGIN
    REF REAL x = LOC REAL ;
    REF REAL y = LOC REAL ;
    read(x) ;
    read(y) ;
    print(x +y)
END
FINISH
```

Notice that there is no semi-colon after the print clause, because there is no clause following it needing to be separated. A semi-colon just before the word END would cause a failure to compile.

The concept of a serial clause as a sequence of steps is extremely important because it applies not only to complete programs but to parts of programs. Chapter 2 will show how a serial clause can be nested inside a single main step of the program, and how it defines the life-times of any identifiers declared within it. A serial clause need not contain any declarations, but if it does, it must *start with a declaration and end with a unitary clause.* The serial clause which forms the main body of a program must always contain at least one declaration, so the first BEGIN must always be followed by a declaration. However, it is not necessary to group all declarations together.

1.5 Notation

The full character set for Algol 68-R includes upper and lower case alphabets and various other symbols. In this section, we take a brief look at how these are used.

Upper case is used for all the fixed language words such as BEGIN, END, IF, THEN and ELSE. These words shape the grammatical structure of a program and their meanings cannot be altered. Upper case is also used for modes and some operators, while lower case is reserved for words made up by the user to suit his own program. For the most part, these are identifiers introduced by declarations, but lower case is also used for labels and field selectors in STRUCT modes.

If the full character set is not available, a restricted set with only one alphabet can be used. The true upper-case words must then be distinguished by enclosure within primes,

```
full set        REAL a ;
restricted set  'REAL' A ;
```

The full set is very much easier to read, and is used throughout this Guide.

Various symbols are used for bracketing; as will be seen from the list, the same symbols are sometimes used for opening and closing.

(. . .) Ordinary round brackets are used in formulae and around pieces of program. An alternative pair is BEGIN . . . END, customarily used for vertical grouping, as in Example 1.

[. . .] Always associated with array indexing.

'. . .' In addition to their use for contending with a restricted character set, primes can be used to enclose various operator symbols so as to increase the effective number of combinations available. For example '/' is different from / (Appendix 2).

". . ." Used for character denotations and around sequences of characters.

{
. . .} Program comment, enclosed in this way, can be inserted in any reasonable place in a program. Alternatives to curly brackets are

COMMENT . . . COMMENT
C . . . C

Note that curly brackets are also used in this Guide for a purpose outside the language' indicating optional elements of the grammar.

$ . . . $ Used as the opening and closing symbols for format denotations.

The principal punctuation marks are the semi-colon and the comma. Although they have different meanings, both are used as *separators*, not terminators. For example, a bracketed list of items is (a, b, c) not (a, b, c,). The same principle applies to the semi-colons in a serial clause.

The only strict rule of layout is that there must be no interruption of an upper case language word by spaces or new lines, nor must two language words be run together. Nevertheless, layout is supremely important for program legibility. Trouble taken over indenting and alignment is amply repaid in the avoidance of errors.

1.6 Glossary of terms

Declaration	A step which introduces an identifier to stand for a given object
Identifier	Sequence of lower-case letters and digits, starting with a letter
Mode	The class of an object, eg INT for an integer, REF INT for an integer variable
Object	An entity which has a mode
Program	A serial clause, enclosed in brackets, with a title at the top and FINISH at the bottom. The serial clause must start with a declaration and end with a unitary clause
Serial clause	A sequence of unitary clauses, or a mixed sequence of declarations and unitary clauses starting with a declaration and ending with a unitary clause
Step	A declaration or a unitary clause
Unitary clause	Describes some action to be taken

2

Structure of a Program

2.1 Sequencing and nesting

In low-level language, each step is a rudimentary operation and the program designer must necessarily adopt a highly sequential attitude of mind—do this, now this, now this. In Algol 68, the steps can be less rudimentary, and any idea that each step will occupy one single line of text should be discarded. Each step is separated from the next by a semi-colon. The semi-colon is an indication that what has gone before must be completed before the next step is taken. It has nothing whatever to do with new lines or cards.

Without some kind of structure beyond simple sequencing, a long program may be difficult to write, for it is easy to miss the wood for the trees. Structure can be built into an Algol program by thinking first what each major step ought to do. If it is a complicated step, sequences of smaller steps can be incorporated *inside it*, and even smaller steps inside those, to whatever extent may be necessary.

The grammar of the language decides where such nesting can occur. For example, one type of construction is the 'conditional'

IF *some condition is true*
THEN *carry out one sequence of steps*
ELSE *carry out another sequence of steps*
FI

This provides us with the opportunity to obey a serial clause within a single step of the program. In due course, we shall see that there are several other ways in which this can be done.

To illustrate program structure we now take a definite problem and discuss the ways in which it might be programmed. The problem is this:

> *An examination result A, B+, B− or C is to be read by the program, and converted into an integer 1, 2, 3 or 4 respectively. The result is to be left in a variable. Errors in the data need not be detected.*

The first solution is shown in Example 2, using conditionals of the type

IF *a condition is true*
THEN *carry out one or more steps*
FI

This construction does nothing at all if the condition is false. Example 2 includes comment to show which are the main steps at the outermost level, and is written as though it were part of a larger program, that is to say without any 'top and tail'.

EXAMPLE 2

```
{declaration}    REF INT class = LOC INT {creates space to hold the integer result} ;
{declaration}    REF CHAR mark = LOC CHAR ;
{unitary clause} read(mark) ;
{unitary clause} IF mark = "A"
                 THEN class := 1
                 FI ;
{unitary clause} IF mark = "B"
                 THEN
                     class := 2{provisionally} ;
                     read(mark) ;
                     IF mark = "-"
                     THEN class := 3
                     FI
                 FI ;
{unitary clause} IF mark = "C"
                 THEN class := 4
                 FI
```

The important thing to see is that the conditionals count as single steps, even though there may be sequences inside them. After testing for B, for instance, the action to be taken is written as a serial clause made up of three separate unitary clauses, the third of which is itself a conditional.

In this problem, there is really no need to consider the three cases A, B and C separately. By using conditionals with an ELSE part the last three steps can be rolled into one, as shown at Example 3.

EXAMPLE 3

```
{unitary clause} IF mark = "A"
                 THEN class := 1
                 ELSE IF mark = "B"
                       THEN read(mark) ;
                            IF mark = "+"
                            THEN class := 2
                            ELSE class := 3
                            FI
                       ELSE class := 4
                       FI
                 FI
```

The indentations help the eye to pick up the nested structure, and some such convention should always be observed when preparing Algol text.

2.2 Results delivered by unitary and serial clauses

Most problems have two closely interwoven aspects, arithmetical calculation and data handling. The example discussed in 2.1 contains no arithmetical element; it is merely a transformation of the way an examination result is classified. To perform a calculation, and get a result, we write an expression such as a + b + c or x * y * z. (Asterisk is the symbol for multiplication.) Details of

how expressions are written are given in chapter 4. Here we are concerned with where they fit into the scheme of things.

An expression is a type of unitary clause which delivers a result, and from a purely grammatical point of view it can be written wherever a unitary clause is allowed. For example, the grammar rule for assignment to a variable (x, say) is

x := *unitary clause*

and this is where expressions in practical programs most often appear. Thus,

```
i := i + 1
```

adds 1 to the integer held in the variable i. In Example 3, the simple assignment

```
class := 3
```

does actually contain a unitary clause, the rather simple expression 3. By exploiting the idea that the right-hand side of an assignment can be any unitary clause which delivers a result, this example can be much improved. An experienced programmer would not write four separate assignments, but would say "I have to obtain a result for assignment to 'class'. Let me think of a unitary clause which will deliver the required number under all conditions." Example 4 is the outcome—shorter and clearer.

EXAMPLE 4

```
class := IF mark = "A" THEN 1
         ELSE IF mark = "B"
              THEN read(mark) ;
                   IF mark = "+" THEN 2 ELSE 3 FI
              ELSE 4
              FI
         FI
```

There is another new point in this example, the concept that a *serial* clause can deliver a result. The clause concerned is

```
read(mark) ; IF mark = "+" THEN 2 ELSE 3 FI
```

There are two steps here, a preliminary step and an expression. The rule is that *the result of a serial clause is the result delivered by the last step obeyed in it.* Serial clauses delivering results are useful whenever we need to take a few preliminary steps before we are ready to write the unitary clause which actually does the calculation.

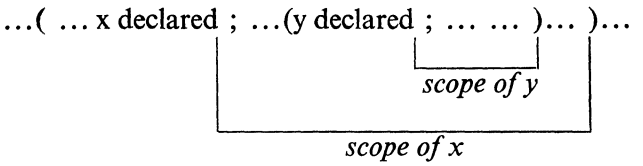
Although a serial clause can always appear inside a conditional, as shown in Example 4, there are some places where the grammar of the language debars the use of a serial clause. For example, the clause on the right-hand side of an assignment must be unitary (for otherwise the first semi-colon in the serial clause would look like the end of the assignment as a whole). This difficulty is easily overcome. By enclosing a serial clause in brackets, it is made unitary. Ordinary round brackets may be used, or BEGIN and END if preferred. An assignment of the form

```
x := BEGIN ... ; ... ; ... END
     └──────────────────┘
       serial clause
```

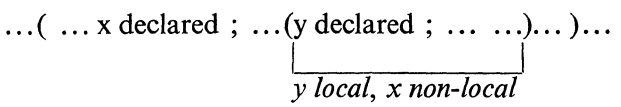
therefore satisfies the grammar and does what we may require.

2.3 Scopes

A program is made easier to follow if the identifiers are chosen as meaningful names for the objects they represent in the problem. Very often it is found that a particular meaning applies to only one small part of the problem, and we would like to introduce an identifier limited to this part only. This is exactly what happens when a declaration is placed in a nested serial clause. The identifier is then valid for the remainder of that serial clause only, including smaller clauses nested within it, as illustrated below. In the diagram, brackets are used to indicate nested serial clauses, which may in fact be delimited by brackets or by words like THEN and ELSE.



For descriptive purposes, the words *local* and *non-local* are used to distinguish identifiers which are newly declared from those which have been declared 'outside'.



Identifiers declared at the outermost level of a program can be used throughout the whole program and are described as *global.*

When a declaration creates a new variable, as in

```
REF REAL x = LOC REAL
```

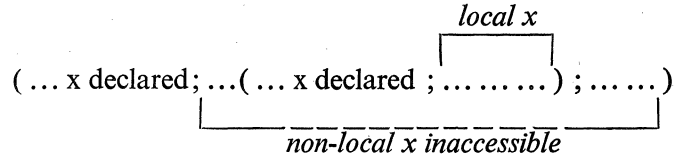
the question of scope arises in two ways. The identifier passes out of scope at the end of the serial clause, but what about the corresponding storage space? In this instance, as we might have anticipated, it has the same scope as the identifier x. However, we shall see later on that a local generator can be used on its own, and its scope is then the same as that of the most local identifier which has been declared, no matter what it might happen to be.

To illustrate scoping, let us now modify Example 4. In the solution already given, the variable 'mark' is used for two different purposes. It is first used as a receptacle for A, B or C, and if the mark was B it is used again for the next character (+ or −). If we wish to preserve A, B or C in mark, we would need a separate variable for the sign, and this can be brought into existence temporarily, as shown in Example 5.

EXAMPLE 5

```
class := IF mark = "A" THEN 1
         ELSE IF mark = "B"
              THEN REF CHAR sign = LOC CHAR ;
                   read(sign) ;
                   IF sign = "+" THEN 2 ELSE 3 FI
                   {the variable 'sign' now disappears}
              ELSE 4
              FI
         FI
```

By an oversight, or perhaps deliberately, an identifier may be declared locally when the same identifier had already been declared non-locally. In such cases, the non-local object continues to exist but becomes temporarily inaccessible because the identifier is always taken to have its local meaning.



As an example of this type of clash, consider the effect obtained by

```
BEGIN INT k = 1 ;
      BEGIN INT k = 2 ;
            print(k)
      END ;
      print(k)
END
```

This program prints 2, then 1.

It is an error to declare the same identifier twice over at the same level, even though the modes may be quite different.

2.4 Labelling

A label is a place marker, and has the same form as an identifier. It does not stand for an object, and it does not have to be declared. It is simply written in front of a unitary clause, with a colon for separation, like 'part 1' and 'part 2' below.

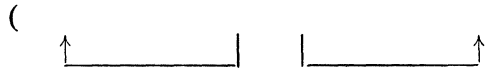
```
BEGIN    --- ;
         --- ;
part 1:  --- ;     no declarations allowed after
         --- ;     the first label, except within
part 2:  ---       their own serial clauses
END
```

Declarations can never be labelled, and no unitary clause can be labelled if there are declarations at the same level coming later in the serial clause.

The purpose of labelling is to make jumps possible. The unitary clause

GOTO label

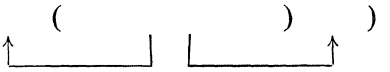
disturbs the natural sequence in which the steps of a program are obeyed, for the next step obeyed after the GOTO is the one marked with the label specified. The jump may be either forwards or backwards in the same serial clause,



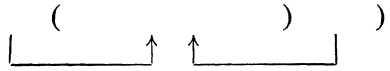
but must not be used to skip past declarations. In the above diagram, there can be no declarations

after the first arrow-head, which represents the position of a label. It is also permitted to jump to a label set at a place outside the serial clause containing the GOTO, but never to a label at an inner level.

right



wrong



The only grammatical contexts in which labels can occur are the places where they are set, and after the word GOTO. One may, of course, go to the same label from any number of different GOTO clauses. It does not matter if the label happens to be the same as a declared identifier, for the two are kept quite distinct.

An abundance of jumps is usually a sign of a poorly structured program, and obscures an understanding of how the program works. It is usually found that GOTO can be avoided by using suitably nested conditionals (4.3) or loops (4.4). The solution to any problem may at first seem to present itself as a plain sequence of tiny steps, which we may then be tempted to modify by including jumps from one place to another. It is nearly always better to proceed by rethinking the problem in terms of fewer, larger steps. It may make little difference to run-time efficiency, but it encourages the 'top-down' attitude of mind so essential to mastery of really complex problems.

3

Declarations

3.1 Identity declarations

A declaration identifies an object. All the declarations in the first two chapters have been written in a form which can be described grammatically as

mode identifier = unitary clause

For example, the declaration at the beginning of chapter 1 was

```
INT two = 2
```

and the correspondence with the grammatical rule should be clear. The mode is INT, the identifier is 'two' and the unitary clause is 2. Similarly, in the declaration

```
REF CHAR sign = LOC CHAR
```

the mode is REF CHAR, the identifier is 'sign' and the unitary clause is the local generator LOC CHAR. Declarations written according to this grammar rule are known as *identity declarations.* Every declaration for a new identifier can be written as an identity declaration, but there is a specially shortened form of declaration which can be used for variables, described below.

3.2 Short declaration of variables

To avoid repeating the mode information when declaring a variable, a shortened type of declaration is permitted. This is formed by omitting *both* of the portions shown below in curly brackets:

{REF} *mode identifier* {= LOC *mode*}

The declaration of 'sign' shown in the introductory paragraph can therefore be shortened to

```
CHAR sign ;
```

The semi-colon is not part of the declaration, but is included here simply to emphasize that there is no more to the declaration. A similar discretion is used throughout the Guide.

The short declaration for a variable will be described as a *variable declaration.* Convenient as it is, it unfortunately results in disguising the true mode of the identifier, for when we see

```
REAL x ;
```

we must consciously remember that the *mode of x is REF REAL, not REAL.* This point can hardly be emphasized too strongly. The abbreviated form is justified by history, as it has been the standard way of declaring variables since the days of Algol 60.

3.3 Initial assignment to a variable

Until an assignment has been made, the value held in a variable is indeterminate and should not be used. An assignment is often the very first step to be taken after a variable declaration, and the following extra abbreviation may then be used:

full form	*abbreviated form*
REAL x ; x := 1.0	REAL x := 1.0

This lessens the danger of forgetting the true mode of the identifier; the assignment symbol serves to remind us that the left-hand side must be a reference, for one cannot assign to a REAL. Compare these:

```
REAL pi = 3.142 ;
REAL stress := 13.8 ;
```

The first is an identity; therefore the mode of pi is REAL. The second declares and initializes a variable; therefore the mode of 'stress' is REF REAL.

3.4 Compounded declarations

If several identifiers are to be declared with the same mode, another abbreviation is possible.

separately	*together*
REAL x ; REAL y ;	REAL x, y ;
INT a := 0 ; INT b := 1 ;	INT a := 0, b := 1 ;
INT s = a + b ; INT d = a − b ;	INT s = a + b, d = a − b ;

Identities must not be mixed with variable declarations (whether initialized or not), but compounded variable declarations need not all be initialized:

INT c ; INT e := 6 ;	INT c, e := 6 ;
INT f ; INT k = 365 ;	*no combination possible*

As an exercise, the reader is advised to write down the modes of all the identifiers declared in this section before looking at the answers below:

x, y	REF REAL
a, b, c, e, f	REF INT
s, d, k	INT

3.5 A note on right-hand sides

The right-hand side of an identity declaration, like that of an initialized variable declaration, is a unitary clause capable of delivering a result of the required mode. When we compare the two kinds of declaration:

REAL t = *unitary clause delivering a REAL*
REAL z := *unitary clause delivering a REAL*

we see that the requirements for the right-hand sides are exactly the same even though the modes of t and z are different. It is useful to remember that the mode of object required in both cases is the mode actually written on the left.

4

Unitary Clauses

In this chapter, all the main types of unitary clause are described, starting with expressions (4.1) and assignments (4.2). Apart from conditionals (4.3), the remaining simple type of unitary clause is the loop (4.4), which delivers no result. Jumps, which can also be treated as unitary clauses, have already been discussed in chapter 2.

4.1 Expressions

The three types of expression are *primaries* (ie single terms), *formulae* consisting of primaries and operators, and *generators*, which create references. Local generators have been briefly described in 1.3 and 2.3, whilst global generators are treated in a later chapter. A generator simply creates a variable without attaching an identifier to it, and the important point is that it creates a *new* reference every time it is obeyed.

4.1.1 Primaries and the use of brackets

A primary is very often an identifier symbolizing an object, or an object expressed as a denotation. However, there are several other types of primary. In the list below, the primaries in italics are to be described in later chapters.

list of primaries	*example*
denotation	1.35 & 6
identifier	x
procedure call	cos(x)
array element or slice	a[i]
structure field	age OF man
conditional	IF u THEN a ELSE b FI
bracketed serial clause	(read(i) ; i + 3)

We have now reached the 'bottom of the grammar', for the primary is the smallest grammatical class in the language. It is the simplest type of expression and can be used by itself in any context where an expression is specified. Similarly, an expression is one type of unitary clause, and a unitary clause with no other steps before or after it can be classed as a one-step serial clause. It follows that a primary can always stand by itself in a context which demands one of the following:

primary, expression, unitary clause, serial clause

Small units can always play the part of larger units.

Large units can also be made to serve as small units, for this is the whole purpose of brackets. (A collection of knives and forks tied together with string becomes one small unit in an auction sale because of the string.) We have seen in chapter 2 that a serial clause becomes unitary if placed in brackets. In reality, the brackets are more powerful, because every one of the above grammatical items, including the serial clause, becomes a *primary* when put in brackets.

4.1.2 Formulae

A formula is a mixture of primaries, and operators such as plus (+), minus (−), times (*) and divide (/). For example, i + 2 is a formula with i and 2 as the *operands* of +. Operators act on primaries or on the results of other operations, as in the formula x + y * z where the operands of + are x and y * z. The y is taken with the multiplication operator because multiplication binds more tightly than addition, as in ordinary mathematics. To overcome this natural binding, brackets must be used. To obtain the product of x + y and z, for instance, we must write

```
(x + y) * z
```

By making x + y into a primary, the brackets force the required grouping of terms.

Operators which take two operands are known as *dyadic*, and all such operators have a priority number which governs the strength of binding. The larger the priority number, the tighter is the binding. Some examples are:

operators	*priority*
+ and −	6
* and /	7
↑ (raised to the power)	8

Dyadic operations of equal priority are performed in their written order, left to right. Thus a/b * c means (a/b) * c.

Some operators take only a single operand; they are known as *monadic* and bind to their operand more tightly than any dyadic operator. This rule usually accords with understood mathematical usage, but may occasionally trip the unwary scientific programmer. For example,

```
−2↑2 means (−2) ↑ 2
```

giving the perhaps unexpected result +4. Monadic operators can be applied in succession when required, so that

```
− ROUND ABS x
```

takes the absolute value (modulus) of x, rounds it to the nearest integer and then makes it negative. Some operators, of which + and − are examples, can be used either dyadically or monadically, and the tightness of binding is governed accordingly.

Beginners are apt to confuse the uses of ↑ and &. The vertical arrow is an operator, and therefore

```
10.0 ↑ 6
```

is a *formula* which will be evaluated at run-time to give the result one million. However, it is less extravagant to write such a constant as the *primary*

```
1.0 & 6
```

which is the denotation for 1.0 times ten to the power 6. This is of course a million, but there is no run-time calculation. The ampersand is as much a part of the denotation as the decimal point, and cannot be used in any other context (so n & 6 is wrong).

A complete list of Algol 68-R operators is given in Appendix 2, where it will be noticed that operators taking numerical operands and delivering numerical results are in the minority. A formula need not give a numerical result. For instance, the right-hand side of the identity declaration

