Programming Algol 68 Made Easy

Sian Mountbatten

Phœnix Engineering

Publishing history

- First edition published by Oxford & Cambridge Compilers Ltd in 1995.
- Second (revised) edition published by Oxford & Cambridge Compilers Ltd in 1997.
- Third edition published by Phœnix Engineering in 2000.
- Revised by Phœnix Engineering in 2001, 2002.
- Fourth edition by Neil Matthew in 2021

Prepared in Scotland.

To

Aad van Wijngaarden

Athair na h-Algol 68

Contents

Preface

It is a fallacy to say that progress consists of replacing the workable by the new. The brick was invented by the Babylonians and has been used virtually unchanged for 2500 years. Even now, despite the advent of curtain-walling, the brick is still the primary building material. Likewise, the long-predicted revolution in computer programming to be produced by the introduction of fourth- and fifth-generation languages has not come to pass, almost certainly because their purported advantages are outweighed by their manifest disadvantages. Third-generation languages are still used for the bulk of the world's programming. Algol 68 has been used as a paradigm of third-generation languages for 32 years.

Each computer programming language has a primary purpose: C was developed as a suitable tool in which to write the Unix operating system, Pascal was designed specifically to teach computer programming to university students and Fortran was designed to help engineers perform calculations. Where a programming language is used for its design purpose, it performs that purpose admirably. Fortran, when it first appeared, was a massive improvement over assembler languages which had been used hitherto. Likewise, C, when restricted to its original purpose, is an admirable tool, but it is a menace in the hands of a novice. However, novices do not write operating systems.

According to the "Revised Report on the Algorithmic Language Algol 68" (see the Bibliography), Algol 68 was "designed to communicate algorithms, to execute them efficiently on a variety of different computers, and to aid in teaching them to students". Although this book has not been geared to any specific university syllabus, the logical development of the exposition should permit its use in such an environment. However, since no programming expertise is assumed, the book is also suitable for home-study.

It is time to take a fresh approach to the teaching of computer programming. This book breaks new ground in that direction. The concept of a variable (a term borrowed from mathematics, applied to analogue computers and then, inappropriately, to digital computers) has been replaced by the principle of value integrity: in Algol 68, every value is a constant. All the usual paraphernalia of pointers, statements and expressions is dispensed with. Instead, a whole new sublanguage is provided for understanding the nature of programming.

This book covers the language as implemented by the DRA a68toc translator. Since the last edition, a new chapter on the Standard Prelude has been added, thereby bringing together all the references to that Prelude in the rest of the book. This edition is an interim edition describing the QAD transput provided with the Algol68toC package.

It has been a conscious aim of the author to reduce the amount of description

to a minimum. It is advisable, therefore, that the text be read slowly, re-reading a point if it is not clear. This is particularly true for chapter 5 where the concept of the name has been introduced rather carefully. The exercises are intended to be worked. Answers to all the exercises have been given except for those which are self-marking.

A program written for use with the book can be found in the same directory as this book.

I should like to thank Wilhelm Klöke for bringing the Algol 68RS compiler to my attention and James Jones and Greg Nunan for their active help in the preparation of the QAD transput.

In 40 years of programming, I have had many teachers and mentors, and I have no doubt that I have benefited from what they have told me, although now it is difficult to pinpoint precisely which part of my understanding is due to which individual. Any errors in the book are my own. If any reader should feel that the book could be improved, I should be grateful if she would communicate her suggestions to the publisher, so that in the event of another edition, I can incorporate those I feel are appropriate (she includes he).

Sian Mountbatten
Inbhir Nis
Am Mart 2008

Preface to the 4th Edition

This 4th edition of "Programming Algol 68 Made Easy" or "PAME" to its friends has been created to reflect changes in technology and the a68toc package since Dr. Mountbatten's last version in 2008.

The book, while mostly general in nature, features the Algol-68RS compiler as a target, through the a68toc Algol 68 to C translator. Examples and exercises are presented in the dialect supported by a68toc.

To preserve Sian's work to the greatest extent, only very minor changes have been made to the text: typos, references to older versions of a68toc and support for more architectures.

The LaTeX source has been updated to use the HyperTex package to provide internal links suitable for producing a PDF with navigation. Exercises now have working links to answers and vice versa.

Starting from a version of Dr. Mountbatten's port of a68toc to 32-bit Linux (v1.15[1]), the current author has made some advances and released further versions[2]. These include:

- A complete port to 64-bit systems, removing assumptions (explicit and implied) about word size in the libraries and the compiler itself.
- The integration of a well-supported garbage collector (Boehm-Demers-Weiser) as a compile-time option for the compiler (default for 64-bit systems).
- Added support for 32-/64-bit ARM processor architectures, removing implicit assumptions about endianness.
- Added support for 64-bit macOS operating system versions.
- A fix for a long-standing bug in the compiling of the `BY` part of `FOR` loops.
- Production of Debian Linux binary packages for i386, amd64, armhf, arm64 architectures.

This edition of the book applies to versions of a68toc v1.22 and later.

Neil Matthew
neil@tilde.co.uk
May 2021

[1] While a v1.19 is known, no source of this version has been located

[2] Releases and work-in-progress available on GitHub:
https://github.com/coolbikerdad

Chapter 1

Introduction

Algol 68 is a high-level, general-purpose programming language ideally suited to modern operating systems. This book will teach you Algol 68 plus the necessary development skills to enable you to write substantial programs which can be executed from the command line.

In principle, you can solve any computable problem with Algol 68. You can write programs which perform word processing, perform complicated calculations with matrices, design graphs or bridges, process pictures, predict the weather, and so on. Or you can write simple programs which count the number of words in a file or list a file with line numbers.

Algol 68 is a powerful language. There are many constructs which enable you to manipulate complicated data structures with ease, and yet it is all easy to understand because one of the guiding principles of Algol 68 was that it was designed to be **orthogonal**. This means that the language is based on a few independent ideas which are developed and applied with generality. The language was designed in such a way that it is impossible to write ambiguous programs. The design is also difficult to describe until it has been fully described, which means that some concepts have to be introduced in a superficial manner, but later reading will deepen your understanding.

You need to have a thorough grasp of the basic ideas if you are going to write powerful programs in Algol 68: these ideas unfold in the first five chapters. The chapters should be read in order, but chapter 5 is a watershed—it forms the basis of much of the computer programming performed in the world today. Its ideas should be mastered before continuing.

Chapter 10 draws together all the various references to grammatical points and clarifies the limitations of the language—you will need to know these if you want to squeeze the last ounce of power out of the language. Chapters 11 and 12 deal with advanced topics which should not be touched until you have mastered preceding material. Chapter 13 describes the standard prelude which, besides providing means of determining the characteristics of an Algol 68 implementation, also provides the transput facilities whose power are characteristic of Algol 68.

In this chapter, some aspects of Algol 68 grammar are described. Don't worry if they seem confusing; all will become clear later in the book. It also covers denotations and the identity declaration, the latter having crucial importance in the language.

1.1 What you will need

The language described in this book is that made available by the a68toc Algol 68RS compiler developed by the *Defence Research Agency* (see section 1.9 for more information about what a compiler does). It implements almost all of the language known as Algol 68, and extends that language in minor respects. To run the programs described in this book you will need a computer with a Linux, macOS[1] or Windows[2] operating system and Intel/AMD i386/x86_64 or ARM armhf/aarch64 processors with a compatible C compiler (GCC and clang should be fine). The source package will occupy ≈ 12Mb on the hard disk while the binary package will also need ≈ 9Mb of space on the hard disk. The source package may be deleted once the binary package has been installed.

The book expects you to be familiar with the usual commands for manipulating files. You will need to know how to use an editor for plain text files (not a word processor). No programming expertise is presumed.

Much program development work takes place at the command line because a graphical user interface is usually too cumbersome to cope with the myriad commands issued by the programmer. See the manual pages for `ca` and `a68toc` for details of how to use the `Algol68toC` program development system.

1.2 Terminology

In describing Algol 68, it is necessary to use a number of technical terms which have a specialist meaning. However, the number of terms used has been reduced to a minimum. Whenever a term is introduced it will be written in **bold**. Parts of programs are printed as though they had been produced by a typewriter like this:

```
BEGIN
```

Some of the terminology may seem pedantic. Describing the parts of an Algol 68 program can, and should be, precise. The power of Algol 68 derives as much from the precision as from the generality of its ideas.

1.3 Values and modes

Two of the guiding principles of Algol 68 are the concepts of **value** and **mode**. Typically, an Algol 68 program manipulates values to produce new values, and, in the process, does useful work (such as word-processing). Values are such entities as numbers and letters, but you will see in later chapters that values can be very complicated and, indeed, can be things that you would not normally think of as a value.

A value is characterised by its mode. Every value has only one mode, and cannot change its mode. Therefore, if you have a mode change you must have a new value as well (but see chapter 8). A mode defines a set of values. The number of values in the set depends on the mode and there can be from none to potentially infinity. For example, the whole number represented by the digits 37 has mode `INT`. The symbol

[1] 64-bit only

[2] With Windows Subsystem for Linux installed

`INT` is called a **mode indicant**. You will be meeting many more mode indicants in the course of this book and they are all written in capital letters and sometimes with digits. The strict definition of a mode indicant is that it consists of a series of one or more characters which starts with a capital letter, and is continued by capital letters or digits. No intervening spaces are allowed. There is no limit to the length of a mode indicant although in practice it is rare to find mode indicants longer than 16 characters. Here are some more mode indicants which you will meet in this and later chapters:

```
BOOL    CHAR    COMPL    FILE    HMEAN
```

Section 7.3 explains how you can define your own mode indicants. Although you can use any sequence of valid characters, meaningful mode indicants can help you to understand your programs.

Exercises

1.1 Is there anything wrong with the following mode indicants? Ans

(a) `RealNumber`
(b) `2NDINT`
(c) `COMPL`
(d) `UPPER CASE`
(e) `ONE.TWO`

1.2 What is the definition of a mode indicant? Ans

1.4 Integers

Although, strictly speaking, there is no largest positive integer, by default the largest positive integer which can be manipulated by `a68toc` is the 32-bit value 2 147 483 647, and the largest negative integer is −2 147 483 647 (the first is $2^{31}-1$ and the second is $-2^{31}+1$)[3]. The representation of a value in an Algol 68 program is called a **denotation** because it denotes the value. It is important to realise that the denotation of a value is not the same as the value itself. To be precise, we say that the denotation of a value represents an instance of that value. For example, three separate instances of the value denoted by the digits `31` occur in this paragraph. All the instances denote the same value.

If you want to write the denotation of an integer in an Algol 68 program, you must use any of the digits `0` to `9`. No signs are allowed. This means that you cannot write denotations for negative integers in Algol 68 (but this is not a problem as you will see). Although you cannot use commas or decimal points, spaces can be inserted anywhere. Here are some examples of denotations of integers separated by commas (the commas are not part of the denotations):

```
0 , 3 , 03 , 3000000 , 2 147 483 647
```

[3] A larger 64-bit integer mode is available, see 13.1

Note that `3` and `03` denote the same value because the leading zero is not significant. However, the zeros in the three million are significant. The mode of each of the five denotations is `INT`. The following are incorrect denotations:

```
3,451    -2    1e6
```

The first contains a comma, the second is a formula, and the third contains the letter `e`. You will see later on that the third expression denotes a number, but by definition this denotation does not have mode `INT`.

Exercises

1.3 Write a denotation for thirty-three. Ans

1.4 What is wrong with the following integer denotations? Ans

(a) `1,234,567`
(b) `5.`
(c) `-4`

1.5 Identity declarations

Suppose you want to use the integer whose denotation is

```
48930767
```

in various parts of your program. If you had to write out the denotation each time you wanted to use it, you would find that

- you would almost certainly make mistakes in copying the value, and
- the meaning of the integer would not be at all clear

It is imperative, particularly with large programs, to make the meaning of the program as clear as possible. Algol 68 provides a special construct which enables you to declare a synonym for a value (in this case, an integer denotation). It is done by means of the construct known as an **identity declaration** which is used widely in the language. Here is an identity declaration for the integer mentioned at the start of this paragraph:

```
INT special integer = 48930767
```

Now, whenever you want to use the integer, you write

```
special integer
```

in your program.

An identity declaration consists of four parts:

```
<mode indicant> <identifier> = <value>
```

You have already met the `<mode indicant>`. An **identifier** is a sequence of one or more characters which starts with a lower-case letter and continues with lower-case letters or digits or underscores. It can be broken-up by spaces, newlines or tab characters. Here are some examples of valid identifiers (they are separated by commas to show you where they end, but the commas are not part of the identifiers):

```
i,  algol,  rate 2 pay,  eigen value 3
```

The following are wrong:

```
2pairs   escape.velocity   XConfigureEvent
```

The first starts with a digit, the second contains a character which is neither a letter nor a digit nor an underscore, and the third contains capital letters.

An identifier looks like a name, in the ordinary sense of that word, but we do not use the term "name" in this sense because it has a special meaning in Algol 68 which will be explained in Chapter 5. The identifier can abut the mode indicant as in

```
INTa = 4
```

but this is unusual. For clarity in your programs, ensure that a mode indicant followed by an identifier is separated from the latter by a space.

The third part is the equals symbol `=`. The fourth part (the right-hand side of the equals symbol) requires a value. You will see later that the value can be any piece of program which yields a value of the mode specified by the mode indicant. So far, we have only met integers, and we can only denote positive integers.

There are two ways of declaring identifiers for two integers:

```
INT i = 2 ; INT j=3
```

The semicolon `;` is called the **go-on symbol** because it means "throw away the value yielded by the previous phrase, and go on to the next phrase". If this statement seems a little odd, just bear with it and all will become clear later. We can abbreviate the declarations as follows:

```
INT i=2, j = 3
```

The comma separates the two declarations, but does not mean that the `i` is declared first, followed by the `j`. On the contrary, it is up to the compiler to determine which declaration is elaborated first. They could even be done in parallel on a parallel processing computer. This is known as collateral elaboration, as opposed to **sequential** elaboration determined by the go-on symbol (the semicolon). We shall be meeting collateral elaboration again in later chapters. Elaboration means, roughly, execution or "working-out". The compilation system translates your Algol 68 program into machine code. When the machine code is obeyed by the computer, your program is elaborated. The sequence of elaboration is determined by the compiler as well as by the structure of your program. Note that spaces are allowed almost everywhere in an Algol 68 program.

Some values are predefined in what is called the **standard prelude**. You will be learning more about it in succeeding chapters. One integer which is predefined in Algol 68 has the identifier `max int`. Can you guess its value?

Exercises

1.5 What is wrong with the following identifiers? Ans

(a) `INT`
(b) `int`
(c) `thirty-four`
(d) `AreaOfSquare`

1.6 What is wrong with the following identity declarations? Ans

(a) `INT thirty four > 33`
(b) `INT big int = 3 000 000 000`

1.7 Write an identity declaration for the largest integer which the Algol 68 compiler can handle. Use the identifier `max int`. Ans

1.6 Characters

All the symbols you can see on a computer, and some you cannot see, are known as characters. The alphabet consists of the characters `A` to `Z` and `a` to `z`. The digits comprise the characters `0` to `9`. Every computer recognises a particular set of characters. The character set recognised by `a68toc` is ASCII (which stands for American Standard Code for Information Interchange). The mode of a character is `CHAR` (read "car" because it is short for character). A character is denoted by placing it between quote characters. Thus the denotation of the lower-case a is `"a"`. Here are some character denotations:

```
"a"  "A"  "3"  ";"  "\"  "'"  """"  " "
```

Note that quote characters are doubled in their denotations. The third denotation is `"3"`. This value has mode `CHAR`. The denotation `3` has mode `INT`: the two values are quite distinct, and one is not a synonym for the other. The last denotation is that of the space character.

Here are some identity declarations for values of mode `CHAR`:

```
CHAR a = "A", zed = "z";  CHAR tilde = "~"
```

Note that the two sets of identity declarations are separated by a semicolon, but the declaration for `tilde` is not followed by a semicolon. This is because the semicolon `;` is not a terminator; it is an action. Identity declarations do not **yield** any value. An identity declaration is a **phrase**. Phrases are either identity declarations or **units**. When a phrase is elaborated, if it is a unit, it will yield a value. That is, after elaboration, a value will be available for further use if required. Again, this may not make much sense now, but it will become clearer as you learn the language.

Here is a piece of program with identity declarations for an `INT` and a `CHAR`:[4]

[4]The a68toc compiler insists on a semicolon between identity declarations for different modes. In the above case, you would have to write `INT ninety nine=99 ; CHAR x = "X"`

```
INT ninety nine=99 , CHAR x = "X"
```

The compiler recognises 256 distinct values of mode `CHAR`, but most of them can only occur in denotations. The space is declared as `blank` in the standard prelude.

Exercises

1.8 Write the denotations for the full-stop, the comma and the digit `8` (not the integer 8). Ans

1.9 Write a suitable identity declaration for the question mark. Ans

---

1.7 Real numbers

Numbers which contain fractional parts, such as 3.5 or 0.0005623956, or numbers expressed in scientific notation, such as 1.95×10^{34} are values of mode `REAL`. Reals are denoted by digits and one at least of the decimal point (which is denoted by a full stop), or the letter `e`. The `e` means $\times 10^{\text{some power}}$. Just as with integers, there are no denotations for negative reals. When the exponent is preceded by a minus sign, this does not mean that the number is negative, but that the decimal point should be shifted leftwards. For example, in the following `REAL` denotations, the third denotation has the same value as the fourth (again, the denotations are separated by commas, but the commas are not part of the denotations):

```
4.5,  .9,  0.000 000 003 4,  3.4e-9,  1e6
```

Although the second denotation is valid, it is advisable in such a case to precede the decimal point with a zero: `0.9`. This is better because a decimal point not preceded by an integer can be missed easily. Here are some identity declarations for values of mode `REAL`:

```
REAL e = 2.718 281 828,
     electron charge = 1.602 10 e-19,
     monthly salary = 2574.43
```

The largest `REAL` which the compiler can handle is declared in the standard prelude as `max real`. Its value is

$$1.79769313486231571e308$$

The value of π is declared in the standard prelude with the identifier `pi` and a value of

```
REAL pi = 3.14159265358979323846264 3
```

It was mentioned above that in an identity declaration, any piece of program yielding a value of the required mode can be used as the value. Here, for example, is an identity declaration where the value has mode `INT`:

```
REAL a = 3
```

However, the mode required is `REAL`. In certain circumstances, a value of one mode can be coerced into a value of another mode. These circumstances are known as **contexts**. There are five contexts defined in the language. Each context will be mentioned as it occurs. The right-hand side of an identity declaration has a **strong** context. In a strong context, a value and its mode can be changed according to six rules, known as **coercions**, defined in the language. Again, each coercion will be explained as it occurs. The coercion which replaces a value of mode `INT` with a value of mode `REAL` is known as **widening**. You will meet a different kind of widening in section 7.4.

You can even supply an identifier yielding the required mode on the right-hand side. Here are two identity declarations:

```
REAL one = 1.0;
REAL one again = one
```

You cannot combine these two declarations into one with a comma as in

```
REAL one = 1.0, one again = one
```

because you cannot guarantee that the identity declaration for `one` will be elaborated before the declaration for `one again` (because the comma is not a **go-on** symbol).[5]

Values of modes `INT`, `REAL` and `CHAR` are known as plain values. We shall be meeting another mode having plain values in chapter 4, and modes in chapter 3 which are not plain. Complex numbers are dealt with in chapter 7.

[5]The a68toc compiler does permit a subsequent declaration to use the value of a previous value, but it is strictly non-standard. You would be wise to restrict your programs to Algol 68 syntax because other Algol 68 compilers will not necessarily be so lax.

Exercises

1.10 Is there anything wrong with the following identity declarations?

```
REAL x = 5.,
     y = .5;
     z = 100
```

Ans

1.11 Given that light travels $2.997\,925 \times 10^8$ metres per second in a vacuum, write an identity declaration for the identifier `light year` in terms of metres to an accuracy of 5 decimal places (use a calculator). Ans

1.8 Program structure

Algol 68 programs can be written in one or more parts Here is a valid Algol 68 program:

```
PROGRAM firstprogram CONTEXT VOID
USE standard
BEGIN
   print(20)
END
FINISH
```

Only the three lines starting with `BEGIN` and ending with `END` are strictly part of the Algol 68 program. The first, second and last lines are specific to the a68toc compiler. The first line gives the identification of the program as `firstprogram` and the fact that this file contains a program. The `CONTEXT VOID` phrase specifies that the program stands on its own instead of being embedded in other parts. The phrase is a vestige of the modular compilation system originally provided by the compiler at the heart of `a68toc`.

A full explanation of the `print` phrase will be found in chapter 9 (Transput). For now, it is enough to know that it causes the value in the parentheses to be displayed on the screen.[6] The standard prelude must be `USE`d if you want to use `print`. You can use any identifier for the operating system file in which to store the Algol 68 **source code** of the program. Although it does not have to be the same as the identifier of the module, it is advisable to make it so.

Both the `print` phrase and the denotation are units. Chapter 10 will explain units in detail. Phrases are separated by the go-on symbol (a semicolon `;`). Because there is only one phrase in `firstprogram`, no go-on symbols are required. Here is another valid program:

[6] When Algol 68 was first implemented there were few monitors around, so `print` literally printed its output onto paper.

```
PROGRAM prog CONTEXT VOID
USE standard
BEGIN
   INT special integer = 48930767;
   print(20) ; print(special integer)
END
FINISH
```

The semicolon between the two `print` phrases is not a terminator: it is a separator. It means "throw away any value yielded by the previous phrase and go on with the succeeding phrase". That is why it is called the go-on symbol. Notice that there is no semicolon after the third phrase.

Algol 68 programs are written in free format. This means that the meaning of your program is independent of the layout of the source program. However, it is sensible to lay out the code so as to show the structure of the program. For example, you could write the first program like this:

```
PROGRAM firstprogram CONTEXT VOID
USE standard(print(20))FINISH
```

which is just as valid, but not as comprehensible. Notice that `BEGIN` and `END` can be replaced by `(` and `)` respectively. How you lay out your program is up to you, but writing it as shown in the examples in this book will help you write comprehensible programs.

Exercises

1.12 What is wrong with this sample program?

```
PROGRAM test
BEGIN
   print("A"))
END
FINISH
```

Ans

1.13 Using an editor, key in the two sample programs given in this section, and compile and execute them. What do they display on your screen? Ans

---

1.9 Comments

When you write a program in Algol 68, the pieces of program which do the work are called "source code". The Algol 68 compiler translates this source code into C source code which is then translated by the GNU C compiler into "object code". This is then converted by a program called a linker into machine code, which is understood by the computer. You then execute the program by typing its name at the command-line plus any arguments needed. This is called **running** the program.

When you write the program, it is usually quite clear to you what the program is doing. However, if you return to that program after a gap of several months, the source code may not be at all clear to you. To help you understand what you have written in the program, it is possible, and recommended, to write comments in the source code. Comments can be put almost anywhere, but not in the middle of mode indicants and not in the middle of denotations. A comment is ignored by the compiler, except that comments can be nested. A comment is surrounded by one of the following pairs:

```
COMMENT ... COMMENT
CO ... CO
#...#
{...}
```

where the ... represent the actual comment. The paired braces are peculiar to the a68toc compiler. Other compilers may not accept them. Here is an Algol 68 program with comments added:

```
PROGRAM prog CONTEXT VOID
USE standard
BEGIN
   INT i = 23,  # My brother's age #
       s = 27; CO My sister's age CO
   CHAR z = "&",  COMMENT An ampersand
   COMMENT  y{acht}="y";
   REAL x = 1.25;
   print(i); print(s); print(z);
   print(y); print(x)
END
FINISH
```

There are four comments in the above program. If you start a comment with `CO` then you must also finish it with `CO`, and likewise for the other comment symbols (except the braces). Here is a program with a bit of source code "commented out":

```
PROGRAM prog CONTEXT VOID
USE standard
BEGIN
   INT i = 1, j = 2 #, k = 3#;
   print(i); print(j)
END
FINISH
```

The advantage of commenting out source is that you only have to remove two characters and that source can be included in the program again. You can use any of the comment symbols for commenting out. Here is another program with a part of the program containing a comment commented-out:

```
PROGRAM prog CONTEXT VOID
USE standard
BEGIN
   INT i = 1;
   COMMENT
   REAL six = 6.0, # Used subsequently #
        one by 2 = 0.5;COMMENT
   CHAR x = "X";
   print(i); {print(six);} print(x)
END
FINISH
```

This is an example of nested comments. You can use any of the comment symbols for this purpose as long as you finish the comment with the matching symbol. However, if the part of your program that you want to comment out already contains comments, you should ensure that the enclosing comment symbols should be different. One way of using comment symbols is to develop a standard method. For example, the author uses the `#...#` comment symbols for one line comments in the code, `CO` symbols for multiline comments and `COMMENT` symbols for extensive comments required at the start of programs or similar code chunks.

Exercises

1.14 Write a short program which will print the letters of your first name. You should declare an identifier of mode `CHAR` for each letter, and write a `print` phrase for each letter. Remember to put semicolons in the right places. Add comments to your program to explain what the program does. Ans

---

1.10 External values

Values denoted or manipulated by a program are called **internal** values. Values which exist outside a program and which are data used by a program or data produced by a program (or both) are known as **external** values.

In the previous sections we have been learning how plain values are denoted in Algol 68 programs. This internal display of values is not necessarily the same as that used for external values. If you copy the following program into a file and compile and run it you will get

␣␣␣␣␣␣␣␣+10A␣+.15000000000000000e␣+1

output on your screen.

```
PROGRAM test CONTEXT VOID
USE standard
BEGIN
   print(10);  print("A");  print(1.5)
END
FINISH
```

Notice that although the denotation for the first letter of the alphabet is surrounded by quote characters, when it is displayed on your screen, the quote characters are omitted. The rules for numbers are as follows: if a number is not the first value in the line it is preceded by a space. Integers are always printed in the space required by `max int` plus one position for the sign. Both positive and negative integers have a sign. A real number is always printed using the print positions required by `max real`, plus a sign for the number. The exponent is also preceded by a sign. If you want extra spaces, you have to insert them.

Try the following program:

```
PROGRAM print2 CONTEXT VOID
USE standard
BEGIN
   print(10); print(blank); print("A");
   print(0.015); print(0.15); print(1.5);
   print(15.0); print(150.0); print(1500.0);
   print(15e15)
END
FINISH
```

1.11 Summary

An Algol 68 program manipulates values. A value is characterised by its mode. A mode is indicated by a mode indicant. Plain values can be denoted. Values occur in contexts, and can sometimes be coerced into values of different modes. Identifiers can be linked to values using identity declarations. The values manipulated by a program are called internal values. External values are data used by, or produced by, a program. Comments describe a program, but add nothing to its elaboration.

Finally, here are some exercises which test you on concepts you have met in this chapter.

Exercises

1.15 Give denotations of the following values: Ans

(a) one thousand nine hundred and ninety six.
(b) The fifth letter of the lower-case Roman alphabet.
(c) The fraction $\frac{1}{7}$ expressed as a decimal fraction to 6 decimal places.

1.16 Is there anything wrong with the following mode indicants? Ans

(a) `C H A R`
(b) `INT.CHAR`
(c) `THISISANEXTREMELYLONGMODEINDICANT`
(d) `2CHAR`

1.17 Write suitable identity declarations for the following identifiers: Ans

(a) `fifty five`
(b) `three times two point seven`
(c) `colon`

1.18 Is there anything wrong with the following identity declarations?

```
REAL x = 1.234,
     y = x;
```

Ans

1.19 What is the difference in meaning between `0` and `0.0`? Ans

1.20 Write a program containing `print` phrases to print the following values on your screen, separated by one space between each value:

```
0.5   "G"   1   ":"   34000000
```

Ans

Chapter 2

Formulæ

Formulæ consist of **operators** with operands. Operators are predeclared pieces of program which compute a value determined by their operands. Algol 68 is provided with a rich set of operators in the standard prelude and you can define as many more as you want. In this chapter, we shall examine all the operators in the standard prelude which can take operands of mode `INT`, `REAL` or `CHAR`. In chapter 6, we shall return to operators and look at what they do in more detail, as well as how to define new ones.

Operators are written as a combination of one or more symbols, or in capital letters like a mode indicant. We shall meet both kinds in this chapter.

2.1 Monadic operators

Operators come in two flavours: **monadic** and **dyadic**. A monadic operator has only one operand, but a dyadic operator has two operands. A monadic operator is written before its operand. For example, the monadic minus `-` reverses the sign of its operand:

```
-3000
```

This could equally well be written `- 3000` since spaces are, generally speaking, not significant. There is, likewise, a monadic `+` operator which doesn't do anything to its operand, but is useful where you want to refer expressly to a positive number. It has been provided for the sake of consistency. You should note that `-3000` is not a denotation, but a formula consisting of a monadic operator operating on an operand which is a denotation. We say that the monadic operator `-` takes an operand of mode `INT` and **yields** a value of mode `INT`. It can also take an operand of mode `REAL` when it will yield a value of mode `REAL`.

A formula can be used as the value part of an identity declaration. Thus the following identity declarations are both valid:

```
INT  minus 2 = -2;
REAL minus point five = -0.5
```

The operator `ABS` takes an operand of mode `INT` and yields the absolute value again of mode `INT`. For example, `ABS -5` yields the value denoted by `5`:

```
INT five = ABS -5
```

Note that when two monadic operators are combined, they are elaborated in right-to-left order, as in the above example. That is, the `-` acts on the `5` to yield `-5`, then the `ABS` acts on `-5` to yield `+5`. This is just what you might expect. `ABS` can also take an operand of mode `REAL` yielding a value of mode `REAL`. For example:

```
REAL x = -1.234;
REAL y = ABS x
```

Another monadic operator which takes an `INT` operand is `SIGN`. This yields -1 if the operand is negative, 0 if it is zero, and $+1$ if it is positive. Thus you can declare

```
INT res = SIGN i
```

if `i` has been previously declared.

2.2 Dyadic operators

A dyadic operator takes two operands and is written between them. The simplest operator is dyadic `+`. Here is an identity declaration using it:

```
INT one = 1;
INT two = one + one
```

This operator takes two operands of mode `INT` and yields a result of mode `INT`. It is also defined for two operands of mode `REAL` yielding a result of mode `REAL`:

```
REAL x = 1.4e5 + 3.7e12
```

The `+` operator performs an action quite different for `REAL` operands from that performed for `INT` operands. Yet the meaning is essentially the same, and so the same symbol is used for the two operators.

Before we continue with the other dyadic operators, a word of caution is in order. As we have seen, the maximum integer which the computer can use is `max int` and the maximum real is `max real`. The dyadic `+` operator could give a result which is greater than those two values. Adding two integers such that the sum exceeds `max int` is said to give "integer overflow". Algol 68 contains no specific rules about what should happen in such a case.[1]

The dyadic `-` operator can take two operands of mode `INT` or two operands of mode `REAL` and yields an `INT` or `REAL` result respectively:

```
INT minus 4 = 3 - 7,
REAL minus one point five = 1.9 - 3.4
```

Note that the dyadic `-` is quite different from the monadic `-`. You can have both operators in the same formula:

[1] The standard prelude supplied with the Linux port of the `a68toc` compiler provides a means of specifying what should be done if integer overflow occurs, if it can be detected. See section 13.3.13 for the details. Likewise for "floating-point overflow" and "floating-point underflow", see section 13.6.1.

```
INT minus ten = -3 - 7
```

The first minus sign represents the monadic operator and the second, the dyadic.

Since a formula yields a value of a particular mode, you can use it as an operand for another operator. For example:

```
INT six = 1 + 2 + 3
```

The operators are elaborated in left-to-right order. First the formula `1+2` is elaborated, then the formula `3+3`. What about the formula `1-2-3`? Again, the first `-` operator is elaborated giving `-1`, then the second giving the value `-4`.

Instead of saying "the value of mode `INT`", we shall sometimes say "the `INT` value" or even "the `INT`"—all these expressions are equivalent.

Exercises

2.1 Write an identity declaration for the `INT` value `-35`. Ans

2.2 What is the value of each of the following formulæ? Ans

(a) `3 - 2`
(b) `3.0 - 2.0`
(c) `3.0 - -2.0`
(d) `2 + 3 - 5`
(e) `-2 + +3 - -4`

2.3 Given the following declarations

```
INT a = 3, REAL b = 4.5
```

what is the value of the following formulæ? Ans

(a) `a+a`
(b) `-a-a`
(c) `b+b+b`
(d) `-b - -b + -b`

2.3 Multiplication

The operand `*` (often said "star") represents normal arithmetic multiplication and takes `INT` operands yielding an `INT` result. For example:

```
INT product = 45 * 36
```

Likewise, `*` is also defined for multiplication of two values of mode `REAL`:

```
REAL real product = 2.4e-4 * 0.5
```

It is important to note that although the actions of the two operators are quite different, they both represent multiplication so they both use the same symbol.

Like `+` and `-`, multiplication can occur several times:

```
INT factorial six = 1 * 2 * 3 * 4 * 5 * 6
```

the order of elaboration being left-to-right.

You can also combine multiplication with addition and subtraction. For example, the value of the formula `2+3*4` is `14`. At school, you were probably taught that multiplication should be done before addition (your teachers may have used the mnemonic BODMAS to show the order in which operations are done. It stands for Brackets, Over, Division, Multiplication, Addition and Subtraction). In Algol 68, the same sort of thing applies and it is done by operators having a **priority**. The priority of multiplication is higher than the priority for addition or subtraction. The priority of the dyadic `+` and `-` operators is 6, and the priority of the `*` operator is 7.

Here are identity declarations using a combination of multiplication and addition and subtraction:

```
INT  i1 = 3, i2 = -7;
INT  result1 = i1 * i2 - 8;
REAL r1 = 35.2, r2 = -0.04;
REAL result2 = r1 * -r2 + 12.67 * 10.0
```

In the elaboration of `result2`, the multiplications are elaborated first, and then the addition.

Remember from chapter 1 that widening is allowed in the context of the right-hand side of an identity declaration, so the following declaration is valid:

```
REAL a = 24 * -36
```

It is important to note that an operand is not in a strong context, so no widening is allowed. The context of an operand is **firm**. Because widening is not allowed in a firm context, it is possible for the compiler to examine the modes of the operands of an operator and determine which declaration of the operator is to be used in the elaboration of the formula. This also applies to monadic operators (see 6.2.1 for details).

Looking again at the above identity declaration, the context of the denotation `36` is firm (it is the operand of the monadic `-`), the contexts of the `24` and the `-36` are also firm because they are the operands of the dyadic `*`, but the value yielded by the formula is on the right-hand side of the identity declaration, so it is in a strong context. It is this value which is coerced to a value of mode `REAL` by

the widening. Note that the value of the formula (which has mode `INT`) does not change. Instead, it is replaced by the coercion with a value of mode `REAL` whose whole number part has the same value as the `INT` value. It is worth saying that the value of the formula obtained by elaboration is lost after the coercion. You could hang on to the intermediate integer value by using another identity declaration:

```
INT intermediate value = 24 * -36;
REAL a = intermediate value
```

Exercises

2.4 In this exercise, these declarations are assumed to be in force:

```
INT  d1 = 12, d2 = -5;
REAL d3 = 4.0 * 3.5, d4 = -3.0
```

What is the value of each of the following formulæ? Ans

(a) `ABS d2`
(b) `- ABS d4 + d3 * d4`
(c) `d2 - d1 * 3 + d2 * 4`

2.4 Division

In the preceding sections, all the operators mentioned yield results which have the same mode as the operand or operands. In this and following sections, we shall see that this is not always the case.

Division poses a problem because division by integers can have two different meanings. For example, $3 \div 2$ can be taken to mean `1` or `1.5`. In this case, we use two different operator symbols.

Integer division is represented by the symbol `%`. It takes operands of mode `INT` and yields a value of mode `INT`. It has the alternative representation `OVER`. The formula `7 % 3` yields the value `2`, and the formula `-7 % 3` yields the value `-2`. The priority of `%` is 7, the same as multiplication. Here are some identity declarations using the operator:

```
INT r = 23 OVER 4, s = -33 % 3;
INT q = r * s % 2
```

Using the given values of `r` and `s`, the value of `q` is `-27`. When a formula containing consecutive dyadic operators of the same priority is elaborated, elaboration is always left-to-right, so in this case the multiplication is elaborated first, followed by the integer division. Of course, `%` can be combined with subtraction as well as all the other operators already discussed.

The modulo operator `MOD` gives the remainder after integer division. It requires two operands of mode `INT` and yields a value also of mode `INT`. Thus `5 MOD 3` yields `2`, and `12 MOD 3` yields `0`. It does work with negative integers, but the results are

unexpected. You can explore `MOD` with negative integers in an exercise. `MOD` can also be written as `%*`. The priority of `MOD` is 7.

Division of real numbers is performed by the operator `/`. It takes two operands of mode `REAL` and yields a `REAL` result. Thus the formula `3.0/2.0` yields `1.5`. Again, `/` can be combined with `*` and the other operators already discussed. It has a priority of 7. The operator is also defined for integer operands. Thus `3/2` yields the value `1.5`. No widening takes place here since the operator is defined to yield a value of mode `REAL` when its operands have mode `INT`.

Here are some identity declarations using the operators described so far:

```
REAL pi by 2 = pi / 2,
     pm3 = pi - 3.0 * -4.1;
INT c = 22 % 3 - 22 MOD 3;
INT d = c MOD 6 + SIGN -36
```

Exercises

2.5 What is the value yielded by each of the following formulæ, and what is its mode? Ans

(a) `5 * 4`
(b) `5 % 4`
(c) `5 / 4`
(d) `5 MOD 4`
(e) `5.0 * 3.5 - 2.0 / 4.0`

2.6 Write a short program to print the results of using `MOD` with negative integer operands. Try either operand negative, then both operands negative. Ans

2.7 Give an identity declaration for the identifier `two pi`. Ans

2.5 Exponentiation

If you want to compute the value of `3*3*3*3` you can do so using the multiplication operator, but it would be clearer and faster if you used the exponentiation operator `**`. The mode of its left operand can be either `REAL` or `INT`, but its right operand must have mode `INT`. If both its operands have the mode `INT`, the yield will have mode `INT` (in this case the right operand must not be negative), otherwise the yield will have mode `REAL`. Thus the formula `3**4` yields the value `81`, but `3.0**4` yields the value `81.0`. Its priority is 8. In a formula involving exponentiation as well as multiplication or division, the exponentiation is elaborated first. For example, the formula `3*2**4` yields `48`, not `1296`.

Every dyadic operator has a priority of between 1 and 9 inclusive, and all monadic operators bind more tightly than all dyadic operators. For example, the formula `-2**2` yields `4`, not `-4`. Here the monadic minus is elaborated first, followed by the exponentiation.

Exercises

2.8 Given these declarations:

```
INT two = 2, m2 = -2;
REAL x = 3.0 / 2.0, y = 1.0
```

what is the value and mode yielded by the following formulæ? Ans

(a) `two ** -m2`
(b) `x ** two + y ** two`
(c) `3 * m2 ** two`

2.6 Mixed arithmetic

Up to now, the four basic arithmetic operators have always had operands of the same modes. In practice, it is quite surprising how often you want to compute something like `2 * 3.0`. Well, fortunately, the dyadic operators `+`, `-`, `*` and `/` (but not `%`) are also defined for mixed modes. That is, any combination of `REAL` and `INT` can be used. With mixed modes the yield is always `REAL`. Thus the following formulæ are all valid:

```
1+2.5    3.1*-4    2*3.5**3    2.4-2
```

The priority of the mixed-mode operators is unchanged. As we shall see later, the priority relates to the operator symbol rather than the flavour of the operator in use.

2.7 Order of elaboration

Even though the order of elaboration is dependent on the priority of operators, it is often convenient to change the order. This can be done by inserting parentheses `(` and `)` (or `BEGIN` and `END`): the formula inside the parentheses is evaluated first. Here are two formulæ which differ only by the insertion of parentheses:

```
3 * 4 - 2
3 *(4 - 2)
```

The first has the value `10`, and the second `6`. Parentheses can be nested to any depth.

```
REAL a = (3*a3*(xmin+eps1)**2)/4;
REAL alpha g=(ymax - ymin)/(xmax - xmin);
INT p=BEGIN 2 * 3**4 % (13-2**3) END - 4.0
```

It is uncommon to find `BEGIN` and `END` in short formulæ. If you use `BEGIN` at the start of a formula, you must use `END` to complete it even though these symbols and parentheses are equivalent.

2.8 Changing the mode

We have seen that in a strong context, a value of mode `INT` can be coerced by widening to a value of mode `REAL`. What about the other way round? Is it possible to coerce a value of mode `REAL` to a value of mode `INT`? Fortunately, it is impossible using coercion. The reason behind this is related to the fact that real numbers can contain fractional parts. In replacing an integer by a real number there is no essential change in the value, but when a real number is changed to an integer, in general the fractional part will be lost. It is undesirable that data should be lost without the programmer noticing.

If you want to convert a `REAL` value to an `INT`, you must use one of the operators `ROUND` or `ENTIER`. The operator `ROUND` takes a single operand of mode `REAL` and yields an `INT` whose value is the operand rounded to the nearest integer. Thus `ROUND 2.7` yields `3`, and `ROUND 2.2` yields `2`. The same rule applies with negative numbers, thus `ROUND -3.6` yields `-4`. At the half way case, for example, `ROUND 2.5`, the value is rounded away from zero if the whole number part is odd, and rounded toward zero if it is even (zero, in this case, is taken to be an even number). This ensures that rounding errors over a large number of cases tend to cancel out.

The operator `ENTIER` (French for "whole") takes a `REAL` operand and likewise yields an `INT` result, but the yield is the largest integer equal to or less than the operand. Thus `ENTIER 2.2` yields `2`, `ENTIER -2.2` yields `-3`.

The operator `SIGN` can also be used with a `REAL` operand. Its yield has mode `INT` with the same values as before, namely: `-1` if the operand is negative, `0` if it is zero, and `+1` if it is positive. We shall see in subsequent chapters that this property of `SIGN` can be useful.

Exercises

2.9 What is the value and mode of the yield of each of the following formulæ? Ans

(a) `ROUND(3.0 - 2.5**2)`
(b) `ENTIER -4.5 + ROUND -4.5`
(c) `SIGN(ROUND 3.6 / 2.0) * 2.0`

2.10 What is the value of the formula

```
(ENTIER -2.9 + 3**2)/4.0
```

Ans

2.9 Miscellaneous operators

The operators `MAX` and `MIN` are defined for any combination of `INT` and `REAL` operands and yield the maximum, or minimum, of two values. They can also be combined in the same formula:

```
INT max min = 345 MAX 249 MIN 1000
```

which yields `345`. Like `+`, `-`, and `*`, they only yield a value of mode `INT` if both their operands are `INT`. Otherwise, they yield a value of mode `REAL`. They both have a priority of 9.

2.10 Operators using `CHAR`

This chapter has been rather heavy on arithmetic up to now. You might wonder whether operators can have operands of mode `CHAR`. The answer is yes. Indeed, the `+` and `*` operators are so declared, and we shall meet them in chapter 3. There are two monadic operators which involve the mode `CHAR`. The operator `ABS` (which we have already met) can take a `CHAR` operand and yields the integer corresponding to that character. For example, `ABS "A"` yields `65` (the number associated with the letter `"A"` as defined by the ASCII standard). The identifier `max abs char` is declared in the standard prelude with the value `255`. Conversely, we can convert an integer to a character using the monadic operator `REPR`. The formula

```
REPR 65
```

yields the value `"A"`. `REPR` can act on any integer in the range `0` to `max abs char`. `REPR` is of particular value in allowing access to control characters. For example, the tab character is declared in the standard prelude as `tab ch`. Consult section 13.2.2 for the details.

2.11 `print` revisited

In chapter 1, we used the `print` phrase to convert internal values to external characters. We ought to say what `print` is and how it works, but we don't yet know enough about the language. Just use it for the moment, and we shall learn more about it later.

Besides being able to convert internal values to external characters, `print` can take two parameters (see chapter 6 for the low-down on parameters) which can be used to format your output. `newline` will cause following output to be displayed on a new line, and `newpage` will emit a form-feed character (`REPR 12`). `newline` and `newpage` will be described in detail in section 13.7.11.

If you want to print the characters emitted by your Algol 68 programs you can use file redirection to redirect your output to a file, which you can later copy to the printer. For example, suppose you have compiled a program called `tt`. To redirect its output to a file called `tt.res`, which you can later copy to the printer, you issue the command

```
tt > tt.res
```

at the command line. Alternatively, you send the output directly to the printer using the command

```
tt | lpr
```

at the command line. Try compiling and running the following program:

```
PROGRAM tt CONTEXT VOID
USE standard
BEGIN
   print(newpage);
   INT a = ENTIER (3.6**5);
   REAL p = 4.3 / 2.7;
   print(a);  print(newline);
   print(b);  print(newline)
END
FINISH
```

2.12 Summary

Operators combined with operands are called formulæ. Operators are monadic or dyadic. Monadic operators take a single operand, bind more tightly than dyadic operators and when combined are elaborated from right to left. Dyadic operators take two operands and have a priority of 1 to 9. Successive dyadic operators having the same priority are elaborated from left to right. Parentheses, or `BEGIN` and `END`, may be used to alter the order of elaboration.

A summary of all the operators described in this chapter, together with their priorities, can be found in chapter 13.

Here are some exercises which test you on what you have learned in this chapter. The exercises involving `ABS` and `REPR` will need to be written as small programs and compiled and run. In fact, it would be a good idea to write all the answers as small programs (or incorporate them all in one large program). Don't forget to use the `print` phrase with `newline` and `newpage` to separate your output.

Exercises

2.11 The following declarations are assumed to be in force for these exercises:

```
INT i = 13, j = -4, k = 7;
CHAR s = "s", t = "T";
REAL x = -2.4, y = 2.7, z = 0.0
```

What is the value of each of the following formulæ? Ans

(a) `(2 + 3) * (3 - 2)`
(b) `j+i-k`
(c) `3*ABS s`
(d) `ABS"t"-ABS t`
(e) `REPR(k**2)`

(f) `ROUND(x**2-y/(x+1))`
(g) `z**9`

2.12 Because of the kind of arithmetic performed by the compiler, division of values of mode `REAL` by zero does not cause a program to fail (but see section 13.6.1). Write a program containing the phrases `REAL z=0.0/0.0;` and `REAL iz=1/0;` and see what happens. In practice, it's probably a good idea to check for division by zero. Ans

2.13 Now try the phrase `print(1%0)`. Ans

2.14 What is wrong with the following formulæ? Ans

(a) `[4-j]*3`
(b) `(((3-j)*x+3)*x+5.6`
(c) `ROUND "e"`
(d) `ENTIER 4 + 3.0`

Chapter 3

Repetition

Up to now, we have dealt with plain values: that is, values with modes `INT`, `REAL` or `CHAR`. In practice, plain values are of limited use when dealing with a lot of data. For example, commercial programs are continually dealing with strings of characters and engineers use vectors and matrices. In this chapter, we start the process of building more complicated modes. Firstly, we consider repetition of values.

3.1 Multiples

A **multiple** consists of a number of **elements**, each of which have the same mode (sometimes known as the **base mode**). The mode of a multiple consists of the mode indicant for each element preceded by brackets, and is said "row of mode". For example, here is an identity declaration of a row of `CHAR` multiple:

```
[]CHAR a = "abcd"
```

The phrase on the left-hand side of the equals symbol is read "row of car a". The phrase on the right-hand side of the equals symbol is the denotation of a value whose mode is `[]CHAR`. Spaces can, of course, appear before, between or after the brackets.

Multiples of mode `[]CHAR` are so common that this denotation was devised as a kind of shorthand. The maximum number of elements in a multiple is equal to the maximum positive integer (`max int`), although in practice, your program will be limited by the available memory. The denotation of a `[]CHAR` may extend over more than one line. There are two ways of doing this. You can simply write the denotation on more than one line in which case every character "between" the starting and ending quote characters is included except the newline characters, or you can split the denotation with quote characters at the end of one line and at the start of the continuation of the denotation on the next line. Here are two declarations which exemplify these rules:

```
[]CHAR long1 = "The first stage in the develo
pment of a new program consists of analysing
the problem that the program must solve.";
[]CHAR long2 = "The first stage in the "
           "development of a new "
           "program consists of "
           "analysing the problem "
           "that the program must "
           "solve."
```

Notice that the second method is neater because you can indent the subsequent parts of the denotation. Everything "between" the second and third quote characters and "between" the fourth and fifth quote characters is ignored, although you should not put anything other than spaces or tabs and newlines there. If you want to place a quote character (") in the denotation, you must double it, just as in the character denotation. Here are two `[]CHAR` denotations, each containing two quote characters:

```
[]CHAR rca = """Will you come today?""",
       rcb = "The minority report stated "
             "that ""in their opinion""";
```

The repeated quote characters are different from the quote characters which chain the two parts of the denotation of `rcb`.

3.1.1 Row-displays

Multiples of other modes cannot be denoted as shown above, but use a construct called a **row-display**. A row-display consists of none or two or more units separated by commas and enclosed by parentheses (or `BEGIN` and `END`). Here is the identity declaration for `a` written using a row-display:

```
[]CHAR a = ("a","b","c","d")
```

It is important to note that the units in the row-display could be quite complicated. For example, here is another declaration for a multiple with mode `[]CHAR`:

```
[]CHAR b = ("a","P",REPR 36,"""")
```

In each of these two declarations, the number of elements is `4`.

Here are identity declarations for a multiple of mode `[]INT` and a multiple of mode `[]REAL`:

```
[]INT  c = (1, 2+3, -2**4, 7, -11, 2, 1);
[]REAL d = (1.0, -2.9, 3e4, -2e-2, -5)
```

Note that the last unit of the row-display for `c` has the same value as the first unit. In a multiple of mode `[]INT`, the individual elements can have any value of mode `INT`: that is to say, any integer or formula yielding an integer. In `d`, the unit yielding the last element is written as a formula yielding a value of mode `INT`. Since the context of the row-display is strong (because it occurs on the right-hand side of an identity declaration), this context is passed on to its constituent units. Thus, the context of the formula is also strong, and so the value yielded by the formula is widened to yield `-5.0`.

An empty row-display can be used to yield a **flat** multiple (one with no elements). For example, here is an identity declaration using an empty row-display:

```
[]REAL empty = ()
```

The denotation for a flat `[]CHAR` is used in the identity declaration

```
[]CHAR none = ""
```

A multiple can also have a single element. However, a row-display cannot have a single unit (because it would be an enclosed clause, which is a different construct). In this case, we use a unit (or a formula, which is another kind of unit) for the only element, and the value of that unit is coerced to a multiple with a single element using the **rowing** coercion. For example,

```
[]INT ri = 4
```

yields a multiple with one element. An enclosed clause can be used instead:

```
[]INT ri1 = (4)
```

since an enclosed clause is also a unit (see section 10.4).

Rowing can only occur in strong contexts (and the right-hand side of an identity declaration is a strong context). Here is another example:

```
[]CHAR rc = "p"
```

A row-display can only be used in a strong context. Because the context of an operand is firm, a row-display cannot appear in a formula (but there is a way round this, see section 10.5). The shorthand denotation for a `[]CHAR` is not a row-display and so does not suffer from this limitation.

3.1.2 Dimensions

One of the properties of a multiple is its number of **dimensions**. All the multiples declared so far have one dimension. The number of dimensions affects the mode. A two-dimensional multiple of integers has the mode

```
[,]INT
```

(said "row-row-of-int"), while a 3-dimensional multiple of reals (real numbers) has the mode

```
[,,]REAL
```

Note that the number of commas is always one less than the number of dimensions. In Algol 68, multiples of any number of dimensions can be declared.[1]

To cater for more than one dimension, each of the units of a row-display can also be a row-display. For example, the row-display for a multiple with mode `[,]INT` could be

```
((1,2,3),(4,5,6))
```

The fact that this is the row-display for a 2-dimensional multiple would be clearer if it were written

```
((1,2,3),
 (4,5,6))
```

For two dimensions, it is convenient to talk of "rows" and "columns". Here is an identity declaration using the previous row-display:

```
[,]INT e = ((1,2,3),
            (4,5,6))
```

The first "row" of `e` is yielded by the row-display `(1,2,3)` and the second "row" is yielded by `(4,5,6)`. The first "column" of `e` is yielded by the row-display `(1,4)`, the second "column" by `(2,5)` and the third "column" by `(3,6)`. Note that the number of elements in each "row" is the same, and the number of elements in each "column" is also the same, but that the number of "rows" and "columns" differ. We say that `e` is a **rectangular** multiple. If the number of "rows" and "columns" are the same, the multiple is said to be **square**. Here is an identity declaration for a square multiple:

```
[,]CHAR f = (("a","b","c"),
             ("A","B","C"),
             ("1","2","3"))
```

All square multiples are also rectangular, but the converse is not true. Note that in the row-display for a multi-dimensional multiple of characters, it is not possible to use the special denotation for a `[]CHAR`.