```
BOOL internal = a < x AND x < b
```

is a boolean formula, in which < binds more tightly than AND. If x lies between a and b, the value of 'internal' would be TRUE.

For economy in the use of symbols, the identity sign used in declarations is shared with the equals operator, and a moment's recollection of grammar may be necessary to interpret the declaration

```
BOOL equal = n = m
```

as meaning "define the identifier 'equal' as an object of mode BOOL, and make its value be that of the boolean formula n = m, ie TRUE if n and m are equal, otherwise FALSE". The equals operator can test for equality of integers *but not reals*. In Algol 68-R there is no way of finding out whether two real numbers are precisely equal, for such equality is likely to be purely accidental bearing in mind the problems of representing real values in a finite machine.

4.1.3 Modes of operands and results of operations

The modes of operands are important because they determine what an operator will actually do. For example, ABS applied to a REAL delivers its modulus, but applied to a CHAR it delivers an integer in the range 0 to 63. We must always take care to see that the operator we think we need is actually defined for the given operands—otherwise the program will fail to compile. This would happen if, for instance, we applied the operator ARG to a REAL, though not if we applied it to a COMPLEX.

Another item of mode information which can be important is that of the result of an operation. Unless we are aware of this mode, we cannot be sure that the result will work as the operand of another operator. For example, the boolean formula a > b > c is wrong, because a > b delivers a BOOL and the operator > is undefined for a boolean operand. Even when no error occurs, an unexpected result can be obtained. Users have been surprised that the formula

```
2 ↑ −2
```

gives the answer 0, for in mathematics 2 to the power −2 is a quarter. But Appendix 2 points out that an INT raised to the power of an INT gives an INT result, and the nearest INT to a quarter is 0.

It is not always possible, nor even desirable, that we should try to present an operator with operands of *exactly* the modes it asks for. It is allowed that operands should be *references* to objects of the correct modes. Suppose that n is an integer variable, mode REF INT, and consider what happens when we write the formula n + 2. Appendix 2 says that + can be used between INT, REAL or COMPLEX objects in any combination, but says nothing about REF INT. The operator will then automatically *dereference* the variable n; in other words it will take its first

operand to be the INT to which n refers. This automatic mode-change is known as a *coercion*, and it only takes place when demanded by the context. The only forms of coercion applied to operands are dereferencing, when necessary, and 'deproceduring' which is described in chapter 9.

The list of standard operators (Appendix 2, Arithmetical assignments) includes PLUS, MINUS, TIMES and DIV. The main purpose of these is to abbreviate unitary clauses like

```
x := x + 1.0
```

which can be expressed simply (and efficiently) as

```
x PLUS 1.0
```

The action of PLUS is very different from that of +. Its priority is different and it requires a reference mode for its first operand. Assuming that x in the above example has mode REF REAL, no coercion is applied to x.

4.2 Assignments

The purpose of an assignment is to store an object in a variable, where it will remain until a further assignment is made or the variable goes out of scope. The mode of the variable determines the kind of object it can hold, and no other kind will do. A variable with mode REF REAL can only hold a REAL value, and similarly for all other modes. The grammar for assignment is

expression := *unitary clause*

Nearly always, the expression we use on the left is simply a primary, though we shall encounter more general cases in due course. In any event, the expression must deliver a reference. A more important matter is to ensure that the right-hand side delivers an object of the correct mode. The mode required is the mode of the left-hand side, less one REF, and various coercions are applied, if necessary. The full rules of coercion can be quite complicated, and a later chapter deals with the subject. The most important forms applied to the right-hand side of an assignment are *dereferencing* and *widening*. Dereferencing has already been met (4.1.3, operands in formulae), but widening is new.

A simple example shows both coercions:

```
INT i := 4 ;
REAL x ;
x := i
```

The mode of x is REF REAL, and the only object x can receive is a REAL. But i has mode REF INT which is unsuitable. It is therefore dereferenced. The mode is now INT, which is still unsuitable, so it is 'widened' to REAL, which converts the integer value 4 to the real value 4.0. The assignment can now take place. REAL is a wider class of object than INT, and similarly COMPLEX is wider than REAL. Other forms of widening are listed in 9.1.3.

It is important to know what coercions *cannot* be performed. For example, there is no such coercion as 'narrowing'. A real cannot be converted to an integer other than by using an operator such as ROUND, which delivers the integer nearest to its real operand:

```
REAL x := 3.8 ;  INT i := ROUND x
```

In the second assignment, the operator ROUND dereferences x and delivers the integer 4, suitable for assignment without further coercion. The following is *wrong*

```
i := 0.0
```

because 0.0 is a real denotation, and REAL cannot be coerced to INT.

The left-hand side of an assignment need not be a primary. It can be an expression which *delivers* a variable, such as

```
IF random > 0.5 THEN x ELSE y FI := 1.0
```

It is assumed here that both x and y are real variables. They are not dereferenced, for dereferencing *never* occurs on the left-hand side of an assignment.

4.2.1 Result of an assignment clause

When an assignment has taken place, the left-hand side variable is delivered as the result of the clause. A few examples will show how this fact can be used. The simplest to understand, as no coercions are entailed, is

```
REAL x ;
REF REAL xx ;
xx := x := 1.0
      |_____|
   unitary clause
```

As might be guessed, this assigns 1.0 to x, and x to xx. The assignment x := 1.0 is the unitary clause which acts as the right-hand side for the assignment to xx (see grammar rule at beginning of section 4.2). Notice that the effect of the grammar is to make multiple assignments work from right to left. The variable xx is twice removed from 1.0; it refers to x, which in turn refers to 1.0.

A similar situation is shown in

```
REF REAL xx ;
xx := LOC REAL := 1.0
```

where the LOC REAL, in which 1.0 has been placed, is assigned to xx. The relation between xx and 1.0 is exactly the same as in the previous example; xx refers to an object (which this time has no identifier) which in turn refers to 1.0.

Another example in which no coercions are needed is

```
REF REAL x = LOC REAL := 0.0
             |___________|
              unitary clause
```

which satisfies the grammar rule for an identity declaration (see first display in 3.1). This generates a working space, puts 0.0 into it, and then attaches the identifier x to the working space or variable. The example is simply the unabbreviated form of the initialized variable declaration

```
REAL x := 0.0
```

Finally, we give overleaf an example in which coercion does take place.

```
REAL x, y ;
x := y := 3.6
```

This is the commonest type of multiple assignment, used for assigning the same object to two different variables. It is a good idea to know how it works! First 3.6 is assigned to y, and y is delivered. Then y is dereferenced to make it suitable for assignment to x, which gives 3.6. This is now assigned to x, and finally the clause as a whole delivers x as its result.

The result of a clause may not always be used. It is most often an assignment clause in which this happens, as we may have no further *immediate* use for the variable. When we write

```
.... ;
x := 1.4 ;
y := 1.6
```

the result of the first assignment is x, but after the semi-colon x is no longer 'in hand'. Failure to use the result of the clause does no harm, for x itself has not been destroyed.

4.3 Conditionals

This section deals with two distinct forms of conditional, the IF construction and the CASE construction. The first decides between two courses of action, depending on whether a boolean value is TRUE or FALSE. The second decides between any number of courses, according to the value of an integer.

4.3.1 The IF construction

The grammar rule for the use of IF is

IF *serial clause* THEN *serial clause* {ELSE *serial clause*} FI

Curly brackets denote a part which can be omitted. When the IF construction is used as a primary in an expression, a result must be delivered in both cases, so the ELSE part must then be present, but when the conditional stands by itself as a step in a serial clause, the ELSE part is optional. Example 2 on page 7 illustrates this.

The clause between IF and THEN must deliver a boolean result. For example,

IF read(x) ; x > 1.0 OR x < 0.0 THEN *etc*

shows a serial clause in which the last step is a boolean formula. The fact that boolean formulae usually occur in conditionals tends to make one lose sight of other possibilities. Suppose, for example, that we wish to assign TRUE or FALSE to a boolean variable b, according to whether i is equal to j or not. It is tempting at first to write

b := IF i = j THEN TRUE ELSE FALSE FI

But this is only a roundabout way of saying

b := i = j

A not uncommon slip is to translate directly from English and write (mistakenly)

IF i = 2 OR 3 THEN *etc*

To see what is wrong, remember that if the first operand of OR has mode BOOL, so must the second.

One often finds it necessary to nest conditionals within conditionals, particularly between ELSE and FI, as happened in Examples 4 and 5 in chapter 2. An abbreviation for ELSE IF ... FI FI is ELSF . . FI, as shown schematically below.

in full
```
IF u THEN a ELSE
IF v THEN b ELSE
IF w THEN c ELSE d FI FI FI
```

abbreviated
```
IF u THEN a ELSF
   v THEN b ELSF
   w THEN c ELSE d
FI
```

For each ELSE IF elided into ELSF, one closing FI is omitted.

There is an extremely concise contracted notation for conditionals, especially useful in formulae where IF, THEN, ELSE and FI seem disproportionately clumsy.

full	*contracted*
IF ... THEN ... FI	(... | ...)
IF ... THEN ... ELSE ... FI	(... | ... | ...)
ELSF	|:

The expression on the right of the assignment to 'class' in Example 4 of 2.2 may now be written on one line:

```
(mark = "A"|1|: mark = "B"|read(mark) ; (mark = "+"|2|3)|4)
```

It is left to the reader to decide whether this is carrying things too far.

4.3.2 The CASE construction and SKIP

The grammar for using the CASE construction is

CASE *serial clause* IN
serial clause, serial clause, ..., serial clause
{OUT *serial clause*} ESAC

where the curly brackets indicate parts which can optionally be omitted. When serial rather than unitary clauses are used after IN, care should be taken with the layout of the program, as in this context the commas are stronger separators than semi-colons. Alternatively any serial clauses may be placed in brackets, for this is always an option.

The clause after the word CASE must deliver an integer. This integer selects which of the clauses in the list after IN is to be obeyed (1 for the first, 2 for the second etc). If an OUT part is included, its clause is obeyed when the selecting integer is out of bounds, ie less than 1 or greater than the number of clauses between IN and OUT. If no OUT part is included, the program will fault when the selecting integer is out of bounds.

The CASE construction can be used as a primary or as a unitary clause standing on its own. As an example of the former,

```
CASE n IN 31, IF leap year THEN 29 ELSE 28 FI,
      31, 30, 31, 30, 31, 31, 30, 31, 30, 31
ESAC
```

delivers the number of days in the nth month. When the CASE construction is used for listing courses of action (such as jumps or procedure calls), it may happen that some of the cases require no action at all to be taken. The dummy primary SKIP can then be used to fill in the blanks,

```
CASE wine number
IN lay down, drink, SKIP, give away ESAC
```

The use of SKIP as the default action between OUT and ESAC is often useful. SKIP does nothing and delivers nothing.

The contracted notation for the CASE construction is similar to that for IF. CASE and ESAC are written as round brackets, IN and OUT as vertical strokes, as in

```
(n | 31, (leap year|29|28), 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
```

4.4 Loops

Loops are expressed by the DO construction, which causes a given unitary clause to be obeyed over and over again. There is a wide choice of forms for this construction, as may be seen from its grammatical form

{FOR *identifier*}
{FROM *unitary clause*}{BY *unitary clause*}{TO *unitary clause*}
{WHILE *serial clause*}
DO *unitary clause*

Parts in curly brackets are optional. The clause which is repeated, known as the 'controlled clause', is the one after the word DO. The preambles are all concerned with controlling the repetition. Broadly, there are two criteria for terminating repetition, one of which is to carry on until a given condition ceases to be true, whilst the other is to repeat a given number of times. The DO construction without any of the optional parts causes indefinite repetition, leaving the programmer to provide his own escape with a suitable jump, as shown in the following example which puts the factorial of n into f.

```
INT i := f := 1 ;
DO IF n >= i
    THEN f TIMES i ; i PLUS 1
    ELSE GOTO label outside
    FI
```

This is clumsy, and can be avoided by using WHILE,

```
WHILE n >= i DO (f TIMES i ; i PLUS 1)
```

(Note the use of brackets to make the controlled clause unitary.) The word WHILE expects a serial clause delivering a boolean result. This clause is obeyed before each repetition, and if its result is FALSE, repetition stops immediately. If the result is false at the outset, the controlled clause is not obeyed at all.

The other way of controlling repetition is to specify in advance how many times the controlled clause is to be obeyed. For the factorial we could have written

```
TO n DO (f TIMES i ; i PLUS 1)
```

The word TO expects a unitary clause delivering an integer result. This is evaluated *once only at the outset* and is taken as the number of repeats required.

A further facility is a loop counter, shown in this version:

```
INT f := 1 ;
FOR counter TO n DO f TIMES counter
```

The counter must be written as an identifier, and its introduction after FOR is treated as its declaration. Its normal values are 1 when the controlled clause is being obeyed the first time, 2 for the second time, and so on. Although the value changes, the loop counter has mode INT, not REF INT, which debars any artificial assignments being made. Its scope is limited to the controlled clause. If the same identifier had, by chance, been declared previously in some other connection, the non-local would of course be masked, and unusable anywhere in the DO clause.

The construction 'FOR counter TO n' is really a default version of

```
FOR counter FROM initial value BY step size TO final value
```

and a few examples may be needed to see how this operates.

	successive value of counter
FROM 2 BY 3 TO 11	2, 5, 8, 11
BY 4 TO 12	1, 5, 9
BY −1 TO −4	1, 0, −1, −2, −3, −4

The initial value, step size and final value can all be unitary clauses delivering integers and these are evaluated once only at the outset. Any or all of these parts can be omitted, and the default values are then taken from

```
FROM 1 BY 1 TO infinity
```

If the TO and WHILE parts are both included, the repetition may stop for either reason, whichever applies first.

Nested loops are very common; as an example, the following clause prints the integers 1 to 100 with five on each line.

```
FOR i FROM 0 TO 19 DO
    (FOR j TO 5 DO print(5 * i + j) ; print(newline))
```

The normal rules of scoping permit the use of i non-locally in the inner controlled clause.

Finally, it must be emphasized that a loop delivers no result. When it has been completed, the counter passes out of scope and cannot be used to find out how many repetitions actually occurred.

4.5 Grammatical summary

The principal types of unitary clause are listed below. Parts in curly brackets need not be present. Items not fully discussed in chapters 1–4 are excluded.

Expression—one of the following three kinds:
primary, ie
denotation, or
identifier, or
conditional*, or
BEGIN *serial clause* END
formula of primaries and operators
generator, of form LOC *mode*

Assignment:
expression := *unitary clause*

Jump:
GOTO *label*

Loop:
{FOR *identifier*}
{FROM *unitary clause*}{BY *unitary clause*}{TO *unitary clause*}
{WHILE *serial clause*}
DO *unitary clause*

* Conditional:
IF *serial clause* THEN *serial clause* {ELSE *serial clause*} FI, or
CASE *serial clause* IN *serial clause*, ..., *serial clause* {OUT *serial clause*} ESAC

5

Data Structuring—Arrays

Algol 68 is one of the few languages in which complicated data structures can be freely defined and then manipulated as simply as integers or reals. For instance, if 'data' is an identifier declared to stand for a group of items and x is a variable of the appropriate mode (REF to the group of items), the assignment

```
x := data
```

will put the entire data-structure into x. The placing of the individual items can be forgotten. It is important not to lose sight of this advantage, and to structure the data in a problem so as to exploit it fully.

Basically there are two types of grouping for individual items of data. A group can be an *array* or a *structure*. From now onward, the word 'structure' is used in a technical sense, though arrays and structures are both instances of data-structuring in the colloquial sense of the phrase. An array is a set of objects all of the same mode; these objects are described as *elements* of the array, and are selected by means of a numerical *index*. For example, if 'vector' is a row (ie one-dimensional array) of reals, vector [i] is the ith element. Similarly, if 'matrix' is a two-dimensional array, matrix [i, j] is the element with coordinates i, j. Indexes are integers, and can be written as variables or unitary clauses which deliver integer results. Selection of an element from an array is therefore a dynamic process; the position in the array is computed while the program is running.

By contrast, a structure is a collection of items which need not all have the same mode; the individual items are known as fields of the structure, and are named with *field selectors*. As an example, if 'person' is a structure which includes a field called 'age', this particular field is selected by writing 'age OF person'. The selection mechanism is not dynamic, as the field selectors are written in the program explicitly and cannot be computed. This makes a structure more efficient than an array, and if its static nature is not a handicap, it is preferable. It will often be found that data is best organized partly in one way and partly in the other, for structures can contain array fields, and arrays can have structured elements.

5.1 Array declarations and assignment

The same principles govern array declarations as have already been described for simple constants and variables. Constant arrays and array variables can be declared by means of identity declarations, and there is once again an abbreviated form of variable declaration. The only new aspects are the modes for arrays, and the manner in which actual values of arrays are displayed when written in the program.

The mode of an array is the mode of its elements preceded by a pair of square brackets [ ] possibly containing commas. The brackets are empty for simple rows, but the space can be subdivided by commas to indicate more than one dimension. Thus [,] implies 2 dimensions, [, ,] implies 3 and so on. For instance, the mode of a row of reals is

```
[ ]REAL
```

and the mode for a two-dimensional array of integers is

```
[ , ]INT
```

The elements of an array can be of any mode except another array mode. Thus [ ][ ]REAL is not allowed; the touching brackets][must be replaced by a comma to give a true two-dimensional array, or an array of *references* to another array can be used (5.4).

When declaring a constant array by an identity declaration, the actual value can be displayed in the form of a list known as a *collateral*, as in

```
[ ]INT lookup = (2, 8, 4, 5, 6, 3, 9, 0, 7, 1)
```

In general, the items in the list can be written as serial clauses which will deliver objects of the mode required for the array. After being declared in this way, the identifier 'look up' can be used to stand for the entire array, or indexed to select a particular element. The index in this case runs from 1 to 10, as the elements of a collateral are numbered from 1 upwards unless some indication is supplied to the contrary (see 5.2). A collateral is not specifically a denotation for an array, as it is also used for displaying a structure. There are, in fact, no denotations for arrays except for arrays of characters, which are discussed separately in section 5.3. Identity declarations for constant arrays are not a common requirement in programming, and their use should be restricted to cases such as the one shown above where there is a collateral on the right-hand side. There is little to be gained from a declaration which defines an array identifier in terms of some other array already declared, as it will not cause a copy to be made.

We turn now to variables. An array variable is a reference to an array and has a mode such as REF[ ]REAL. It is used for holding any suitable array assigned to it, and its declaration must create the required amount of space. Suppose, for example, that we wish to make space for a row of 4 reals. We may write

```
REF[ ]REAL row = LOC[1 :4]REAL ;
```

The generator specifies the range over which the array index is going to run, and a fault will occur if the index goes outside these bounds. The lower and upper bounds (1 and 4 in the above example) can be expressed as integers or more generally as a pair of unitary clauses, which are evaluated when the declaration is encountered.

To abbreviate the above declaration, the rule given in 3.2 is not quite sufficient. It is not good enough to omit the right-hand side and delete the initial REF, as this would eliminate the bounds altogether. These must still be accommodated, and so the abbreviated form is

```
[1 :4]REAL row ;
```

Bounds must always be given in a declaration which generates space.

A variable declaration can be combined with initial assignment, as for example

```
[1 :4]REAL row := (0.0, 1.0, 0.0, 0.0)
```

Care must be taken when assigning arrays to see that the right-hand side has the same index bounds as the variable. The above example is correct, but if the bounds on the left had been 0 : 3, it would have been wrong, as the bounds of the collateral are taken to be 1 : 4. The right-hand side of an assignment to an array variable need not be a collateral; it can be a previously declared array (or any unitary clause which delivers an array). For example,

```
[1 : 4]REAL copy := row
```

declares a new variable into which 'row' will be copied. Assignment always causes a copy to be taken—for that is why space is generated in the first place. Again, care must be taken to see that the bounds on the two sides agree exactly, and furthermore the mode of the array which is being assigned must be suitable for the variable which is to receive it. The elements of an array (unlike the individual terms in a collateral display) cannot be coerced to any other mode. If, in the example shown above, 'row' had had mode [ ]INT, it would not have been possible to perform the assignment.

The manner of dealing with arrays of two or more dimensions is an obvious generalization of the above. To take an example with two dimensions,

```
[1 : n, 1 : m]REAL matrix ;
```

declares a variable for a matrix of n * m real elements. Collaterals can be used for assignment, as in

```
matrix := ((x11, x12), (x21, x22))
```

which assumes n = m = 2. However, display of a two-dimensional array is expensive, as it entails the use of 'heap storage', briefly discussed in section 5.3. If the program would not otherwise require heap facilities, the use of nested collaterals hardly justifies the cost. There is, in any case, an alternative method, mentioned in the next section.

5.2 Indexing

As explained in the introduction to this chapter, individual elements of an array are selected by indexing. Indexing can also be used to select subsets of arrays, such as a row or column of a matrix. First, however, it is important to describe the mode of object obtained as a result of indexing.

Consider the array constant c, and the variable v, declared by

```
[ ]INT c = (1, 2, 3) ;
[1 :3]INT v := c
```

The mode of c is [ ]INT, and the mode of its elements c[1], c[2] and c[3] is INT. The mode of v is REF[ ]INT, and the mode of v[1], v[2] and v[3] is REF INT. The effect of indexing a reference to an array is to produce a *reference* to the selected element*. Consequently, we can assign to indexed variables, thus

```
v[1] := 1 ;
v[2] := 2 ;
v[3] := 3
```

* This applies however many levels of reference there are to the array. The effect of indexing is to remove all but one REF.

The indexes used for selecting array elements will normally be variables or expressions, computed whilst the program is running, for otherwise there is no advantage in using arrays rather than structures.

To select a subset of the elements of an array, a more general type of indexing is used. In the explanation which follows, we shall take as a starting point the array variables declared by

```
[n :m]REAL a ;
[n :m, r :s]REAL b
```

Subset	*Elements of the original array*	*Bounds of the subset*
a[5 :7]	a[5], a[6], a[7]	1 :3
b[5 :7, j]	b[5, j], b[6, j], b[7, j]	1 :3
b[i, 5 :7]	b[i, 5], b[i, 6], b[i, 7]	1 :3
b[5 :7, 6 :8]	b[5, 6 :8], b[6, 6 :8], b[7, 6 :8] (nine elements in all)	1 :3, 1 :3
b[i,]	the ith row of b, complete	r :s
b[i]	alternative form of above	r :s
b[, j]	the jth 'column' of b, complete	n : m

It will be seen from these examples that a subset formed by indicating the lower and upper limits of an index of the original array acquires new bounds starting at 1. Thus the elements of a[5 :7] are individually

a[5:7][1], a[5 : 7][2] and a[5 : 7][3]

In terms of the original array variable, these are

a[5], a[6] and a[7]

On the other hand, if a dimension is left alone, by means of a blank index, the undisturbed dimension retains its original bounds. Thus

b[5 :7,] has bounds 1 :3, r :s

but

b[5 :7, r :s] has bounds 1 :3, 1 :(s − r + 1)

In this last case, the subset of the second dimension includes the whole of its original range, but according to rule it acquires the new lower bound of 1. In practice, this change of bound is almost always what is actually required. To see this, consider the simple case

```
[1 :10]INT row ;
```

To assign collaterally to a part of this array, we can write

```
row[6 :10] := (6, 7, 8, 9, 10)
```

and the bounds of the two sides will agree as they should. The integers shown for indexing in all of the above examples have been chosen merely for simplicity. In every case, unitary clauses can also be used.

The indexing range for any array can be made to be different from the actual bounds by means of the 'AT' construction. For example, in the declaration

```
[0:4]INT w := (1, 2, 3, 4, 5)[AT 0]
```

the lower bound for the collateral is shifted to 0, making it suitable for assignment to w. Alternatively,

```
w[AT 1] := (1, 2, 3, 4, 5)
```

is valid, as the 'indexer' used with w makes the bounds go from 1 to 5. The AT construction can be used by itself in this way, or as part of a subsetting indexer. For example,

```
w[3:4 AT 3]
```

refers to a row of two integers, and has the lower bound 3 when otherwise it would have been 1.

There are many applications of array subsetting (often described as *slicing*). Problems in physics often call for the selection of rows or columns from matrices, and we may note in passing that slicing is one way of avoiding the nested collateral shown at the end of 5.1. That assignment would have been more efficiently written as

```
matrix[1] := (x11, x12) ;
matrix[2] := (x21, x22)
```

Another application occurs in connection with procedures, where it may be required to supply a part of an array, or an array with fewer dimensions, as a parameter.

When using arrays, it should always be borne in mind that indexing, being dynamic, consumes time while the program is running. Repeated use of the same index is usually avoidable by means of a suitable identity declaration. For example, let us suppose that we find a particular element of an array g, say the element g[1], is required many times over. If g is a real array variable, g[1] has mode REF REAL, and the reference can be separately identified by a declaration such as

```
REF REAL g1 = g[1]
```

Thereafter, g1 will be exactly equivalent to g[1] but without its indexing overhead. Similarly, with h declared by

```
[1 : n, 1 : m]REAL h ;
```

repeated double indexing, when the first index (say) is always the same, can be avoided by declaring

```
REF[ ]REAL hi = h[i]
```

and using hi[j] rather than h[i, j]. This too represents a saving.

5.3 Sequences of characters and flexible arrays

The principles applying to arrays of characters are the same as for other arrays, but there are some additional and alternative facilities meriting separate discussion. A row of characters can be held in an array, which provides the facility of selecting individual characters or subsets by indexing, but if indexing is not really required there are alternatives. An object of mode BYTES is a group

of 4 characters with no indexing facility (and no associated overhead). Groups of 8 or 12 characters may similarly be treated as objects of modes LONG BYTES or LONG LONG BYTES respectively. The denotations for such objects are character sequences of the appropriate length enclosed by quote symbols; for example

```
LONG LONG BYTES title = "USERS' GUIDE"
```

In practice, it is usually tolerable to make lengths of sequences up to 4, 8 or 12 by padding with spaces.

Sequences of these and other lengths can be handled as rows of characters with mode [ ]CHAR. For example,

```
[ ]CHAR alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

or, in the case of a variable,

```
[1 :6]CHAR parent := "FATHER" ;
IF female THEN parent[1 :2] := "MO" FI
```

As they are not of length 4, 8 or 12, the above right-hand sides are row-of-character denotations, usually described as *string denotations*. However, this does not preclude the use of "JACK" in a declaration or assignment such as

```
[1 : 4]CHAR name := "JACK"
```

as coercion from mode BYTES to [ ]CHAR is performed automatically in such a context.

A common requirement in character manipulation work is to handle sequences of unknown length. Whilst one may always use *computed* bounds in the declaration of an array, the difficulty may be that the size of the array is not known in time. The characters may, for instance, have been read in, and the space to receive them has to be generated in advance. When a problem seems unmanageable with fixed length character sequences, the mode STRING should be used. This is what is known as a 'flexible array' of characters. Variables of mode REF STRING automatically adjust their bounds to receive any character sequence assigned to them. For example,

```
STRING s ; {declares s, but creates no space yet}
s := name ; {name was declared earlier as a reference to "JACK". The variable s now holds
             these 4 characters, and has the same bounds as name had}
s := "PIECE WORK" {the bounds of s change to become 1 to 10}
```

The storage space required by a STRING variable changes whenever an assignment demands that it should do so. This facility is achieved by the use of a special system of storage allocation known as 'heaping' and it causes some overhead to be added to a program, not only in the storage space it occupies, but in the speed with which it runs. Although no universal figures can be provided, the extra running time can be ten or twenty per cent. It should be clear from this that the STRING mode should be avoided in work of a mainly numerical nature, where rows of characters occur only in arranging titles for output or similar purposes. However, for problems in fields such as linguistics, or symbol manipulation, it is a facility much to be valued.

Flexible array variables can be declared for other modes (at the same cost) by inclusion of the word FLEX after the bounds. Examples are

```
[1 :0 FLEX]REAL array1 ;
[1 :4 FLEX]REAL array2 ;
```

The first of these declarations creates an array variable with no initial space, for the bounds 1 :0 correspond to an array of zero size (1 :1 being an array with one element). Space is created automatically when an assignment is made to the *unindexed* array variable, for example

```
array1 := (1.0, 2.0, 3.0)
```

The bounds of array1 will now agree with those of the object assigned, in this instance 1 :3. The array cannot be extended by assigning to a non-existent element,

```
array1[4] := 4.0 {wrong}
```

This is wrong. A larger array must be created afresh, by a further complete assignment. The declaration of array2 shown above creates some space for the array initially, so that we could write

```
array2[1 :3] := array1 ;
array2[4] := 4.0
```

The mode STRING is defined to mean [1 :0 FLEX]CHAR.

5.4 Arrays of references to arrays

A common practical requirement is to be able to select one array from a set of similar arrays by means of an index. The ability to select rows or columns from a two-dimensional array is not quite the same thing. The data may have been already grouped in one-dimensional arrays, not necessarily all of the same size. If a two-dimensional array were declared, the data would have to be copied into it, and would not completely fill it unless the original arrays were of equal size. The solution to this problem is to declare an array whose elements are the original array *variables.* This is an array of references to the given arrays. As an initial example, let the original array variables be v1, v2 and v3 declared by

```
[1 : n1]REAL v1 ;
[1 : n2]REAL v2 ;
[1 : n3]REAL v3 ;
```

These can be grouped into a 'higher' array, r say, as follows:

```
[ ]REF[ ]REAL r = (v1, v2, v3)
```

The effect of indexing r is to select one of the original variables; thus r[2] is v2, and the ith element of v2 can be selected by writing r[2] [i]. Notice that both pairs of square brackets in the above declaration are empty. This is because the declaration creates no new working space. The first pair of brackets simply indicates that (v1, v2, v3) is to be treated as an array. The second pair is part of the mode for the elements. Since these are merely *references* to other arrays no actual array space need be created; it is claimed in the separate declarations of v1, v2 and v3.

Although it illustrates the idea, the above example is not wholly typical of the way arrays of array references arise in practice, as the size of the array of references will probably not be known when the program is written. This raises a difficulty, for if the number of component arrays is unknown, they cannot be declared like v1, v2 and v3. The space must be generated anonymously as shown in the following problem, which gives a more realistic picture of a practical situation.

The problem is to read in a set of arrays of reals, creating space for each array just before its items are read, and to assign each array reference to a suitable variable, whose mode will be REF REF[ ]REAL. We shall assume that the data stream is arranged in the following sequence:

number of rows (an integer)
number of reals in the first row (an integer)
the first row of reals
number of reals in the second row (an integer)
the second row of reals
etc

The piece of program shown below reads the first item and uses this to declare an array of array references. This is followed by a loop; each repetition creates the space required for the next row of reals (by a local generator), fills it, and assigns it to the main array of references (though the last two steps must necessarily be done in reverse order).

```
INT rows, items ;
read(rows) ;
[1 :rows]REF[ ]REAL array ;
FOR i TO rows DO
BEGIN read(items) ;
        array[i] := LOC[1 :items]REAL ;
        read(array[i]) {reads the row of reals}
END
```

The scope of the local generator is the same as that of 'array'. It is essential in this example that there be no declarations after DO BEGIN, as the effect would be to restrict the scope of the generator and prevent the program working properly (see section 10.5 forbidding assignment of scoped objects to variables of wider scope).

6

Structures and Mode Declarations

6.1 Structure declaration and field selection

A structure is a collection of objects whose modes may all be different. Each object occupies a certain position or field in the structure and is named with a *field selector*. The selectors are made up by the programmer in the same way as identifiers, but there is no confusion between the two, as they are used in different contexts.

A constant structure can be declared by an identity declaration such as

```
STRUCT(BYTES name, INT day, month)feast = ("XMAS", 25, 12)
```

The above mode is an obvious abbreviation of

```
STRUCT(BYTES name, INT day, INT month)
```

The field selectors are part of the mode. To declare a structure variable, the identity declaration is particularly clumsy, as will be seen from the example

```
REF STRUCT(BYTES name, INT day, month)occasion
= LOC STRUCT(BYTES name, INT day, month)
```

This can be abbreviated in the usual way for variable declarations, and may also include an initial assignment, thus

```
STRUCT(BYTES name, INT day, month)occasion := feast
```

The effect of this assignment is to copy ("XMAS", 25, 12) into the new variable, 'occasion'.

When a single field of a structure is required, it is selected by the construction

selector OF *primary*

Usually the primary will be a structure identifier, but it could be a bracketed clause delivering a structure or reference to a structure. The mode of the field is given by the specification in the structure declaration; thus 'name OF feast' has mode BYTES, and 'day OF feast' has mode INT. However, when field selection is applied to a structure *variable*, the REF belonging to the variable is transferred to the field.

object	*mode*
occasion	REF STRUCT(BYTES name, INT day, month)
name OF occasion	REF BYTES
day OF occasion	REF INT
month OF occasion	REF INT

Thus it is possible to assign individually to the fields of a variable structure, as in

```
name OF occasion := "WHIT"
```

This is similar to the way array indexing works and the same generalization about multiple references applies. The object after OF is automatically dereferenced until its mode is REF STRUCT(. . .), and the remaining REF is transferred to the selected field.

Grammatically, the construction *selector* OF *primary* is itself a primary. The word OF cannot be omitted, even if the selector is unique in the program, and it binds more tightly than any operator. This makes it easy to write expressions, such as

```
100 * month OF occasion + day OF occasion
```

without having to bracket month OF occasion, and day OF occasion.

As the fields of a structure can have any mode, they may themselves be structures. For example, lettered points on a diagram could be held in variables such as p, q and r declared as

```
STRUCT(CHAR label, STRUCT(REAL x, y)coords)p, q, r ;
```

and values could be assigned to all or part of such variables in such ways as

```
x OF coords OF p := y OF coords OF p := 1.0 ;
label OF p := "O" ;
q := p ;
r := ("X",(1.0, 0.0))
```

6.2 Mode declarations

It will already be obvious that the mode of a structure is likely to be quite long, and the need for some means of abbreviation is quickly felt. This is one of the purposes of *mode declarations*. Any mode—not just a structure mode—can be given a concise name by a declaration of the form

```
MODE modename = mode
```

where *modename* can be any upper case word chosen by the programmer. For example, the declarations

```
MODE VECTOR = STRUCT(REAL x, y) ;
MODE POINT = STRUCT(CHAR label, VECTOR coords) ;
```

enable us to declare p, q and r of section 6.1 by writing simply

```
POINT p, q, r ;
```

A new mode name can be used as soon as it has been declared (or 'mentioned'—see chapter 11), and is scoped in the usual way.

When a mode name is declared for an array, the question of bounds has to be considered. As the mode name may be used in a variable declaration, its definition must *always* include bounds in places where they would then be necessary. For example, to define a mode name for a mode such as

```
[ , ]REAL
```

the mode declaration would have to include bounds, as in

```
MODE MATRIX = [1 :n, 1 :m]REAL ;
```

so that a variable declaration like

```
MATRIX h ;
```

would contain all the information required. The bounds are evaluated when the mode declaration is encountered, and will therefore be the same whenever the mode name MATRIX is used. In contexts where bounds are not required, the hidden bounds are simply ignored. For example, consider the declaration

```
[1 :6]REF MATRIX row of refs ;
```

This would be taken to mean

```
[1 :6]REF[ , ]REAL row of refs ;
```

as bounds are never required after the word REF (and would indeed be wrong).

The mode names STRING and COMPLEX can be used without declaration in a user's program. Their standard meanings are

```
MODE STRING = [1 :0 FLEX]CHAR
MODE COMPLEX = STRUCT(REAL re, im)
```

Care should be taken not to use the construction [ ]STRING, as this would mean [ ] [ ]CHAR, and rows of rows are not allowed.

6.3 Structures and arrays in combination

Structures can have array fields and arrays can have structures as their elements. This enables us to build modes of arbitrary complexity to match the structure of the data we are dealing with. The rules for selecting elements of an array or fields of a structure have already been described, but when arrays and structures occur in combination, one further rule is needed. Briefly, *brackets bind more tightly than OF*. This applies to the square brackets used for indexing arrays and also to round brackets when they are used for holding the actual parameters of a procedure. To see how this rule works, consider the following situation. A variable t is declared by

```
[1 :n]STRUCT(INT i, REAL x)t ;
```

and we wish to select the integer field of the first structure in the array. The construction

```
i OF t[1]
```

is correct, because t is bound more tightly to [1] than to the preceding OF. However, the binding rule will not always produce the effect we actually require. Take the declaration

```
STRUCT([1 :n] INT row, INT i)s ;
```

and consider how to select the first element of the row field. The constructions

```
row OF s[1] {wrong}
row[1] OF s {wrong}
```

are both wrong. The first is wrong because s is not an array, and therefore cannot be indexed. The

second is wrong because 'row' is not an object at all! It is a field selector. The correct answer is

```
(row OF s) [1]
```

Round brackets can always be used to form the primary which is required.

6.4 Bounds in declarations

A variable declaration for a structure must create space for all of its fields. If any of the fields are arrays, they will require appropriate space generation, and index bounds must therefore be given. Beginners are not always entirely clear about this, and it has been found useful to give a general rule for determining when bounds must be included in declarations involving arrays and when not.

Identity declarations

The mode must never include bounds. Examples:

```
[ ]COMPLEX w = (0.0?1.0, 1.0?0.0, 0.0?0.0) ;
STRUCT([ ]CHAR s, INT i)t = ("ABCDE", 8)
```

Variable declarations

First write the declaration without any index bounds, then apply the following rules.

1 Does the mode start with [ ]?
  If so, insert bounds and apply rule 2 to the rest of the mode.
  If not,
2 Does it start with STRUCT?
  If so, apply rule 1 to each field in turn.
  If not,
3 Do not put bounds in any subsequent square brackets, as this declaration is not concerned with creating array space.

Example:

```
[1 :5]STRUCT(REF[ ]INT reference field,
             STRUCT([1 :3]BYTES b, BOOL c)struct field,
             [1 :n, 1 :m]REAL matrix)row of structures ;
```

This makes space in all for 5 references, 15 bytes', 5 bools and 5 * n * m reals. The space for the integer arrays would be generated in the declarations which created the references to be assigned to these fields.

7

Procedures

Procedures are objects, such as cos or print, which happen to be processes rather than data in the normal sense of the word. They are objects which can be *obeyed.* The purpose of a procedure is to enable a piece of program to be written and given a name at one place in a program, without being obeyed there and then. Once named, it can be obeyed at any time by merely writing its name and coupling it with such data as it needs for its operation. The economy which results may be very great indeed, for commonly needed processes have only to be written out once. It is found, also, that the mental stress entailed in writing a complicated program is enormously reduced if subsidiary parts are hived off as procedures.

In the present chapter, we concentrate first on obeying procedures which have already been written. This should give an understanding of the proper way to use library procedures, which do not have to be declared by the user. The remaining sections demonstrate how a user may write his own procedures and declare them in his program.

7.1 Mode of a procedure

A procedure is a piece of program named with an identifier. For example, the standard procedure for taking a square root is named 'sqrt'. In order to use such a procedure correctly, we must ascertain (i) what type of data it needs, and (ii) what type of result it delivers. This information is contained in the *mode of the procedure.* (What a library procedure actually does will normally be described in words in the library catalogue. If the procedure has been written by the user himself, it is good practice for him to keep a brief description for future reference.)

As an example, we find from Appendix 2 that the mode of sqrt is

PROC(REAL)REAL

This indicates that sqrt

—is the name of a procedure
—needs to be supplied with a REAL object of data
—delivers a result of mode REAL

Other examples of procedure modes are given overleaf.

mode of procedure	*remarks*
PROC(INT, INT)BOOL	Requires 2 integers as data. Delivers boolean result.
PROC REAL	Requires no data. Delivers REAL result (for example 'random', Appendix 2).
PROC(REF[ ]INT)VOID	Requires an integer array variable. Delivers no result.
PROC VOID	Requires no data. Delivers no result.
PROC(REAL,REAL,PROC(REAL)REAL)REAL	Requires three objects of data, two real numbers and a procedure of mode PROC(REAL)REAL. Delivers a real result.

It should be clear from these examples that a procedure can demand as data any number of objects of any mode, and can deliver a result of any mode or no result at all. The word VOID means that no result is delivered.

7.2 Obeying a procedure

A procedure is obeyed ('called') when its identifier is followed by a list of the objects of data it requires. These are known as the *actual parameters* and are separated by commas and enclosed in brackets. For example, suppose that f is a procedure of mode PROC(INT,INT)REAL. Such a procedure could be called, with two integer parameters, in any context where a REAL would be appropriate, as for instance in the expression:

```
f(3,5) * sqrt(9.0) + f(4,6) * sqrt(13.0) + 1.0
```

The actual parameters (3, 5, etc) can be unitary clauses. In practice, they will usually be expressions, as in

```
f(n+m, ROUND sqrt(1+x↑2))
```

Before the procedure f is obeyed, the two parameters are evaluated, and it must not be assumed that the first will be evaluated before (or after) the second. The order is undefined. In this example, the second parameter contains a call of sqrt; any amount of such nesting of procedure calls is allowed.

Each actual parameter must be capable of delivering an object of the mode specified in the mode of the procedure. If necessary, coercions such as dereferencing will be applied automatically. For example, remembering that the mode of sqrt is PROC(REAL)REAL, we can see that the use of a real variable x in the call sqrt(x) would cause x to be dereferenced before sqrt was obeyed.

A procedure like sqrt, whose sole purpose is to deliver a result, is akin to a mathematical function. But procedures are not restricted to functions; a procedure can perform any process whatever, and its call need not appear in a formula. It may stand by itself as an individual step in a program, for example:

```
BEGIN
    REAL x, y ;
    read(x) ;
    y := 1 + sqrt(1 + x↑2) ;
    print(y/(y − 2))
END
```

The calls of 'read' and 'print', which are standard input and output procedures, do not deliver any result. They carry out processes. Grammatically, a procedure call is a primary and as such can always be used as a unitary or serial clause.