The base mode of a multiple can be any mode, including another row mode. For example:

[1] The a68toc Algol 68 compiler supports dimensions up to three. If you try to declare rows having more than three dimensions, the translation proceeds without error messages, but the resulting C code will fail to compile.

```
[][]CHAR days =
   ("Monday","Tuesday","Wednesday",
    "Thursday","Friday","Saturday",
    "Sunday")
```

The mode is said "row of row of CHAR". Note that days is one-dimensional, each element consisting of a one-dimensional []CHAR. The shorthand denotation for a []CHAR can be used in this case. Because the base mode is []CHAR, the individual []CHARs can have different lengths. Here is another example using integers:

```
[][]INT trapezium = ((1,2),(1,2,3),(1,2,3,4))
```

3.1.3 Subscripts and bounds

Each element of a multiple has one integer associated with it for each dimension. These integers increase by 1 from the first to the last element in each dimension. For example, in the declaration

```
[]INT r1 = (90,95,98)
```

the integers associated with the elements are [1], [2] and [3] (see the next section for an explanation of why the integers are written like this). Remember that the first element in a row-display always has an associated integer of [1]. These integers are known as subscripts[2] or indexers. Thus the subscript of 98 in r1 is [3]. In the two-dimensional multiple

```
[,]INT r2 = ((-40, -30, -20),
             (100, 130, 160))
```

the subscripts for -40 are [1,1] and the subscripts for 160 are [2,3].

We say that the **lower bound** of r1 is 1, and its **upper bound** is 3. The multiple r2 has a lower bound of 1 for both the first and second dimensions, an upper bound of 2 for the first dimension (2 "rows") and an upper bound of 3 for the second dimension (3 "columns"). We shall write the bounds of r1 and r2 as [1:3] and [1:2,1:3] respectively. The bounds of a flat multiple, unless specified otherwise (see the section on trimming), are [1:0].

The bounds of a multiple can be interrogated using the operators LWB for the lower bound, and UPB for the upper bound. The bounds of the first, or only, dimension can be interrogated using the monadic form of these operators. For example, using days defined above, LWB days yields 1, and UPB days yields 7. Where the multiple is multi-dimensional, the bounds are interrogated using the dyadic form of LWB and UPB: the left operand is the dimension while the right operand is the identifier of the multiple. For example, 1 UPB r2 yields 2 and 2 UPB r2 yields 3. The priority of the dyadic operators is 8.

[2] From the practice of mathematicians who write $x_1, x_2, \ldots$

Exercises

3.1 What is wrong with the following identity declarations? Ans

(a) `()CHAR c1 = "Today"`
(b) `[]CHAR c2 = 'Yesterday'`
(c) `[]INT i1 = (1, 2.0, 3)`

3.2 Using the identifier `first 4 odd numbers`, write an appropriate identity declaration. Ans

3.3 Given the identity declarations

```
[]CHAR s = "abcdefgh";
[]REAL r = (1.4e2, 3.5e-1, -4.0);
[,]INT t = ((2,3,5),
            (7,11,13),
            (17,19,23))
```

what is the value of the following: Ans

(a) `UPB s`
(b) `LWB r`
(c) `2 UPB t - 1 LWB t + 1`

3.4 Write the formulæ which give the upper and lower bounds of each of the following multiples: Ans

(a)

```
[,,]INT a = (((1,2,3),
              (4,5,6)),
             ((7,8,9),
              (10,11,12)))
```

(b) `[]REAL b = ()`

3.2 Slicing

In the previous section, it was mentioned that a subscript is associated with every element in a multiple. The lower-bound of the multiple for a dimension determines the minimum subscript for that dimension and the upper-bound for that dimension determines the maximum subscript. Thus there is a set of subscripts for each dimension. The individual elements can be accessed by quoting all the subscripts for that element. For example, the elements of the multiple

```
[]INT odds = (1,3,5)
```

can be accessed as `odds[1]`, `odds[2]` and `odds[3]`. The first of these is read "odds sub one bus" ("bus" is the opposite of "sub"). In a multi-dimensional multiple, two or more subscripts are required to access a single element, the subscripts being separated by commas. For example, in the multiple

```
[,]REAL rs = ((1.0, 2.0, 3.0),
              (4.0, 5.0, 6.0))
```

`rs[1,2]` yields `2.0`. Similarly, `rs[2,3]` yields `6.0`. Thus one can declare

```
REAL rs12 = rs[1,2],
     rs23 = rs[2,3]
```

Although, technically, a multiple with all its subscripts specified is called a **slice**, the term is usually reserved for a multiple with less than the maximum number of subscripts (in other words, at least one of the dimensions does not have a subscript). For example, using `rs` declared above, we can write

```
[]REAL srs = rs[1,]
```

which yields the multiple denoted by `(1.0,2.0,3.0)`. The comma must be present in the slice on the right-hand side otherwise the compiler will report an error of "wrong number of indices".

Vertical slicing is also possible. The phrase `rs[,2]` yields the multiple `(2.0,5.0)`. In the context of the declaration

```
[,]CHAR rs2 = (("a","b","c","d"),
               ("e","f","g","h"),
               ("i","j","k","l"))
```

the slice `rs2[,3]` yields the value `"cgk"` with a mode of `[]CHAR`. Note, however, that vertical slicing is only possible for multiples with at least two dimensions. The multiple `days`, declared in the previous section, is one-dimensional and so cannot be sliced vertically.

In a 3-dimensional multiple, both 2-dimensional and 1-dimensional slices can be produced. Here are some examples:

```
[,,]INT r3 = (((1,2),(3,4),((5,6),(7,8)));
[,]INT r31 = r3[1,,],
       r32 = r3[,2,],
       r33 = r3[,,3];
[]INT r312 = r31[2,], r4 = r31[,2]
```

Exercises

3.5 The declaration

```
[,]INT r = (( 1, 2, 3, 4),
            ( 5, 6, 7, 8),
            ( 9,10,11,12),
            (13,14,15,16))
```

is in force for this and the following exercise. Give the value of the following slices: Ans

(a) `r[2,2]`
(b) `r[3,]`
(c) `r[,2 UPB r]`

3.6 Write slices for the following values Ans

(a) `10`
(b) `(5,6,7,8)`
(c) `(3,7,11,15)`

3.3 Trimming

The bounds of a multiple can be changed using the `@` construction. For example, in the declaration

```
[]CHAR digits = "0123456789"[@0]
```

the bounds of `digits` are `[0:9]`. Bounds do not have to be non-negative. For example,

```
[,]INT ii = ((1,2,3),(4,5,6));
[,]INT jj = ii[@-3,@-50]
```

whence the bounds of `jj` are `[-3:-4,-50:-48]`. Notice that you cannot change the bounds of a row-display (except by using a **cast**—see section 10.5). For now, always declare an identifier for the display, and then alter the bounds. The bounds of a slice can be changed:

```
[,]INT ij = ((1,3,5),(7,9,11),(13,15,17));
 []INT ij2 = ij[2,][@0]
```

The declaration for `ij2` could also be written

```
[]INT ij2 = ij[2,@0]
```

`@` can also be written `AT`.

Wherever an integer is required in the above, any unit yielding an integer will do. Thus it is quite in order to use the formula

```
(a+b) UPB r
```

where the parentheses are necessary if `a+b` is expected to yield the dimension of `r` under consideration (because the priority of `UPB` is greater than the priority of `+`).

A **trimmer** uses the `:` construction. In the context of the declaration of `digits` above, the phrase `digits[1:3]` yields the value `"123"` with mode `[]CHAR`. Again, using the declaration of `r` in the last set of exercises, `r[1:2,1]` yields `(1,2)`, and `r[1:2,1:2]` yields `((1,2),(5,6))`.

Trimming is particularly useful with values of mode `[]CHAR`. Given the declaration

```
[]CHAR quote = "Habent sua fata libelli"
```

(the quotation at the start of the acknowledgements in the "Revised Report"),

```
quote[:6]
quote[8:10]
quote[12:15]
```

yield the first three words. Note that when the first subscript in a trimmer is omitted, the lower bound for that dimension is assumed, while omission of the second subscript assumes the corresponding upper bound. Again, any unit yielding `INT` may be used for the trimmers. The context for a trimmer or a subscript is **meek**.

Omission of both subscripts yields the whole slice with a lower bound of 1. So, the upper bound of the phrase `digits[:]` is `10` which is equivalent to `digits[@1]`.

The lower bound of a `trimmer` is, by default, `1`, but may be changed by the use of `@`. For example, `digits[3:6]` has bounds `[1:4]`, but `digits[3:6@2]` has bounds `[2:5]`. The bounds of `quote[17:]` mentioned above are `[1:7]`.

Exercises

3.7 Write an identity declaration for `months` on the lines of the declaration of `days` in section 3.1. Ans

3.8 Given the declarations

```
[,]INT i = ((1,-2,3,4),(-5,6,7,8));
[]REAL r= (1.4,0,-5.4,3.6);
[]CHAR s= "abcdefghijklmnopqrstuvwxyz"
                                [@ ABS"a"]
```

what are the values of the following phrases? Ans

(a) `2 UPB i + UPB s[@1]`
(b) `r[2:3]`
(c) `i[2,2] - r[3]`
(d) `i[2,2:]`
(e) `s[ABS"p":ABS"t"]`

3.4 Printing multiples

We have already used `print` to convert plain values to characters displayed on your screen. In fact, `print` can be supplied with a row of values to be converted, so it is quite valid to write

```
[]INT i1 = (2,3,5,7,11,13);  print(i1)
```

You can also present an actual row-display. Instead of using

```
print(2); print(blank); print(3)
```

you can write `print((2,blank,3))`. The doubled parentheses are necessary: the outer pair are needed by `print` anyway, and the inner pair are part of the row-display. Notice that the modes of the elements of the row-display are quite different. We shall learn in chapter 8 how that can be so.

Here is a program which will print the answers to the last exercise.

```
PROGRAM test CONTEXT VOID
USE standard
BEGIN
 [,]INT i = ((1,-2,3,4),(-5,6,7,8));
 []REAL r= (1.4,0,-5.4,3.6);
 []CHAR s= "abcdefghijklmnopqrstuvwxyz"
                                [@ ABS"a"];

 print(("i=",i,newline,
        "r=",r,newline,
        "s=[",s,"]",newline,
        "2 UPB i + UPB s[@1]=",
        2 UPB i+UPB s[@1],newline,
        "r[2:3]=",r[2:3],newline,
        "i[2,2] - r[3]=",
        i[2,2] - r[3],newline,
        "i[2,2:]=",i[2,2:],newline,
        "s[ABS""p"":ABS""t""]=",
        s[ABS"p":ABS"t"],
        newline))
END
FINISH
```

As you can see, `print` will quite happily take values of modes `[]CHAR`, `[,]INT`, `[]REAL` and so on[3]. Notice also that in order to get quote symbols in the last line to be printed, they are doubled. A common mistake is to omit a quote symbol or a closing comment symbol. If your editor provides lexical highlighting (usually called "syntax" highlighting), an omitted quote or comment symbol will cause a large part of your program to be highlighted as though it were a `[]CHAR` or a comment. The mistake will be very clear. If your editor does not support lexical highlighting, you will get an odd message from the compiler (usually to the effect that it has run out of program!).

[3] but a68toc supports multiples of up to three dimensions.

Exercises

3.9 Write short programs to print the answers to all the exercises in this chapter from 3.5. You should insert multiples of `CHAR` at suitable points, as in the example above, so that you can identify the printed answers. Ans

3.5 Operators with multiples

No operators are defined in the standard prelude for multiples whose elements have modes `INT` or `REAL`. This is not a drawback as you will learn in chapter 6. Nor are there any monadic operators in the standard prelude for multiples of `CHAR`. However, multiples of `CHAR` occur so often, that two dyadic operators are available for them.

The operator `+` is defined for all combinations of `CHAR` and `[]CHAR`. Thus, the formula

```
"abc" + "d"
```

yields the value denoted by `"abcd"`. With these operands, `+` acts as a concatenation operator. The operator has a priority of 6 as before.

Multiplication of values of mode `CHAR` or `[]CHAR` is defined using the operator `*`. The other operand has mode `INT` and the yield has mode `[]CHAR`. For example, in the declaration

```
[]CHAR repetitions = "ab" * 3
```

`repetitions` identifies `"ababab"`. The formula could have been written with the integer as the left operand. In both cases, the operator only makes sense with a positive integer.

Exercises

3.10 Given the identity declarations

```
[]CHAR s = "Dog bites man",
       t = "aeiou"
```

what is the value of the following formulæ? Ans

(a) `"M"+s[UPB s-1:]+s[4:10]+"d"+s[2:3]`
(b) `s[5]*3+2*s[6]`

3.6 Ranges

If you cast your mind back to the form of an Algol 68 program, you will remember that it consists of a number of phrases enclosed by `BEGIN` and `END` (or parentheses) preceded by a `PROGRAM` phrase with an optional `USE` phrase. The part of the program enclosed by `BEGIN` and `END` (including the `BEGIN` and `END`) is called a **closed clause**. The important point here is that a closed clause consists of one or more phrases separated by semicolons `;` (the last phrase being a unit), surrounded by parentheses (or `BEGIN` and `END`). Since a declaration is not a unit, the last phrase cannot be a declaration. We say that the value of a closed clause is the value yielded by the final unit. As an example, here is a closed clause with a value of mode `INT`:

```
BEGIN
   INT i = 43;
   print((i,newline));
   i
END
```

An important adjunct of a closed clause is that any identifiers declared in the clause do not exist outside the clause. We say that the **range** of an identifier is confined to that section of the closed clause from its declaration to the end of the clause.

3.7 Program repetition

Having investigated the construction and use of multiple values, it is now time to address repetition of program actions. For example, suppose you wanted to output 8 blank lines. You could write

```
print((newline,newline,newline,newline,
       newline,newline,newline,newline))
```

A simpler way would be to write

```
TO 8 DO print(newline) OD
```

The integer following the `TO` can be any unit yielding an integer (not necessarily positive) in a meek context. If the value yielded is zero or negative, then the ensuing clause enclosed by `DO` and `OD` will not be elaborated at all. The `TO` ... `OD` construct is called a **loop clause** or, more simply, a loop.

If you omit the `TO` integer construct, the loop will be repeated indefinitely. In that case, you would need some way of terminating the program inside the loop.

A more useful form of the loop clause is shown by the following example

```
FOR i TO 10
DO
   print((i,newline))
OD
```

The `i` is an identifier, whose declaration occurs at that point and whose mode is `INT`. The example will print the numbers `1` to `10`, each on its own line. The range

of `i` is the whole of the loop clause, but does not include the unit following `TO`. Any identifier may be used in place of `i`. When the `TO` part is omitted, it is as though `TO` ∞ had been written.

It is possible to modify the number of times the loop is obeyed. The simplest way is to define the starting point using the `FROM` construct. Here is an example:

```
FOR n FROM -10 TO 10 DO print((n,blank)) OD
```

This prints the numbers from `-10` to `+10` on the screen. The integer after `FROM` can be any unit which yields a value of mode `INT` in a meek context. When `FROM` is omitted, it is assumed that the first value of the identifier following `FOR` is 1.

This example prints the square of each of the numbers from `0.2` to `0.9`:

```
FOR number FROM 2 TO 9
DO
   REAL value = number / 10;
   print((value," squared =",
          value * value,newline))
```

In these examples, the value of the identifier has always increased by `1`. The increase can be changed using the `BY` construct. For example, to print the cubes of the even numbers between 30 and 50 inclusive, you could write

```
FOR n FROM 30 BY 2 TO 50
DO
   print((n**3,newline))
OD
```

The `BY` construct is particularly useful for decreasing the value of the identifier:

```
[]CHAR title =
       "Programming Algol 68 Made Easy";

FOR c FROM UPB title BY -1 TO LWB title
DO
   print(title[c])
OD
```

This last example shows how useful the loop clause can be for accessing some of or all of the elements of a multiple. Here is another example:

```
[]INT hh=(7,17,27,37,47);
INT two=2;

FOR i BY 2 TO UPB hh
DO
   print(hh[i] * hh[i])
OD
```

which will print

```
      +49        +729       +2209
```

on one line. Omitting the `BY` construct assumes a default step of 1.

Notice how use of the `LWB` and `UPB` operators ensures that your program does not try to use a subscript outwith the bounds of the multiple. If you try to access an element whose subscript is greater than the upper bound (or less than the lower bound), the program will fail at run-time with an appropriate error message.

An important use of the identity declaration is that of optimisation. In the previous example, the computation of the i^{th} element of `hh` takes a little time, and there is no point in repeating it. In the following example, the identity declaration computes the value of `hh[i]` and the `print` statement uses the resulting value twice:

```
FOR i BY 2 TO UPB hh
DO
   INT hhi = hh[i];
   print((hhi * hhi,newline))
OD
```

Everything said about multiples with elements of mode `INT` or `CHAR` applies equally well to multiples whose elements have mode `REAL`. A `FOR` loop yields no value (cf section 6.1.5).

Exercises

3.11 Write an Algol 68 program which will print the cubes of the numbers from `1` to `25`. Ans

3.12 Write a program which will print the characters of the alphabet backwards, all on one line. Ans

---

3.8 Nested loops

When dealing with two-, and higher-dimensional multiples, it is often necessary to run a subsidiary loop. For example, suppose we wanted to print the square of each element in the multiple declared as

```
[,]INT primes = (( 2, 3, 5, 7),
                 (11,13,17,19),
                 (23,29,31,37),
                 (41,43,47,53))
```

with each row on one line. Here is a piece of program which will do it:

```
FOR i FROM 1 LWB primes TO 1 UPB primes
DO
   []INT pri=primes[i,];

   FOR j FROM LWB pri TO UPB pri
   DO
      INT prij = pri[j];
      print(prij * prij)
```

```
    OD;
    print(newline)
OD
```

Notice the optimisations. The first defines the i^{th} "row", and the second defines the j^{th} element in that "row". The point is that any piece of program can appear inside the loop clause. Loop clauses can be nested to any depth. Because the loop clause is an enclosed clause, it must contain at least one phrase, and the last phrase must be a unit (see chapter 10 for a thorough discussion of units).

Exercises

3.13 Using a nested loop, write a short program to display the first 25 letters of the alphabet on your screen in five rows of five letters. Separate each letter with a comma. Ans

3.14 Write a program to print the value of a 3-dimensional multiple of real numbers which you have declared in your program. Ans

---

3.9 Program structure

In chapter 1, it was mentioned that the basic structure of an Algol 68 program consists of

```
BEGIN
   phrases
END
```

This is not strictly true. It is quite possible to write a program consisting solely of a `DO` loop! For example:

```
PROGRAM dosum
USE standard
FOR i TO 5
DO
   print((i**2,newline))
OD
FINISH
```

3.10 The `FORALL` loop

The `FORALL` loop is not part of Algol 68, but an extension introduced by the a68toc compiler. It is similar to the `FOR` loop, but the identifier has the mode of an element of the multiple under consideration. Look at this example:

```
[]REAL r1 = (1.0,2.0,3.0,4.0,5.0);
FORALL e IN r1 DO print(e * e) OD
```

In the FORALL loop, e takes the value of each element in r1 and so has mode REAL. The compiler generates more efficient code using the FORALL loop by avoiding the normal overheads of the subscripting mechanism. However, the FORALL loop can only be used when all the elements of a dimension are required. If you want to limit the processing to a few elements, you can trim the multiple or use the FOR loop.

The elements of more than one multiple can be combined simultaneously. For example:

```
[]INT i = (1,2,3,4,5),
      j = (11,12,13,14,15);
FORALL ii IN i, jj IN j
DO
   print((ii * jj,newline))
OD
```

The comma between ii IN i and jj IN j means that the constructs are elaborated collaterally. The bounds of i must be the same as the bounds of j.

FORALL clauses can be nested as in the case of FOR clauses. If we use l and m declared in a previous example, then

```
FORALL ll IN l
DO
   FORALL mm IN m
   DO
      print(ll * mm)
   OD
OD
```

could be used to print the products of all the integers.

3.11 Summary

Modes of multiples start with brackets ([]). A multiple of characters has a special denotation. All multiples can be constructed using a row-display. Rows have bounds and dimensions. Rows can be sliced and trimmed, and their bounds can be changed using the @ construct.

The FOR loop has the form

```
FOR id FROM a BY b TO c DO ... OD
```

where the default values of a, b and c are 1, 1 and ∞ respectively, but may take any value of mode INT in a meek context. If c is greater than or equal to a and b is negative, the loop will not be executed. If b is zero, the loop will be executed indefinitely. The range of id excludes the units a, b and c. The FORALL loop has the form

```
FORALL id1 IN row1 DO ... OD
```

We have covered a good deal of ground in this chapter, so here are some more exercises revising what you have learnt. It is most instructive to verify your answers by writing appropriate Algol 68 programs.

Exercises

3.15 What is wrong with the following identity declarations? Ans

(a) `[]REAL r1 = [2.5,-2.5,3.5]`
(b) `[,]INT i1 = ((1,2,3),(4,5,6,7))`
(c) `[]CHAR s1 = "abcde'fg"`

3.16 What are the upper and lower bounds of the following? Ans

(a) `((10,20,30),(-10,-20,-30))`
(b) `("a","b","c")`
(c) `"abcdef"[3:4]`

3.17 If `a` is declared as

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

what is the value and mode of Ans

(a) `a[2,]`
(b) `a[,2]`
(c) `a[:2,3]`
(d) `a[2:,:2]`

3.18 What value does `"abc"*3+"defg"` yield? Ans

3.19 Write a program to display every fifth letter of the alphabet' all on one line. Ans

Chapter 4

Choice

One of the essential properties of a computer program is its ability to modify its actions depending on its circumstances and environment. In other words, its behaviour is not predetermined, but can vary from one execution to another. In this chapter, we shall introduce a new plain mode, describe the operators using or yielding values of the new mode, and then investigate the program structures which allow an Algol 68 program to choose between alternatives.

4.1 Boolean values

The mode `BOOL` is named after George Boole, the distinguished nineteenth century mathematician who developed the system of logic which bears his name. There are only two values of mode `BOOL`, and their denotations are `TRUE` and `FALSE`. Let us declare two identifiers:

```
BOOL t = TRUE,
     f = FALSE
```

The `print` phrase, when fed with Boolean values prints `T` for `TRUE`, and `F` for `FALSE`, with spaces neither before nor after. Thus

```
print((t,f,t,f,t))
```

produces `TFTFT` on the screen.

4.2 Boolean operators

The simplest operator which has an operand of mode `BOOL` is `NOT`. If its operand is `TRUE`, it yields `FALSE`. Conversely, if its operand is `FALSE`, it yields `TRUE`. The operator `ODD` yields `TRUE` if its operand is an odd integer and `FALSE` if it is even. The operators can be combined, so

```
NOT ODD 2
```

yields `TRUE`.

`ABS` converts its operand of mode `BOOL` and yields an integer: `ABS TRUE` yields `1`, `ABS FALSE` yields `0`.

Boolean dyadic operators come in two kinds: those that take operands of mode `BOOL`, yielding `TRUE` or `FALSE`, and those that operate on operands of other modes.

Two dyadic operators are declared in the standard prelude which take operands of mode `BOOL`. The operator `AND` (alternative representation `&`) yields `TRUE` if, and only if, both its operands yield `TRUE`, so that

```
t AND f
```

yields `FALSE` (`t` and `f` were declared earlier). Both the operands are elaborated before the operator (but see the section later on pseudo-operators). The priority of `AND` is 3.

The operator `OR` yields `TRUE` if at least one of its operands yields `TRUE`. Thus

```
t OR f
```

yields `TRUE`. It has no alternative representation. Again, both operands are elaborated before the operator. The priority of `OR` is 2.

You will learn in chapter 6 how to define new operators if you need them.

4.3 Relational operators

Values of modes INT, REAL, CHAR and []CHAR can be compared with each other. The expression

```
3 = 1+2
```

yields TRUE. Similarly,

```
1+1=1
```

yields FALSE. The equals symbol = can also be written EQ. Likewise, the formula

```
35.0 EQ 3.5e1
```

should also yield TRUE, but you should be chary of comparing two REALs for equality or inequality because the means of transforming the denotations into binary values may yield values which differ slightly. The operator is also defined for both operands being CHAR or []CHAR. In the latter case, the two multiples must have the same number of elements, and corresponding elements must be equal if the operator is to yield TRUE. Thus

```
"a" = "abc"
```

yields FALSE. Notice that the bounds do not have to be the same. So a and b declared as

```
[]CHAR a = "Dodo" [@0],
       b = "Dodo"
```

yield TRUE when compared with the equals operator. Because the rowing coercion is not allowed in formulæ, the operator is declared in the standard prelude for mixed modes (such as REAL and INT).

The converse of = is /= (not equal). So the formula

```
3 /= 2
```

yields TRUE, and

```
"r" /= "r"
```

yields FALSE. An alternative representation of /= is NE. The priority of both = and /= is 4. The operands of = and /= can be any combination of values of mode INT and REAL. No widening takes place, the operators being declared for the mixed modes.

The ordering operators <, >, <= and >= can be used to compare values of modes INT, REAL, CHAR and []CHAR in the same way as = and /=. They are read "less than", "greater than", "less than or equal to" and "greater than or equal to" respectively. The formula

```
3 < 3.1
```

yields TRUE.

If the identifiers b and c are declared as having mode CHAR, then the formula

```
c < b
```

will yield the same value as

```
ABS c < ABS b
```

and similarly for the operator `>`. The operators `<=` and `>=` can both be used with equal values. For example,

```
24 <= 24.0
```

yields `TRUE`.

For values of mode `[]CHAR`, the formula

```
"abcd" > "abcc"
```

yields `TRUE`. Two values of mode `[]CHAR` of different length can be compared. For example, both

```
"aaa" <= "aaab"
```

and

```
"aaa" <= "aaaa"
```

yield `TRUE`. Alternative representations for these operators are `LT` and `GT` for `<` and `>` and `LE` and `GE` for `<=` and `>=` respectively. The priority of all four ordering operators is 5.

Note that apart from values of mode `[]CHAR`, no operators are defined in the standard prelude for multiples.

Exercises

4.1 What is the value of each of the following formulæ? Ans

(a) `ABS NOT TRUE`
(b) `3.4 + ABS TRUE`
(c) `-3.5 <= -13.4`
(d) `2e10 >= 3e9`
(e) `"abcd" > "abc"`

4.2 In the context of these declarations

```
[]INT i1 = (2,3,5,7);
[]CHAR t = "uvwxyz"
```

what is the value of each of the following? Ans

(a) `UPB i1 < UPB t`
(b) `t[2:4] >= t[2:3]`
(c) `i1[3] < UPB t[2:]`

4.4 Compound Boolean formulæ

Formulæ yielding `TRUE` or `FALSE` can be combined. For example, here is a formula which tests whether π lies between `3` and `4`

```
pi > 3 & pi < 4
```

which yields `TRUE`. The priorities of `<`, `>` and `&` are so defined that parentheses are unnecessary in this case. Likewise, we may write

```
"ab" < "aac" OR 3 < 2
```

which yields `FALSE`. More complicated formulæ can be written:

```
3.4 > 2 & "a" < "c" OR "b" >= "ab"
```

which yields `TRUE`. Because the priority of the operator `&` is higher than the priority of `OR`, the `&` in the above formula is elaborated first. The order of elaboration can be changed using parentheses.

There does not seem much point to these formulæ since everything is known beforehand, but all will become clear in the next chapter.

Compound Boolean formulæ can be confusing. Being aware of the converse of a compound condition helps you to ensure you have considered all possibilities. For example, the converse of the formula

```
a < b & c = d
```

is the formula

```
a >= b OR c /= d
```

One of the formulæ would yield `TRUE` and the other `FALSE`.

Exercises

4.3 What is the value of each of the following: Ans

(a) `NOT ODD 3 OR 3 < 4`

(b) `3 > 2 & (5 > 12 OR 7 <= 8)`

(c) `(TRUE OR FALSE) AND (FALSE OR TRUE)`

(d)

```
NOT("d">"e")
   AND
   FALSE
    OR
NOT(ODD 5 & 3.6e12 < 0)
```

(e) `3<4 & 4<5 & 5<6 & 6>7`

4.4 For each condition, write out its converse: Ans

(a) `FALSE`

(b) `4 > 2`

(c) `a > b AND b > c`

(d) `x = y OR x = z`

4.5 Conditional clauses

Now we can discuss clauses which choose between alternatives. We have met the enclosed clause consisting of at least one phrase enclosed by `BEGIN` and `END` (or parentheses) in the structure of an Algol 68 program, and also in the `DO` ... `OD` loop of a `FOR` or `FORALL` clause. The part of the enclosed clause inside the parentheses (or `BEGIN` and `END`) is called a **serial clause** because, historically, sequential elaboration used to be called "serial elaboration". The value of the serial clause is the value of the last phrase which must be a unit.

There are two kinds of clause which enable programs to modify their behaviour. They are called **choice clauses**. The **conditional clause** allows a program to elaborate code depending on the value of a boolean serial clause, called a `BOOL` **enquiry clause**. Here is a simple example:

```
IF   salary < 5000
THEN 0
ELSE (salary-allowances)*rate
FI
```

The enquiry clause consists of the formula

```
salary < 5000
```

which yields a value of mode `BOOL`. Two serial clauses, both containing a single unit can be elaborated. If the value yielded by `salary` is less than `5000`, the value `0` is yielded. Otherwise, the program calculates the tax. That is, if the `BOOL` enquiry clause yields `TRUE`, the serial clause following `THEN` is elaborated, otherwise the serial clause following `ELSE` is elaborated. The `FI` following the `ELSE` serial clause must be there.

The enquiry clause and the serial clauses may consist of single units or possibly declarations and formulæ and loops. However, the last phrase in an enquiry clause must be a unit yielding `BOOL`. The range of any identifiers declared in the enquiry clause extends to the serial clauses as well. The range of any identifiers declared in either serial clause is limited to that serial clause. For example, assuming that `a` and `i` are predeclared, we could write:

```
IF   INT ai = a[i];  ai < 0
THEN print((ai," is negative",newline))
ELSE print((ai," is non-negative",newline))
FI
```

The conditional clause can be written wherever a unit is permitted, so the previous example could also be written

```
INT ai = a[i];
print((ai,IF   ai < 0
          THEN "is negative"
          ELSE "is non-negative"
          FI,newline))
```

The value of each of the serial clauses following `THEN` and `ELSE` in this case is `[]CHAR`. Here is an example with a conditional clause inside a loop:

```
FOR i TO 100
DO
   IF i MOD 10 = 0
   THEN print((i,newline))
   ELSE print((i,blank))
   FI
OD
```

The ELSE part of a conditional clause can be omitted. Thus the above example could also be written

```
FOR i TO 100
DO
   print((i,blank));
   IF i MOD 10 = 0 THEN print(newline) FI
OD
```

The whole conditional clause can appear as a formula or as an operand. The short form of the clause is often used for this: IF and FI are replaced by (and) respectively, and THEN and ELSE are both replaced by the vertical bar |[1]. For example, here is an identity declaration which assumes a previous declaration for x:

```
REAL xx = (x < 3.0|x**2|x**3)
```

If the ELSE part is missing then its serial clause is regarded as containing the single unit SKIP. In this case, SKIP will yield an undefined value of the mode yielded by the THEN serial clause. This is an example of **balancing** (explained in chapter 10). This is particularly important if a conditional clause is used as an operand.[2]

Since the right-hand side of an identity declaration is in a strong context, widening is allowed. Thus, in

```
REAL x = (i < j|3|4)
```

whichever value the conditional clause yielded would be widened to a value of mode REAL.

Since the enquiry clause is a serial clause, it can have any number of phrases before the THEN. For example:

```
IF []CHAR line =
   "a growing gleam glowing green";
   INT sz = UPB line - LWB line + 1;
   sz > 35
THEN
   ...
```

[1] Some editors insert a different character when you press the key marked |. Check that the character produced is accepted by the Algol 68 compiler.

[2] In this case the a68toc compiler requires an ELSE part and will warn if it is missing. It generates code which will cause a run-time fault if your program tries to execute an ELSE part which has been omitted. You can get around this restriction by explicitly using SKIP.

Conditional clauses can be nested

```
IF a < 4.1
THEN
   IF b >= 35
   THEN print("yes")
   ELSE print("no")
   FI
ELSE
   IF c <= 20
   THEN print("perhaps")
   ELSE print("maybe")
   FI
FI
```

The `ELSE IF` in the above clause could be replaced by `ELIF`, and the final `FI FI` with a single `FI`, giving:

```
IF a < 4.1
THEN
   IF b >= 35
   THEN print("yes")
   ELSE print("no")
   FI
ELIF c <= 20
THEN print("perhaps")
ELSE print("maybe")
FI
```

Here is another contracted example:

```
INT p = IF   c = "a" THEN 1
        ELIF c = "h" THEN 2
        ELIF c = "q" THEN 3
        ELSE 4
        FI
```

The range of any identifier declared in an enquiry clause extends to any serial clause beyond its declaration but within the overall conditional clause. Consider this conditional clause:

```
IF   INT p1 = ABS(c="a");  p1=1
THEN  p1+2
ELIF INT p2 = p1-ABS(c="h");  p2 = -1
THEN INT i1 = p1+p2;  i1+p1
ELSE INT i2 = p1+2*p2;  i2-p2
FI
```

The range of `p1` extends to the enclosing `FI`; likewise the range of `p2`. The ranges of `i1` and `i2` are confined to their serial clauses.

In the abbreviated form, `|:` can be used instead of `ELIF`. For example, the above identity declaration for `p` could be written

```
INT p = (c="a"|1|:c="h"|2|:c="q"|3|4)
```

In both identity declarations, the opening parenthesis is an abbreviated symbol for IF.

Sometimes it is useful to include a conditional clause in the IF part of a conditional clause. In other words, a BOOL enquiry clause can be a conditional clause yielding a value of mode BOOL. Here is an example with a and b predeclared with mode BOOL:

```
IF IF a
   THEN NOT b
   ELSE b
   FI
THEN print("First possibility")
ELSE print("Second possibility")
FI
```

4.5.1 Pseudo-operators

As was mentioned in chapter 2, both the operands of an operator are elaborated before the operator is elaborated. The a68toc compiler implements the **pseudo-operator** ANDTH which although it looks like an operator, has its right-hand operand elaborated only if its left-hand operand yields TRUE. Compare ANDTH (which is read "and then") with the operator AND. The priority of ANDTH is 1. The phrase IF p ANDTH q THEN ... FI is equivalent to

```
IF IF NOT p
   THEN FALSE
   ELIF q
   THEN TRUE
   ELSE FALSE
   FI
THEN ...
FI
```

You should be chary of using ANDTH in a compound boolean expression. For example, given the condition

```
   UPB s > LWB s
      ANDTH
   s[UPB s]="-"
      AND
(CHAR c=s[UPB s-1];
 c>="a" & c<="z")
```

the intention of the compound condition is to determine whether a terminating hyphen is preceded by a lower-case letter. Clearly, testing for a character which precedes the hyphen can only be elaborated if there are at least two characters in s. The first boolean formula (the left operand of ANDTH) ensures that the second formula (the right operand of ANDTH) is only elaborated if s identifies at least two characters. Unfortunately, because the priority of AND is greater than the priority

of ANDTH and because both operands of an operator must be elaborated before the operator is elaborated, the right-hand operand of AND will be elaborated whatever the value of the left operand of ANDTH. In order to achieve the above aim, the compound condition should be written

```
 UPB s > LWB s
     ANDTH
(s[UPB s]="-"
    AND
 (CHAR c=s[UPB s-1];
  c>="a" & c<="z"))
```

Note the additional parentheses which ensure that the boolean formula containing AND is treated *as a whole* as the right-hand operand of the pseudo-operator ANDTH.

There is another pseudo-operator OREL (read "or else") which is similar to the operator OR except that its right-hand operand is only elaborated if its left-hand operand yields FALSE. Like ANDTH, the priority of OREL is 1. The remarks given above about the use of ANDTH in compound boolean formulæ apply equally to OREL.

Neither ANDTH nor OREL are part of Algol 68.

Exercises

4.5 Write a conditional clause which tests whether a REAL value is less than π, and prints "Yes" if it is and "No" otherwise. Ans

4.6 Write a conditional clause inside a loop clause to display the first 96 multiples of 3 (including 3) in lines of 16. Use the operator MOD for the test. Ans

4.7 Replace the operator OREL in the following program with a suitable conditional clause:

```
PROGRAM p CONTEXT VOID
USE standard
IF INT a=3, b=5, c=4;
   a > b OREL b > c
THEN print("Ok")
ELSE print("Wrong")
FI
FINISH
```

Ans

4.6 Multiple choice

Sometimes the number of choices can be quite large or the different choices are related in a simple way. For example, consider the following conditional clause:

```
IF n = 1
THEN action1
ELIF n = 2
THEN action2
ELIF n = 3
THEN action3
ELIF n = 4
THEN action4
ELSE action5
FI
```

This sort of choice can be expressed more concisely using the **case clause** in which the boolean enquiry clause is replaced by an integer enquiry clause. Here is the above conditional clause rewritten using a case clause:

```
CASE n
IN
   action1
      ,
   action2
      ,
   action3
      ,
   action4
OUT
   action5
ESAC
```

which could be abbreviated as

```
(n|action1,action2,action3,action4|action5)
```

Notice that `action1`, `action2`, `action3` and `action4` are separated by commas (they are not terminators). Each of `action1`, `action2` and `action3` is a unit, so that if you want more than one phrase for each action, you must make it an enclosed clause by enclosing the action in parentheses (or `BEGIN` and `END`). If the `INT` enquiry clause yields `1`, `action1` is elaborated, `2`, `action2` is elaborated and so on. If the value yielded is negative or zero, or exceeds the number of actions available, `action5` in the `OUT` part is elaborated. The `OUT` part is a serial clause so no enclosure is required if there is more than one unit.

In the following case clause, the second unit is a conditional clause to show you that any piece of program which happens to be a unit can be used for one of the cases:

```
CASE i IN 3,(x>3.5|4|-2),6 OUT i+3 ESAC
```

The first action yields `3`, the second yields `4` if `x` exceeds `3.5` and `-2` otherwise, and the third action yields `6`.

Sometimes the `OUT` clause consists of another case clause. For example,

```
CASE n MOD 4
IN
   print("case 1"),
   print("case 2"),
   print("case 3")
OUT
   CASE (n-10) MOD 4
   IN
      print("case 11"),
      print("case 12"),
      print("case 13")
   OUT
      print("other case")
   ESAC
ESAC
```

Just as with `ELIF` in a conditional clause, `OUT CASE` ... `ESAC ESAC` can be replaced by `OUSE` ... `ESAC`. So the above example can be rewritten

```
CASE n MOD 4
IN
   print("case 1"),
   print("case 2"),
   print("case 3")
OUSE (n-10) MOD 4
IN
   print("case 11"),
   print("case 12"),
   print("case 13")
OUT print("other case")
ESAC
```

Here is a case clause with embedded case clauses:

```
CASE command
IN
   action1,
   action2,

   (subcommand1
   |subaction1,subaction2
   |subaction3)
OUSE subcommand2

IN subaction4,
   subaction5,
```

```
   subaction6
OUT
   subaction7
ESAC
```

Calendar computations, which are notoriously difficult, give examples of case clauses:

```
INT days = CASE month IN
            31,
            IF year MOD 4 = 0
                  &
               year MOD 100 /= 0
                   OR
               year MOD 400 = 0
            THEN 29
            ELSE 28
            FI,
            31,30,31,30,31,31,30,31,30,31
            OUT -1
           ESAC
```

And here is one in dealing cards:

```
[]CHAR suit=(i|"spades",
               "hearts",
               "diamonds",
               "clubs"
              |"")
```

Like the conditional clause, if you omit the `OUT` part, the compiler assumes that you wrote `OUT SKIP`. In the following example, when `i` is `4`, nothing gets printed:[3]

```
PROGRAM prog CONTEXT VOID
USE standard
FOR i TO 5
DO
   print((i MOD 4|"a","g","r"))
OD
FINISH
```

[3]The a68toc compiler does not support missing `OUT` parts that are executed. It will warn about this and will generate a run-time error. It is good practice to ensure that at least `OUT SKIP` occurs in every case clause.

Exercises

4.8 What is wrong with the following identity declaration, assuming that `p` has been predeclared as a value of mode `BOOL`:

```
INT i = (p|1,2,3|4)
```

Ans

4.9 Write a program consisting solely of a case clause which uses the `SIGN` operator to give three different actions depending on the sign of a number of mode `REAL`. Ans

4.7 Summary

There are two values having mode `BOOL`. Operators with operands of mode `BOOL` are predeclared in the standard prelude. A conditional clause uses an enquiry clause yielding a value of mode `BOOL`. A case clause uses an enquiry clause yielding a value of mode `INT`. Both conditional and case clauses can be abbreviated. Extended conditional and case clauses can be written using `ELIF` and `OUSE` respectively. Conditional clauses and case clauses are sometimes grouped together and termed choice clauses. Choice clauses are examples of enclosed clauses, and are units.

Here are some exercises which test you on the material covered in this chapter.

Exercises

4.10 Which values have the mode `BOOL`? Ans

4.11 What is the value of each of the following formulæ? Ans

(a) `3 < 4`
(b) `4.0 >= 0.4e1`
(c) `2 < 3 & 3 > 2`
(d) `11 < 2 OR 10 < ABS TRUE`
(e) `NOT TRUE & ABS "A" < ABS "D"`
(f) `NOT(3 > 2 & 3 > 1 OR 10 < 6)`

4.12 What is wrong with the following (`m` is predeclared):

```
IF m>4|print("ok")ELSE print(".")ESAC
```

Ans

4.13 What would be displayed on your screen by the following:

```
FOR i TO 10 DO print(ODD i) OD
```

Ans

4.14 Use a conditional clause to print `"Units"` if `m` (which has mode `INT`) is less than `10`, `"Tens"` if it is less than `100`, `"Hundreds"` if it is less than `1000` and `"Too big"` otherwise. Ans

4.15 Use a case clause to print the value of a card in words. For example, if it is a queen, print `"Queen"`. Ans

---

Chapter 5

Names

Previous chapters dealt with values that have always been known when the program was written. If a program is to be able to react to its environment, it must be able to convert external values into internal values and then manipulate them. Analogous to `print`, the conversion can be done by `read` which constructs internal values from external character sequences. In order to manipulate such converted values, we need some way of referring to them. Algol 68 can generate values which can refer to other values. This kind of value is called a **name**. Although a name has a value, it is quite different from the value referred to. The difference is rather like your name: your name refers to you, but is quite distinct from you.

For example, suppose `read` is presented with the character sequence "`123G`" and is expecting an integer. `read` will convert the digits into the number "one hundred and twenty-three", held in a special internal form called "2's-complement binary". To manipulate that value, a name must be generated to refer to it. The mode of a name is called a "reference mode".

A name which can refer to a value of mode `INT` is said to have the mode `REF INT`. Likewise, we can create names with modes

```
REF BOOL    REF[]CHAR    REF[,]REAL
```

As you can see, `REF` can precede any mode. It can also include a mode already containing `REF`. Thus it is possible to construct modes such as

```
REF REF INT
REF[]REF REAL
REF[]REF[]CHAR
REF REF REF BOOL
```

but we shall defer discussion of these latter modes to chapter 11.

Names are created using **generators**. There are two kinds of generator: local and global. The extent to which a name is valid is called its **scope**. The scope of a local name is restricted to the smallest enclosing clause which contains declarations. The scope of a global name extends to the whole program. In general, values have scope, identifiers have range. We shall meet global generators in chapters 6 and 11.

The phrase `LOC INT` generates a name of mode `REF INT` which can refer to a

value of mode `INT`.[1] The `LOC` stands for local. It is quite reasonable to write the phrase

```
read(LOC INT)
```

Unfortunately, the created name is an anonymous name in the sense that it has no identifier so that once the `read` has completed, the name disappears. We need some way of linking an identifier with the generated name so that we can access the name after `read` has finished. This is done with an identity declaration. Here is an identity declaration with a local generator:

```
REF INT a = LOC INT
```

The value identified by `a` has the mode `REF INT` because the phrase `LOC INT` generates a name of mode `REF INT`. Thus it is a name, and it can refer to a value (as yet undefined) of mode `INT` (the value referred to always has a mode of one less `REF`). So now, we can write

```
read(a)
```

After that phrase has been elaborated, `a` identifies a name which now refers to an integer.

Names can also be declared using a predeclared name on the right-hand side of the identity declaration. Here is another identity declaration using `a`:

```
REF INT b = a
```

In this declaration, `b` has the mode `REF INT` so it identifies a name. `a` also has the mode `REF INT` and therefore also identifies a name. The identity declaration makes `b` identify the same name as `a`. This means that if the name identified by `a` refers to a value, then the name identified by `b` (the same name) will always refer to the same value.

5.1 Assignment

The process of causing a name to refer to a value is called **assignment**. Using the identifier declared above, we can write

```
a := 3
```

We say "a assign 3". Note that the mode of the name identified by `a` is `REF INT`, and the mode of the denotation `3` is `INT`. After the assignment, the name identified by `a` refers to the value denoted by `3`.

Suppose now we want the name identified by `a` to refer to the value denoted by `4` (this may seem pedantic, but as you will see below, it is necessary to distinguish between the denotation of a value and that value itself). We write

```
a := 4
```

[1] Historically, programmers were more interested in the value referred to than the name (Algol 68 was the first language to distinguish clearly between a name and the value referred to), so the generator is followed by the mode of the value to which the name will refer.

Let us juxtapose these two assignments:

```
a := 3
  ;
a := 4
```

If you look carefully at the two assignments, a number of things spring to mind. Firstly, an assignment consists of three parts: on the left-hand side is an identifier of a name, in the middle is the **assignment token**, and on the right-hand side is a denotation. Secondly, the left-hand side of the two assignments is the same identifier: `a`. Since the identifier is the same, the value must be the same.[2] That is, in the two assignments, `a` is synonymous with a value which *does not change*. The value is a name and has the mode `REF INT` (in this case). Thus the value of the left-hand side of an assignment is a name.

Thirdly, the values on the right-hand side of the two assignments differ. Firstly, `a` is assigned the value denoted by `3`, then (after the go-on symbol), `a` is assigned the value denoted by `4`.

After the second assignment, `a` refers to `4`. Of course, when we say "`a` refers to", we mean "the name identified by `a` refers to". What has happened to the value `3`? To understand this, we need to look a little more closely at what we mean by the value `3`. The denotation `3` represents the number three. Now, of course, the number three exists independently of a computer program. When the digit `3` is elaborated in an Algol 68 program, an **instance** of the number three is created. Likewise, elaborating the digit `4` creates an instance of the number four. When `a` is assigned an instance of the value four, the instance of the value three disappears. This property of assignment is very important. Because an assignment causes data to disappear, it is dangerous to use. You have to be careful that the data which disappears is not data you wanted to keep. So the instance of a value can disappear, but the value still exists (like the number three).

It is worth reiterating that however many times a name is assigned a value, the value of the name remains unchanged. It is the value referred to which is superseded. Outwith the realm of computers, if an individual is assigned to a department of an organisation, clearly the department hasn't changed. Only its members have changed.

When an identifier for a name has been declared, the name can be made to refer to a value immediately after the declaration. For example

```
REF REAL x = LOC REAL := pi
```

where `pi` is the value declared in the standard prelude. `LOC REAL` generates a name of mode `REF REAL`.

The right-hand side of an assignment is a strong context so widening is allowed. Thus we can write

```
x := 3
```

where the `3` is widened to `3.0` before being assigned to `x`. In reality, the value denoted by `3` is not changed to the value denoted by `3.0`: it is replaced by the new value. There is an important principle here. It is called the "principle of value

[2] Provided that both identifiers appear in the same range.

integrity": once an instance of a value has been created, it does not change until such time as it disappears. Thus, in Algol 68, every value is a constant. Every coercion defined in Algol 68 replaces a value of one mode with a related value of another mode.

5.1.1 Copying values

Here is another identity declaration with an initial assignment:

```
REF INT c = LOC INT := 5
```

Using the identifier `a` declared earlier, we can write

```
a := c
```

and say "a assign c". The name on the left-hand side of the assignment has mode `REF INT`, so a value which has mode `INT` is required on the right-hand side, but what has been provided is a name with mode `REF INT`. Fortunately, there is a coercion which replaces a name with the value to which it refers. It is called **dereferencing** and is allowed in a strong context. In the above assignment, the name identified by `c` is dereferenced yielding an instance of the value five which is a copy of the instance referred to by `c`. That new instance is assigned to `a`. It is important to remember that the process of dereferencing yields a *new* instance of a value.

Try the following program:

```
PROGRAM assign CONTEXT VOID
USE standard
BEGIN
   REF INT a = LOC INT,
           b = LOC INT := 7;
   print(("b=",b,newline));
   print("Please key 123G:"); read(b);
   a := b;
   print(("a now refers to",a,newline,
          "b now refers to",b,newline))
END
FINISH
```

This should convince you that dereferencing involves copying.

Every construct in Algol 68 has a value except an identity declaration. We said above that the value of the left-hand side of an assignment is a name. In fact, the value of the whole of the assignment is the value of the left-hand side. Because this is a name, it can be used on the right-hand side of another assignment. For example:

```
a := b := c
```

You should note that an assignment is not an operator. The assignments are performed from *right to left*: firstly, `c` is dereferenced and the resulting value assigned to `b`. Then `b` is dereferenced and the resulting value is assigned to `a`.

5.1.2 Assigning operators

The following assignment

```
a := a
```

does not do anything useful, but serves to remind us that the name identified by `a` on the right-hand side of the assignment is dereferenced, and the resulting value is assigned to `a`. However, `a` now refers to a new instance of the value it previously referred to and the previous instance has now disappeared.

Now consider the phrases

```
c := 5; a := c+1
```

The right-hand side of the second assignment is now a formula. The name identified by `c` is now in a firm context (it is the left-operand of the `+` operator). Fortunately, dereferencing is also allowed in a firm context. Thus the value of `c` (a name with mode `REF INT`) is replaced in the formula by a copy of the value to which it refers (`5`), which is added to `1`, and `a` is assigned the new value (`6`). We say "a is assigned c plus one".

What about the phrase

```
a := a+1
```

In exactly the same way as the previous phrase, the name on the right-hand side is dereferenced, the new value created is added to `1`, and then the same name is assigned the new value.

One of the features of assignment is that the elaboration of the two sides is performed collaterally. This means that the order of elaboration is undefined. This does not matter in the last example because the value of the name identified by `a` is the same on the two sides of the assignment. Remember that the value of `a` is a name with mode `REF INT`. It is the value to which `a` referred which was superceded.

Assignments of this kind are so common that a special operator has been devised to perform them. The above assignment can be written

```
a +:= 1
```

and is read "a plus-and-assign one". The operator has the alternative representation `PLUSAB`.[3] Note that the left-hand operand must be a name. The right-hand operand must be any unit which yields a value of the appropriate mode in a firm context.

The operator `+:=` is defined for a left-operand of mode `REF INT` or `REF REAL`, and a right-operand of mode `INT` or `REAL` respectively. The yield of the operator is the value of the left-operand (the name). If the left-operand has mode `REF REAL`, the right-operand can also have mode `INT`. No widening occurs in this case, the operator having been declared for operands having these modes. Because the operator yields a name, that name can be used as the operand for another assigning operator. For example

[3] `PLUSAB` stands for "plus-and-becomes". When Algol 68 was first designed, people were more concerned with the values referred to than the names, so `PLUSAB` was intended to describe what happens to the value referred to. Bearing in mind the principle of value integrity, the value referred to by `a` does not become anything, but is replaced by its value plus 1.

```
x +:= 3.0 *:= 4.0
```

which results in `x` referring to `4.0*(x+3.0)`. The formula is elaborated in left-to-right order because the operators have the same priority. The operators are more efficient than writing out the assignments in full.