7.3 Writing a procedure—a simple example

Suppose that we require a procedure which will deliver as its result the period of swing of a simple pendulum for any given values of the length of the pendulum and angle of swing (ie semi-arc in radians). Denoting these two parameters by 'length' and 'angle', the mathematical formula for the period is

$$2\pi\sqrt{\frac{\text{length}}{g}}\cdot\left(1+\frac{\text{angle}^2}{16}\right)$$

where g is the acceleration due to gravity. The following bracketed serial clause would perform the calculation and deliver the required result:

```
BEGIN
    REAL length = ... , angle = ... ;
    REAL g = 981.3 ;
    2 * pi * sqrt(length/g) * (1 + angle↑2/16)
END
```

provided that we could fill in the blanks with the actual values of length and angle required. However, this cannot be done. Length and angle must be left as mere identifiers until actual values are supplied at the call of the procedure. To express this, the *denotation* for the procedure is written, not as shown above, but as

```
(REAL length, REAL angle)REAL:
BEGIN
    REAL g = 981.3 ;
    2 * pi * sqrt(length/g) * (1 + angle↑2/16)
END
```

This is a form of construction peculiar to procedures. There must be a heading which gives the modes and identifiers for the parameters, all in brackets, followed by the mode of the result delivered, and a colon. The parameter names, length and angle, are known as *formal parameters.*

7.4 Procedure declarations

Procedures are declared in the same way as other objects. Variable declarations can be used to create variables to which different procedures may be assigned (7.4.1), or identity declarations can be used to name particular procedures. We shall consider the identity declaration first, as this is what is most often required. The form of an identity declaration is

mode identifier = unitary clause

When this is used to declare a procedure, the mode of the procedure is written on the left. The unitary clause on the right must deliver an object of this mode, and in the simplest case it will be the denotation of the procedure.

As an example, consider how we might attach a name, 'period' say, to the procedure discussed in 7.3. This procedure takes two REAL parameters and delivers a REAL result. Its mode is

```
PROC(REAL,REAL)REAL
```

The identity declaration is therefore written as

```
PROC(REAL, REAL)REAL period =
(REAL length, REAL angle)REAL :
BEGIN
    REAL g = 981.3 ;
    2 * pi * sqrt(length/g) * (1 + angle↑2/16)
END
```

It will be seen that the mode information appears on both sides, making the declaration rather cumbersome. When the right-hand side of the identity is a procedure denotation*, as in this example, an abbreviated form is allowed,

PROC *identifier = procedure denotation*

The example may therefore be written

```
PROC period = (REAL length, angle)REAL:
BEGIN
    REAL g = 981.3 ;
    2 * pi * sqrt(length/g) * (1 + angle↑2/16)
END
```

In practice, most procedure declarations are expressed in this way. Notice that the mode REAL can apply without repetition to both length and angle.

Procedures may or may not have parameters, and may or may not deliver a result. In the heading of the denotation, if there are no parameters the parameter list is simply omitted. If there is no result the word VOID is used, as also in the mode of the procedure. The different forms for the (abbreviated) declaration are therefore as shown on the next page.

* It need not be; for consider

```
PROC(REAL)REAL trig = IF u THEN sin ELSE cos FI;
```

which makes trig synonymous with sin or cos.

With parameters and result—

PROC *identifier* = (*parameter list*) *mode*: (*serial clause*)

With parameters and no result (in this case the use of VOID is optional)—

PROC *identifier* = (*parameter list*) {VOID}: (*serial clause*)

With result and no parameters—

PROC *identifier* = *mode*: (*serial clause*)

With no result and no parameters—

PROC *identifier* = VOID: (*serial clause*)

In Algol 68-R, the serial clause must always be enclosed in brackets, even when it consists of a single step. If this step is a conditional, double brackets will arise, eg (IF ... FI) or ((...|...|...)).

7.4.1 Procedure variables

A procedure variable is a reference to a procedure, and may be used as an indirect means of calling it. This makes it possible to assign different procedures to the same variable in different parts of the program, so varying the effect of the call. The various procedures which may be assigned must all have precisely the same mode. As an example,

```
PROC(REAL)REAL f ;
f := sin
```

declares the procedure variable f, to which sin is assigned. When we now write f(x), the variable will be automatically dereferenced and the effect will be sin(x). The procedure itself must, of course, be fully in scope when it is called, as discussed in 7.8.

7.5 Parameters

The rules for declaring and calling a procedure have already been described. This section is intended to comment on their practical application, which inevitably revolves around the subject of parameters, as these are the plugs and sockets through which a program communicates with its procedures.

The mechanism by which formal parameters are identified with their actuals is just as though the procedure body began with a series of identity declarations, one for each parameter. The formal parameter provides the identifier and the unitary clause supplied as the actual parameter gives the right-hand side:

mode identifier = *unitary clause*

↑ formal ↑ actual

All facets of identity declarations therefore apply with equal force to parameter substitution.

7.5.1 Simple values as parameters

Readers accustomed to Algol 60 should be made aware that a REAL parameter in Algol 68 is not quite the same as a 'value parameter' in the earlier language. In Algol 68, a REAL parameter is identified with a real number, and cannot be used as though it were a variable; it cannot appear on the left of assignments in the body of the procedure.

When a REAL parameter is specified in a procedure declaration, any expression can be supplied as the actual parameter, provided that it is capable of delivering a REAL result. If a variable is supplied, as will usually be the case, it will automatically be dereferenced. If an integer is supplied, it will be converted to the equivalent real number. But if the parameter is specified as INT, a real must *not* be supplied. This is a good restriction. If a procedure needs to be given the number of teeth on a gear, an actual parameter such as 8.5, possibly disguised as a real variable currently holding 8.5, probably indicates a programming error or misconception. The compiler will report errors of this sort. Reals can, if necessary, be converted to integers by use of operators ENTIER or ROUND.

7.5.2 Reference parameters

The purpose of a reference parameter is to supply a procedure with a *variable*. This enables it to perform assignments to the actual parameter. As a trivial example, consider

```
PROC step = (REF INT i)VOID: (i PLUS 1)
```

The clause i PLUS 1 uses the operator PLUS, defined in Appendix 2, whose effect is to perform the assignment i := i + 1. This clause delivers the result i, which is discarded as 'step' is defined to have no result. The action of this procedure is therefore to take the integer out of the parameter variable, add 1, and put it back. A reference parameter can be regarded as a receptacle for use by the procedure. Depending on how the procedure is designed, it can take an item of data from the receptacle, place a result in it, or both—in this instance both.

Although the mode INT can be widened to REAL, the mode REF INT cannot be coerced to REF REAL. They are different kinds of variable. A procedure which specifies a REF REAL parameter must be supplied with a *real variable*. This does not prevent an expression being used as the actual parameter, but it must be an expression capable of delivering a real variable—a normal arithmetic expression will obviously not do. Valid expressions as actuals for 'step' are shown in

```
INT n := 1, m := 1 ;
step(IF random < 0.5 THEN n ELSE m FI) ;
step(CASE n IN n, m ESAC)
```

Reference parameters can, of course, be of many different modes. References to arrays are discussed in the following section. References to structures are another possibility. Example 6 shows a procedure which operates on a REF DATE parameter, where DATE is defined by

```
MODE DATE = STRUCT(INT day, BYTES month, INT year)
```

The procedure advances the date referred to in the parameter by one day (without doing anything about leap years).

EXAMPLE 6

```
PROC procrastinate = (REF DATE date):
BEGIN
    [ ]STRUCT(BYTES name, INT days)month =
        (("JAN ", 31), ("FEB ", 28), ("MAR ", 31),
        ("APR ", 30), ("MAY ", 31), ("JUN ", 30),
        ("JUL ", 31), ("AUG ", 31), ("SEP ", 30),
        ("OCT ", 31), ("NOV ", 30), ("DEC ", 31)) ;
    REF INT d = day OF date ;
    REF BYTES m = month OF date ;
    INT i := 1 ;
    WHILE m ≠ name OF month[i] DO i PLUS 1 ;
    IF d < days OF month[i]THEN d PLUS 1 ELSE
       m := name OF month[(i = 12|year OF date PLUS 1 ; 1|i + 1)] ;
       d := 1
    FI
END
```

The procedure is called very simply:

```
DATE meeting := (31, "DEC ", 1970) ;
procrastinate(meeting)
```

alters the meeting date to (1, "JAN ", 1971).

7.5.3 Array parameters

Array parameters occur with great frequency in practical programming. There are no new basic principles to be explained; it should therefore suffice to give an example with comments. As an elementary case, consider a procedure for squaring every element of a two-dimensional array of reals. Such a procedure would take an array variable as its only parameter, assuming that the original array can be over-written with the result.

```
PROC square2 = (REF[ , ]REAL a)VOID :
BEGIN
    FOR i FROM 1 LWB a TO 1 UPB a DO
    FOR j FROM 2 LWB a TO 2 UPB a DO
    (REF REAL aij = a[i, j] ; aij TIMES aij)
END
```

The mode of a formal parameter *never* includes bounds, and yet in nearly every practical case, a procedure does need to know what they are. They may, of course, be different for different calls of the procedure, and the body of the procedure must then include operations on the parameter to determine them. The operators LWB and UPB (Appendix 2) are provided for such a purpose. Alternatively, the bounds could be passed to the procedure through extra parameters, but this is less convenient in use and less foolproof. Occasionally it may be reasonable to design a procedure on the assumption that the bounds are known; it will then be essential to see that the actual parameters have these bounds and no others.

If an array variable x has been declared as

```
[3:36, 0:7]REAL x ;
```

and has had numbers assigned to all of its elements, these will become squared by writing the unitary clause

```
square2(x)
```

The 2 has been included in the identifier merely as a reminder that square2 must be given a 2-dimensional array variable.

Array parameters of mode [ ]REAL, as distinct from REF[ ]REAL, are not often used, unless the actuals are to be supplied as collaterals. This is a useful device for obtaining the effect of an unknown number of parameters. For example, the mode of the formal parameter of a procedure p might be

```
[ ]REF REAL
```

and the corresponding actual could be written as (x, y, z), say. Notice that this introduces double brackets in the call:

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

If, in a particular call, we should wish to supply the procedure p with only one real variable, we can write simply

```
p(x)
```

as a coercion known as rowing (chapter 9) will convert x into an array of one element automatically.

7.5.4 Procedure parameters

So far, we have concentrated on the use of parameters for supplying a procedure (which we shall call p) with objects of data such as numbers, references, arrays and so on. The actual parameter is reduced to the object required (eg by evaluating an expression) upon entry to the procedure p. A more general type of requirement is to supply p with a *process* to be carried out somewhere within its body. This is done by giving a *procedure* as a parameter of p. Within the body of p, the corresponding formal is treated as the name of the (unknown) procedure, which will be obeyed wherever the formal parameter is called. Formal parameters treated in this manner have a mode beginning PROC.

As an example, we might have a procedure for finding the area under any mathematical curve from a given lower to a given upper limit.

```
PROC area = (PROC(REAL)REAL f, REAL lower, upper)REAL :
            BEGIN ... END
```

To use this, we have to give 'area' the process for calculating a particular curve, as a procedure of mode PROC(REAL)REAL. To find the area under sin(x) from x = 0.0 to x = 0.3, we would write simply

```
area(sin, 0.0, 0.3)
```

Another application is to provide an escape from a procedure to some given label in the program. Typically, this may be a useful means by which some breakdown in a numerical process within the

procedure could give rise to a jump to some place in the program where the appropriate steps for continuing the problem would be found. To illustrate this most simply, we show below a square root procedure with an escape mechanism for negative arguments.

```
PROC root = (REAL x, PROC REAL q)REAL:
            (IF x > 0.0 THEN sqrt(x)
                        ELSE q FI)
```

This procedure 'root' might be called as follows:

```
REAL answer, y ;
        --- ;
        answer := root(y, GOTO remedy) ;
        --- ;
remedy: --- ;
        ---
```

This illustrates a new concept. Instead of supplying the name of a procedure as the actual for q, we have written GOTO a label. Any GOTO-clause may be regarded as a parameterless procedure in this context.

A procedure parameter can have as its actual either the name of a procedure or a procedure denotation, provided as always that the mode is appropriate. The device shown for GOTO is, in reality, a concession to brevity. The call

```
root(y, GOTO remedy)
```

might have been written more fully with an actual procedure denotation,

```
root(y, REAL: (GOTO remedy))
```

but this is unnecessary.

7.6 The procedure body

The 'body' of a procedure is the bracketed serial clause following the heading. This piece of program will be obeyed when the procedure is called, and is best written so that it is as self-contained as possible. If, for example, working spaces or constants are needed (like the real 'g' in the 'period' example) they should be declared in the serial clause itself. This makes such identifiers local, ie limits their scope to within the procedure and reduces confusion with other parts of the program. However, the body of a procedure will often need to make use of library procedures, and to that extent will not be strictly self-contained. Indeed it is sometimes convenient, and even desirable, to trespass further than this. A procedure can make use of any objects declared outside provided they are still in scope when the procedure is called (see 7.8).

The choice always to be made is whether to pass data to the procedure through the proper channels (parameters) or whether to allow it to use whatever is available from outside. In the latter case, a proper specification of the procedure may become difficult, and much of the purpose of using a procedure may be nullified. If a procedure is to be used in many different contexts, possibly in different programs, it should be as self-contained as possible. It should assume no more of an environment than is certain to be available, such as the system library, or other procedures in a clearly defined suite.

7.7 The result

The result of a procedure has the mode specified in its heading, and is the object delivered by the last unitary clause obeyed in the bracketed serial clause which forms the body of the procedure. If necessary, this object is coerced to the required mode. If no result is specified, the object delivered by the body is simply discarded, and therefore cannot be used.

When writing a procedure, it is essential to ensure that the unitary clause delivering the required object is indeed the very last clause obeyed. A beginner could make the mistake of writing:

```
BEGIN
        IF a>b THEN GOTO middle FI ;
        --- ;
        y ;
        GOTO end ;
middle: --- ;
        x ;
end:    SKIP
END
```

meaning to deliver x if a>b is true, otherwise y. It is easy to see that the last clause obeyed here is the dummy clause SKIP; the results x and y are eliminated, before this, by the semi-colons which follow them. One way to overcome this is to use EXIT as shown in

```
BEGIN
        IF a>b THEN GOTO middle FI ;
        --- ;
        y EXIT
middle: --- ;
        x
END
```

The effect of EXIT is to take us to the end of the serial clause containing it, without discarding any result we may have in hand. It is used *in place of a semi-colon* and must always be followed by a label (for otherwise there would be no way of reaching the clauses which follow on).

The word EXIT need not be confined to the body of a procedure, but it must be used with care. It must always be remembered that it takes us only to the end of the serial clause in the immediate context. Consider

```
BEGIN --- ;
        IF u THEN ... EXIT after: ... FI ;
        --- ;
        ---
END
```

In this example, EXIT takes us to FI, not to the END.

The result of a procedure can be an object of any mode, including procedure modes and reference

modes. In these more advanced applications, it is necessary to make quite sure that the scope of the object delivered is not restricted to the procedure body. For example, it is impossible to deliver as the result of a procedure a variable which was declared within its body. Compare the following procedures:

right (*formal parameter list*)REAL: (REAL x ; --- ; --- ; x)

wrong (*formal parameter list*)REF REAL: (REAL x; --- ; --- ; x)

right (REF REAL x)REF REAL: (--- ; --- ; x)

In the first of these examples, x is a REF REAL local to the procedure. The space created by the variable declaration is available for use inside the body only, but the REAL to which x refers is a basic value which has no scoping limitation. As x is dereferenced to deliver the REAL result, no complications arise.

In the second example, we are attempting to deliver x itself, ie to deliver a working-space. As this space is available within the body only, the program is wrong.

In the third example, the object delivered (when the procedure is called) is the actual parameter supplied for x. As this must have been in scope at the procedure call, it can be the object delivered as the result of the procedure. An application of this idea is shown in the following example. The procedure p searches for the largest positive element of an array and delivers a reference to it. If all the elements are zero or less, it delivers NIL, which is a reference to a non-existent object.

```
PROC p = (REF[ ]INT a)REF INT:
BEGIN
    INT max := 0, n ;
    FOR i FROM LWB a TO UPB a DO
        IF a[i] > max THEN max := a[i] ; n := i FI ;
    IF max = 0 THEN NIL ELSE a[n]FI
END
```

As an illustration of a procedure which delivers a procedure,

```
PROC trig = (INT i)PROC(REAL)REAL:
(CASE i IN sin, cos, tan, sec, cosec, cot ESAC)
```

would be a possibility. A typical call would be

```
x := a * trig(3)(pi/4)
```

and the effect would be x := a * tan(pi/4)

7.8 Scope of a procedure denotation

A procedure cannot be obeyed if any of the objects it requires are out of scope when it is called. There can be no scoping trouble if the procedure is named in an identity declaration and called by using the declared name, but if the procedure denotation is handled as an object, for example,

assigned to a procedure *variable* or delivered as the result of some other procedure, care must be taken. The following artificial example shows the type of mistake which could easily be made:

```
PROC INT f ;
BEGIN
    INT i ;
    f := INT:(i + 1) ;
    i := 2
END ;
print(f)
```

Between BEGIN and END, an assignment causes the procedure variable f to refer to a procedure denoted by INT:(i + 1). It is important to notice that this procedure uses a non-local variable i which will cease to exist after END. The use of f in the print clause is therefore meaningless, and is a situation which must not be allowed to occur. The source of the trouble is the assignment to f; an object must never be assigned to a variable which has a greater scope than that of the object itself. A similar situation can arise in computing with references, and is discussed in 10.5.

If a procedure denotation uses no external objects other than those which are passed as its own parameters, its scope is not limited in any way. For example, consider

```
PROC(REF INT)f ;
BEGIN --- ;
        f := (REF INT i):(i PLUS 1) ;
        ---
END ;
INT j := 2 ;
f(j)
```

The procedure assigned to f in the third line is entirely self-contained, and its calls need not be restricted to the serial clause in which it appears. The effect of f(j) will be to dereference f, and apply the procedure to the actual parameter j, thus making j refer to 3.

8

Defining New Operators

A list of the standard operators is given in Appendix 2. Within a program, additional operators can be declared and existing operators can be given additional meanings. The effect of any operator depends not only on which operator it is, but on the modes of the operands; extra meanings can therefore be given to an existing operator if the modes of the operands for the new meaning are distinct from those for existing meanings.

Operator declarations are especially useful for defining operations between data-structures peculiar to the program.

8.1 Operator declarations

An operator is not an 'object', and therefore has no mode, but an operator declaration resembles a procedure declaration with the word OP in place of PROC. The simple form is

OP *opsymbol* = *procedure denotation*

A procedure denotation is entirely appropriate on the right, as an operator is, in effect, a procedure with one or two parameters (one if monadic, two if dyadic). As an example, the following declaration adds an extra dyadic meaning to the multiplication operator (*), defining it to give the scalar product of two vectors of n elements:

```
OP * = ([ ]REAL a, b)REAL:
       BEGIN REAL s := 0.0 ;
             FOR i FROM LWB a TO UPB a DO
                 s PLUS a[i] * b[i] ;
             s
       END
```

The significant distinction between operators and procedures is that several declarations of the same operator symbol may appear in the same serial clause and be valid *simultaneously*, provided that each applies to a distinct set of operand modes. A monadic definition is always distinct from a dyadic, and dyadic definitions are distinct provided that no ambiguity can arise in use. An operator symbol defined for both the following pairs of modes would be ambiguous.

	left operand	*right operand*
a	REAL	REAL
b	REF REAL	REAL

Ambiguity between *a* and *b* is due to the fact that operands are automatically dereferenced if necessary. A REF REAL operand could therefore select definition *b*, or be dereferenced and select definition *a*. The only other possible source of ambiguity would be between a REAL (say) and a PROC REAL, where 'deproceduring' (9.1.1) might or might not occur, but this situation rarely arises in practice. The following combination does not lead to ambiguity:

	left operand	*right operand*
c	REAL	REAL
d	REAL	INT

because 'widening' (9.1.3) is not a form of coercion applied to operands.

As a further example, the operator declaration given below defines 'equals' between two objects of mode DATE, where

```
MODE DATE = STRUCT(INT day, BYTES month, INT year) ;
```

Thus,

```
OP = = (DATE a, b)BOOL:
        (day OF a = day OF b AND
         month OF a = month OF b AND
         year OF a = year OF b)
```

As with procedure declarations, a more general form of operator declaration takes one or other of the forms:

monadic

OP (*mode*) {*mode*} *opsymbol* = *unitary clause*

dyadic

OP (*mode, mode*) {*mode*} *opsymbol* = *unitary clause*

where the mode for the result may be replaced by the word VOID if (exceptionally) the operator is not required to deliver a result. The unitary clause must deliver a procedure with one or two formal parameters respectively. For instance,

```
OP(REAL)REAL SIN = sin
```

provides a new, if rather superfluous, way of finding a sine. (Instead of writing sin(x), one could write SIN x.)

8.2 Declaration of priority

Monadic operators bind more tightly than dyadic operators, and dyadic operators bind with a tightness depending on their priority. The larger the priority number, the tighter the binding. In the absence of a priority declaration, the standard priority for the operator symbol applies, regardless of whether the operator declaration supplements the standard definition or overrides it (see Appendix 2, Default Priorities). If the operator declaration is for an entirely new operator symbol, the default priority is taken to be 1, the weakest binding of all.