There are four other operators like `+:=`. They are `-:=`, `*:=`, `/:=`, `%:=` and `%*:=`. Their alternative representations are respectively `MINUSAB`, `TIMESAB`, `DIVAB`, `OVERAB` and `MODAB`. The operators `OVERAB` and `MODAB` are only declared for operands with modes `REF INT` and `INT`. The priority of all the operators is 1.

The assignment operators are operators, not assignments (although they perform an assignment), so that the previous example is not an assignment, but a formula.

The right-hand side of an assignment can be any unit which yields a value whose mode has one less `REF` than the mode of the name on the left-hand side. Names whose mode contains more than one `REF` will be considered in chapter 11.

Exercises

5.1 The following identity declarations

```
REF CHAR s = LOC CHAR,
REF INT  i = LOC INT,
REF REAL r = LOC REAL
```

hold in this and the following exercises.[4] What is the mode of `i`? Ans

5.2 After the assignment `r := -2.7` has been elaborated, what is the mode of the value referred to by `r`? Ans

5.3 What is wrong with the assignment `i := r` and how would you correct it? Ans

---

[4]The a68toc compiler requires that you write semicolons instead of commas to separate these three declarations.

5.2 Assignments in formulæ

Since an assignment yields a name, it can be used in a formula. However, the assignment must be converted into an enclosed clause (using parentheses or `BEGIN` and `END`) ensuring that the assignment is elaborated first. For example, in

```
3*(a := c+4)+2
```

if `c` refers to `3`, the value of the formula will be `23` with mode `INT`, `a` will refer to `7`, the value of the assignment is a name of mode `REF INT` and `c` will still refer to `3`. Remember that assignment is not an operator.

Here is an example of two assignments in a conditional clause:

```
IF a<2 THEN x := 3.2 ELSE x := -5.0 FI
```

This can be written with greater efficiency as

```
x := IF a < 2 THEN 3.2 ELSE -5.0 FI
```

The left-hand side of an assignment has a **soft** context. In a soft context, dereferencing is not allowed (it is the only context in which dereferencing is not allowed). In the following phrase, the conditional clause on the left yields a name which is then assigned the value of the right-hand side:

```
IF a < 2 THEN x ELSE y FI := 3.5
```

In the next assignment, a conditional clause appears on both sides of the assignment:

```
(a<2|x|y) := (b<2|x|y)
```

The result depends on the values referred to by both `a` and `b` as much as on the values referred to by both `x` and `y`.

Exercises

5.4 What is wrong with the following program fragment?

```
REF REAL x = LOC REAL,
         y = LOC REAL := 3.5;
y := 4.2+x
```

Ans

5.5 If `x` refers to `3.5` and `y` refers to `-2.5`, what is the mode and value yielded by the following phrases: Ans

(a) `x := -y`
(b) `ABS y`

5.6 What does `x` refer to after

```
x := 1.5;  x PLUSAB 2.0 DIVAB 3.0
```

(try it in a small program). Ans

5.3 Multiple names

Here is an identity declaration for a name which can refer to a multiple::

```
REF[]INT i7 = LOC[1:7]INT
```

There are two things to notice about this declaration. Firstly, the mode on the left-hand side is known as a **formal-declarer**. It says what the mode of the name is, but it says nothing about how many elements there will be in any multiple to be assigned, nor what its bounds will be. All the identity declarations for multiples in chapter 3 used formal-declarers on the left-hand side. In fact, only formal-declarers are used on the left-hand side of *any* identity declaration.

Secondly, the generator on the right-hand side is an **actual-declarer**. It specifies how many elements can be assigned. In fact, the trimmer represents the bounds of the multiple which can be assigned. If the lower bound is `1` it may be omitted, so the above declaration could well have been written

```
REF[]INT i7 = LOC[7]INT
```

which can be read as "ref row of int i7 equals loc row of seven int". The bounds of a multiple do not have to start from `1` as we saw in chapter 3. In this identity declaration

```
REF[]INT i7 at 0 = LOC[0:6]INT
```

the bounds of the multiple will be `[0:6]`.

5.4 Assigning to multiple names

We can assign values to the elements of a multiple either individually or collectively.

5.4.1 Individual assignment

You may remember from chapter 3 that we can access an individual element of a multiple by specifying the subscript(s) of that element. For example, suppose that we wish to access the third element of `i7` as declared in the last section. The rules of the language state that a subscripted element of a multiple name is itself a name. In fact, the elaboration of a slice of a multiple name creates a new name. Thus the mode of `i7[3]` is `REF INT`. We can assign a value to `i7[3]` by placing the element on the left-hand side of an assignment:

```
i7[3]:=4
```

Unless you define a new identifier for the new name, it will cease to exist after the above assignment has been elaborated (see below for examples of this).

Since each element of `i7` has an associated name (created by slicing) of mode `REF INT`, it can be used in a formula:

```
i7[2]:=3*i7[i7[1]] + ENTIER(4.0/i7[3])
```

As you can see, an element was used to compute a subscript. It has been presumed that the value obtained after dereferencing lies between `1` and `7` inclusive. If this were not so, a run-time error would be generated. In the above assignment, all three elements on the right-hand side of the assignment would be dereferenced before being used in the formula. Note that subscripting (or slicing or trimming) binds more tightly than any operator. Thus, in the last term in the above example, `i7` would be sliced first, then the yielded name dereferenced, and finally, the new value would be divided into `4.0`.

Here is a `FOR` loop which assigns a value to each element of `i7` individually:

```
FOR e FROM LWB i7 TO UPB i7
DO
   i7[e]:=e**3
OD
```

Using the bounds interrogation operators is useful because:

1. The fact that the lower bound of `i7` is `1` is masked, but the formula `LWB i7` ensures that the correct value is used.
2. If the bounds of `i7` are changed when the program is being maintained, the loop clause can remain unchanged. This simplifies the maintenance of Algol 68 programs.
3. The compiler can omit bounds checking. For large multiples, this can speed up processing considerably.

Here is a program which uses a name whose mode is `REF[]BOOL`. It computes all the prime numbers less than 1000 and is known as Eratosthenes' Sieve:

```
PROGRAM sieve CONTEXT VOID
USE standard
BEGIN
   INT size = 1000;

   REF[]BOOL flags = LOC[2:size]BOOL;

   FOR i FROM LWB flags TO UPB flags
   DO
      flags[i] := TRUE
   OD;

   FOR i FROM LWB flags TO UPB flags
   DO
      IF flags[i]
      THEN
         FOR k
         FROM 2*i BY i TO UPB flags
         DO
            flags[k] := FALSE
            CO Remove multiples of i CO
         OD
```

```
      FI
   OD;

   FOR i FROM LWB flags TO UPB flags
   DO
      IF flags[i] THEN print((i,blank)) FI
   OD
END
FINISH
```

5.4.2 Collective assignment

There are two ways of assigning values collectively. Firstly, it can be done with a row-display or a `[]CHAR` denotation. For example, using the declaration of `i7` above:

```
i7:=(4, -8, 11, ABS "K",
     ABS TRUE, 0, ROUND 3.4)
```

Notice that the bounds of both `i7` and the row-display are `[1:7]`. In the assignment of a multiple, the bounds of the multiple on the right-hand side must match the bounds of the multiple name on the left-hand side. If they differ, a fault is generated. If the bounds are known at compile-time, the compiler will generate an error message. If the bounds are only known at run-time (see section 5.8 on dynamic names), a run-time error will be generated. The bounds can be changed using a trimmer or the `@` symbol (or `AT`). See chapter 3 for details.

The second way of assigning to the elements of a multiple collectively is to use an identifier of a multiple with the required bounds. For example:

```
[]INT i3 = (1,2,3);
REF[]INT k = LOC[1:3]INT := i3
```

The right-hand side has been assigned to the multiple name `k`.

As mentioned above, parts of a multiple can be assigned using slicing or trimming. For example, given the declarations

```
REF[,]REAL x = LOC[1:3,1:3]REAL,
           y = LOC[0:2,0:2]REAL
```

and the assignment

```
x:=((1,2,3),
    (4,5,6),
    (7,8,9))
```

we can write

```
y[2,0]:=x[3,2]
```

The multiple name `y` is sliced yielding a name of mode `REF INT`. Then[5] the multiple name `x` is sliced also yielding a name of mode `REF INT` which is then dereferenced

[5] But because the two sides of an assignment are elaborated collaterally, the RHS might be elaborated before the LHS or even in parallel.

yielding a new instance of the value to which it refers (8) which is then assigned to the new name on the LHS of the assignment. Here is an identity-declaration which makes the new name permanent:

```
REF INT y20 = y[2,0];  y20:=x[3,2]
```

which has its uses (see below).

Here are some examples of slicing with (implied) multiple assignments:

```
y     := x[@0,@0];
y[2,] := x[ 1,@0];
y[,1] := x[ 2,@0]
```

In the first example, the right-hand side is a slice of a name whose mode is `REF[,]REAL`. Because the slice has no trimmers its mode is also `REF[,]REAL`. Using the `@` symbol, the lower bounds of both dimensions are changed to `0`, ensuring that the bounds of the multiple name thus created match the bounds of the multiple name `y` on the left. After the assignment (and the dereferencing), `y` will refer to a copy of the multiple `x` and the name created by the slicing will no longer exist.

In the second assignment, the multiple `x` has been sliced yielding a name whose mode is `REF[]REAL`. It refers, in fact, to the first "row" of `x`. The `@0` ensures that the lower bound of the second dimension of `x` is `0`. The left-hand side yields a name of mode `REF[]REAL` which refers to the last "row" of the multiple `y`. The name on the right-hand side is dereferenced. After the assignment `y[2,]` will refer to a copy of the first "row" of `x` and the name produced by the slicing will no longer exist.

In the third assignment, the second "row" of `x` is assigned to the second "column" of `y`. Again, the `@0` construction ensures that the lower bound of the second dimension of `x` is zero. After the assignment, the name created by the slicing will no longer exist.

Notice how the two declarations for `x` and `y` have a common formal-declarer on the left-hand side, with a comma between the two declarations. This is a common abbreviation. The comma means that the two declarations are elaborated collaterally (and on a parallel processing computer, possibly in parallel).

It was stated in the section on names that names can be put on the right-hand side of an identity declaration. This is particularly useful for accessing elements of rows. Consider the following:

```
REF[]INT r = LOC[100]INT;

FOR i FROM LWB r TO UPB r DO r[i]:=i*i OD;

FOR i FROM LWB r TO UPB r-1
DO
   IF   REF INT ri=r[i], ri1=ri[i+1];
        ri > ri1
   THEN ri:=ri1
   ELSE ri1:=ri
   FI
OD
```

This is another example of optimisation, but in this case, we need names because the `THEN` and `ELSE` clauses contain assignments. Both `ri` and `ri1` are used thrice

in the conditional clause, but the multiple `r` is only subscripted twice in each loop. In the condition following the `IF`, both `ri` and `ri1` would be dereferenced (but not in the identity declarations). The values of `ri` and `ri1` remain constant: the names are assigned new values. You can see from the identity declarations that the modes of the names `ri` and `ri1` are both `REF INT`.

Here is a program fragment which uses a `REF[]REAL` identity declaration for optimisation:

```
REF[,]REAL m = LOC[3,4]REAL;   read(m);

FOR i FROM 1 LWB m TO 1 UPB m
DO
   REF[]REAL mi = m[i,];
   FOR j FROM LWB mi TO UPB mi
   DO
      REF REAL mij = mi[j];
      mij*:=mij
   OD
OD;

print((m,newline))
```

As you can see, `read` behaves just like `print` in that a whole multiple can be read at one go (see chapter 3 for the use of `print` with multiples). The only difference between the way `read` is used and the way `print` is used is that the values for `read` must be names (or identifiers of names) whereas `print` can use denotations or identifiers of names or identifiers which are not names.

Exercises

5.7 After the assignments of `x` to `y` discussed above, what is the final value of `y` (careful)? Ans

5.8 Given these declarations

```
REF[,]INT m = LOC[3:5,-2:0]INT,
REF[]INT n = LOC[1:3]INT:=(1,2,3)
```

Ans

(a) What is wrong with the assignment `m[1,]:=n`?
(b) How would you assign the second "column" of `m` to its third "row"?

5.9 Modify Eratosthenes' Sieve to compute the 365^{th} prime. Ans

---

5.5 Flexible names

In the previous section, we declared mutliple names. The bounds of the multiple to which the name can refer are included in the generator. In subsequent assignments, the bounds of the new multiple to be assigned must be the same as the bounds given in the generator. In Algol 68, it is possible to declare names which can refer to a multiple of any number of elements (including none) and, at a later time, can refer to a different number of elements. They are called **flexible** names. Here is an identity declaration for a flexible name:

```
REF FLEX[]INT fn = LOC FLEX[1:0]INT
```

There are several things to note about this declaration. Firstly, the mode of the name is not `REF[]INT`, but `REF FLEX[]INT`. The `FLEX` means that the bounds of the multiple to which the name can refer can differ from one assignment to the next. Secondly, the bounds of the name generated at the time of the declaration are `[1:0]`. Since the upper bound is less than the lower bound, the multiple is said to be **flat**; in other words, it has no elements at the time of its declaration[6]. Thirdly, `FLEX` is present on both sides of the identity declaration (but in the last section of this chapter we shall see a way round that).

We can now assign multiples of integers to `fn`:

```
fn:=(1,2,3,4)
```

The bounds of the multiple to which `fn` now refers are `[1:4]`. Again, we can write

```
fn:=(2,3,4)
```

Now the bounds of the multiple to which `fn` refers are `[1:3]`. We can even write

```
fn:=7
```

in which the right-hand side will be rowed to yield a one-dimensional multiple with bounds `[1:1]`, and

```
fn:=()
```

giving bounds of `[1:0]`.

In the original declaration of `fn` the bounds were `[1:0]`. The compiler will not ignore any bounds other than `[1:0]`, but will generate a name whose initial bounds are those given. So the declaration

```
REF FLEX[]INT fn1 = LOC FLEX[1:4]INT
```

will cause `fn1` to have the bounds `[1:4]` instead of `[1:0]`.

The lower bound does not have to be `1`. In this example,

```
REF[]INT m1 = LOC[-1:1]INT;
FOR i FROM LWB m1 TO UPB m1 DO m1[i]:=i+3 OD;
REF FLEX[]INT f1 = LOC FLEX[1:0]INT := m1
```

[6] The Revised Report mentions a "ghost element" in this context (see section 10.11 for details)

the bounds of `f1` after the initial assignment are `[-1:1]`.

If a flexible name is sliced or trimmed, the resulting name is called a **transient name** because it can only exist so long as the flexible name stays the same size. Such names have a restricted use to avoid the production of names which could refer to nothing. For example, consider the declaration and assignation

```
REF FLEX[]CHAR c1 = LOC FLEX[1:0]INT;
 c1:="abcdef";
```

Suppose now we have the declaration

```
REF[]CHAR lc1=c1[2:4]; #WRONG#
```

followed by this assignment:

```
c1:="z";
```

It is clear that `lc1` no longer refers to anything meaningful. Thus transient names cannot be assigned without being dereferenced, nor given identifiers, nor used as parameters for a routine (whether operator or procedure). However there is nothing to prevent them being used in an assignment. For example,

```
REF FLEX[]CHAR s=LOC[1:0]CHAR:=
   "abcdefghijklmnopqrstuvwxyz";
s[2:7]:=s[9:14]
```

where the name yielded by `s[9:14]` is immediately dereferenced. Note that the bounds of a trim are fixed even if the value trimmed is a flexible name. So the assignment

```
s[2:7]:="abc"
```

would produce a run-time fault.

Exercises

5.10 The declaration

```
REF FLEX[]CHAR s = LOC FLEX[1:0]CHAR
```

applies to the following: Ans

(a) What is the value of `s`?
(b) After the assignment

```
s:="aeiou"
```

what are the bounds of `s`?

5.6 The mode STRING

The mode STRING is defined in the standard prelude as having the same mode as the expression FLEX[1:0]CHAR. That is, the identity declaration

```
REF STRING s = LOC STRING
```

has exactly the same effect as the declaration

```
REF FLEX[]CHAR s = LOC FLEX[1:0]CHAR
```

You will notice that although the mode indicant STRING appears on both sides of the identity declaration for s, in the second declaration the bounds are omitted on the left-hand side (the mode is a formal-declarer) and kept on the right-hand side (the actual-declarer). Without getting into abstruse grammatical explanations, just accept that if you define a mode like STRING, whenever it is used on the left-hand side of an identity declaration the compiler will ignore the bounds inherent in its definition.

We can now write

```
s:="String"
```

which gives bounds of [1:6] to s. We can slice that row to get a value with mode REF CHAR which can be used in a formula. If we want to change the bounds of s, we must assign a value which yields a value of mode []CHAR to the whole of s as in

```
s:="Another string"   or   s:=s[2:4]
```

Wherever []CHAR appears in chapter 3, it may be safely replaced by STRING. This is because it is only names which are flexible so the flexibility of STRING is only available in REF STRING declarations.

There are two operators defined in the standard prelude which use an operand of mode REF STRING: PLUSAB, whose left operand has mode REF STRING and whose right operand has mode STRING or CHAR, and PLUSTO, whose left operand has mode STRING or CHAR and whose right operand has mode REF STRING. Using the concatenation operator +, their actions can be summarised as follows:

```
a PLUSAB b   ≡   a:=a+b
a PLUSTO b   ≡   b:=a+b
```

Thus PLUSAB concatenates b onto the end of a, and PLUSTO concatenates a to the beginning of b. Their alternative representations are +:= and +=: respectively. For example, if a refers to "abc" and b refers to "def", after a PLUSAB b, a refers to "abcdef", and after a PLUSTO b, b refers to "abcdefdef" (assuming the PLUSAB was elaborated first).

Exercises

5.11 Write a program which declares a name with mode `REF STRING` and then consecutively assigns the rows of characters `"ab"`, `"abc"`, upto the whole alphabet and prints each row on a separate line. Use a `FOR` loop clause. Ans

5.12 Declare a flexible name which can refer to a 2-dimensional row whose elements have mode `REAL`. Assign a one-dimensional row whose elements are

```
5.0 10.0 15.0 20.0
```

Write the `print` phrase which will display each bound on the screen followed by a space, all on one line. Ans

5.7 Reference modes in transput

Wherever previously we have used a value of mode `INT` with `print`, we can safely use a name with mode `REF INT`, and similarly with all the other modes (such as `[,]REAL`). This is because the parameters for `print` (the identifiers or denotations used for `print`) are in a firm context and so can be dereferenced before being used.

In the preamble to this chapter, `print`'s counterpart `read` was mentioned. It is now time to examine `read` more closely. Generally speaking, values displayed with `print` can be input with `read`. The main differences are that firstly, the parameters for `read` must be names. For example, we may write

```
REF REAL r = LOC REAL;
read(r)
```

and the program will skip spaces, tabs and end-of-line and new-page characters until it meets an optional sign followed by optional spaces and at least one digit, when it will expect to read a number. If an integer is present, it will be read, converted to the internal representation of an integer and then widened to a real.

Likewise, `read` may be used to read integers. The plus and minus signs (`+` and `-`) can precede integers and reals. Absence of a sign is taken to mean that the number is positive. Any non-digit will terminate the reading of an integer except for a possible sign at the start. Reals can contain `e` as in `3.41e5`. It is best to ensure that each number is preceded by a sign so that the reading of any preceding number will be terminated by that sign.

For a name of mode `REF CHAR`, a single character will be read, `newline` or `newpage` being called if necessary. In fact, tabs and any other control characters (whose absolute value is less than `ABS blank`) will also be skipped.

If `read` is used to read a `[]CHAR` with fixed bounds as in

```
REF[]CHAR sf = LOC[36]CHAR;
read(sf)
```

then the number of characters specified by the bounds will be read, `newline` and `newpage` being called as needed. You can call `newline` and `newpage` explicitly to ensure that the next value to be input will start at the beginning of the next line or page.

Just like `print`, `read` can take more than one parameter by enclosing them in a row-display.

You should note that the end of a line or page will not terminate the reading of a number. So if you want to read a number from the keyboard, you should follow the number with a non-digit before pressing "Enter". In this case, you don't have to read a newline as well, but the "Enter" generates a newline and that newline will be pending in the input.[7]

The only flexible name for which `read` can be used is `REF STRING`. When reading values for `REF STRING`, the reading pointer will not go past the end of the current line.[8] If the reading position is already at the end of the line, the row will have no elements. When reading a `STRING`, `newline` must be called explicitly for transput to continue. The characters read are assigned to the name.

Exercises

5.13 Write a program to read two real numbers and then print their sum and product. Ans

5.14 Write a program which will input text line by line (the lines being of different length) and which will then write out each line with the characters reversed. For example, the line `"and so on"` will be displayed as `"no os dna"`. Continue reading until a line of zero length is read. Ans

---

5.8 Dynamic names

Hitherto, all the names which can refer to rows were declared with bounds whose values were given by integer denotations. In fact, the bounds given on the right-hand side of the identity declaration can be any unit which yields an integer in a **meek** context. So it is quite reasonable to write

```
REF INT size = LOC INT;  read(size);
REF[]INT a = LOC[1:size]INT
```

or even

```
REF[]INT r=
   LOC[1:(REF INT i=LOC INT;
          read(i);
          i)]INT
```

since an enclosed serial clause has the value of its last unit. The value of the clause in the parentheses is a name of mode `REF INT` and since the context of the clause is meek, dereferencing is allowed. The context is passed on to the last unit in the clause. Thus the integer read by `read` will be passed to the generator.

A **dynamic** name is one which can refer to a multiple whose bounds are determined at the time the program is elaborated. It means that you can declare names referring to multiples of the size you actually require, rather than the maximum size that you might ever need.

[7] Console input is better handled using the `kbd channel` described in section 13.7.2.

[8] See section 9.4 for details of string terminators.

Exercises

5.15 Declare a name which can refer to a multiple of reals whose upper bound is determined by reading an integer from the keyoard. Ans

5.16 Write a program which will read an integer which says how many integers follow it. Compute the sum of all the integers and print it. Ans

5.9 Loops revisited

In section 3.7, we introduced the loop clause whose start, step and finish were specified by integer denotations. Instead of an integer, a unit which yields a value of mode `INT` in a meek context can be supplied. The principle coercions not available in a meek context are rowing and widening. In practice, almost any unit yielding `INT` will do. In particular, a name with mode `REF INT` can be given.

There is an extra construct which is extremely useful for controlling the execution of the `DO` ... `OD` loop. It is very common to execute a loop while a particular condition holds. For example, while integers are negative:

```
WHILE
   REF INT int=LOC INT; read(int); int < 0
DO
   print((ABS int,newline))
OD
```

In this example, no loop counter was needed and so the `FOR id` part was omitted. The phrase following the `WHILE` must be an enquiry clause yielding `BOOL`. In this case, an integer is read each time the loop is elaborated until a non-negative integer is read. The range of any declarations in the enquiry clause extends to the `DO` ... `OD` loop.

It happens quite often that the `WHILE` enquiry clause performs all the actions which need repeating and nothing is required in the `DO` part. Since the loop clause must contain at least one unit, `SKIP` can be used as in

```
FOR i FROM LWB a TO UPB a
WHILE (sum+:=a[i]) <= max
DO
   SKIP
OD
```

The complete loop clause thus takes the form:

```
FOR id FROM from-unit BY by-unit TO to-unit
WHILE boolean-enquiry-clause
DO
   serial clause
OD
```

Exercises

5.17 Write a program which will read integers until zero is encountered. The program should print the sums of the negative and positive integers. Ans

5.18 Write a program which will read lines from the keyboard and then compute a unique code for each line as follows: if `"did"` is read, compute the value of

```
ABS"d" + ABS"i"*2 + ABS"d"*3
```

Display the string and its corresponding number on the screen. Terminate the program when a zero-length line has been read (if the result exceeds `max int`, you will normally not get an error: just erroneous results—see section 13.3.13). Ans

5.10 Abbreviated declarations

You have now met many identity declarations. When declaring names, it is apparent that much of the declaration is repeated on both sides. For example:

```
REF[]REAL r = LOC[10]REAL
```

Declarations of names are very common in Algol 68 programs and abbreviated declarations are available. The above declaration can be written

```
LOC[10]REAL r
```

or, most commonly

```
[10]REAL r
```

An abbreviated declaration uses the actual-declarer (the right-hand side of an identity declaration) followed by the identifier; and if the actual-declarer contains the generator `LOC`, you can omit the `LOC` (see section 6.1 which explains actual-declarers and formal-declarers).

Here are some of the declarations given as examples in this chapter rewritten in their abbreviated form:

```
INT a;
REAL x:=pi;
CHAR s;
[7]INT i7;
[0:6]INT i7 at 0;
[3]INT k:=(1,2,3);
[3,3]REAL x; [0:2,0:2]REAL y;
FLEX[1:0]INT fn;
[36]CHAR sf;
[(INT i; read(i); i)]INT r
```

It is important to note that identity declarations should not be mixed with abbreviated name declarations because the modes are quite different. For example, in

```
REAL a:=2.4;
REAL b = a+2.1
```

the mode of `a` is `REF REAL`, but the mode of `b` is `REAL`. In the abbreviated declaration of a name, the mode given is that of the value to which the name will refer (the actual-declarer).

When you declare a new object, if you do not intend assigning to it, use an identity declaration. Only declare it as a name if you intend superseding the value to which it will refer. Remember that assignment can be dangerous because values are superseded.

Exercises

5.19 Write abbreviated declarations for the following:
Ans

(a) `REF[]CHAR rc = LOC[1000]CHAR`
(b) `REF FLEX[]INT fi = LOC FLEX[1:0]INT`
(c) `REF BOOL b = LOC BOOL := TRUE`

5.20 Write full identity declarations for the following:
Ans

(a) `INT a,b,c`
(b) `REAL x;[5]CHAR y;[3,3]REAL z`
(c) `FLEX[1:0]CHAR s`

5.11 Summary

A name is a value whose mode always begins with the mode constructor `REF`. A name can refer to a value whose mode starts with one less `REF` than the mode of the name. An assignment causes a name to refer to a value. The value to which a name refers can be superseded using a further assignment. An assignment is a kind of unit and can appear in a formula if it is enclosed by parentheses (or `BEGIN` and `END`). Multiple assignments can be used to assign the same value to more than one name.

A name can be generated using a local or global generator and can be made to refer to a value in the same phrase in which it is declared.

Algol 68 provides flexible names as well as fixed names for multiples. The mode indicant for `FLEX[]CHAR` is defined in the standard prelude as `STRING`. Names for multiples can have bounds determined at run-time.

`read` will convert external character sequences into internal values. Its parameters must be names or `newline` or `newpage`.

Name declarations may be written as identity declarations or in an abbreviated form.

Before continuing with chapter 6, it would be wise to revise the material in the first five chapters since these comprise the basis of the language.

Exercises

5.21 Declare a name to refer to a multiple of 1000 integers, first as an identity declaration, and secondly in abbreviated form. Ans

5.22 Write a program which will compute the average of a number of salaries (*eg*, 1010.53) read from the keyboard until the number `-1` is read. Display the average on the screen. Ans

5.23 Write a program which will read a line and then scan it, writing out the individual words on one line apiece. The program should read the line into a `REF STRING` name, then remove leading and trailing spaces and add a space to the end. Use a boolean name called `in word` and make it refer to `FALSE`. As you step along the line, make `in word` refer to `FALSE` if you read a space and `TRUE` otherwise. Keep a track of the length of the current word. Whenever the value `in word` changes from `TRUE` to `FALSE`, extract the word using an appropriate trimmer and print it. Allow for there being more than one space between words. Ignore the possibility of commas, brackets etc. Ans

---

Chapter 6

Routines

Routines consist of two types: operators and procedures. They have much in common, so the first section covers their common aspects. These are followed by a section on operators and a section on procedures. The length of this chapter reflects the importance of routines in the language.

6.1 Routines

A **routine** is a number of encapsulated actions which can be elaborated in their entirety in other parts of the program. A routine has a well-defined mode. The value of a routine is expressed as a **routine denotation**. Here is an example:

```
([]INT a)INT:
(
   INT sum:=0;

   FOR i FROM LWB a TO UPB a DO sum+:=i OD;
   sum
)
```

In this example, the **header** of the routine is given by

```
([]INT a)INT:
```

which could be read as "with (parameter) row of `INT` `a` yielding `INT`". The mode of the routine is given by the header, less the colon and any identifiers. So the mode of the above routine is

```
([]INT)INT
```

We say that the routine takes one **parameter** of mode `[]INT` and yields a value of mode `INT`.

As you can see from the **body** of the routine (everything except the header), the routine yields the sum of the individual elements of the parameter. The body of a routine is a unit. In this case, it is an enclosed clause.

We have met parameters before in a different guise. The formal definition of an identity declaration is

```
<formal-mode-param> = <actual-mode-param>
```

The formal-mode-param consists of an identifier preceded by a formal-mode-declarer (referred to in the last chapter as a formal-declarer). An actual-mode-param is a piece of program which yields an internal object which henceforth is identified by the identifier. For example, in the identity declaration

```
[]INT a = (2,3,5,7,11)
```

`[]INT a` is the formal (mode) parameter, `[]INT` is the formal (mode) declarer, the identifier is `a`, and the actual (mode) parameter is the row-display `(2,3,5,7,11)`. The word "mode" was placed in parentheses because it is common usage to omit it. Henceforth, we shall talk about formal parameters and actual parameters.

In the header of the above routine, `a` is declared as a formal parameter. The mode of `a` is `[]INT`. At the time the routine is declared, `a` does not identify a value. That is why it is called a "formal" parameter. It is only when the routine is used that `a` will identify a value. We'll come to that later. Any identifier may be used for the formal parameter of a routine.

In the body of the routine, `a` is treated as though it has a value. Since its mode is `[]INT`, it is a multiple and so it can be sliced to access its individual elements.

The body of the routine written above consists of an enclosed clause. In this case, the enclosure consists of the parentheses `(` and `)`, but it might well have been written using `BEGIN` and `END`. Inside the enclosure is a serial clause consisting of three phrases. The first is a declaration with an initial assignment. Although an assignment yields a name, an identity declaration with an initial assignment, even an abbreviated one, does not. This is the only exception.

The second phrase is a `FOR` loop clause which yields `VOID` (see section 6.1.4). The third phrase consists of the identifier `sum` which yields its name of mode `REF INT`.

Now, according to the header of the routine, the routine must yield a value of mode `INT`. The context of the body of a routine is strong. Although a serial clause cannot be coerced, the context of the serial clause is passed to the last phrase of that clause. In this case, we have a value of mode `REF INT` which, in a strong context, can be coerced to a value of mode `INT` by dereferencing.

Exercises

6.1 What is the formal definition of an identity declaration? Ans

6.2 Why is the parameter of a routine denotation called a formal parameter? Ans

6.3 In the routine denotation

```
(REAL r)INT: ENTIER r;
```

Ans

(a) What is the mode of the formal parameter?

(b) What is the mode of the value yielded?
(c) What is the context of the body of the routine?
(d) If the value of `r` were `-4.6`, what value would the routine yield?

6.4 Write a routine which takes a parameter of mode `[]INT` and yields a value of mode `[]CHAR`, where each element of the result yields the character equivalent of the corresponding element in the parameter (use `FOR` and `REPR`). Ans

---

6.1.1 Routine modes

In general, a routine may have any number of parameters, including none, as we shall see. The mode of the parameters may be any mode, and the value yielded may be any mode. The modes written for the parameters and the yield are always formal declarers, so no bounds are used if the modes of the parameters or yield involve multiples.

Here is a possible header of a more complicated routine:

```
(INT i,REF[,]CHAR c,REAL a,REAL b)BOOL:
```

A minor abbreviation would be possible in this case. The

```
REAL a,REAL b
```

could be written `REAL a,b` giving

```
(INT i,REF[,]CHAR c,REAL a,b)BOOL:
```

Notice that the parameters are separated by commas. This means that when the routine is used, the actual parameters are evaluated collaterally. We shall see later that this is important when we consider **side-effects**.

The order in which parameters are written in the header is of no particular significance.

The mode of the routine whose header is given above is

```
(INT,REF[,]CHAR,REAL,REAL)BOOL
```

("with int ref row row of car real real yielding bool").

6.1.2 Multiples as parameters

Since a formal parameter which is a multiple has no bounds written in it, any multiple having that mode could be used as the actual parameter. This means that if you need to know the bounds of the actual multiple, you will need to use the bounds interrogation operators. For example, here is a routine denotation which finds the smallest element in its multiple parameter:

```
([]INT a)INT:
(
   INT min:=a[LWB a];

   FOR i FROM LWB a TO UPB a
   DO
       min:=min MIN a[i]
   OD;
   min
)
```

6.1.3 Names as parameters

The second parameter in the more complicated routine header given in section 6.1.1 had the mode `REF[,]CHAR`. When a parameter is a name, the body of the routine can have an assignment which makes the name refer to a new value. For example, here is a routine denotation which assigns a value to its parameter:

```
(REF INT a)INT:  a:=2
```

Notice that the unit in this case is a single phrase and so does not need to be enclosed. Here is a routine denotation which has two parameters and which yields a value of mode `BOOL`:

```
(REF[]CHAR rc,[]CHAR c)BOOL:
IF UPB rc - LWB rc /= UPB c - LWB c
THEN FALSE
ELSE rc[:]:=c[:];  TRUE
FI
```

Here, the body is a conditional clause which is another kind of enclosed clause. Note the use of trimmers to ensure that the bounds of the multiples on each side of the assignment match.

If a flexible name could be used as an actual parameter, then the mode of the formal parameter must include the mode constructor `FLEX`. For example,

```
(REF FLEX[]CHAR s)INT:
(CO Code to compute the number of words CO)
```

Of course, in this example, the mode of `s` could have been given as `REF STRING`.

6.1.4 The mode `VOID`

A routine must yield a value of some mode, but it is possible to throw away that value using the voiding coercion. The mode `VOID` has a single value whose denotation is `EMPTY`. In practice, because the context of the yield of a routine is strong, use of `EMPTY` is usually unnecessary (but see section 8.2). Here is another way of writing the last routine in the previous section:

```
(REF[]CHAR rc,[]CHAR c)VOID:
IF UPB rc - LWB rc /= UPB c - LWB c
THEN
   print(("Bounds mismatch",newline));
   stop
ELSE rc[:]:=c[:]
FI
```

This version produces an emergency error message and terminates the program prematurely (see section 4 for details of `stop`). Since the yield is `VOID`, any value the conditional clause might yield will be thrown away. A `FOR` loop yields `EMPTY` and a semicolon voids the previous unit. Declarations yield no value, not even `EMPTY`.

6.1.5 Routines yielding names

Since the yield of a routine can be a value of any mode, a routine can yield a name, but there is a restriction: the name yielded must have a scope larger than the body of the routine. This means that any names declared to be *local*, cannot be passed from the routine. Names which exist outwith the scope of the routine can be passed via a parameter and yielded by the routine. For example, here is a routine denotation which yields the name passed by such a parameter:

```
(REF INT a)REF INT:  a:=2
```

Compare this routine with the first routine denotation in section 6.1.3.

In chapter 5, we said that a new name can be declared using the generator `LOC`. For example, here is an identity declaration for a name:

```
REF INT x = LOC INT
```

The range of the identifier `x` is the smallest enclosed clause in which it has been declared. The scope of the value it identifies is limited to that smallest enclosed clause because the generator used is the **local generator** `LOC`. Here is a routine which tries to yield a name declared within its body:

```
(INT a)REF INT:
  (REF INT x = LOC INT:=a;  x) #wrong!#
```

This routine is wrong because the scope of the name identified by `x` is limited to the body of the routine. Note, however, the a68toc Algol 68 compiler provides neither compile-time nor run-time scope checking so that it is possible to yield a locally declared name. However, the rest of the program would be undefined—you might or might not get meaningful things happening. When scopes are checked, this sort of error cannot occur.

However, there is a way of yielding a name declared in a routine. This is achieved using a global generator. If `x` above were declared as

```
REF INT x = HEAP INT
```

or, in abbreviated form, `HEAP INT x`, then the scope of the name identified by `x` would be from its declaration to the end of the program even though the range of the *identifier* `x` is limited to the body of the routine:

```
(INT a)REF INT:  (HEAP INT x:=a;  x)
```

Notice that the mode of the yield is still `REF INT`. All that has changed is the scope of the value yielded. Of course, you would not be able to identify the yielded name using `x`, but we'll come to that problem when we deal with how routines are used. Notice that the global generator is written `HEAP` instead of `GLOB` as you might expect. This is because global generators use a different method of allocating storage for names with global scope and, historically, this different method is recorded in the mode constructor.

The difference between range and scope is that identifiers have range, but values have scope. Furthermore, denotations have global scope.

Exercises

6.5 Write the header of a routine with a parameter of mode `REF REAL` and which yields a value of mode `REAL`. Ans

6.6 Write the header of a routine which takes two parameters each of which is a name of mode `REF CHAR`, and yields a name of mode `REF CHAR`. Ans

6.7 Write a routine which takes a parameter of mode `STRING` and yields a value of mode `[]STRING` consisting of the words of the parameter (use your answer to exercises in section 5.11). Ans

6.1.6 Parameterless routines

A routine can have no parameters. In the header, the parentheses normally enclosing the formal parameter **list** (either one parameter, or more than one separated by commas) are also omitted. Here is a routine with no parameters and which yields a value of mode `INT`:

```
INT: 2*3**4 - ENTIER 36.5
```

It would be more usual to use identifiers which had been declared in some enclosing range. For example,

```
INT: 2*a**4 - ENTIER b
```

Routines which have no parameters and yield no value are fairly common. For example,

```
VOID:  print(2)
```

Strictly speaking, there is one value having the mode `VOID`, but there's not a lot you can do with it. In practice, `VOID` routines usually use identifiers declared in an enclosing range (they are called identifiers global to the routine). For example:

```
VOID: (x:=a;  x<=2|print(x)|print("Over 2"))
```

where the body is an abbreviated conditional clause, and `x` and `a` have been declared globally with appropriate modes.

Assignment of values to names declared globally[1] to the routine is known as a **side-effect**. We shall deal with side-effects when we describe how routines are used, and we shall show why side-effects are undesirable. If you write parameterless routines, it is preferable not to put any assignments to globally-declared names in them. In fact, it would be safer to say: "In a routine, don't assign to names not declared in the routine or not provided as parameters". Side-effects are messy and are usually a sign of badly designed programs. It is better to use a parameter (or an extra parameter) using a name, and then assign to that name. This ensures that values can only get into or out of your routines via the header, and results in a much cleaner design. Cleanly designed programs are easier to write and easier to maintain.

[1] The phrase "names declared globally" is intended to mean here that the names have been declared in a range which encloses the routine, not that `HEAP` has necessarily been used in the declaration. We use the phrase "a global name" in the latter case.

Exercises

6.8 Write the header of a routine which yields a value of mode `REAL`, but takes no parameters. Ans

6.9 Write a routine of mode `VOID` which prints

```
Hi, there
```

on your screen. Ans

6.2 Operators

In the preamble to this chapter, it was mentioned that routines consist of two kinds: procedures and operators. See section 6.3 for details of procedures.

An operator has a mode and a value (its routine denotation) and, if dyadic, a priority. The parameters to routines which are defined as operators are called operands. Monadic operators, while not having a priority, behave as though they had a priority greater than any dyadic operator and take one operand and yield a value of some mode.

Here is an identity declaration of the monadic operator `B`:

```
OP(INT)INT B = (INT a)INT: a
```

There are several points to note.

1. The mode of the operator is `OP(INT)INT`. That is, it takes a single operand of mode `INT` and yields a value of mode `INT`.
2. The symbol for the operator looks like a mode indicant. It isn't a mode indicant, but obeys the same rules (starts with an uppercase letter and possibly continues with uppercase letters or digits, and no spaces are allowed inside the symbol).
3. The right-hand side of the identity declaration is a routine denotation. A special identity declaration is used for operators: instead of the mode of the operator, the mode constructor `OP` is used followed by the operator symbol. The abbreviated declaration of the operator `B` is

```
OP B = (INT a)INT: a
```

Chapter 2 described how operators are used in formulæ. A possible formula using `B` could be

```
B 2
```

which would yield `2`.

6.2.1 Identification of operators

This section is more difficult than preceding sections and could be omitted on a first reading. You are unlikely to fall afoul of what is described here unless you are declaring many new operators.

One of the most useful properties of operators is that there can be more than one declaration of the same operator symbol using an operand having a different mode. This is called "operator overloading". How does the compiler know which version of the operator to use? Before answering this question, consider the following program fragment:

```
 1 BEGIN
 2    OP D = (INT a)INT:   a+2;
 3    OP D = (REAL a)REAL: a+2.0;
 4    REAL x:=1.5, a:=-2.0; INT i:=4;
 5
 6    x:=IF  OP D = (REF REAL a)REF REAL:
 7                      a+:=2.0;
 8           ENTIER(D a:=x) > i
 9       THEN D i
10       ELSE D x
11       FI;
12
13    OP D = (REF REAL a)REF REAL:  a+:=3.0;
14    x:=D a
15 END
```

The numbers on the left-hand side are not part of the program. As you can see, there are four declarations of `D`: one with an `INT` operand, one with a `REAL` operand and two with a `REF REAL` operand. If you try compiling this you will get the error

```
ERROR (146) more than one version of D
```

for the last declaration. There are two points to be made here.

1. Outside the conditional clause, there are three declarations of `D`: on lines 2, 3 and 13. Now, an operator is used in a formula and the context of the operand of an operator is firm. Of the coercions we have met so far, only one, namely dereferencing, is allowed in a firm context. If you look at the assignment on line 14, you can see that the mode of the operand of `D` is `REF REAL` (from the declaration of `a` on line 4). Now a value of mode `REF REAL` is firmly coercible to `REAL` (by dereferencing). So there are two declarations of `D` which could be used: the declaration on line 3 and the declaration on line 13 (the range of the declaration on line 6 is confined to the conditional clause). According to the rules for the identification of operators (see below), the compiler would not be able to distinguish between the two declarations. Hence the error message.

2. Why did the identical declaration of `D` on line 6 not cause a similar error message? Answer: because the declaration on line 6 is at the start of a new range: the enclosed clause starting on line 6 and extending to the `FI` on line 11. Since that is a new range, any operator declarations with a

mode which is **firmly related** to the mode of an operator declared in an outer range makes the declaration in the outer range inaccessible. Thus, the assignment on line 8 will use the version of `D` declared on line 6, the `D` on line 9 identifies the `D` declared on line 2, and the `D` on line 10 again uses the `D` declared on line 6.

Thus, in determining which operator to use, the compiler firstly finds a declaration whose mode can be obtained from the operands in question using any of the coercions allowed in a firm context (chapter 10 will state all the coercions allowed). Secondly, it will use the declaration in the smallest range enclosing the formula.

The declaration of an object is known as its **defining occurrence**. Where the object is used is called its **applied occurrence**. In practice, it is rare to find like operator declarations in nested ranges.

Exercises

6.10 This and the following exercise use the following program fragment:

```
 1 IF
 2    OP T = (INT a)INT:  a*a;
 3    OP T = (CHAR a)INT: ABS a * ABS a;
 4    INT p:=3, q:=4; CHAR c:=REPR 3;
 5    T p < T c
 6 THEN
 7    OP T = (REF INT a)REF INT: a*:=a;
 8    IF T 4 < T q
 9    THEN "Yes"
10    ELSE T REPR 2
11    FI
12 ELSE T c > T q
13 FI
```

There are 3 defining occurrences of the operator `T` on lines 2, 3 and 7. There are 7 applied occurrences of the operator (on lines 5, 8, 10 and 12). Which defining occurrence is used for each applied occurrence? Ans

6.11 What is the mode and value yielded by Ans

(a) `T q` on line 8
(b) `T q` on line 12
(c) `T c` on line 12
(d) `T REPR 2` on line 10

6.12 What is wrong with these two declarations occurring in the same range:

```
OP TT = ([]INT a)[]INT:
        FOR i FROM LWB a TO UPB a
        DO print(a[i]*3) OD;
OP TT = (REF[]INT a)[]INT:
        FOR i FROM LWB a TO UPB a
        DO print(a[i]*3) OD
```

Ans

6.2.2 Operator usage

Before we go on to dyadic operators, there is one more point to consider. Given the operator declaration

```
OP PLUS2 = (REAL a)REAL:  a+2.0
```

what is the mechanism by which the formal parameter gets its value? Firstly, we must remember that a particular version of the operator is chosen on the basis of firmly relatedness. In other words, only coercions allowed in a firm context can determine which declaration of the operator to use. Secondly, in elaborating the formula

```
PLUS2 x
```

where `x` has the mode `REF REAL`, the compiler elaborates the identity declaration

```
REAL a = x
```

where `REAL a` is the formal parameter. Since the context of the right-hand side of an identity declaration is strong, any of the strong coercions would normally be allowed (all coercions, including dereferencing). However, because the version of the operator was chosen on the basis of firmly relatedness, the coercions available in a strong context which are not available in a firm context (that is, widening and rowing) are not available in the context of an operand. If an operand of mode `INT` is supplied to an operator requiring a `REAL`, the compiler will flag an error: widening would not occur. This is the only exception to the rule that the right-hand side of an identity declaration is a strong context.

It was pointed out in section 6.1.5 that a routine can yield a name. An operator does not usually yield a name because subsequent use of the name usually involves dereferencing and using the value the name refers to. However, here is an operator declaration which yields a name of a multiple which is used in a subsequent phrase:

```
OP NAME = (INT a)REF[]INT:
            (HEAP[2]INT x:=(a,a);  x);
REF[]INT a = NAME 3
```

After the elaboration of the identity declaration, the name could be accessed wherever necessary.

Exercises

6.13 Given the declarations

```
OP M3 = (INT i)INT:  i-3;
OP M3 = ([]INT i)[]INT:
        FORALL n IN i DO n-3 OD;
INT i:=1,[3]INT j:=(1,2,3)
```

which operator declarations would be used for the following formulæ Ans

(a) `M3 i`
(b) `M3 j[2]`
(c) `M3 j`
(d) `M3 j[:2]`

6.2.3 Dyadic operators

The only differences between monadic and dyadic operators are that the latter have a priority and take two operands. Therefore the routine denotation used for a dyadic operator has two formal parameters. The priority of a dyadic operator is declared using the indicant `PRIO`:

```
PRIO HMEAN = 7; PRIO WHMEAN = 6
```

The declaration of the priority of the operator uses an integer denotation in the range 1 to 9 on the right-hand side.

Consecutive priority declarations do not need to repeat the `PRIO`, but can be abbreviated in the usual way. The priority declaration relates to the operator symbol. Hence the same operator cannot have two different priorities in the same range, but there is no reason why an operator cannot have different priorities in different ranges. A priority declaration does not count as a declaration when determining the scope of a local name.

If an existing operator symbol is used in a new declaration, the priority of the new operator must be the same as the old if it is in the same range, so the priority declaration should be omitted.

The identification of dyadic operators proceeds exactly as for monadic operators except that the most recently declared priority in the same range is used to determine the order of elaboration of operators in a formula. Again, two operators using the same symbol cannot be declared in the same range if they have firmly related modes (see section 6.2.1).

These declarations apply to the remainder of this section:

```
PRIO HMEAN = 7, WHMEAN = 6;
OP HMEAN  = (REAL a,b)REAL:
              2.0/(1.0/a+1.0/b);
OP WHMEAN = (REAL a,b)REAL:
              2.0/(1.0/a+2.0/b)
```

If `HMEAN` appears in the formula

```
x HMEAN y
```

where `x` and `y` both have mode `REF REAL`, the compiler constructs the identity declarations

```
REAL a = x, REAL b = y
```

Notice that the two identity declarations are elaborated collaterally (due to the comma separating them), which could be important (see below). If `x` refers to `2.5` and `y` refers to `3.5`, the formula will yield

```
2.0/(1.0/2.5 + 1.0/3.5)
```

which is $2.91\dot{6}$. Likewise, the formula

```
x WHMEAN y
```

would yield $2 \cdot 058\,823\,529\,411\,76$. Now consider the formula

```
(x+:=1.0) WHMEAN (x+:=1.0)
```

which cause the value referred to by `x` to be incremented twice as a side-effect. The resulting identity declarations are

```
REAL a = (x+:=1.0), REAL b = (x+:=1.0)
```

The definition of Algol 68 says that the operands of a dyadic operator should be elaborated collaterally, so the order of elaboration is unknown. Suppose `x` refers to `1.0` before the formula is elaborated. There are three cases:

1. The identity declaration for `a` is elaborated first, giving `a=2.0` and `b=3.0`. The formula will yield $1.714\,285\,714$.
2. The identity declaration for `b` is elaborated first, giving `b=2.0` and `a=3.0`. The formula will yield 1.5.
3. The identity declarations are elaborated in parallel. In this case, the result could be indeterminate.

If you compile a program using a68toc with the declaration for `WHMEAN` and try to compute the formula given above, you get the result `+1.5000000000000000` which suggests that case 2 holds.

If `x` refers to `1.0`, then the formula

```
1.0/(x+:=1.0) + 1.0/(x+:=1.0)
```

yields `+.83333333333333339e +0` which is correct provided that the two operands are elaborated sequentially. The moral of all this is: avoid side-effects like the plague.

What happens if the identifier of an actual parameter is the same as the identifier of the formal parameter? There is no clash. Consider the identity declaration

```
INT a = a
```

where the `a` on the left-hand side is the formal parameter for a routine denotation, and the `a` on the right-hand side is an actual parameter declared in some surrounding range. The formal parameter occurs at the start of a new range. Within that range, the identifier `a` in the outer range becomes inaccessible, but at the moment that the identity declaration is being elaborated, the formal parameter is made to identify the value of the actual parameter which, of course, is not an identifier. So go ahead and use identical identifiers for formal parameters and actual parameters.

6.2.4 Operator symbols

Most of the operators described in chapters 2 to 5 used symbols rather than upper-case letters. You may use any combination of the `<=>*/:` symbols (and any number of them) except `:=`, `:=:` and `:/=:` (the latter two are described in chapter 11). Any of the symbols `+-?&%` can only start a compound symbol. Of course, they can stand on their own for an operator. In chapter 11, you will find the `<<` and `>>` operators described as well as more declarations for existing operators. Here are some declarations of operators using the above rules:

```
OP *** = (INT a)INT: a*a*a;
OP %< = (CHAR c)CHAR: (c<" "|" "|c);
OP -:: = (CHAR c)INT: (ABS c-ABS"0")
```

We have now covered everything about operators in the language.

Exercises

6.14 Why are side-effects undesirable? Ans

6.15 What is wrong with these operator symbols: Ans

(a) `M*`
(b) `%+/`
(c) `:=:`

6.16 Declare an operator using the symbol `PP` which will add `1` to the value its `REF INT` operand refers to, and which will yield the name of its parameter. Ans

---

6.3 Procedures

The second way of using routines is to declare them as procedures. We have seen that an operator can be declared and used, have a mode and a value (its routine denotation), but apart from having an operator symbol, it cannot be identified with an identifier in the way that a name or a denotation of a `CHAR` value can. Procedures are quite different.

Firstly, here are some general remarks on the way procedures differ from operators. The mode of a procedure always starts with the mode constructor `PROC`. A procedure can have any number of parameters, including none. Two procedures having the same identifier cannot be declared in the same range (so "overloading" is not allowed). When a procedure is used, its parameters, if any, are in a strong context. This means that rowing and widening are available.