To define a new priority for a dyadic operator, the operator declaration must be preceded by a priority declaration having the simple form

PRIORITY *opsymbol* = *digit*

where the digit given must be in the range 1 to 9 inclusive. The normal rules of scoping apply to operator declarations and priority declarations. For example, consider

```
BEGIN PRIORITY £ = 3 ;
      OP £ = (INT a, b)INT:( --- ) ;
      --- ;
      BEGIN PRIORITY £ = 4 ;
            OP £ = (INT a, b)INT:( --- ) ;
            ---
      END ;
ww:   ---
END
```

Within the inner serial clause, the new local priority (4) and meaning of £ override the non-local definitions. If the local priority declaration had been omitted, the new definition of £ would have had priority 3. In any case, at ww the priority 3 is restored, together with the original definition of the operator.

8.3 Operator symbols

Basic symbols which have been set aside for use as operators are:

standard operators	+ − * / ↑ = # < > ?
spare operator characters	£ % @

Other permitted forms of operator symbol are:

upper case words	any upper case word not already used for some other purpose
combinations	any combination of symbols, unspaced and between primes, for example '/:='.

Note that the standard operators <= and >= are exceptional. These are the only permitted instances in which a combination of operator characters may be used without enclosure in primes. (The operators <= and '<=' are regarded as quite distinct.) Spaces must not be inserted between < and = or > and =.

9

Coercion

Modes are important because they control to a great extent the processes to be carried out as a program is obeyed. For example, the mode of the variable on the left of an assignment determines absolutely the mode of object which the right-hand side must deliver, and the process of evaluating the right-hand side is influenced accordingly. If at first an object of the wrong mode appears in such a context (the right-hand side), it will be acted upon so as to produce an object of the right mode *if possible*. Not all contexts act as strongly as this. On the left-hand side of an assignment, the demands made by the context are much less strong; the only requirement here is that the object should be a variable, ie its mode must begin with REF. The present chapter is therefore concerned with two aspects of coercion, firstly the types of mode change which are possible and secondly the degree to which various contexts will bring them about.

9.1 Forms of coercion

9.1.1 Deproceduring

A procedure which takes parameters is obeyed when actual parameters are supplied, and is not obeyed if they are not. Unfortunately, this simple rule is inapplicable for procedures which have no parameters in their definition. Whether they are obeyed or not has to depend on the mode of object required. For example, the library procedure 'random' has mode PROC REAL. When random is mentioned, it is not obeyed if the context expects a PROC REAL, but is obeyed if the context expects a REAL. This occurrence is known as the deproceduring coercion, because an object of mode PROC . . . has given place to an object of the same mode without the PROC. In the following example, the deproceduring coercion occurs once:

```
REAL x ;
PROC(REAL)REAL f1 ;
PROC REAL f2 ;
x := sin(0.3) ;  x := random ;  {both procedures are obeyed}
f1 := sin ;      f2 := random   {neither procedure is obeyed}
```

9.1.2 Dereferencing

This is the most familiar coercion, and has been discussed in earlier chapters. Typically, if x has mode REF REAL and the context demands a REAL, then x is dereferenced.

9.1.3 Widening

If a context demands REAL, then INT can be supplied instead. REAL is a wider class than INT, and the coercion is known as widening. Other forms of widening are:

INT	to	REAL
REAL		COMPLEX
INT		BITS
BYTES		[ ]CHAR
LONG BYTES		[ ]CHAR
LONG LONG BYTES		[ ]CHAR

9.1.4 Rowing

If the context demands [ ]INT, say, and an INT is supplied, the required mode change to an array of one element (with lower and upper bound 1) takes place.

An example is given in 7.5.3 where an actual parameter is made into an array to suit the mode required by a procedure. Similarly, if the context demands REF[ ]INT, the mode REF INT can be coerced to it. Thus,

```
REAL y ;
REF[ ]REAL x = y ;
```

causes x to refer to an array of one element, such that x[1] is the same REF REAL as y.

Rowing can also be applied to arrays, in order to increase the number of dimensions by one. For example,

```
[ ]INT i1 = (1, 2, 3) ;
[, ]INT i2 = i1 ;
```

makes i2 an array with bounds [1:1, 1:3].

9.1.5 Uniting

In Algol 68, the programmer is bound to associate a mode with every identifier introduced in the program. Occasionally, we would wish that the mode of an object could be dynamic, ie subject to change in the course of running. A special class of modes,

UNION(*mode, mode, ...*)

caters for this requirement. For example, if we declare x by writing

```
UNION(INT, CHAR)x ;
```

the mode of x is REF UNION(INT, CHAR). The 'united mode' permits x to hold *either* an integer *or* a character. If we make the assignment

```
x := "H"
```

the mode of the right-hand side, CHAR, is coerced to UNION(INT, CHAR), which is suitable for assignment to x. This is known as uniting. After the assignment has been performed, x holds "H" (and knows it holds a character). *There is no de-uniting coercion.*

```
CHAR a := x
```

is illegal. If this assignment were allowed, the principle of mode-checking would be violated. The only way to extract "H" from x is by a conformity clause (9.1.5.1).

There are some restrictions on the modes which can be included in a union. As an example, UNION(INT, REF INT) is not allowed. For suppose u were a variable of this mode, and we were to write the assignment u := i; where i is an integer variable. If i were dereferenced, it could be united as an INT, but if not, it could be united as a REF INT. Such ambiguity of interpretation is never allowed, and this combination is therefore forbidden. However, in order not to place too many constraints on the combinations allowed in a union, the forms of coercion applied before uniting are restricted to dereferencing and deproceduring. In consequence a mode like UNION (REAL, INT) is allowed, since widening does not occur before uniting.

9.1.5.1 Conformity clauses

The purpose of a conformity clause is to retrieve an object which has been united—and to retrieve it in its original mode. This could be any one of the modes included in the original declaration of the union, depending on which particular path the program takes when it runs. For this reason, a 'collateral conformity clause' is usually written in conjunction with the CASE construction,

CASE (*expression, expression, . . .*) ::= *unitary clause*
IN *serial clause, serial clause, . . .* {OUT *serial clause*} ESAC

The conformity clause is shown between CASE and IN. The unitary clause on the right of the special assignment must deliver the united object (after such dereferencing as may be necessary). Each expression on the left-hand side should deliver a variable capable of holding an object of one of the possible modes of the union. The current value of the united object is assigned to whichever of these variables has the correct mode to receive it. If the nth expression is suitable, ie 'conforms', the integer n is delivered, and acts as the selecting integer of the case clause. If no expression conforms, 0 is delivered and no assignment occurs.

As an example, let us extract "H" from x (into which it was united in 9.1.5),

```
INT i ;
CHAR a ;
CASE (i, a) ::= x IN serial clause, serial clause ESAC
```

The mode of x is REF UNION(INT, CHAR). After dereferencing, the dynamic mode will therefore be INT or CHAR, so two separate variables are provided. In fact the mode is CHAR, so the "H" is assigned to a, whilst i remains untouched. The integer 2 is delivered as the result of the conformity and the second serial clause is obeyed. (This would probably use the variable a, whilst the first serial clause would use i.) A collateral conformity clause used in this manner ensures a complete run-time check of the mode of a united object, corresponding to the checks which can be carried out by the compiler for all non-union modes.

To obtain values of only one of the modes in the union, a simpler construction can be used. This is

expression ::= *unitary clause*

which carries out an assignment only if the variable delivered by the expression conforms. The clause as a whole then delivers the boolean value TRUE. Otherwise no assignment is performed and FALSE is delivered. The boolean result should normally be checked by using the conformity as the condition of an IF clause.

If the symbol ::= is replaced by :: the conformity tests are performed in the same way, and boolean or integer results delivered, but no assignments take place.

9.1.6 De-voiding

This coercion is, in effect, a rationalization of the way in which GOTO and SKIP may be used.

A GOTO-clause does not deliver a result, and might be thought unusable in a context such as

```
sqrt(IF x > 0 THEN x ELSE GOTO label FI)
```

The parameter of sqrt must be a REAL, and yet, if x is less than 0, a GOTO will be obeyed. To satisfy the mode rules it has to be said that GOTO can be coerced to any mode—in this case REAL. This is a pure technicality. After all, when the GOTO is obeyed, nothing whatever is delivered to sqrt.

A variation of this situation occurs if a procedure takes a parameterless procedure parameter. For example,

```
PROC f = (PROC REAL p) : (--- p ---) ;
```

In this case, 'GOTO label' can be supplied as the actual parameter for p. It is reasonable that GOTO label can be treated as though it had mode PROC VOID, but it may seem less convincing that it should be used as a PROC REAL. However, this can prove useful in practical programming. Notice that the GOTO would not be obeyed until p was deprocedured within the body of f. Then, when a REAL was expected, the GOTO would cause a jump and in fact deliver nothing.

Similar considerations apply to the use of SKIP, a dummy clause which does nothing and delivers nothing. It can be coerced to the mode of any variable or basic object. As an example, consider

```
x := CASE i IN a, b, SKIP, u, v ESAC
```

The SKIP is used merely to fill a position in the list. Care should be taken to avoid an assignment of SKIP being actually performed, as the result is always undefined and may even cause the program to fault while it is running.

9.1.7 Voiding

Voiding is the act of discarding a value. Two situations in which this occurs have already been described. The object delivered by a unitary clause is thrown away if it is a step followed by a semi-colon, and the object delivered by the body of a procedure is thrown away if no result is specified in its heading. To describe these occurrences formally as 'voiding' coercions is necessary for cases can arise where voiding causes some positive action to be taken before a result is discarded. Suppose, for example, that 'consolidate' is the name of a parameterless procedure of mode PROC VOID. In the following context,

```
--- ; consolidate ; ---
```

we would expect the procedure to be obeyed, not simply thrown away. This comes about as a result of the way voiding works, as objects of mode PROC VOID are deprocedured first. The same goes for an object of mode REF PROC VOID (or REF REF PROC VOID, etc), which will be dereferenced and deprocedured. However, an object of mode REF PROC VOID is not obeyed before voiding if it is the result of an assignment. For example, in the assignment f2 := random (section 9.1.1), we have seen that random cannot be obeyed before assignment, and neither is it obeyed when f2, the result, is voided.

9.2 Types of context

9.2.1 Strong positions

When the context clearly demands a unique mode, every permitted form of coercion is applied in an attempt to obtain that mode. A few obvious examples are indicated below:

REAL k = *unitary clause*[1] ;
x := *unitary clause*[2] ;
IF *serial clause*[3] THEN a[*unitary clause*[4]]FI ;
OP(REAL, REAL)REAL £ = *unitary clause*[5] ;

Key:

1 must deliver REAL
2 mode of x, less one REF
3 BOOL
4 INT
5 PROC(REAL, REAL)REAL

If the context does not itself demand a unique mode, the mechanism of coercion cannot come fully into play, but the programmer can force any of the coercions described in section 9.1 by saying what mode is required. The construction is

mode VAL *unitary clause*

which can be used as a unitary clause. It delivers an object of the mode given, provided that the coercions necessary are of the allowed forms. This is most useful for controlling dereferencing where several levels of reference are involved. For example, assuming that a is a real array variable, consider the assignments in

REF REAL ai ;
ai := a[i] ;
(REF REAL VAL ai) := 0.0

After the first assignment, in which no coercions are applied, ai holds the variable a[i]. To assign a real to this variable, using ai, a forced coercion is necessary. Another example, also concerned with levels of reference, is shown in 10.2.

9.2.2 Reference positions

The ultimate modes of left-hand sides of assignments and conformity clauses must always begin with REF. The number of REF's is important and no dereferencing occurs. In such positions the only coercion applied is deproceduring.

9.2.3 Operand positions in expressions

The coercions to operands are dereferencing and deproceduring. These are applied far enough to match the modes of the operand(s) to those demanded by the operator.

9.2.4 Procedure calls

The actual parameters of a procedure are in strong positions, but the procedure itself may have to be reached by coercion. For example, consider

```
PROC(REAL)REAL f := sin ;
REAL x := 1 + f(pi/3)
```

Here f has mode REF PROC(REAL)REAL. It must be dereferenced at the call to give sin, which can then be applied to the given actual parameter. The coercions applied in this context are dereferencing and deproceduring.

9.2.5 Array indexing and field selection

An array can be indexed, and an array variable can be indexed, but a *reference* to an array variable (REF REF[ ]...) must be dereferenced once before the indexing can be carried out. Similarly, a reference to a structure variable must be dereferenced to a single REF before field selection can take place. In both cases, multiple REF's must be reduced to a single REF. The only other coercion applied in these contexts is deproceduring.

9.2.6 Coercion of alternatives to one mode

If a conditional or case clause delivers an object, it must be of one definite mode, whichever alternative is chosen. If the clause appears in a strong position, this mode is decided by the context, and all the alternatives will be coerced to it, but in other types of context, some uncertainty may appear to arise unless all the alternatives have already the same mode. If not, one mode must be found to which all alternatives can be coerced. The rule for this type of coercion is that one of the alternatives will, if necessary, be dereferenced and deprocedured until it is of a mode to which all the other alternatives can be (strongly) coerced.

a
```
INT i, j ;
REF INT ii := i ;
IF u THEN ii ELSE j FI := 4
```

b
```
REAL x ;
IF u THEN random ELSE x FI + 123.4
```

In *a*, ii is dereferenced so that the mode of the conditional is REF INT whether u be TRUE or FALSE. In *b*, the mode of the conditional is REAL. This will have been reached by deproceduring random and dereferencing x.

9.2.7 Result of a serial clause containing declarations

With the currently available Algol 68-R compiler, the result of a serial clause which contains declarations, but is not the body of a procedure, is subject to special restriction. It is always deprocedured and dereferenced, and if the result is an array, or structure containing an array, it is voided. The result is also voided if it is delivered by a conditional clause with alternatives of different modes.

10

Computing with References

References have always been used in computer programs as the means by which basic data is accessed. In any numerical computation, assignment of REAL to REF REAL and dereferencing of REF REAL to REAL is commonplace. In certain types of program, a need arises for carrying out manipulations one level further away from the basic data. The essential objects of manipulation become the *references* to basic data, and assignments of REF to REF REF become the order of the day. The work which gives rise to this form of manipulation is nearly always concerned mainly with the *structure* of data, and with transformations to structure. This chapter describes the language features designed to make reference manipulations really practicable.

10.1 Avoiding duplication of data

The word REF is commonly to be found in the specification of the mode of a field in a structure. It will be useful to set up an example of such a context:

```
MODE MAN = STRUCT(LONG BYTES name, INT age) ;
MODE DOG = STRUCT(LONG BYTES breed, REF MAN owner) ;

MAN m := ("MR SMITH", 70) ;
DOG d := ("SHEEPDOG", m) ;
```

Suppose that the word REF in the definition of the mode DOG were omitted. Then the second field of d would hold an object of mode MAN, obtained by dereferencing m. In other words, "MR SMITH" and 70 would be copied into d, and would duplicate Mr Smith's particulars. Subsequently, as Mr. Smith grew older, the assignment

```
age OF m PLUS 1
```

would update the original record but not the copy in d. As it is, the 'owner' field having mode REF MAN, it can be seen that Mr Smith's particulars are kept in one place only, and the assignment to d ensures that 'owner OF d' refers indirectly to the same data as m does. Avoidance of duplication of data is, in reality, the main justification for the use of REF.

10.2 IS and ISNT

The number of operations which can be carried out on references (as such) is quite limited. Arithmetic operators like +, −, etc cause dereferencing to take place, and so also does the equals operator, =. However, in reference manipulation work it is essential to have some means of testing whether two references are the very same; the construction used for this purpose is a unitary clause of the form

expression IS *expression*

where the expressions deliver the references. The clause delivers a boolean result, TRUE if the references are the same and FALSE otherwise. (ISNT gives the opposite result.)

To see the distinction between = and IS quite clearly, consider the declarations

```
INT i := 8 ;
INT j := i ;
```

For numerical purposes, i and j are now equal and the expression i = j delivers the result TRUE. Both variables refer to 8, but they are nevertheless two different variables. The fact that 8 is *copied* from i into j by the second assignment brings this point fully home. The test

```
i IS j
```

detects that the references are distinct and delivers FALSE. On the other hand, consider the identity declaration

```
REF INT h = j
```

which introduces a new identifier h for a reference which already exists; h and j identify the same working space in store, and the test

```
h IS j
```

therefore gives TRUE.

In the above examples, the objects being compared both have the same mode, REF INT. The modes can be different provided they can be equalised by dereferencing one side or the other a suitable number of times. (If not, a fault occurs.) For example, consider the clause

```
owner OF d IS m
```

in relation to the piece of program given in section 10.1. Here the modes are different. The mode of 'owner OF d' is REF REF MAN, whilst that of m is REF MAN. The former is therefore dereferenced once automatically, so that the test effectively decides whether 'owner OF d' *refers* to m or not. It does, and the result delivered is therefore TRUE.

The last example shows one of the important uses of reference tests. At the level of the basic data, the test would be

```
name OF owner OF d = name OF m
AND age OF owner OF d = age OF m
```

which is undoubtedly clumsier, and would be even more so with a more complicated data structure. Testing the basic data can always be avoided by working at a reference level, but the

two are *equivalent only if the basic data is unduplicated.* The principle should be to create one reference only to each object of data, and then use this unique reference consistently throughout the problem.

Reference work is by no means without pitfalls, and it seems worthwhile to give an example of what can go wrong if insufficient thought is given to modes. Consider the following continuation of the dog program:

```
DOG d1 := ("ALSATIAN", m) ;
DOG d2 := ("LABRADOR", m) ;
```

Do both of these dogs have the same owner? Clearly so, but the construction

```
owner OF d1 IS owner OF d2
```

delivers the result FALSE. The reason is that 'owner OF d1' identifies one field of the space reserved for d1, whilst 'owner OF d2' identifies the corresponding space in d2, which is distinct. The modes are both REF REF MAN, and this provides the clue. To test whether the two fields contain the same REF MAN, it is necessary to carry out the IS test at the REF MAN level. This can be done by using VAL to force the required mode, thus

```
(REF MAN VAL owner OF d1)IS owner OF d2
```

Only one side need be forced, as the other will then be automatically coerced to the same mode. The result will now be TRUE.

When performing multiple tests, such as

```
(x IS y)AND(u ISNT v)
```

brackets are needed. It is important to remember that 'x IS y' is a unitary clause, and unitary clauses cannot be used as primaries in expressions unless they are bracketed. The words IS and ISNT are not operators. Like the assignment symbol, they bind more weakly than any operator.

10.3 Generators

To see why generators are important in reference manipulation work, it is helpful to draw a parallel with ordinary numerical computation. In the course of any long calculation, numbers are continually being created and destroyed. They are created as results of expressions, can be held for future use by assignment to variables and are destroyed automatically whenever fresh assignments cause them to be overwritten. The processes required for reference manipulation are exactly similar. We need the ability to create new references freely, hold them temporarily in variables and have them disappear automatically when (and only when) they are no longer required. Local generators, the only kind discussed up to this point in the Guide, do not meet these requirements fully for two closely related reasons. Firstly, if a locally generated reference has been assigned to a variable, eg

```
owner OF d := LOC MAN := ("MR YOUNG", 18) ;
```

the *space* it represents does not disappear when another assignment is made to that variable, such as

```
owner OF d := m
```

The space occupied by ("MR YOUNG", 18), though now inaccessible, is reserved throughout the serial clause which defines the scope of LOC MAN. In certain types of work, this may cause a program to run out of data space unnecessarily. Secondly, the fact that the space created by a local generator is subject to scoping prevents one from using procedures for grafting new references on to a global data structure. Space generated locally in the body of a procedure ceases to exist upon exit from the procedure, and if there were no alternative, a powerful technique would be denied to us. Both of these objections are overcome by the use of *global generators*, sometimes known as 'heap generators' because the heap system of storage allocation is used for their implementation.

A global generator resembles a local generator in form, but the word LOC is omitted. References created by global generators, like numerical values, do not go out of scope. The space continues to exist for as long as required. However, once it has become inaccessible, it is automatically made available for re-allocation by the system. Such treatment is usually known as *garbage collection*, and it occurs when the program would otherwise run short of data space. The programmer need not be aware that it happens; from his point of view, global generators are safe and trouble-free, unhampered by scoping restrictions. However, the heap storage allocation system carries an overhead, which applies to the whole program if a global generator or a flexible array (5.3) appears anywhere within it.

10.4 Chaining of data and the use of NIL

A characteristic feature of most non-numerical computing is the unpredictable amount of space required for data. For example, when processing the state of a chess board, a natural language sentence, or an algebraic expression, the extent of the data base keeps changing as the work proceeds. What we require is a means of extending the data structure without copying it all out. The only way of doing this dynamically is to chain the data together by using structures with reference fields pointing to similar structures which can be generated to any extent required. In this section of the present chapter, an indication of the technique will be given, although the subject of list-processing as a whole is too big for a full treatment.