Procedures are declared using the mode constructor `PROC`. Here is a procedure which creates a range of characters:

```
PROC(CHAR,CHAR)[]CHAR range =
   (CHAR a,b)[]CHAR:
   BEGIN
```

```
        CHAR aa,bb;

        (a<=b|aa:=a; bb:=b|aa:=b; bb:=a);

        [ABS aa:ABS bb]CHAR r;

        FOR i
        FROM LWB r TO UPB r
        DO
           r[i]:=REPR i
        OD;
        r
    END
```

This procedure identity declaration resembles the declaration for a multiple: much of the mode is repeated on the right-hand side and the formal-declarer on the left-hand side has no identifiers for the modes of the parameters. Notice that the modes of the parameters must be repeated in the formal-declarer, but that the mode of the procedure on the right-hand side can contain the usual abbreviation. Here is the abbreviated header:

```
PROC range = (CHAR a,b)[]CHAR:
```

The formal-declarer is important for creating synonyms:

```
PROC(REAL)REAL sine = sin
```

Two or more procedure declarations can be separated by commas, even if the procedures have different modes. Consider, for example:

```
PROC pa = (INT i)INT:  i*i,
     pb = (INT i)CHAR: REPR(i*i),
     pc = (INT i)REAL: (i=0|0|1/i)
```

6.3.1 Parameterless procedures

Procedures can have no parameters. Suppose the following names have been declared:

```
INT i,j
```

Here is a procedure with mode `PROC INT` which yields an `INT`:

```
PROC INT p1 = INT: i:=3+j
```

A procedure can be invoked or called by writing its identifier. For example, the procedure `p1` would be called by

```
p1
```

or

```
INT a = p1
```

The right-hand side of this identity declaration requires a value of mode `INT`, but it has been given a unit of mode `PROC INT`. This is converted into a value of mode `INT` by the coercion known as **deproceduring**. This coercion is available in every context (even soft).

Have you realised that `print` must be the identifier of a procedure? Well done! However, we cannot talk about its parameters yet because we don't know enough about the language.

Here is another procedure which yields a name of mode `REF INT`. The mode of the procedure is `PROC REF INT`:

```
PROC p2 = REF INT: IF i < 0 THEN i ELSE j FI
```

and assumes that the names identified by `i` and `j` had already been declared. Here is an identity declaration which uses `p2`:

```
REF INT i or j = p2
```

Because `p2` yields a name, it can be used on the left-hand side of an assignment:

```
p2:=4
```

Here, `4` will be assigned to `i` or `j` depending on the value `i` refers to. The left-hand side of an assignment has a soft context in which only the deproceduring coercion is allowed.

In procedures `p1` and `p2`, the identifier `i` had been declared globally to the procedures. Assignment to such an identifier is, as already stated, a side-effect. Here is another procedure of mode `PROC INT` which uses a global identifier, but does not assign to it:

```
PROC p3 = REAL:
(
   [i]REAL a;  read((a,newline));
   REAL sum:=0.0;

   FOR i FROM LWB a TO UPB a
   DO
      sum+:=a[i]
   OD;

   sum
)
```

and here is a call of `p3`:

```
print(p3)
```

In the identity declaration

```
REAL r = p2
```

`p2` is deprocedured to yield a name of mode `REF INT`, dereferenced to yield an `INT`, and then widened to yield a `REAL`. All these coercions are available in a strong context (the right-hand side of an identity declaration).

The call of a procedure can appear in a formula without parentheses. Here is an example:

```
p2:=p1 * ROUND p3
```

If we call the procedure `p1`, declared above, its value does not have to be used. For example, in

```
p1;
```

the value yielded by `p1` has been voided by the following semicolon after the procedure had been called.

In the section on routines, we introduced the mode `VOID`. Here is a procedure yielding `VOID`:

```
PROC p4 = VOID:  print(p3)
```

and a possible use:

```
; p4;
```

where the semicolons show that the call stands on its own.

When a parameterless procedure yields a multiple, the call of that procedure can be sliced to get an individual element. For example, suppose we declare

```
PROC p5 = [,]REAL:
(
   [i,j]REAL a;
   read((a,newline));
   a
)
```

where `i` and `j` were declared above, and then call `p5` in the formula

```
REAL x = p5[i-3,j] * 2
```

When `p5` is called, it yields a two-dimensional multiple of mode `[,]REAL` which is then sliced using the two subscripts (assuming that `i-3` is within the bounds of the first dimension) to yield a value of mode `REAL`, which is then used in the formula.

Procedure `p2`, declared above, yielded a name declared globally to the procedure. As explained in the sections on routines, a procedure cannot yield a locally-generated name. However, if the name is generated using `HEAP`, then the name can be yielded as in `p6`:

```
PROC p6 = REF INT: (HEAP INT i:=3;  i)
```

Here is a call of `p6` where the yielded name is captured with an identity declaration:

```
REF INT global int = p6
```

Then `print(global int)` will display `3`.

The yield of a procedure can be another procedure. Consider this program fragment:

```
PROC q2 = INT: max int % 2,
     q3 = INT: max int % 3,
     q4 = INT: max int % 4,
     q5 = INT: max int % 5;

INT i;  read((i,newline));

PROC q = PROC INT:
   CASE i+1 IN q2,q3,q4 OUT q5 ESAC
```

Procedure `q` will yield one of the predeclared procedures depending on the value of `i`. Here, the yielded procedure will not be deprocedured because the mode required is a procedure.

One parameterless procedure is provided in the standard prelude. Its identifier is `random`, and when called returns the next pseudo-random real number of a series. If called a large number of times, the numbers yielded are uniformly distributed in the range $[0, 1)$.

Exercises

6.17 Write a procedure which assigns a value to a name declared globally to the procedure. Ans

6.18 Write a procedure which reads an integer from the keyboard, then declares a dynamic name of a multiple of one dimension, and reads that number of integers from the keyboard. Now compute the sum of all the integers, and yield its value as the yield of the procedure. Ans

6.19 Write a procedure which yields the name of a two dimensional multiple containing characters read from the keyboard. The mode of the multiple should be `REF[,]CHAR`. Ans

---

6.3.2 Procedures with parameters

Parameters of procedures can have any mode (including procedures). Unlike operators, procedures can have any number of parameters. The parameters are written as a parameter **list** which consists of one parameter, or two or more separated by commas.

Here is a procedure with a single parameter:

```
PROC(INT)CHAR p7 = (INT i)CHAR: REPR(i>0|i|0)
```

This is a full identity declaration for `p7`. It can be abbreviated to

```
PROC p7 = (INT i)CHAR: REPR(i>0|i|0)
```

The mode of `p7` is `PROC(INT)CHAR`. That is, `p7` is a procedure with a single integer parameter and yielding a character. Here is a call of `p7`:

```
CHAR c = p7(-3)
```

Note that the single parameter is written between parentheses. Since the context of an actual parameter of a procedure is strong, a name of mode `REF INT` could be used:

```
CHAR c = p7(i)
```

or

```
CHAR c = p7(ai[j])
```

where `ai` has mode `REF[]INT` and `j` has mode `INT` or `REF INT` or `PROC INT` (or even `PROC REF INT`).

Here is a procedure which takes three parameters:

```
PROC char in string =
   (CHAR c,REF INT p,STRING s)BOOL:
(
   BOOL found:= FALSE;
   FOR k FROM LWB s TO UPB s
   WHILE NOT found
   DO
      (c = s[k] | i:=k; found:= TRUE)
   OD;
   found
)
```

The procedure (which is in the standard prelude) tests whether a character is in a string, and if it is, returns its position in the parameter `p`. The procedure yields `TRUE` if the character is in the string, and `FALSE` if not. Here is a possible call of the procedure:

```
IF INT p;  char in string(char,"abcde",p)
THEN ...
```

where `char` was declared in an outer range. Notice that the `REF INT` parameter of `char in string` is not assigned a new value if the character is not found in the string.

When calling a procedure, the call must supply the same number of actual parameters, and in the same order, as there are formal parameters in the procedure declaration.

If a multiple is one of the formal parameters, a row-display can be supplied as an actual parameter (remember that a row-display can only occur in a strong context). In this case, the row-display counts as a single parameter, but the number of elements in the row-display can differ in successive calls since the bounds of the multiple can be determined by the procedure using the bounds interrogation operators. Here is an example:

```
PROC pb = ([]INT m)INT:
(INT sum:=0;
 FOR i FROM LWB m TO UPB m DO sum+:= m[i] OD;
 sum)
```

and here are some calls of `pb`:

```
pb((1,2,3))   pb((2,3,5,7,11,13))
```

Again, procedures with parameters can assign to, or use, globally declared names and other values, but it is better to include the name in the header of the procedure. Here is a procedure which reads data into a globally declared multiple using that multiple as a parameter:

```
PROC rm = (REF[]REAL a)VOID:
   read((a,newline))
```

It could now be called by

```
rm(multiple)
```

where `multiple` had been previously declared as having mode `REF[]REAL`.

As described in section 6.1.3, a flexible name can be used as an actual parameter provided that the formal parameter has also been declared as being flexible. For example, here is a procedure which takes a single parameter of mode `REF STRING` and which yields an `INT`:

```
PROC read line = (REF STRING s)INT:
(
   read((s,newline));
   UPB s #LWB is 1#
)
```

`read line` reads the next line of characters from the keyboard, assigns it to its parameter, which is a flexible name, and yields the length of the line.

Exercises

6.20 Write a procedure which takes a `REF REAL` parameter, divides the value it refers to by π, multiplies it by 180, assigns the final value to its parameter, and yields the parameter (that is, its name). Ans

6.21 Write a procedure which takes two parameters: the first should have mode `STRING` and the second mode `INT`. Display the string on the screen the number of times given by the integer. If the integer is negative, display a newline first and then use the absolute value (use the operator `ABS`) of the integer. Yield the mode `VOID`. Ans

6.22 Write a procedure, identified as `num in multiple`, which does for an integer what `char in string` does for a character. Ans

---

6.3.3 Procedures as parameters

Here is a procedure which takes a procedure as a parameter:

```
PROC sum = (INT n,PROC(INT)REAL p)REAL:
(
   REAL s:=0;
   FOR i TO n DO s+:=p(i) OD;
   s
)
```

Notice that the mode of the procedure parameter is a formal mode so no identifier is required for its `INT` parameter in the header of the procedure `sum`. In the loop clause, the procedure is called with an actual parameter.

When a parameter must be a procedure, there are two ways in which it can be supplied. Firstly, a predeclared procedure identifier can be supplied, as in

```
PROC pa = (INT a)REAL: 1/a;
sum(34,pa)
```

Secondly, a routine denotation can be supplied:

```
sum(34,(INT a)REAL: 1/a)
```

A routine denotation is a unit. In this case, the routine denotation has the mode `PROC(INT)REAL`, so it can be used in the call of `sum`. Notice also that, because the routine denotation is an actual parameter, its header includes the identifier `a`. In fact, routine denotations can be used wherever a procedure is required, so long as the denotation has the required mode. The routine denotation given in the call is on the right-hand side of the implied identity declaration of the elaboration of the parameter. It is an example of an **anonymous** routine denotation.

Exercises

6.23 Given the declaration of `sum` in the text, what is the value of: Ans

(a) `sum(4,(INT a)REAL: a)`
(b) `sum(2,(INT b)REAL: 1/(5*b))`
(c) `sum(0,pa)` (`pa` is declared in the text)

6.3.4 Recursion

One of the fun aspects of using procedures is that a procedure can call itself. This is known as **recursion**. For example, here is a simplistic way of calculating a factorial:

```
PROC factorial = (INT n)INT:
   (n=1|1|n*factorial(n-1))
```

Try it with the call

```
factorial(7)
```

Here is another recursively defined procedure which displays an integer on the screen in minimum space:

```
PROC ai = (INT i)VOID:
IF   i < 0  THEN print("-");  ai(ABS i)
ELIF i < 10 THEN print(REPR(i+ABS"0"))
ELSE ai(i%10);  ai(i MOD 10)
FI
```

In each of these two cases, the procedure includes a test which chooses between a recursive call and phrases which do not result in a recursive call. This is necessary because, otherwise, the procedure would never complete. Neither of these procedures uses a locally declared value. Here is one which does:

```
PROC new fact = (INT i)INT:
IF INT n:=i-1;   n = 1
THEN 2
ELSE i*new fact(n)
FI
```

The example is somewhat artificial, but illustrates the point. If `new fact` is called by, for example, `new fact(3)`, then in the first call, `n` will have the value `2`, and `new fact` will be called again with the parameter equal to `2`. In the second call, `n` will be `1`, but this `n` this time round will be a new `n`, with the first `n` inaccessible (it being declared in an enclosing range). `new fact` will yield `2`, and this value will be used in the formula on line 4 of the procedure. The first call to `new fact` will then exit with the value `6`.

Apart from being fun, recursive procedures can be an efficient way of programming a particular problem. Chapter 11 deals with, amongst other topics, recursive modes, and there, recursive programming comes into its own.

A different form of recursion, known as **mutual recursion**, is exemplified by two procedures which call each other. You have to ensure there is no circularity. The principal difficulty of how to use a procedure before it has been declared is overcome by first declaring a procedure name and then assigning a routine denotation to the procedure name after the other procedure has been declared. Here is a simple example:[2]

[2] A compiler which implements the Algol 68 defined by the Revised Report would not have to resort to this device because the declaration of each procedure would be available everywhere in the enclosing range (but see section 6.3.6).

```
PROC(INT)INT pb;
PROC pa = (INT i)INT: (i>0|pb(i-1)|i);
pb:=(INT i)INT: (i<0|pa(i+1)|i);
```

Then `pa(4)` would yield `3` and `pa(-4)` would yield `-4`. Similarly, `pb(4)` would yield `4` and `pb(-4)` would yield `-3`. Notice that the right-hand side of the assignment is an anonymous routine denotation.

Exercises

6.24 Write a recursive procedure to reverse the order of letters in a value of mode `[]CHAR`. It should yield a value also of mode `[]CHAR`. Ans

6.25 Write two mutually recursive procedures which take an integer parameter and which yield an `INT`. The first should call the second if the parameter is odd, and the second should call the first if the parameter is even. The alternative option should yield the square of the parameter for the first procedure and the cube of the parameter for the second procedure. Use `square` and `cube` as the procedure identifiers. Ans

---

6.3.5 Standard procedures

The standard prelude contains the declarations of more than 60 procedures, most of them concerned with transput (see chapter nine). A number of procedures, all having the mode

```
PROC(REAL)REAL
```

are declared in the standard prelude and yield the values of common mathematical functions. These are `sqrt`, `exp`, `ln`, `cos`, `sin`, `tan`, `arctan`, `arcsin` and `arccos`. Naturally, you must be careful to ensure that the actual parameter for `sqrt` is non-negative, and that the actual parameter for `ln` is greater than zero. The procedures `cos`, `sin` and `tan` expect their `REAL` parameter to be in radians.

New procedures using these predeclared procedures can be declared:

```
PROC sinh =
   (REAL x)REAL: (exp(x) + exp(-x))/2
```

A variety of pseudo-random numbers can be produced using `random int`. The mode of the procedure `random int` is

```
PROC(INT)INT
```

and yields a pseudo-random integer greater than or equal to one, and less than or equal to its integer parameter. For example, here is a procedure which will compute the percentage of each possible die throw in 10 000 such throws:

```
PROC percentage = []REAL:
(
   PROC throw = INT: random int(6);

   [6]REAL result:=(0,0,0,0,0,0);

   TO 10 000 DO result[throw]+:=1 OD;

   FOR i FROM LWB result TO UPB result
   DO result[i] /:= 10 000 OD;

   result
)
```

Notice that `percentage` has another procedure (`throw`) declared within it. There is no limit to such nesting.

6.3.6 Other features of procedures

Since a procedure is a value, it is possible to declare values whose modes include a procedure mode. For example, here is a multiple of procedures:

```
[]PROC(REAL)REAL pr = (sin,cos,tan)
```

and here is a possible call:

```
pr[2](2)
```

which yields $-0.416\,146\,836\,5$. We could also declare a procedure which could be called with the expression

```
pr(2)[2]
```

but this is left as an exercise.

Similarly, names of procedures can be declared and can be quite useful. Instead of declaring

```
PROC pc = (INT i)PROC(REAL)REAL: pr[i]
```

using `pr` declared above, with a possible call of `pc(2)` we could write

```
PROC(REAL)REAL pn:=pr[i]
```

and then use `pn` instead of `pc`. The advantage of this would be that `pr` would be subscripted only once instead of whenever `pc` is elaborated. Furthermore, another procedure could be assigned to `pn` and the procedure it refers to again called. Using `pn` would usually involve dereferencing.

There are scoping problems involved with procedure names. Although the scope of a denotation is global, procedure denotations may include an identifier whose range is not global. For this reason, the scope of a procedure denotation is limited to the smallest enclosing clause containing a declaration of an identifier or mode or operator indicant which is used in the procedure denotation.

For example, in this program fragment

```
PROC REAL pp;  REAL y;
BEGIN
   REAL x:=3.0;
   PROC p = REAL:  x:=4.0;
   print(p);
   pp:=p; CO wrong CO
   print(x)
END;
print(("pp=",pp)) #wrong#
```

the assignment in line 6 is wrong because the scope of the right-hand side is less than the scope of the left-hand side. Unfortunately, the a68toc compiler does not perform scope checking and so will not flag the incorrect assignment.

There are times when `SKIP` is useful in a procedure declaration:

```
PROC p = REAL:
IF x<0
THEN print("Negative parameter"); stop; SKIP
ELSE sqrt(x)
FI
```

The yield of the procedure is `REAL`, so each part of the conditional clause must yield a value of mode `REAL`. The construct `stop` yields `VOID`, and even in a strong context, `VOID` cannot be coerced to `REAL`. However, `SKIP` will yield an undefined value of any required mode. In this case, `SKIP` yields a value of mode `REAL`, but the value is never used, because the program is terminated just before.

Grouping your program into procedures helps to keep the logic simple at each level. Nesting procedures makes sense when the nested procedures are used only within the outer procedures. This topic is covered in greater depth in chapter 12.

6.4 Summary

The fact that this is one of the longer chapters in the book reflects the importance of routines in Algol 68 programs. Every formula uses operators, and procedures enable a program to be written in small chunks and tested in a piecewise manner.

A routine denotation forms the basis of both operators and procedures. Routine denotations have a well-defined mode, the value being the denotation itself. A routine can declare identifiers within its body, including other routines (whether operators or procedures).

Operators can have one or two operands (as the parameters are called) and usually yield a value of some mode other than `VOID`. Dyadic operators have a priority of 1 to 9. Firmly related operators cannot be declared in the same range. The operator symbol can be a bold indicant (like a mode indicant) or one of or a combination of various symbols.

Procedures can have none or more parameters of any mode, and can yield a value of any mode (including `VOID`). Procedures can call themselves: this is known as recursion.

Rows of procedures, names of procedures and other modes using procedure modes can all be declared and, on occasion, can be useful.

Here are some exercises which cover some of the topics discussed in this rather long chapter.

Exercises

6.26 At the time of the call of a procedure or operator, what is the relationship between the formal parameters and the actual parameters? Ans

6.27 Write an operator which will find the largest element in its two-dimensional row-of-reals parameter. Ans

6.28 Write a procedure, identified by `pr`, which can be called by the phrase `pr(2)[2]`. Ans

6.29 Write a procedure which computes the length of a line read from the keyboard. Ans

Chapter 7

Structures

Structures are a powerful piece of Algol 68, particularly when combined with the unions described in the next chapter. In this chapter, we shall meet another mode constructor, examine complex numbers and their associated operators and learn how to construct new modes. In doing so, you will deepen your understanding of the language.

7.1 Structure denotations

In chapter 3, we saw how a number of individual values can be collected together to form a multiple whose mode was expressed as "row of" the base mode. The principal characteristic of multiples is that all the elements have the same mode. A structure is another way of grouping data elements, but in this case, the individual parts can be, but need not be, of different modes. In general, accessing the elements of a multiple is determined at run-time by the elaboration of a slice. In a structure, access to the individual parts, called **fields**, are determined at compile time. Structures are, therefore, an efficient means of grouping data elements.

The mode constructor `STRUCT` is used to create structure modes. Here is a simple identity declaration of a structure:

```
STRUCT(INT i,CHAR c) s = (2,"e")
```

The mode of the structure is

```
STRUCT(INT i,CHAR c)
```

and its identifier is `s`. The `i` and the `c` are called **field selectors** and are part of the mode. They are not identifiers, even though the rule for identifier construction applies to them, because they are not values in themselves. You cannot say that `i` has mode `INT` because `i` cannot stand by itself. We shall see in the next section how they are used.

The expression to the right of the equals symbol is called a **structure-display**. Like row-displays, structure-displays can only appear in a strong context. In a strong context, the compiler can determine which mode is required and so it can tell whether a row-display or a structure-display has been provided. We could now declare another such structure:

```
STRUCT(INT i,CHAR c) t = s
```

and `t` would have the same value as `s`.

Here is a structure declaration

```
STRUCT(INT j,CHAR c) ss = (2,"e")
```

which looks almost exactly like the first structure declaration above, except that the field selector `i` has been replaced with `j`. The structure `ss` has a different mode from `s` because not only must the constituent modes be the same, but the field selectors must also be identical.

Structure names can be declared:

```
REF STRUCT(INT i,CHAR c) sn =
          LOC STRUCT(INT i,CHAR c)
```

Because the field selectors are part of the mode, they appear on both sides of the declaration. The abbreviated form is

```
STRUCT(INT i,CHAR c) sn
```

We could then write

```
sn:=s
```

in the usual way.

The modes of the fields can be any mode. For example, we can declare

```
STRUCT(REAL x,REAL y,REAL z) vector
```

which can be abbreviated to

```
STRUCT(REAL x,y,z)vector
```

and here is a possible assignment:

```
vector:=(1.3,-4,5.6e10)
```

where the value `-4` would be widened to `-4.0`.

Here is a structure with a procedure field:

```
STRUCT(INT int,
       PROC(REAL)REAL p,
       CHAR char) method = (1,sin,"s")
```

Here is a name referring to such a structure:

```
STRUCT(INT int,
       PROC(REAL)REAL p,
       CHAR char) m := method
```

A structure can even contain another structure:

```
STRUCT(CHAR c,
       STRUCT(INT i,j)s) double=("c",(1,2))
```

In this case, the inner structure has the field selector `s` and *its* field selectors are `i` and `j`.

Exercises

7.1 Declare a structure containing three integer values with field selectors `i`, `j` and `k`. Ans

7.2 Declare a name which can refer to a structure containing an integer, a real and a boolean using field selectors `i`, `r` and `b` respectively. Ans

7.2 Field selection

The field-selectors of a structure mode are used to "extract" the individual fields of a structure. For example, given this declaration for the structure `s`:

```
STRUCT(INT i,CHAR c) s = (2,"e")
```

we can select the first field of `s` using the **selection**

```
i OF s
```

The mode of the selection is `INT` and its value is `2`. Note that the construct `OF` is not an operator. The second field of `s` can be selected using the selection

```
c OF s
```

whose mode is `CHAR` with value `"e"`. The field-selectors cannot be used on their own: they can only be used in a selection.

A selection can be used as an operand. Consider the formula

```
i OF s * ABS c OF s
```

In the structure `method`, declared in the previous section, the procedure in the structure can be selected by

```
p OF method
```

which has the mode `PROC(REAL)REAL`. For a reason which will be clarified in chapter 10, if you want to call this procedure, you must enclose the selection in parentheses:

```
(p OF method)(0.5)
```

Remembering that the context of the actual-parameters of a procedure is strong, you could also write

```
(p OF method)(int OF method)
```

where `int OF method` would be widened to a real number and the whole expression would yield a value of mode `REAL`.

The two fields of the structure `double` (also declared in the previous section), can be selected by writing

```
c OF double
s OF double
```

and their modes are `CHAR` and `STRUCT(INT i,j)` respectively. Now the fields of the inner structure `s` of `double` can be selected by writing

```
i OF s OF double
j OF s OF double
```

and both selections have mode `INT`.

Now consider the structure name `sn` declared by

```
STRUCT(INT i,CHAR s) sn;
```

The mode of `sn` is

```
REF STRUCT(INT i,CHAR s)
```

This means that the mode of the selection

```
i OF sn
```

is not `INT`, but `REF INT`, and the mode of the selection

```
c OF sn
```

is `REF CHAR`. That is, the modes of the fields of a structure name are all preceded by `REF` (they are all names). This is particularly important for recursively defined structures (see chapter 11). Thus, instead of assigning a complete structure using a structure-display, you can assign values to individual fields. That is, the assignment

```
sn:=(3,"f")
```

is equivalent to the assignments

```
i OF sn := 3;
c OF sn := "f"
```

except that the assignments to the individual fields are separated by the go-on symbol (the semicolon `;`) and the two units in the structure-display are separated by a comma and so are elaborated collaterally.

Given the declaration and initial assignment

```
STRUCT(CHAR c,STRUCT(INT i,j)s)dn:=double
```

the selection

```
s OF dn
```

has the mode `REF STRUCT(INT i,j)`, and so you could assign directly to it:

```
s OF dn:=(-1,-2)
```

as well as to one of its fields:

```
j OF s OF dn:=0
```

Exercises

7.3 Given the declarations

```
STRUCT(STRUCT(CHAR a,INT b)c,
       PROC(STRUCT(CHAR a,INT b))INT p,
       INT d)st;
STRUCT(CHAR a,INT b)sta
```

what is the mode of Ans

(a) `c OF st`
(b) `a OF c OF st`
(c) `a OF sta`
(d) `(p OF st)(sta)`
(e) `b OF c OF st * b OF sta`
(f) `sta:=c OF st`

7.4 Declare a procedure which could be assigned to the selection `p OF st` in the last question. Ans

7.3 Mode declarations

Structure declarations are very common in Algol 68 programs because they are a convenient way of grouping disparate data elements, but writing out their modes every time a name needs declaring is error-prone. Using the **mode declaration**, a new mode indicant can be declared to act as an abbreviation. For example, the mode declaration

```
MODE VEC = STRUCT(REAL x,y,z)
```

makes `VEC` synonymous for the mode specification on the right-hand side of the equals symbol. Henceforth, new values using `VEC` can be declared in the ordinary way:

```
VEC vec = (1,2,3);
VEC vn := vec;
[10]VEC va;
PROC(VEC v)VEC pv=CO a routine-denotation CO;
STRUCT(VEC v,w,x) tensor
```

Here is a mode declaration for a structure which contains a reference mode:

```
MODE RV = STRUCT(CHAR c,REF[]CHAR s)
```

but we shall consider such advanced modes in chapter 11. Using a mode declaration, you might be tempted to declare a mode such as

```
MODE CIRCULAR =
      STRUCT(INT i,CIRCULAR c)  CO wrong CO
```

but this is not allowed. However, there is nothing wrong with such modes as

```
MODE NODE  = STRUCT(STRING s,
                    REF NODE next),
     PNODE = STRUCT(STRING s,
                    PROC(PNODE)STRING proc)
```

because the `NODE` inside the `STRUCT` of its declaration is hidden by the `REF`. Likewise, the `PNODE` parameter for `proc` in the declaration of `PNODE` is hidden by the `PROC`.