The simplest type of chain can be built from a structure with two fields, one for an item of data and the other as a pointer to the next. The mode required for building a chain of integers, for instance, is

```
MODE LINK = STRUCT(INT item, REF LINK next)
```

Let us use this to build a chain containing as items the integers 1, 2, 3 and 4. One obvious way to proceed is to declare a variable for each link in the chain, thus

```
LINK a, b, c, d ;
a := (1, b) ;
b := (2, c) ;
c := (3, d) ;
item OF d := 4
```

This has left the REF LINK field of d unset—for we cannot go on for ever! However, it is bad practice to work with unset variables; what is needed is a dummy reference. The word NIL is provided for just such a purpose. Its mode begins with REF, though it does not refer to any actual

object at all, and it can never be dereferenced. The chain can now be terminated more satisfactorily by

```
d := (4, NIL)
```

An alternative, to be preferred in practice, is to declare a null reference with the specific mode required, and to use this in place of NIL. This can be done by writing

```
REF LINK empty = NIL ;
d := (4, empty)
```

The reason why this is preferable is that it enables us to test for the end of a chain by means of the clause

```
next OF d IS empty
```

The mode of 'empty' causes 'next OF d' to be dereferenced to REF LINK, and (in this case) the result TRUE will be obtained.

The method of chain building outlined above is of very limited application. It is easy to see that it cannot be generalized for chains of unknown length, as this would entail declaring an unknown number of variables. The main objective of chaining is therefore vitiated. A solution to this problem can be found by applying two principles:

1 The space required for new links must be generated without declaring identifiers. (Global generators are recommended.)

2 For referring to such space, at least one variable of mode REF REF LINK must be declared.

Using these principles, a chain of four integers can easily be constructed by generating the links in reverse order, thus

```
REF LINK chain := empty ;
chain := LINK := (4, chain) ;
chain := LINK := (3, chain) ;
chain := LINK := (2, chain) ;
chain := LINK := (1, chain)
```

Before proceeding further, it may be found helpful to analyse the modes in the above assignments.

```
    chain     :=     LINK     :=    ( i ,      chain )
      |              |               |          |
REF REF LINK      REF LINK          INT      REF LINK
```

In the collateral, chain is dereferenced to REF LINK to satisfy the mode specified for the second field of a LINK structure (see the original mode declaration).

Now that the identifiers a, b, c, d have been eliminated, the repeated assignments can be carried out in a loop. The final solution is then as follows:

```
MODE LINK = STRUCT(INT item, REF LINK next) ;
REF LINK empty = NIL ;
REF LINK chain := empty ;
FOR i FROM 4 BY -1 TO 1 DO
chain := LINK := (i, chain)
```

As a simple model of how a chain may be processed, let us consider the problem of adding 1 to each of the items in the above chain. These items can be reached by the expressions

```
item OF chain
item OF next OF chain
item OF next OF next OF chain
item OF next OF next OF next OF chain
```

Once again, it is advisable to study the modes involved, starting with the mode of 'chain', which can be written out as

```
REF REF STRUCT(INT item, REF LINK next)
```

The effect of field selection is to remove the REF at the front, and transfer the remaining REF to the selected field. Hence the mode of 'next OF chain' is REF REF LINK, which is exactly the same as the mode of 'chain' itself. However many times 'next OF' is applied, the same mode results. Finally, 'item OF' again removes one REF and transfers the other to the selected field, giving a REF INT. We see, then, that all four of the expressions are simply integer variables holding the successive items of data. To complete the problem, therefore, we could write

```
item OF chain PLUS 1
item OF next OF chain PLUS 1
etc.
```

though it would obviously be better, and more general, to use a loop. This can be done by introducing a second variable, with the same mode as 'chain', and using it to move progressively down the chain as each item is processed. Thus,

```
REF LINK here := chain ;
TO 4 DO (item OF here PLUS 1 ;
            here := next OF here)
```

The items in the chain are now 2, 3, 4, 5, and 'here' is left referring to 'empty'. For a chain of unknown length, this can be used as the test for stopping. Such a test has already been described; it can now be incorporated in the loop:

```
WHILE here ISNT empty DO
    (item OF here PLUS 1 ;
     here := next OF here)
```

To complete this brief look at list-processing techniques, one final problem remains. We have seen how a chain of data can be constructed, and how processed. It is equally important to be able to release the space when no longer required. One assignment suffices,

```
chain := empty
```

There is now no way of accessing the various links which were generated, and to all intents and purposes the space they occupied has ceased to exist. In actuality (as global generators were used), it becomes available for further constructions. The amount of data space needed by a list-processing program is no greater than required for the items and linkages in use at any one time.

10.5 Assignment of scoped values

When computing with references, we are faced with a scoping complication which does not arise in elementary programming of the Algol 60 type. For suppose that ii is a REF REF INT and i is a REF INT, and consider the assignment

```
ii := i
```

Both ii and i have scopes depending on the levels of the program in which they were declared. If the scope of i *includes* the scope of ii, no trouble can arise, but if the scope of i is *smaller* than the scope of ii, the assignment cannot be satisfactory. The definition of assignment is:

> If B is assigned to A, then A refers to B for as long as A exists, or until a fresh assignment is made to A.

This rule cannot be satisfied if the scope of B is less than that of A. For this reason, even quite temporary assignments of local references to global variables must be avoided.

11

Recursion

Any language construction which defines a new name opens up the possibility of circularity—a recurrence of the name within its own definition. This is known as *simple recursion.* If more than one definition is involved in completing the circle, the definitions are described as *mutually recursive.* Recursion can often be usefully exploited, as we shall see, but certain rules must be observed. In Algol 68-R, any new identifier, operator or mode name must be the subject of a declaration before it is actually used. The purpose of this chapter is to show that such a restriction does not debar the use of either simple or mutual recursion.

11.1 Recursive procedures

Recursive procedures enable us to express repetitive processes without the use of explicit loops. It is usual to take the definition of factorial(n) as an example. In section 4.4, several ways of calculating a factorial using DO loops were shown. The following method uses recursion instead:

```
PROC factorial = (INT n)INT:
(IF n = 0 THEN 1 ELSE n * factorial(n - 1)FI)
```

The call of factorial(4) causes a call of factorial(3) to take place before the evaluation of factorial (4) is completed, and so on down to factorial(0) which is 1. There must always be a final escape of some kind to prevent recursion from going on perpetually. Simple recursion in procedure declarations should be confined to the abbreviated type of declaration

PROC *identifier* = *procedure denotation*

For mutually recursive procedures, however, a difficulty presents itself. Suppose that we wish to declare a procedure p which calls a procedure q and that q calls p. At first sight, neither can be declared first without using an identifier which has not yet been declared. To show the method of solution, let us assume that q has mode PROC(REAL)REAL. Then, instead of declaring and using q itself, we declare a procedure *variable* qq which can be 'used' before having an actual procedure assigned to it. This is legitimate as long as the assignment has taken place before qq is actually obeyed. The form the program would take is

```
PROC(REAL)REAL qq ;
PROC p = denotation using qq ;
qq := denotation using p ;
--- ;
call of p
```

It will be seen that no identifier is used before it has been declared, and that by the time p is called, and its denotation is actually obeyed, the assignment to qq will have been carried out.

11.2 Recursive modes

The definition of a mode name can include a reference to itself, as shown in the declaration for LINK used in chapter 10:

```
MODE LINK = STRUCT(INT item, REF LINK next)
```

Before deciding on such mode definitions, it is advisable to check that it would be possible to declare an object with the specified mode. It is plain to see that a mode such as

```
MODE INF = STRUCT(CHAR letter, INF tail)
```

would be impossible, as the declaration of an object

```
INF impossible = ("A", ("B", ("C", .....
```

would never terminate.

LINK is an example of a mode name whose declaration is *simply* recursive. Mutual recursion presents the same problem for mode declarations as for procedures, for if one mode uses another which in turn uses the first, neither of them can be declared before the other if we are not to violate the rule about declaration before use. To overcome this problem, a vestigial type of declaration is allowed, having the form

```
MODE modename
```

with no identity symbol or right-hand side. This acts in lieu of a proper declaration for the purpose (and only for the purpose) of satisfying the rule about declaration before use. Eventually the new mode name must be properly declared. No actual objects can be declared with the new mode until this has been done.

As an example of how this works, we give below definitions of CELL and ITEM which can be useful for programming with tree-like data structures:

```
MODE ITEM ;
MODE CELL = STRUCT(ITEM head, tail) ;
MODE ITEM = UNION(INT, REF CELL)
```

CELL is a generalization of LINK defined earlier. Instead of having an INT field and a REF CELL field, each field can have either of these modes. This makes it possible to model a data structure which splits into many branches; the chain divides into two whenever the head and tail fields are both REF CELL's. For processing such data, it is found that recursive procedures are ideal, once the knack of writing them has been acquired. Below is a procedure which makes a copy of a complete data structure, irrespective of how many branches it has.

```
REF CELL empty = NIL ;
PROC copy = (ITEM a)ITEM :
(REF CELL b ;
IF b ::= a THEN
   IF b IS empty THEN empty
   ELSE CELL := (copy (head OF b), copy (tail OF b))
   FI
ELSE a FI)
```

The advantage of recursion in this type of problem becomes fully evident if we attempt the same thing with ordinary DO loops.

12

Transput

The word transput is short for input/output and applies to the transfer of data into or out of the computer under program control. In this chapter we are concerned with *character transput*, meaning that the data outside the computer is a stream of characters. Inside, it is represented differently and the necessary conversions are performed by transput procedures such as read and print. The programmer need be concerned only with the external representations. Each individual value, such as an integer or real number, is represented by a short sequence of characters, and the forms which these sequences can take will be described at some length.

The style of data representation is to a lesser or greater extent adaptable, depending on the transput procedure selected. The simplest procedures to describe and use are read and print; read is particularly adaptable as it accepts numerical data in free format. Numerical values do not need to have a fixed number of figures, and spaces or new lines can be freely interspersed between items. By contrast, print can only produce standard forms. A real number is always printed to full accuracy and an integer invariably takes the amount of space needed for its largest possible value. Spaces and new lines are inserted by means of layout procedures.

To print results in special styles, with varying numbers of figures or with a complicated scheme of headings and layout, the system known as *formatting* should be used. This is extremely flexible and well repays the small effort required in learning its special language. Instead of print, the procedures 'outf' or 'out' are used. The difference between these alternative procedures is that outf must be supplied with the required format as one of its parameters, whilst out uses a format set up in advance. A format can specify a different style of printing for every value and include instructions about layout.

Although mainly useful for output, formats can also be used for input with the procedures 'inf' or 'in'. The style and layout of the data presented to the computer must then conform exactly to the specification. This extra checking may often be desirable, particularly for data which has been generated by a machine, but for general use the most flexible combination is input in free format using the procedure 'read' and formatted output by 'outf' or 'out'.

Most simple problems can be programmed to use only one channel each of input and output. The procedures read and print automatically select standard channels, but the formatted procedures have parameters for selecting any one of a number of channels. For the standard channels, the selection parameter is given as the library identifier 'standin' for input and 'standout' for output. In this Guide, no other options are described. The use of the standard channels is assumed throughout.

12.1 The transput parameter

Every transput procedure has a parameter which lists the items to be transput. For output, this transput parameter is described as the *value-list* and consists of a list of the values to be printed, or expressions which deliver them. For input, the parameter is a *variable-list* giving destinations for the values read in. The items in a variable-list can be plain variables or—less commonly—expressions which deliver variables.

As an example, let us read in two integers and print their sum and difference with headings.

```
INT i, j ;
read ((i, j)) ; {note the double brackets}
print (("SUM =", i + j, newline, "DIF =", i – j))
```

The item 'newline' takes a new line where it occurs; this method of inserting layout is detailed in 12.3.3. Typically the output from this program would appear as

```
SUM =      +346
DIF =        +8
```

The double brackets arise from the need to bracket lists. The outer brackets are those belonging to the procedure call and the inner ones are part of the parameter. A list of only one item does not need these extra brackets; thus the call of read in the above example could have been expressed as two separate calls, read(i); read (j)—bracketed lists are only a way of abbreviating repeated calls of the transput procedure. There is no constraint on the number of items or mixture of modes in the list.

When reading data, the correct number of values for the variable list must of course be present in the data stream, in the right order and represented in the appropriate style for the mode of the value. For example

```
STRUCT ([1:3] REAL p, INT i)s ;
read(s)
```

expects three reals and one integer on the data-stream, all of which will be input by the call read(s).

References cannot be transput. Just as there are no denotations for references in a program, there are no representations externally. When a variable is used in a value-list, however, the fact that it has a REF mode does not cause trouble. It is automatically dereferenced. Such dereferencing does not occur inside arrays or structures; to print the two integers from an object declared as

```
STRUCT(INT direct, REF INT indirect)object ;
```

it is necessary to decompose the structure and write

```
print((direct OF object, indirect OF object))
```

A call of print(object) would cause a fault, for although 'object' itself is dereferenced automatically, the reference field within it is not. (It would be misleading if a structure of this kind were printed as though the two fields both had mode INT.)

The question of dereferencing will occasionally arise in connexion with input. If the variable is a multiple reference, it will be stripped of all REF's but one. As an example, consider

```
INT i ;
REF INT ii := i ;
read(ii)
```

The effect of this will be to dereference ii *once*, giving i. The integer on the data stream will therefore be put into i. This only works by virtue of the previous assignment, for there must always be some actual space to receive the value; no input procedure will generate space for itself.

12.2 The read procedure

This section includes the permitted forms for values in the data stream when input by the read procedure. It must be emphasized that these forms apply *only* to data streams—they are not to be confused with denotations for constants in a program (see 1.1).

Most values in a data stream are composed of several characters, so it is important to know the rules which decide where one item ends and the next begins. These are particularly simple for numerical values, several of which can be written on the same line or card provided that they are separated by one or more spaces. The end of the line must not split a value, as it is an alternative way of indicating the end of one value and the start of the next. The general principle is that the read procedure will read as many characters as it needs to complete a value. It does this by looking to see if the next character would be a legal continuation or not. If not, the value is terminated and the character which would have been illegal is left unread. However, it is not discarded; it becomes the first character of the next item. For example, if the data stream contained the integers 24 and 25 separated by one space, the space would indicate the end of 24 and be read as the first character of the value 25. Preliminary spaces and new lines are ignored for numerical values, so any amount of layout can be placed between such items.

Especial care is necessary when reading literal character sequences, that is to say items of modes BYTES and [ ]BYTES, including the LONG forms, and STRING and [ ]CHAR. For these items spaces are read as characters and all the characters of the item must be on the *current line or card.* This rule has an important consequence, for the reading position will very often be at the end of a line after reading an item of data. It will *never* be at the beginning of a new line after reading in a value. If our literal character sequence is on the next line, the reading position does not go to the new line automatically as it would for a numerical value. The position must be moved to the start of the new line explicitly, by including 'newline' at the appropriate place in the variable-list or by writing read(newline) as a separate call. It is worth showing an example of this. Suppose we have to read alternate BYTES and INT values in pairs on separate lines:

```
JACK  30
JILL  26
etc
```

This could be programmed as

```
[1:n]STRUCT (BYTES name, INT age) person ;
FOR i TO n DO
read ((newline, person[i]))
```

It is also instructive to consider an alternative arrangement, in which the fields are interchanged and separated by one space:

30 JACK
26 JILL
etc

This character stream could be read by

```
FOR i TO n DO
read ((age OF person[i], space, name OF person[i]))
```

The 'space' in the variable-list is important for moving the reading position to the start of the name. No newline is necessary, because it is taken automatically when a numerical value is expected as the next item.

12.2.1 Forms required by the read procedure

In the following, curly brackets are placed round components which may optionally be omitted, and 'layout' means spaces, new lines or new pages.

REAL

All layout is ignored until the first non-space character is read. This must be a sign or a digit.

Form {sign}decimalnumber{&{sign}digits}

where decimal number is a sequence of digits possibly including a point. The symbol '&' means 'times 10 to the power'. Spaces may optionally be included after +, −, or &. Elsewhere a space or new line will terminate the number. The limits of a real are approximately 5&−78 to 5&76.

Examples 123 123&4 123.4 −123.45&−6

COMPLEX

Form REAL{layout}?{layout}REAL
Example 123.4&−5 ? −23.4&−5

INT

All layout is ignored until the first non-space character is read. This must be a sign or a digit.

Form {sign{spaces}}digits
Examples 8388607 (largest positive INT)
−8388608 (most negative INT)

LONG INT

As for INT, with limits $\pm (2^{46}-1)$

BITS

All layout is ignored until the first non-space character is read.

Form 24 0's or 1's
Example 101010101010101010101010

BOOL

All layout is ignored until the first non-space character is read. This must be T (for TRUE) or F (for FALSE).

CHAR

If the reading position is at the end of a line or page, read takes a new line or page first. (This does not apply to arrays of characters—see separate headings below.) Space is a character, but newline is not.

Form One character (not in quotes) from the table displayed under CHAR in Appendix 1. Ignore the remarks before and after the table, which apply to denotations only.

BYTES

If the reading position is at the end of a line (or page), read does *not* take a new line (or page) first. This must be done independently, for example by read(newline).

Form 4 characters
Example 100%

LONG BYTES

As for BYTES, but 8 characters.

LONG LONG BYTES

As for BYTES, but 12 characters.

STRING

Form Any number of characters up to the end of the current line
Example USERS GUIDE TO ALGOL 68-R

An object of mode STRING is an array of characters with flexible bounds. The characters on the data stream are therefore read up to the end of the current line, after which the lower bound of the string will be 1 and the upper bound will be the number of characters read, regardless of what it was previously.

If the reading position is at the end of a line and the string starts on the next line, read(newline) must be carried out first. Otherwise no characters will be read and the upper bound will become zero.

The use of STRING entails heap storage; in work of a mainly numerical nature the mode [ ]CHAR without flexible bounds is usually adequate, and more efficient.

ARRAY OF CHARACTERS

Form The number of characters in the array, or less (see below)

The characters must be on the current line, as no new lines are taken when reading arrays of characters. Sufficient characters will be read to fill the array if possible. If the end of the line is reached first, the upper bound will be reduced to suit the number of characters actually read.

If the reading position is at the end of a line and the array of characters starts on the next line, read(newline) must be carried out first. Otherwise no characters will be read, and the upper bound will be reduced to one less than the lower bound, eg 0 if the lower bound was 1.

OTHER ARRAYS

Apart from STRING and [ ]CHAR, the reading of any array is equivalent to a succession of reads, one for each array element. Thus,

```
[3:5]BYTES b ; read(b)
```

is equivalent to read((b[3], b[4], b[5])). In this example, all 12 characters must be consecutive on the current line, but for numerical arrays, the elements can always be separated by

spaces or new lines. For multi-dimensional arrays, the elements are taken in lexicographic order, eg

[1:2, 1:3]INT a ; read(a)

is equivalent to read((a[1, 1], a[1, 2], a[1, 3], a[2, 1], a[2, 2], a[2, 3])).

12.3 The print procedure

12.3.1 Items printed with layout

For the following items, if there is not enough room on the line, a new line is taken automatically before the value is printed.

REAL

Except at the beginning of a line, each complete value is preceded by one space. The length is 18 characters, or 17 at the start of a line.

Examples +1.0123456789& +0
−1.0123456789&−12

COMPLEX

Except at the beginning of a line, the complex value is preceded by one space. The length is 37 characters, or 36 at the start of a line.

Example +1.0123456789&+12 ?−1.0123456789& −2

INT

Except at the start of a line, the integer is preceded by one space. The length is 9 characters, or 8 at the start of a line, of which up to 7 are digits.

Examples −8388608
+8388607
−123

LONG INT

As for INT, but 7 digits longer.

BOOL

Except at the start of a line, the value T (for true) or F (for false) is preceded by one space.

BITS

Except at the start of a line, the item is preceded by one space. Then 24 binary digits are printed consecutively.

ARRAYS OF ABOVE ITEMS

Arrays are printed as a succession of elements, in lexicographic order of indices. For example,

[1:2, 1:2]INT a ; print (a)

is equivalent to print ((a[1, 1], a[1, 2], a[2, 1], a[2, 2])). For strings and arrays of characters, see 12.3.2.

CHAR

For a *single* character, a new line is taken automatically if the printing position is at the end of a line. The character is then printed. For arrays of characters, see 12.3.2.

12.3.2 Items printed with no layout

Items of the following modes are printed as plain sequences of characters on the current line. No new line is taken automatically, and overshooting the end of the line will cause a fault. (The number of characters is shown in brackets.)

BYTES (*4*)
LONG BYTES (*8*)
LONG LONG BYTES (*12*)
ARRAYS OF BYTES (*according to size of array*)
STRING (*according to current bounds*)
ARRAY OF CHARACTERS (*according to size of array*)

12.3.3 Layout procedures

The following procedures may be used as items in the variable-list of read or the value-list of print.

space

Moves the reading or printing position ahead by one character, without reading or printing. Each new line of output is initially filled with spaces, and the gap in printing will therefore contain a space already, unless printing has already been carried out and backspace has been used. Under these circumstances, print(space) does not overwrite the old character, but print(" ") does. For reading, space has the effect of ignoring one character.

backspace

Moves the reading or printing position back by one character, enabling a character to be read twice or overprinting to be carried out. When overprinting, the old character is replaced by the new one on the printed output.

newline

Moves the reading or printing position to the start of the next line. The call read(newline) is used mainly for preparing to read an array of characters on the next line, or for ignoring comment on the remainder of the current line.

newpage

Moves the reading or printing position to the start of the first line on the next page, ie causes a new page to be taken when printing. The call read(newpage) is only applicable when reading data previously output with pages.

12.4 Formatted transput

The remainder of this chapter deals with formatting, which is most easily introduced in terms of output. Input can be related to the description afterwards, for it works on exactly the same lines and uses the same notation.

Two procedures are available for formatted output, 'outf' and 'out'. For the time being we shall concentrate on outf, which takes three parameters,

standout— the library identifier to be supplied in order to select the standard output channel,

a *format*— usually supplied as a format denotation, which is bracketed between a pair of $ symbols,

a *value-list*— expressed in the same form as the transput parameter for the print procedure (section 12.1).

The general idea is as follows. The format can be thought of as a program of instructions which outf must obey in synchronism with its progress through the value-list:

outf(standout, $ ... , ... , ... , ... $, (... , ... , ... , ...))

progression through the format →

progression through the value-list →

The items between commas in the value-list can be of mixed types, including arrays and structures, and each item can therefore produce one or more basic values. Items separated by commas in the format denotation are known as *pictures* and for every basic value printed, one picture is obeyed. Each picture describes

a the style of representation of the value, such as the number of digits before and after the point; this is known as the *pattern* for the value

b any layout to be done at the same time as the value is printed, such as headings, spaces or new lines before or after the value itself; these are described as *insertions* and placed where they are wanted in the picture. In formatted transput, new lines are never taken automatically. New line insertions must be used to ensure that no output value overshoots the end of a line.

A picture thus comprises a pattern and insertions. When a value is ready for output, the whole of the corresponding picture is obeyed. If a picture happens to contain insertions but no pattern, the insertions are not obeyed until the next pattern is dealt with.

To illustrate the method, consider the problem of reading an array and printing its elements on separate lines. The program could be

```
[1:3]REAL v ;
read(v) ;
outf (standout, $l<1.3>, l<1.3>, l<1.3>$, v)
```

and the output would appear in the form

```
1.000
1.414
1.732
```

The format here contains three identical pictures, one for each value. The use of italic type has no significance except to make a clear distinction between *l* and 1, and to distinguish the special formatting symbols. In actual program preparation, ordinary characters must be used.

The example uses the insertion *l* which stands for new line, and the pattern <1.3> which specifies 1 digit before the point and 3 after it. Repetition of the same picture can usually be avoided, and in this case very simply. The call of outf could equally well have been

```
outf(standout, $ l<1.3> $, v)
```

with only one picture in the format. If the pictures in a format are exhausted before all the values have been dealt with, the format starts again at its first picture. Care must be taken always to ensure that the format is *completed* a whole number of times in any one call of outf. If it is left only partly completed, any subsequent attempt to use the format—as would occur if outf were in a program loop—would cause a fault. This is intended to guard against mistakes in format writing, which can cause the pictures to get out of step with values.

The format shown above could be used to input the same array of reals

```
1.000
1.414
1.732
```

since input is compatible with output unless otherwise stated. The procedure corresponding to outf is inf, and its call would be

```
inf(standin, $l<1.3>$, v)
```

However, it should be realized that this imposes tight constraints on the way the data is represented—for that is its purpose. The reading of each number would be preceded by the taking of a new line, and it would be essential to ensure that the reading position was on the immediately preceding line before the first number was input. The values themselves would have to start with a space or a minus sign, have 1 digit before the point and at least 3 afterwards. Only 3 digits after the point would be read; any further digits would be ignored because input of the next value begins by taking a new line. In spite of the constraints, formatted input has some valuable uses. It can act as a check on mass-produced data, and it can be used for recognizing control messages in a convenient way (section 12.8).

Within a format denotation, the ordinary rules of program layout still hold good. Spaces and new lines are ignored, though within string quotes spaces will, of course, be treated as characters.

12.5 Simple number patterns, insertions and replication

For transput of integers and reals, there are four simple patterns, exemplified by

<5> an integer pattern, describing a 5-digit integer preceded by space or minus. The number 5 is only an example, but must always be numerical—a variable cannot be used. This integer pattern cannot be used for transput of a real value.
Examples −12345
−2

<3.2> a real pattern, describing a value with three digits before the point and two after. The sign is represented by space or minus.
Examples −123.45
0.10

<1.2&1> a real pattern with an exponent part. The exponent consists of &, plus or minus and one digit. Exponents cannot have more than 2 digits.
Example −3.48&+2

<3&2> a real pattern with an exponent part but no decimal point.
Example −123&+12

The pattern for a complex number is made up of two real patterns separated by *i*, which is the format symbol corresponding to the 'imaginary' symbol (?) in the data stream. For example, <2.1>*i*<2.1> describes a number like 12.3?−45.6

In all cases, leading zeros are replaced by spaces and the sign is displaced to the right so that it comes immediately in front of the first digit.

Insertions for headings and layout can be placed before or after any pattern, but not inside the pointed brackets. The notation used is:

x space
y backspace
l new line
p new page
k sets character position (see text)
"..." the characters inside the quotes are inserted. Within the string, two quote symbols in succession are taken to mean that the quote character is to be transput. Two string insertions must not occur one after the other.

These insertions can be preceded by a (numerical) integer known as a *replicator*. For example 3*x* means 3 spaces, 3*lx* means 3 new lines and a space, 3*l*2*x* three new lines and 2 spaces. Before *k*, a replicator specifies a character position in the line; the first character in a line is at position 1. Thus 5*k* moves the reading or printing position forwards or backwards to position 5. The absence of a replicator is the same as a replicator of 1. Variable replication is described later in this section.

Although insertions can be replicated, patterns cannot be. On the other hand, whole pictures or sequences of pictures can be replicated to give the effect of looping in the format. The pictures to be replicated are enclosed in round brackets with a replicator in front. Thus, in the format

$ ... , 3(... , ...), ... $

there are three items, a simple picture, a composite picture and a simple picture. The composite

picture which contains two simple pictures is repeated three times over, and the format as a whole therefore handles eight values altogether. To repeat a single picture, brackets must still be used, as in

$ 2(...), 3(...)$

which deals with the transput of five values. Insertions can be placed before or after a composite picture. Before it, they are obeyed as part of the first picture (first time round) and after it as part of the last picture (last time round). For example,

$ *l* "MATRIX" *l* 7"–" *l* 2(<1>3*x*, <1>*l*) 7"–" $

is equivalent to

$ *l* "MATRIX" *l* 7"–" *l* <1>3*x*, <1>*l*, <1>3*x*, <1> *l* 7"–" $

and, given four small integer values, produces output of the form

```
MATRIX
------------
-2      4
 0     -2
------------
```

Formatting is particularly useful for output of arrays of one or more dimensions because layout can so easily be inserted between array elements without having to list them all separately in the value-list. With unformatted transput, the need for layout between elements usually does make this necessary.

Arrays are often of unknown size, because the bounds have been input or computed in the course of the program. In such cases, it may be necessary to have an unknown number of replications of a simple or composite picture. Variable numbers of insertions can also be useful. A variable replicator is written in the form

n(*serial clause*)

The letter *n* is a symbol of the formatting language, and must always be followed by a bracketed variable, expression or serial clause to deliver the integer value of the replicator. (If a negative value is delivered, there will be zero replications.) The serial clause is obeyed at the time when the picture is obeyed, no sooner. To emphasize this fact, variable replication is usually described as *dynamic*. Dynamic replication can be illustrated by repeating the last example for a matrix of size n by m. For the sake of definiteness, we assume that the array elements are one-digit integers, as before. The fancy lines above and below the matrix, now consisting of 5m-3 dashes, are retained as an example of dynamically replicated insertions.

```
[1:n, 1:m] INT matrix ;
read(matrix) ; {or assign values in some other way}
outf(standout, $ l "MATRIX" l n(5 * m - 3)"–" l
    n(n) (n(m)(<1>3x)l) n(5 * m - 3)"–" $, matrix)
```

The essential part of the example is

n(n)(*n*(m)(<1>3*x*)*l*)

which deals with the layout of the numerical data in rows and columns.

12.6 Patterns for booleans and characters

The pattern for any value must conform to the mode of that value. We have seen that integers and reals have their own distinct patterns; those for values of other modes are listed below. The inclusion of the modes STRING and [ ]CHAR is occasioned by the fact that sequences of characters are treated as 'basic values' for transput purposes. A sequence of characters is transput under the control of a single picture. For all other arrays, including arrays of BYTES, LONG BYTES and LONG LONG BYTES, each array element has a picture of its own.

mode	*pattern*	
BOOL	*b*	T or F
CHAR	*a*	one character
BYTES	4*a*	4 characters
LONG BYTES	8*a*	8 characters
LONG LONG BYTES	12*a*	12 characters
STRING	*t*	on output, the number of characters in the string, and on input, all characters from the current reading position to the end of the line.
[ ]CHAR	*t*	on output, the number of characters in the array, and on input, sufficient characters to fill the array unless the end of the line comes first. The upper bound is then reduced automatically to suit.

The use of *t* is equivalent to unformatted transput, as the pattern gives no explicit indication of the number of characters to be input or output. The main disadvantage of this is apparent when we recall that a fault is caused by attempting to output a sequence of characters which is too long for the current line. An alternative system for dealing with strings and rows of characters overcomes this difficulty. Instead of using *t*, we can use replicated *a*'s to form the pattern for any character sequence, and the pattern can be interrupted by insertions for layout. For example, an array of 100 characters can be transput with the picture

l 25*a l* 25*a l* 25*a l* 25*a*

in which there are four new line insertions. The remainder of the picture is a broken-up form of the pattern 100*a*. When this system is used for strings or arrays of characters, the sum of the *a*-replicators must be exactly equal to the size of the array, both for input and output; otherwise a fault will occur. The same system can be used to break up the patterns 4*a*, 8*a* and 12*a* in the transput of bytes.

It is instructive to consider the question of whether the repetition of 25*al* could be avoided, in the example given above, by some form of replication. In fact, it cannot be shortened. The temptation to write 4(*l*25*a*) must be resisted, as this is a way of expressing four separate *pictures*. It would therefore expect four separate arrays of 25 characters each in the transput parameter.

12.7 The *g* pattern

Formatting is less convenient for input than for output in most ordinary applications because every character in the data stream must be in exactly the right place and on the right line. Spaces and new lines in front of numbers are not ignored. It is possible to have the best of both worlds by using the *g* pattern, which causes a value of any mode to be transput as though by read or print. This does not necessarily make the format entirely pointless, for on input there are places where new lines must always be taken, and formatting may be worthwhile for this convenience alone. For example, consider how to read data such as

```
BRASS    23.5&-4
IRON     2.86&-4
WOOD     0.02
GLASS    15.0
```

into a variable declared as

```
[1:4]STRUCT(LONG BYTES substance, REAL coefficient)data ;
```

The format in

```
inf(standin, $l8a, g$, data)
```

takes the new lines required before reading the LONG BYTES items, and saves us from having to decompose the array in the variable-list. At the same time, the numerical values can be expressed in freely varying styles.

12.8 Choice patterns

Formatting provides the useful facility of representing boolean values and integers in ways other than T, F, 1, 2, 3 etc. Arbitrary character sequences can be specified instead, for example TRUE, FALSE, ONE, TWO, THREE.

To transput boolean values as arbitrary strings, the pattern is

b(" ... ", " ... ")

The characters in the first string denotation are taken as the representation of true, and the characters in the second represent false. For example,

```
BOOL female := FALSE ;
outf(standout, $ l "THERE ARE"<2>, xb("GIRLS", "BOYS")x
    "IN THE CLASS" $, (24, female))
```

gives the output

```
THERE ARE 24 BOYS IN THE CLASS
```

The same format could be used for input of the values embedded in this sentence. The wording would have to be reproduced exactly, with freedom only to vary the 2-digit integer and write GIRLS instead of BOYS. When a choice pattern is used for input, the second string must not

start with the first. For example

b("WITH", "WITHOUT")

would be useless, because the input stream would match the first string as soon as four characters had been read. The answer is to reverse the order and adjust the program accordingly.

Exactly the same technique can be used for integer values by means of the pattern

c(" ... ", " ... ", " ... ", etc)

the strings representing 1, 2, 3, etc. As in the case of boolean choice patterns, when used for input the first of the strings to match the characters in the data stream determines the value read in. Care must be taken to see that no string is the start of a later one. On output, the integer value must be equal to or greater than 1, and must not exceed the number of strings given, otherwise a fault will occur.

One of the most useful applications of choice patterns is to interpret control information on an input data stream, as illustrated by the following example.

```
REAL x, y ;
INT n ;
WHILE inf(standin, $ c("END", " ", "X=", "Y=", "RUN", "")$, n) ;
        n > 1
DO CASE n-1 IN
    SKIP, read(x), read(y), serial clause, read(newline) ESAC
```

Note the use of an empty string as the final item of the choice pattern. This acts as a default condition; if none of the preceding patterns has been matched by the input, the empty string is taken as the match and nothing is actually read. In this particular example, the resulting action is to take a new line. The variables x and y in the program will be set by 'commands' like

Y=0.538, (the comma and this comment will be ignored)

in the data stream. The serial clause will be obeyed by the command

RUN

and the program will be stopped by the command

END

Spaces before commands will be ignored, and so also will whole lines which do not start like commands.

12.9 The procedures in, out and format

The procedures inf and outf are simple to understand and use because the format for each call is supplied alongside the variable-list or value-list. This keeps related items close together and makes the program easy to follow. But there is a drawback. The format must be left in a completed state at the end of each call, and in some applications this can be too restrictive. This section describes an alternative method.

For output, the format is specified in advance by a call of the procedure 'format' which takes two parameters:

format(standout, *format*)

This call does not cause any transput to take place; it simply attaches the required format to the standard output channel. The procedure 'format' must not be called a second time until the format has been completed once or more times. The actual printing is effected by calls of

out(standout, *value-list*)

which work through the given format progressively, each call carrying on where the previous one left off. Input works in a similar way. The input format is attached by

format(standin, *format*)

and transput effected by calls of

in(standin, *variable-list*)

This method of formatting is most useful when the call of in or out occurs in a program loop and the format is set up outside the loop. For example, consider the problem of printing the integers 1 to max on separate lines, with a blank line after 5, 10, 15 etc, and assume for the present that max is an exact multiple of 5. Using outf, the programming is cumbersome (to say the least), as may be judged by comparing the two solutions:

layout of a table using 'outf'

```
FOR integer TO max DO
outf(standout, $ <3>n(1 + ABS(integer = integer '/' 5 * 5))l $, integer)
```

the same using 'format' and 'out'

```
format(standout, $ 5(<3>l)l $) ;
FOR integer TO max DO out(standout, integer)
```

The obscurity of the first solution is a result of the fact that only one value can be formatted at a time. A test must be devised so that a new line is replicated twice or once according to whether 'integer' is a multiple of 5 or not. The details of this test are an exercise in the use of operators. In the second solution, the format can embrace 5 values at a time even though 'out' deals with them singly.

If max is not an exact multiple of 5 in the second solution, the format will be left uncompleted. Any attempt to repeat this piece of program in a loop would then cause a fault, as the procedure 'format' cannot be called a second time if the format is not ready to start at its beginning. This difficulty can, if necessary, be overcome by artificially clearing the format before setting it up. The method is shown by

```
format(standout, clear format($ 5(<3>l)l $)) ;
FOR integer TO n DO out(standout, integer)
```

The procedure 'clear format' takes a format as its parameter, clears it, and delivers it ready for use from its beginning.

13

Formats Continued

The previous chapter provides sufficient information for the use of formatted transput in many applications, but for more challenging work there are further possibilities. The present chapter introduces the concept of formats as *objects*, similar in many ways to procedures. Objects have modes, and all the facilities of Algol 68 can be applied to them. In the last part of this chapter, the topic is extended in the opposite direction, by going into the finer details of format denotations. It will be shown that a format can be built up from units smaller than patterns. These are known as *frames*, and can be used to construct patterns for numerical values character by character, so providing many alternatives to the simple patterns defined in the previous chapter.

13.1 Formats as objects

A format is an object of mode FORMAT, which means that identifiers can be declared with modes such as FORMAT or REF FORMAT, that procedures can be declared with formats as parameters, and so on.

It should be understood from the outset that a format is a piece of program for which—as in the case of a procedure—the denotation constitutes the only copy. We shall see why this is important later. Meanwhile, consider the declarations

```
FORMAT style 1 := $ l<1.4>, 10k<1.4>, 20k<1.4>$,
                  style 2 := style 1 ;
```

These two variables refer to the one and only copy of the format. By contrast,

```
FORMAT copy 1 := $ l<1.4>, 10k<1.4>, 20k<1.4>$,
                 copy 2 := $ l<1.4>, 10k<1.4>, 20k<1.4>$ ;
```

provides two similar but distinct formats. The same considerations apply to identity declarations. No amount of naming or re-naming will produce extra copies of a format from one denotation.

The number of copies is important because a format in Algol 68-R cannot be attached simultaneously to more than one channel. In order to progress through a given format for input and output in parallel, a different copy is needed for each channel. This is illustrated by

```
[1:3] REAL vector ;
format(standin, copy 1) ;
format(standout, copy 2) ;
FOR i TO 3 DO
BEGIN in(standin, vector[i]) ;
      ... ;
      out(standout, vector[i])
END
```

However, the same copy of a format can always be attached to different channels at different times, provided it is used to completion (or not at all) each time. If, instead of interleaving the input and output of the elements of 'vector', we input the whole vector to complete the format before starting on the output, one copy will do; thus

```
format(standin, copy 1) ; in(standin, vector) ;
format(standout, copy 1) ; out(standout, vector)
```

In this example, inf and outf could well have been used, as inf is identical to a call of format followed by a call of in, and outf likewise. The above lines of program are identical to

```
inf(standin, copy 1, vector) ;
outf(standout, copy 1, vector)
```

It is safer, when sharing a format between different channels, to use inf and outf if possible, as these procedures are designed to attach the format to a specified channel afresh each time and use it to completion.

The ability to name formats, as we have now seen, enables the same format to be used in several contexts, and a context often required is within a larger format. A named format can then serve as a 'procedure' within the framework of the formatting language. The construction which allows this is

f(*serial clause delivering a format*)

This can be placed inside a format (with insertions before and after) to make one picture. It cannot be replicated, but it is a composite picture in the sense that the format delivered can contain several pictures. As an example,

```
STRUCT(INT day, month, year) date 1, date 2 ;
... ;
FORMAT datestyle = $ <2>, "/" <2>, "/" <2> x $ ;
outf(standout, $ l "THE CONCERT ON" f(datestyle),
    "IS POSTPONED UNTIL" f(datestyle)$, (date 1, date 2))
```

When using the *f* construction (as with the *n* construction described in 12.5), it is important to remember that a format is not obeyed until transput actually takes place. Any non-local variables used within *f*(serial clause) will therefore refer to the values they have at this time.

13.2 Scope of a format denotation

A format denotation may be written at one place in a program and obeyed at another. This is due to the fact that the denotation is not obeyed until transput actually takes place. Care must then be taken to see that the denotation is properly in scope when it is obeyed.

The critical parts of the denotation are the serial clauses in the *f* and *n* constructions, which are obeyed as part of the format. Such clauses are almost certain to use non-local identifiers, and these must be in scope at the relevant time. If not, the format denotation as a whole is out of scope and cannot meaningfully be obeyed. Consider, for example,

```
        BEGIN INT m := 3 ;
                format(standout, $ . . . n(m) . . . $) ;
                out(standout, . . . )
        END ;
wrong}  out(standout, . . . )
```

The format denotation uses the program variable m, whose scope is bounded by the BEGIN and END. The format can therefore be obeyed in the first call of out, but not in the second. This is, perhaps, a rather unlikely mistake to make. The following is more easily done, and equally wrong.

```
{wrong} PROC char display = (INT m)FORMAT:($n(m)a3x$) ;
        outf(standout, char display(p), s)
```

The error is in the procedure denotation, which attempts to deliver a format containing the parameter m. The scope of the parameter is restricted to the procedure body; outside it, m has no existence. The format delivered by this procedure is therefore meaningless. There are, of course, correct ways of achieving the desired effect, such as

```
[1:p]CHAR s ;
. . . ;
FORMAT char display = $n(p)a3x$ ;
outf(standout, char display, s)
```

This is correct in that p, assumed to be in scope in the first line, is still in scope when transput actually takes place.

13.3 Frames

There are occasions when the simple number patterns described in the previous chapter may prove inadequate for some particular purpose, especially on output. The remainder of the present chapter conveys a flavour of the facilities available in Algol 68-R for constructing arbitrary number patterns from suitably chosen *frames*, which apply to individual characters. It will be recalled that the pattern for a row of characters can be made up from replicated *a*'s; similarly the pattern for a number can be made up from replicated digit frames. Insertions can then be placed within the pattern if required.

The frames needed for numerical values are listed below; in the descriptions the word 'number' should be understood to apply to a complete integer, or separately to the two parts of a real. Thus the real value 1.23&4 is made up of two separate numbers, 1.23 and the integer exponent 4.

- \+ frame for the sign, indicating that a positive number is to have a plus sign (and a negative number a minus)
- – alternative sign frame, indicating that a negative number is to have a minus, but a positive number a space
- *d* one digit—this frame can be replicated
- *u* one digit, but zero is suppressed (replaced by space) if it is the first digit of a number or if the previous digit of the number was a suppressed zero. This frame can be replicated.
- . frame for the decimal point(.)
- *e* frame for the exponent symbol (&)
- *i* frame for the imaginary symbol (?) between two real patterns

The pattern for an INT value is constructed from a sign frame, which is optional, and frames for the digits. If the sign frame is omitted, there will be no sign or space and negative values cannot then be transput. The pattern for a REAL value has, in addition, frames for a decimal point or an exponent symbol or both.

Examples of integers 24 and 0 with various patterns 4 characters wide

+3d	*+3u*	*−3u*	*−2ud*
+024	+ 24	24	24
+000	+		0

Examples of reals 24.0 and 0.0 with various patterns

+ud.d	*+d.de+d*	*+d.de−d*
+24.0	+2.4&+1	+2.4& 1
+ 0.0	+0.0&+0	+0.0& 0

It will be seen that the effect of using *u* to obtain zero suppression is to isolate the sign from the number, as the spaces occupy the positions vacated by the leading zeros. This can be remedied by writing the sign frame after the *u* frame rather than before it; the sign is then placed immediately in front of the first actual digit without altering the total width of the number. For example, the effect of $3u + d$ on the integer value zero would be $+0$ with three spaces before the sign.

The following complete program shows the use of insertions to obtain digit grouping, as usually found in published tables.

program

```
comrie
BEGIN PROC factorial = (INT n)INT:
        ((n = 0|1|n * factorial(n−1))) ;
        format(standout, $5(ud2x, ux3ux3ul)l$) ;
        FOR i FROM 0 TO 10 DO
        out(standout, (i, factorial(i)))
END FINISH
```

output produced

```
 0          1
 1          1
 2          2
 3          6
 4         24

 5        120
 6        720
 7      5 040
 8     40 320
 9    362 880

10  3 628 800
```

Octal integers (and octal long integers) can be transput by placing 8r in front of an integer pattern; this does not apply to the simple patterns given in 12.5. If no sign frame is given after 8r, the value is the direct octal representation of the word in the computer.

13.4 Frame suppression

It is possible to suppress a character which would normally be printed, or expected for input, by prefixing its frame with the letter *s*. Used in conjunction with insertions, this facility enables a standard character such as decimal point to be replaced by a character of one's own choice. Any of the frames described in the previous section, except for the sign frame, can be treated in this way, though *su* is available only for output. The effects are as follows

se	*s.*	*si*	on output the character is omitted and on input nothing is read
3*sd*			on output, causes omission of the 3 digits; when used for input, the 3 missing digits are assumed to have been zeros
3*su*			available only on output, this does not suppress any digits; it removes spaces which have resulted from suppression of leading zeros

The 3's above are simply examples. Any replicator would have done, or no replicator at all (equivalent to replicator of 1). As an example of how *s* can be used, the pattern

$"£"2*suds*." – "2*d*$

causes any positive reals up to 999.99 to be printed in forms such as

£100–00
£20–50
£0–00

with no hidden or other spaces at all.

14

Program Segmentation

A program need not be written in one unit, but can be made up of separately compiled segments, each of which can use identifiers, operators or modes declared in previous segments. There are two main applications of this facility:

i each programmer can build up his own private library of compiled procedures (etc) to supplement the system library

ii long programs can be divided up into manageable portions for separate compilation

A programmer's compiled segments can be kept in a file known as an album; the Algol 68-R system also allows a party of programmers, each with his own album, to share in addition a common album, a principle which can be extended in a hierarchical manner if necessary. The system library can be regarded as an album which is common to everybody.

14.1 Keeping names

A compiled program is expressed in machine language; the names used in the original Algol text have disappeared. When a *segment* of program is compiled, however, it is essential to make provision for subsequent segments to use its names. A special record of such names must be kept along with a compiled segment. To bring this about, any names to be kept for further use are listed between END and FINISH as shown here:

title of segment
BEGIN *serial clause* END
KEEP *name, name, . . . name*
FINISH

As an example, the following segment defines the mode VECTOR, the operators + (for sums of vectors) and * (for products and scalar products), and unit vectors ux, uy and uz.

```
vectorseg
BEGIN MODE VECTOR  =  STRUCT(REAL x, y, z) ;
    OP +   =  (VECTOR a, b)VECTOR:
                ((x OF a + x OF b,
                    y OF a + y OF b,
                        z OF a + z OF b)) ;
    OP *   =  (REAL a, VECTOR b)VECTOR:
                ((a * x OF b,
                    a * y OF b,
                        a * z OF b)) ;
    OP *   =  (VECTOR a, b)REAL:
                (x OF a * x OF b
                    + y OF a * y OF b
                        + z OF a * z OF b) ;
    VECTOR ux  =  (1.0, 0.0, 0.0) ;
    VECTOR uy  =  (0.0, 1.0, 0.0) ;
    VECTOR uz  =  (0.0, 0.0, 1.0) ;
    SKIP
END
KEEP VECTOR, +, *, ux, uy, uz
FINISH
```

The order of the names in the KEEP list (i.e. VECTOR, +, *, ux, uy, uz) is immaterial, and need not correspond to the order of declaration. Names kept can only be those of objects declared in the outermost level of the segment.

14.2 Requesting previously compiled segments

Suppose that segments entitled seg1, seg2 and seg3 have been compiled and entered into an album with the name file1. A program, tryout, embodying these segments would be written

```
tryout WITH seg1, seg2, seg3 FROM file1
BEGIN serial clause END
FINISH
```

This becomes one unified program

```
seg1 ;
seg2 ;
seg3 ;
serial clause
```

which is *obeyed from the beginning of seg1*. We shall say that tryout 'requested' seg1, seg2 and seg3 (because their names were included after WITH).

We have described tryout as a *program* because it gets assembled as four segments; in terms of source text it is really the title of the final segment.

In addition to the requests issued by tryout itself, there may be other requests. For instance, seg2 may have requested seg1 in order that it might use names from seg1. Although this would *not* cause seg1 to appear twice over in the program tryout, the question arises of whether tryout need request seg1 if seg2 has already done so. The answer is that seg2's request would ensure the presence of seg1, but if the keep list of seg1 is relevant to the tryout segment, then tryout must explicitly request seg1. The linkage of any name is established by a keep, and a *direct* request for the segment which kept it.

If the above rules should result in some segment being requested twice over, or more, there may be some doubt about the order in which the various segments are finally assembled. This will always correspond to the filing order, which in simple cases where albums are not nested, corresponds to the order in which the segments were compiled and entered into the album. The order in which segment names are listed after WITH must correspond to the order in which those segments were filed.

Appendix 1 List of Basic Modes

INT

Integers can be denoted in the normal scale of 10, or with radix 2, 4 or 8. The following are equivalent integer denotations:

128, 2r10000000, 4r2000, 8r200

The limits of INT values for ICL 1900 machines are

-2^{23} to $2^{23}-1$ (inclusive)

The denotations with radix 2, 4 or 8 can be used for negative integers. For example 8r77777777 is equivalent to the integer −1. There is no decimal denotation for a negative integer.

LONG INT

A long integer takes twice the computer space of an INT. Its denotation is the word LONG followed by a radix 10 integer denotation. The limits on ICL 1900 machines are

-2^{46} to $2^{46}-1$ (inclusive).

REAL

The denotation must contain either a decimal point or a tens exponent or both, eg

5.1 or 51&−1 or 0.51&1

A decimal point must always be followed by a digit.
The limits of reals in ICL 1900 machines are approximately 5&−78 to 5&76, and the accuracy is limited to just over 11 decimal figures.

BOOL

The two boolean values have denotations TRUE and FALSE.

CHAR

A character is denoted in quotes ("), and is one of the set:

0	1	2	3	4	5	6	7	8	9
:	;	<	=	>	?	sp	!	"	#
£	%	&	'	(	)	*	+	,	−
.	/	@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	[
$	]	↑	←						

The exception to the quote rule (eg "A") is the denotation for the quote character itself, which is """".

BITS

Indicates a value consisting of 24 binary digits. There is no denotation for BITS values, but in most contexts an integer denotation will be automatically coerced to mode BITS when required. See INT, above.

BYTES

A value of mode BYTES is a unit of 4 characters, and is denoted in quotes, eg "ABCD". A quote character in the BYTES value is written as a pair of quotes, eg "AB""D" is the denotation for the four characters A, B, ", D. An individual character may be selected from a BYTES value by means of the operator ELEM (see Appendix 2); a BYTES denotation cannot be indexed.

LONG BYTES

As above, but the unit is 8 characters.

LONG LONG BYTES

As above, but 12 characters.

FORMAT

See chapter 12.

The following modes, though strictly not 'basic' modes, are included here for the sake of completeness:

COMPLEX — Equivalent to STRUCT(REAL re, im). There is no denotation for a complex number; it is written in a program as a pair of real numbers with the *operator* ? between them, eg

```
1.234 ? 5.678   or   x ? y
```

STRING — Equivalent to [1 : 0 FLEX] CHAR. The denotation for a string is a sequence of characters enclosed in quotes, where the number of characters is not 4, 8 or 12 (see BYTES above). A quote character in the string must be written as a *pair* of quote symbols.

Appendix 2 Standard Constants, Procedures and Operators

Standard constants, procedures and operators can be used in any program without declaration; the same is also true for all entities defined in the Algol 68-R system library. Modes and procedures concerned with standard transput are not included in this Appendix but are grouped with the system library, for which documentation is available at user installations.

Constants

REAL pi = 3.1415 9265 359

Functions

sqrt	square root
exp	exponential function
ln	natural logarithm
sin	sine
cos	cosine
tan	tangent
arcsin	inverse sine in range (−pi/2, pi/2)
arccos	inverse cosine in range (0, pi)
arctan	inverse tangent in range (−pi/2, pi/2)

All of the above are of mode PROC(REAL)REAL.

Random numbers

The procedure "random", mode PROC REAL, delivers a pseudo-random number in the range (0.0, 1.0), with a non-repeatable pseudo-random starting point in every program run.

Operators

The integer following the operator gives its priority. The larger the integer, the more tightly the operator binds to its operands. Priorities of dyadic operators are in the range 1 to 9; the highest priority (10) applies only to monadic definitions.

Arithmetical

↑	8	x↑n gives x raised to the power n, where n must be an INT. If x is REAL, the result is REAL; if x is INT, the result is INT.
* } / }	7	times and divide, between INT, REAL or COMPLEX in any combination, and between LONG INT numbers. The quotient of two integers is REAL.
'/'	7	between n and m of mode INT, gives result ENTIER(n/m), mode INT. Similarly for LONG INT, with LONG INT result.
+ } − }	6	plus and minus, between INT, REAL or COMPLEX in any combination, and between two LONT INT numbers. May also be used monadically (priority 10).

Arithmetical comparisons

< <= > >=	5	less, less or equal, greater and greater or equal, between INT or REAL in any combination, and between two LONG INT numbers. BOOL result.
= #	4	equals and not equals, between two INT numbers or between two LONG INT numbers. BOOL result.

Arithmetical assignments

PLUS	1	x PLUS y means x := x+y, where x is REF INT, REF REAL or REF COMPLEX, and x+y must deliver a result suitable for assignment. (For example, if x is REF INT, y must not be REAL.)
MINUS	1	x MINUS y means x := x−y and similar remarks apply.
TIMES	1	x TIMES y means x := x * y, where x is REF INT or REF REAL, and x * y must deliver a result suitable for assignment. (For example, if x is REF INT, y must not be REAL.)
DIV	1	x DIV y means x := x/y, where x must be REF REAL and y may be INT or REAL.
'/:='	1	used between n and m, where n has mode REF INT, and m has mode INT, performs the assignment n := ENTIER(n/m) and delivers the remainder with the same sign as m. For example, if n refers to −7 and m is 3, it would assign −3 to n and deliver +2.

Other numerical operators

ABS	10	modulus of an INT, REAL or COMPLEX.
SIGN	10	applied to INT or REAL, gives +1 for positive operand, 0 for zero operand and −1 for negative operand. INT result.
ODD	10	applied to INT, gives TRUE if the operand is odd, and FALSE otherwise.
ROUND	10	applied to REAL, gives the nearest INT. In critical cases, rounds up.
ENTIER	10	applied to REAL, gives largest INT less than or equal to the REAL.
LENG	10	applied to an INT value, delivers the LONG INT value.
SHORT	10	applied to a LONG INT value, delivers the INT value.
?	9	used between INT or REAL in any combination, delivers a COMPLEX such that the left operand is the real part and right operand the imaginary part.
ARG	10	applied to COMPLEX, delivers the argument, ie phase, in the range (−pi, pi), mode REAL.
CONJ	10	applied to COMPLEX, delivers the conjugate.

Operations on Booleans

NOT	10	delivers TRUE if the operand is FALSE and vice versa.
AND	3	delivers TRUE if both operands are TRUE, otherwise delivers FALSE.
OR	2	delivers TRUE if at least one operand is TRUE, otherwise delivers FALSE.
= #	4	equals and not equals; delivers TRUE if the relation is true, otherwise delivers FALSE.
ABS	10	delivers integer 1 if operand is TRUE, otherwise delivers integer 0.

Operations on arrays

LWB	10	applied to an array, gives the *lower bound* of its first dimension as an INT.
UPB	10	applied to an array, gives the *upper bound* of its first dimension as an INT.
LWB	8	m LWB p, where m is INT and p an array, gives the *lower bound* of the mth dimension of p. Hence 1 LWB p gives the same value as LWB p.
UPB	8	similar to LWB, but *upper bound.*
FLEXIBLE	10	applied to an array, delivers TRUE if the array has flexible bounds, otherwise delivers FALSE.
CLEAR	10	applied to a reference to an array with any number of dimensions, 'clears' the array. For the following modes of array element the resulting values are as shown:

INT	0
LONG INT	LONG 0
REAL	0.0
COMPLEX	0.0 ? 0.0
BOOL	FALSE
BITS	BIN 0
CHAR	space character
BYTES	"0000"
LONG BYTES	"00000000"
LONG LONG BYTES	"000000000000"

Operations with characters and bytes

ABS	10	applied to a CHAR, gives the internal integer form in the range 0 to 63, corresponding to the position in the CHAR table in Appendix 1.
REPR	10	applied to an INT in the range 0 to 63, gives the corresponding CHAR from the table in Appendix 1.
< <= > >=	5	less than, less than or equal, greater than and greater than or equal, between CHAR, [ ]CHAR, STRING, BYTES, LONG BYTES or LONG LONG BYTES in any combination, delivering TRUE or FALSE. If c and d are characters, c<d means ABSc< ABSd. When comparing sets of characters, successive characters from each set are compared until a decision is made or the smaller set is exhausted. The ordering is lexicographic, ie "AA" < "AAA" is TRUE, and "AAB" < "AB" is TRUE.
= ≠	4	equals and not equals, between operands with modes as above.
CTB CTLB CTLLB	10	applied to words of 4, 8 or 12 characters respectively, delivers the characters as a BYTES, LONG BYTES or LONG LONG BYTES value.
ELEM	7	used between m and c, where m is an INT and c is a BYTES, LONG BYTES or LONG LONG BYTES, gives the mth character of c. The characters are numbered from 1 to 4, 8 or 12 respectively, starting from the left.

Operations with bits

ABS	10	applied to a BITS value gives the corresponding signed INT value.
BIN	10	applied to an INT, gives the BITS value which represents that integer.
NOT	10	applied to a BITS value, delivers the BITS value with every digit reversed.

AND	3	used between two BITS values, delivers the BITS value obtained by applying 'and' to each binary digit:

	0	1
0	0	0
1	0	1

OR	2	used between two BITS values, delivers the BITS value obtained by applying 'or' to each binary digit:

	0	1
0	0	1
1	1	1

SL	7	shift left. b SL m, where b is a BITS value and m a positive INT, gives the BITS value obtained by shifting b left m places. Binary digits are lost on the left, and zeros introduced on the right.
SR	7	shift right; similar to SL.
SLC	7	shift left cyclically. Operands as for SL, but binary digits lost on the left are introduced on the right.
SRC	7	shift right cyclically, similar to SLC.
<= } >= }	5	between BITS values, delivers a BOOL result. p<=q is TRUE if each binary digit which is 1 in p is also 1 in q.
= } # }	4	between BITS values, delivers TRUE if the relation is true, and FALSE otherwise.
ELEM	7	m ELEM b, where m is an INT and b is a BITS value, delivers TRUE if the mth binary digit of b is 1, and FALSE if it is 0.
SET	7	m SET b, where m is an INT and b a BITS value, delivers the bit-pattern b but with the mth binary digit as 1.
CLEAR	7	m CLEAR b, where m is an INT and b is a BITS value, delivers the bit-pattern b but with the mth binary digit as 0.

Default priorities

If one of the standard operator symbols is re-declared dyadically in a program and no priority declaration is given, the standard *dyadic* priority of the operator applies. If there is no standard dyadic priority, the default priority is 1, with the exception of ENTIER, which will have priority 8. For example, if NOT is declared as a dyadic operator, its default priority is 1 (not 10). The complete table of default priorities for dyadic use is as follows:

ABS	1	NOT	1	?	9
AND	3	ODD	1	↑	8
ARG	1	OR	2	*	7
BIN	1	PLUS	1	/	7
CLEAR	7	REPR	1	'/'	7
CONJ	1	ROUND	1	+	6
CTB	1	SET	7	−	6
CTLB	1	SHORT	1	<	5
CTLLB	1	SIGN	1	<=	5
DIV	1	SL	7	>	5
ELEM	7	SLC	7	>=	5
ENTIER	8	SR	7	=	4
FLEXIBLE	1	SRC	7	#	4
LENG	1	TIMES	1	'/:='	1
LWB	8	UPB	8		
MINUS	1				

Appendix 3 **Syntax of Algol 68-R**

Each line represents a separate grammatical alternative, with optional parts in curly brackets. Symbols (except hyphens) and upper case words stand for themselves; *etc* when not preceded by a comma indicates the possibility of repetition from the last unmatched { (or [. Otherwise, n hyphens, *etc* means that the previous n items may be repeated with comma separation, where an item is a word, symbol, thing in curly brackets or repeated item.

segment

title{WITH title-, *etc* FROM album} (sec){KEEP name-, *etc*}FINISH

sec (short for serial clause)

{declaration; {unc; *etc*} *etc*}unclist

unclist

{{label: *etc*}unc; *etc*}{label: *etc*}unc{EXIT label: unclist}

unc (short for unitary clause)

expression := unc
{FOR identifier}{FROM unc}{BY unc}{TO unc}{WHILE sec}DO unc
GOTO label
expression IS expression
expression ISNT expression
proceduredenotation
mode VAL unc
expression ::= unc
expression :: unc
(expression-, *etc*) ::= unc
(expression-, *etc*) :: unc
expression

expression

primary
combination of primaries and operators
{LOC}mode§

primary

identifier
denotation
primary(unc-, *etc*)
primary[indexer-, *etc*]
selector OF primary (*brackets bind more tightly than OF*)
IF sec THEN sec {ELSE sec} FI
CASE sec IN sec-, *etc* {OUT sec} ESAC
(sec)
(sec, sec-, *etc*)

indexer

unc
{{unc} : {unc}}{AT unc}

declaration

mode§ identifier {:= unc}--, *etc*
PROC identifier = proceduredenotation
PROC identifier := proceduredenotation
mode identifier = unc---, *etc*
MODE modename {= mode§}--, *etc*
PRIORITY opsymbol = digit---, *etc*
OP opsymbol = proceduredenotation
OP(mode){mode}opsymbol = unc
OP(mode, mode){mode}opsymbol = unc

proceduredenotation

(mode identifier-, *etc*--, *etc*){mode}: (sec)
mode: (sec)

mode

[{ , } *etc*]notarraymode
notarraymode

notarraymode

STRUCT(mode selector-, *etc* --, *etc*)
simplemode

simplemode

basicmode (*see Appendix 1*)
modename
REF mode
PROC(mode-, *etc*){mode}
PROC mode
UNION(mode-, *etc*)
VOID (*VOID is used only for mode of result of procedure or operator*)

album *a file identifier of the operating system*

denotation *see Appendix 1*

digit *digit in the range 1–9 inclusive*

identifier *see 1.2*

modename *any upper case word not already defined as a word in the language*

name *occurring after KEEP, this can be an identifier, opsymbol or modename*

opsymbol *see 8.3*

title, label and **selector** each have the same form as an identifier

§ If this mode contains square brackets, bounds in the form unc:unc{FLEX} are needed for each dimension in the places described in 6.4.

Index

Numbers refer to sections, A standing for Appendix.

Printed in England for Her Majesty's Stationery Office by McCorquodale Printers Ltd., London
Dd 596204 K40 3/79 McC 3309 HM(S) 9834

HER MAJESTY'S STATIONERY OFFICE

Government Bookshops

49 High Holborn, London WC1V 6HB
13a Castle Street, Edinburgh EH2 3AR
41 The Hayes, Cardiff CF1 1JW
Brazennose Street, Manchester M60 8AS
Southey House, Wine Street, Bristol BS1 2BQ
258 Broad Street, Birmingham B1 2HE
80 Chichester Street, Belfast BT1 4JY

Government publications are also available through booksellers

ISBN 0 11 771600 6