Suppose you want a mode which refers to another mode which hasn't been declared, and the second mode will refer back to the first mode. Both mode declarations cannot be first. In Algol 68 proper, you simply declare both modes in the usual way. However, the a68toc compiler is a single-pass compiler (it reads the source program once only) and so all applied-occurrences must occur later in the source program than the defining-occurrences. In this case, one of the modes is declared using a **stub declaration**. Here is an example:

```
MODE MODE2,
     MODE1 = STRUCT(CHAR c,REF MODE2 rb),
     MODE2 = STRUCT(INT i,REF MODE1 ra)
```

There is nothing circular about these declarations. This is another example of mutual recursion. Go ahead and experiment.

This raises the point of which modes are actually permissible. We shall deal with this in chapter 10. For now, just ensure that you don't declare modes like `CIRCULAR`, and avoid modes which can be strongly coerced into themselves, such as

```
MODE WRONG = [1:5]WRONG
```

If you inadvertently declare a disallowed mode, the compiler will declare that the mode is not legal.

Mode declarations are not confined to structures. For example, the mode `STRING` is declared in the standard prelude as

```
MODE STRING = FLEX[1:0]CHAR
```

and you can write declarations like

```
MODE FDES     = INT,
     MULTIPLE = [30]REAL,
     ROUTINE  = PROC(INT)INT,
     MATRIX   = [n,n]REAL
```

Notice that the mode declarations have been abbreviated (by omitting `MODE` each time and using commas). In the declaration of `ROUTINE`, notice that no identifier is given for the parameter of the procedure. In the last declaration, the bounds will be determined at the time of the declaration of any value using the mode `MATRIX`. Here, for example, is a small program using `MATRIX`:

```
PROGRAM tt CONTEXT VOID
USE standard
BEGIN
INT n;
```

```
MODE MATRIX = [n,n]REAL;

WHILE
   print((newline,
          "Enter an integer ",
          "followed by a blank:"));
   read(n);
   n > 0
DO
   MATRIX m;

   FOR i TO 1 UPB m
   DO
      FOR j TO 2 UPB m
      DO
         m[i,j]:=random*1000
      OD
   OD;

   FOR i TO 1 UPB m
   DO
      print((m[i,],newline))
   OD
OD
END
FINISH
```

Exercises

7.5 Declare a mode for a structure containing two fields, one of mode `REAL` and one of mode `PROC(REAL)REAL`. Ans

7.6 Declare a mode for a structure which contains three fields, the first being the mode of the previous exercise, the second a procedure which takes that mode as a parameter and yields `VOID`, and the third being of mode `CHAR`. Ans

7.7 What is wrong with these two definitions?

```
MODE AMODE = STRUCT(INT i,BMODE b),
     BMODE = STRUCT(CHAR c,AMODE a)
```

Try writing a program containing these declarations, with names of modes `AMODE` and `BMODE` and finish the program with the unit `SKIP`. Ans

7.4 Complex numbers

This section describes the mode used to perform complex arithmetic. This kind of arithmetic is useful to engineers, particularly electrical engineers. Even if you know nothing about complex numbers, you may still find this section useful.

The standard prelude contains the mode declaration

```
MODE COMPL = STRUCT(REAL re,im)
```

You can use values based on this mode to perform complex arithmetic. Here are declarations for values of modes `COMPL` and `REF COMPL` respectively:

```
COMPL z1 = (2.4,-4.6);
COMPL z2:=z1
```

Most of the operators you need to manipulate complex numbers have been declared in the standard prelude.

You can use the monadic operators `+` and `-` which have also been declared for values of mode `COMPL`.

The dyadic operator `**` has been declared for a left-operand of mode `COMPL` and a right-operand of mode `INT`. The dyadic operators `+ - * /` have been declared for all combinations of complex numbers, real numbers and integers, and so have the boolean operators `=` and `/=`. The assigning operators `TIMESAB`, `DIVAB`, `PLUSAB`, and `MINUSAB` all take a left operand of mode `REF COMPL` and a right-operand of modes `INT`, `REAL` or `COMPL`. In a strong context, a real number will be widened to a complex number. So, for example, in the following identity declaration

```
COMPL z3 = -3.4
```

`z3` will have the same value as if it had been declared by

```
COMPL z3 = (-3.4,0)
```

This is the only case where a real number can be widened into a structure.

The dyadic operator `I` takes left- and right-operands of any combination of `REAL` and `INT` and yields a complex number. It has a priority of 9. For example, in a formula, the context of operands is firm and so widening is not allowed. Nevertheless, the yield of this formula is `COMPL`:

```
2 * 3 I 4
```

Some operators act only on complex numbers. The monadic operator `RE` takes a `COMPL` operand and yields its `re` field with mode `REAL`. Likewise, the monadic operator `IM` takes an operand of mode `COMPL` and yields its `im` field with mode `REAL`. For example, given the declaration above of `z3`, `RE z3` would yield `-3.4`, and `IM z3` would yield `0.0`. Given the complex number `z` declared as

```
COMPL z = 2 I 3
```

then `CONJ z` would yield `RE z I - IM z` or `(2.0,-3.0)`. The operator `ARG` gives the argument of its operand: `ARG z` would yield 0.982 793 723 2, lying in the interval $(-\pi, \pi]$. The monadic operator `ABS` with a complex number may be defined as

```
OP ABS = (COMPL z)REAL:
   sqrt(RE z**2 + IM z**2)
```

Remember that in the formula `RE z**2`, the operator `RE` is monadic and so is elaborated first.

As described in the previous section, the mode `COMPL` can be used wherever a mode is required. In particular, procedures taking `COMPL` parameters and yielding `COMPL` values can be declared. Structures containing `COMPL` can be declared as above.

From the section on field selection, it is clear that in the declarations

```
COMPL z = (2.0,3.0);
COMPL w:=z
```

the selection

```
re OF z
```

has mode `REAL` (and value `2.0`), while the selection

```
re OF w
```

has mode `REF REAL` (and its value is a name). However, the formula

```
RE w
```

still yields a value of mode `REAL` because `RE` is an operator whose single operand has mode `COMPL`. In the above phrase, the `w` will be dereferenced before `RE` is elaborated. Thus it is quite legal to write

```
im OF w:=RE w
```

or

```
im OF w:=re OF w
```

in which case the right-hand side of the assignment will be dereferenced before a copy is assigned.

Exercises

7.8 If the complex number `za` has the mode `COMPL` and the value yielded by `(2,-3)`, what is the value of Ans

(a) `CONJ za`
(b) `IM za * RE za * RE za`
(c) `ABS za`
(d) `ARG za`

7.9 What is the value of the formula `23 - 11 I -10`? Ans

7.10 Given the declarations

```
COMPL a = 2 I 3;
COMPL b:= CONJ a
```

what is the mode and value of each of the following: Ans

(a) `im OF b`
(b) `IM b`
(c) `im OF a`
(d) `IM a`

7.5 Multiples in structures

If multiples are required in a structure, the structure declaration should only contain the required bounds if it is an actual-declarer. For example, we could declare

```
STRUCT([]CHAR forename,
              surname,
              title)
   lecturer=
      ("Nerissa","Leitch","Dr")
```

where the mode on the left is a formal-declarer (remember that the mode on the left-hand side of an identity declaration is always a formal-declarer). We could equally well declare

```
STRUCT([]CHAR forename,
              surname,
              title)
   student=
         ("Tom","MacAllister","Mr")
```

As you can see, the bounds of the individual multiples differ in the two cases.

When declaring a name, because the mode preceding the name identifier is an actual-declarer (in an abbreviated declaration), the bounds of the required multiples must be included. A suitable declaration for a name which could refer to `lecturer` would be

```
STRUCT([7]CHAR forename,
       [6]CHAR surname,
       [3]CHAR title)
         new lecturer := lecturer
```

but this would not be able to refer to `student`. A better declaration would use `STRING`:

```
STRUCT(STRING forename,surname,title)person
```

in which case we could now write

```
person:=lecturer;
person:=student
```

Using field selection, we can write

```
title OF person
```

which would have mode `REF STRING`. Thus, using field selection, we can assign to the individual fields of `person`:

```
surname OF person:="McRae"
```

When slicing a field which is a multiple, it is necessary to remember that slicing binds more tightly than selecting (see chapter 10 for a detailed explanation). Thus the first character of the surname of `student` would be accessed by writing

```
(surname OF student)[1]
```

which would have mode `CHAR`. The parentheses ensure that the selection is elaborated before the slicing. Similarly, the first five characters of the forename of `person` would be accessed as

```
(forename OF person)[:5]
```

with mode `REF[]CHAR`.

Consider the following program:

```
PROGRAM t1 CONTEXT VOID
BEGIN
   MODE AMODE = STRUCT([4]CHAR a,INT b);
   AMODE a = ("abcde",3);
   AMODE b:=a;
   SKIP
END
FINISH
```

In the identity declaration for `a`, the mode required is a formal-declarer. In this case, the a68toc compiler will ignore the bounds in the declaration of `AMODE` giving the mode

```
STRUCT([]CHAR a,INT b)
```

which explains why the structure-display on the right is accepted (`"abcde"` has bounds `[1:5]`). However, although the program compiles without errors, when it is run, it fails with the error message

```
Run time fault (aborting):
ASSIGN2: bounds do not match in [] assignment
```

because the mode used in the declaration of the name `b` is an actual-declarer (it contains the bounds given in the mode declaration) and you cannot assign a `[]CHAR` with bounds `[1:5]` to a `REF[]CHAR` with bounds `[1:4]`.

With more complicated structures, it is better to use a mode declaration. For example, we could declare

```
MODE EMPLOYEE =
   STRUCT(STRING name,
          [2]STRING address,
          STRING dept,ni code,tax code,
          REAL basic,overtime rate,
          [52]REAL net pay,tax);

EMPLOYEE emp
```

and then read specific values from the keyboard (chapter 9 covers reading data from files):

```
read((name OF emp,newline,
      (address OF emp)[1],newline,
      (address OF emp)[2],newline,
...
```

The modes of

```
name OF emp
address OF emp
net pay OF emp
```

are

```
REF STRING
REF[]STRING
REF[]REAL
```

respectively. The phrase

```
(net pay OF emp)[:10]
```

has the mode `REF[]REAL` with bounds `[1:10]` and represents the net pay of `emp` for the first 10 weeks. Note that although the monetary values are held as `REAL` values, the accuracy with which a `REAL` number is stored is such that no rounding errors will ensue. See section 12.1.5 which describes which modes are suitable for storing monetary values.

Exercises

7.11 Given the declaration of `emp` in the text, what would be the mode of each of the following: Ans

(a) `address OF emp`
(b) `basic OF emp`
(c) `(tax OF emp)[12]`
(d) `(net pay OF emp)[10:12]`

7.12 What are the bounds of the name in (d) above? Ans

7.6 Rows of structures

In the last section, we considered multiples in structures. What happens if we have a multiple each of whose elements is a structure? No problem. If we had declared

```
[10]COMPL z4
```

then the selection `re OF z4` would yield a name with mode `REF[]REAL` and bounds `[1:10]`.[1] It would be possible, because it is a name, to assign to it:

```
re OF z4:=(1,2,3,4,5,6,7,8,9,10)
```

Selecting the field of a sliced multiple of a structure is straightforward. Since the multiple is sliced before the field is selected, no parentheses are necessary. Thus the real part of the third `COMPL` of `z4` above is given by the expression

```
re OF z4[3]
```

Now consider a multiple of a structure which contains a multiple. Here is its declaration:

```
[100]STRUCT(CHAR c,[5]INT i)s
```

Then the fourth integer in the 25th structure of `s` is given by

```
(i OF s[25])[4]
```

and all the characters are given by the selection

```
c OF s
```

with mode `REF[]CHAR` and bounds `[1:100]`.[2]

[1] Unfortunately, there is a limitation in the a68toc compiler whereby this selection (and similar selections) are disallowed.

[2] But this is not supported by the a68toc compiler.

Exercises

7.13 Suppose a firm had 20 employees, and in writing one of the programs in their payroll system, the modes of section 7.5 were used. Suppose now that we have the declaration

```
[20]EMPLOYEE employee;
```

What would be the mode of each of the following: Ans

(a) `(dept OF employee[3])[3]`
(b) `dept OF employee[10:12]`
(c) `ni code OF employee[1]`
(d) `net pay OF employee[15]`
(e) `(tax OF employee[2])[50:51]`

7.7 Transput of structures

The following program fragment will print the details of the name `emp` declared in section 7.5:

```
print((emp,newline))
```

For details of how this works, see the remarks on "straightening" in section 9.2. However, the individual strings would be printed together and so, in this case, it would be better to write the following:

```
print((name OF emp,newline));
FOR i TO UPB address OF emp
DO
   print((address OF emp)[i],newline))
OD;
print((dept OF emp,newline,
       ni code OF emp,newline,
       tax code OF emp,
       basic OF emp,
       overtime OF emp,
       net pay OF emp,
       tax OF emp,newline))
```

In practice, it would be sensible to declare a procedure or an operator which would print the structure and then call it as required.

7.8 Summary

This chapter has significantly increased the number of different modes we can construct. Structures are constructed using the mode constructor `STRUCT`. Complicated structures are best declared using the mode declaration (using `MODE`). Structures can have any number of fields from one up, and the fields can have any mode, including the same modes. The mode `COMPL` has been declared in the standard prelude together with the necessary operators to manipulate complex numbers.

Structures can contain procedures and multiples and multiples of structures can be declared. Although structures containing reference modes can be declared, they are covered in chapter 11. Operators and procedures which have structure parameters or yield can be declared.

Here are some exercises to check what you have learned.

Exercises

7.14 Write a suitable mode for a football team which contains the names of its 11 members, the name of the team (ordinary name, not the Algol 68 meaning), the number of games played, won and drawn, and the number of goals scored for and against. Ans

7.15 Given the declaration

```
STRUCT(INT i,[3]REAL r)s
```

explain why parentheses are needed in the phrase

```
(r OF s)[2]
```

Ans

7.16 Given the declaration

```
[3]STRUCT(INT i,REAL r)s
```

explain why parentheses are not needed in the phrase `r OF s[2]`. Ans

7.17 Given the declarations

```
MODE S2,
     S1 = STRUCT([3]CHAR n,
                 PROC S2 p),
     S2 = STRUCT([3]CHAR m,
                 PROC(S1)S2 p);
S1 s1;  S2 s2;
```

what are the modes of each of the following: Ans

(a) `p OF s1`
(b) `p OF s2`
(c) `(n OF s1)[2:]`

Chapter 8

Unions

From time to time, you have been using the procedure `print` to display values on your screen. You must have noticed that it seems to be able to take a large variety of values of different modes and that it can process more than one value in one call. You may therefore be wondering how the parameter of `print` is specified. It cannot be a structure because a structure has a fixed number of fields, but if it is a row, how can a row have different modes for its elements? Although the elements of a row must each have the same mode, the explanation is that `print` takes one parameter which is a row of a **united** mode.

This very short chapter introduces the final mode constructor available in Algol 68. It shows the principles behind the construction and use of united modes. It does not and cannot show all the possible usages.

8.1 United mode declarations

`UNION` is used to create a united mode. Here is a declaration for a simple united mode:

```
UNION(INT,STRING) u = (random < .5|4|"abc")
```

The first thing to notice is that, unlike structures, there are no field selectors. This is because a united mode does not consist of constituent parts. The second thing to notice is that the above mode could well have been written

```
UNION(STRING,INT) u = (random < .5|4|"abc")
```

The order of the modes in the union is irrelevant.[1] What is important is the actual modes present in the union. Moreover, a constituent mode can be repeated, as in

```
UNION(STRING,INT,STRING,INT) u =
   (random < .5|4|"abc")
```

This is equivalent to the previous declarations.[2]

[1] However, for the a68toc compiler, this is not the case. Unions with the same modes in a different order are distint united modes

[2] They are distinct for the a68toc compiler.

Like structured modes, united modes are often declared with the mode declaration. Here is a suitable declaration of a united mode containing the constituent modes `STRING` and `INT`:

```
MODE STRINT = UNION(STRING,INT)
```

We could create another mode `STRINTR` in two ways:

```
MODE STRINTR = UNION(STRINT,REAL)
```

or

```
MODE STRINTR = UNION(STRING,INT,REAL)
```

Using the above declaration for `STRINT`, we could declare `u` by

```
STRINT u = (random < .5|4|"abc")
```

In this identity declaration, the mode yielded by the right-hand side is either `INT` or `STRING`, but the mode required is `UNION(STRING,INT)`. The value on the right-hand side is coerced to the required mode by **uniting**.

The united mode `STRINT` is a mode whose values either have mode `INT` or mode `STRING`. It was stated in chapter 1 that the number of values in the set of values defined by a mode can be zero. Any value of a united mode actually has a mode which is one of the constituent modes of the union. So there are no **new** values for a united mode. `u` identifies a value which is either an `INT` or a `STRING`. Because `random` yields a pseudo-random number, it is not possible to determine when the program is compiled (that is, at **compile-time**) which mode the conditional clause yields. As a result, all we can say is that the underlying mode of `u` is either `INT` or `STRING`. We shall see later how to determine that underlying mode.[3]

Because a united mode does not introduce new values, there are no denotations for united modes, although denotations may well exist for the constituent modes.

Almost any mode can be a constituent of a united mode. For example, here is a united mode containing a procedure mode and `VOID`:

```
MODE PROID = UNION(PROC(REAL)REAL,VOID)
```

and here is a declaration using it:

```
PROID pd = EMPTY
```

The only limitation on united modes is that none of the constituent modes may be firmly related (see the section 6.2.1) and a united mode cannot appear in its own declaration. The following declaration is wrong because a value of one of the constituent modes can be deprocedured in a firm context to yield a value of the united mode:

```
MODE WRONG = UNION(PROC WRONG,INT)
```

Names for values with united modes are declared in exactly the same way as before. Here is a declaration for such a name using a local generator:

[3]Note that an Algol 68 union is quite different from a C union. The latter is simply a remapping of a piece of memory. In an Algol 68 union, where the underlying value is kept is the business of the compiler and it cannot be remapped by the programmer.

```
REF UNION(BOOL,INT) un = LOC UNION(BOOL,INT);
```

The abbreviated declaration gives

```
UNION(BOOL,INT) un;
```

Likewise, we could declare a name for the mode `STRINT`:

```
STRINT sn;
```

In other words, objects of united modes can be declared in the same way as other objects.

Exercises

8.1 Write a mode declaration for the united mode `BINT` whose constituent modes are `BOOL` and `INT`. Ans

8.2 Write an identity declaration for a value of mode `BINT`. Ans

8.3 What is wrong with the mode declaration

```
MODE UB = UNION(REF UB,INT,BOOL)
```

Ans

8.4 Declare a name for a mode united from `INT`, `[]INT` and `[,]INT`. Ans

---

8.2 United modes in procedures

We can now partly address the problem of the parameters for `print` and `read`. If we extend the answer to the last exercise, we should be able to construct a united mode which will accept all the modes accepted by those two procedures. In fact, the united modes used are almost the same as the two following declarations:

```
MODE SIMPLOUT = UNION(CHAR, []CHAR,
                      INT,  []INT,
                      REAL, []REAL,
                      COMPL,[]COMPL,
                      BOOL, []BOOL,
                     ),

     SIMPLIN  = UNION(REF CHAR, REF[]CHAR,
                      REF INT,  REF[]INT,
                      REF REAL, REF[]REAL,
                      REF COMPL,REF[]COMPL,
                      REF BOOL, REF[]BOOL,
                     );
```

As you can see, the mode `SIMPLIN` used for `read` is united from modes of names.

The modes `SIMPLOUT` and `SIMPLIN` are a little more complicated than this because they include modes we have not yet met (see chapters 9 and 11), but you now have the basic idea.

The uniting coercion is available in a firm context. This means that operators which accept operands with united modes will also accept operands whose modes are any of the constituent modes. We shall return to this in the next section.

Here is an example of the uniting coercion in a call of the procedure `print`. If `a` has mode `REF INT`, `b` has mode `[]CHAR` and `c` has mode `PROC REAL`, then the call

```
print((a,b,c))
```

causes the following to happen:

1. `a` is dereferenced to mode `INT` and then united to mode `SIMPLOUT`.
2. `b` is united to mode `SIMPLOUT`.
3. `c` is deprocedured to produce a value of mode `REAL` and then united to mode `SIMPLOUT`.
4. The three elements are regarded as a row-display for a `[]SIMPLOUT`.
5. `print` is called with its single parameter.

`print` uses a **conformity clause** (see next section) to extract the actual value from each element in the row.

In section 6.3.2, we gave the declaration of a procedure identified as `char in string`. The header of that procedure was

```
PROC char in string=
   (CHAR ch,REF INT pos,[]CHAR s)BOOL:
```

The procedure yielded `TRUE` if `ch` was present in `s`, in which case `pos` contained the position. Otherwise, the procedure yielded `FALSE`. The same procedure could be written to yield the position of `ch` in `s` if it is present, and `VOID` if not:

```
PROC ucis = (CHAR ch,[]CHAR s)
                  UNION(INT,VOID):
```

The body of the procedure has been left as an exercise.

Exercises

8.5 A procedure has the header

```
PROC pu = ([]UNION(CHAR,[]CHAR) up)VOID:
```

Explain what happens to the parameters if it is called by the phrase

```
pu((CHAR: REPR(ABS"a"+1),LOC[4]CHAR))
```

Ans

8.6 Write the body of the procedure `ucis` given in the text. Ans

---

8.3 Conformity clauses

In the last section, we discussed the consequences of the uniting coercion; that is, how values of various modes can be united to values of united modes. This raises the question of how a value of a united mode can be extracted since its constituent mode cannot be determined at compile-time. There is no de-uniting coercion in Algol 68. The constituent mode or the value, or both, can be determined using a variant of the case clause discussed in chapter 4 (see section 4.6). It is called a **conformity clause**. For our discussion, we shall use the declaration of `u` in section 8.1 (`u` has mode `STRINT`).

The constituent mode of `u` can be determined by the following:

```
CASE u IN
   (INT):    print("u is an integer")
   ,
   (STRING): print("u is a string")
ESAC
```

If the constituent mode of `u` is `INT`, the first case will be selected. Notice that the mode selector is enclosed in parentheses and followed by a colon. Otherwise, the conformity clause is just like the case clause (in fact, it is sometimes called a conformity case clause). This particular example could also have been written

```
CASE u
IN
   (STRING): print("u is a string")
OUT
   print("u is an integer")
ESAC
```

This is the only circumstance when a `CASE` clause can have one choice. Usually, however, we want to extract the value. A slight modification is required:

```
CASE u IN
   (INT i):
      print(("u identifies the value",i))
   ,
   (STRING s):
      print(("u identifies the value ",s))
ESAC
```

In this example, the mode selector and identifier act as the left-hand side of an identity declaration. The identifier can be used in the following unit (or enclosed clause).

The two kinds of conformity clause can be mixed. For example, here is one way of using the procedure `ucis`:

```
CASE ucis(c,s) IN
   (VOID):
      print("The character was not found"),
   (INT p):
      print(("The position was",p))
ESAC
```

We mentioned in the last section that operators with united-mode operands can be declared. Here is one such:

```
MODE IC = UNION(INT,CHAR);
OP * = (IC a,b)IC:
CASE a IN
   (INT ai):
      (b|(INT bi): ai*bi,
         (CHAR bc): ai*bc),
   (CHAR ac):
      (b|(INT bi): ac*bi,
         (CHAR bc): ABS ac*ABS bc)
ESAC
```

In each of the four cases, the resulting product is united to the mode `IC`.

You can have more than one mode in a particular case. For example:

```
MODE ICS = UNION(INT,CHAR,STRING);
OP * = (ICS a,INT b)ICS:
CASE a
IN
   (UNION(STRING,CHAR) ic):
      (ic|(CHAR c): c*b,(STRING s): s*b),
   (INT n): n*b
ESAC
```

Note that conformity clauses do not usually have an `OUT` clause because the only way of extracting a value is by using the `(MODE id):` construction. However, they do have their uses. See the standard prelude for more examples of conformity clauses.

Although `read` and `print` use united modes in their call, you cannot read a value of a united mode or print a value of a united mode (remember that united modes do not introduce new values). You have to read a value of a constituent mode and then unite it, or extract the value of a constituent mode and print it.

Exercises

8.7 The modes

```
MODE IRC = UNION(INT,REAL,COMPL),
     MIRC= UNION([]INT,[]REAL,[]COMPL)
```

are used in this and the following exercises.

Write a procedure which takes a single parameter of mode `MIRC` and which yields the sum of all the elements of its parameter as a value with mode `IRC`. Ans

8.8 Write the body of the operator `*` whose header is declared as

```
OP * = (IRC a,b)IRC:
```

Use nested conformity clauses. Remember that there are 9 separate cases. Ans

---

8.4 Summary

United modes introduce no new values. A united mode can have any mode as one of its constituents except a mode which can be firmly coerced to itself. The uniting coercion is available in firm contexts. Because the values supplied to `print` or `read` are united, the context of the parameter of those procedures is firm. A conformity clause is used to extract the constituent mode or value. The mode `VOID` can be one of the constituents of a united mode and is useful to signal an exceptional yield from a procedure. United modes are used in a variety of ways.

Exercises

8.9 Write a declaration for the united mode `CRIB` whose constituent modes are `CHAR`, `REAL`, `INT` and `BOOL`. Ans

8.10 Write a declaration for the operator `UABS` which has a single operand of mode `CRIB` and which yields the absolute value of its operand. Ans

8.11 Write four formulæ which use `UABS` and a denotation for each of the constituent modes of `CRIB`. Ans

---

Chapter 9

Transput

At various points you have been reading external values from the keyboard and displaying internal values on the screen. This chapter addresses the means whereby an Algol 68 program can obtain external values from other sources and send internal values to places other than the screen. **straightening** is the only new language construct involved and all the matters discussed are available in the standard prelude.

Algol 68 transput gives the first taste of "event-driven programming". In effect, all programs are event-driven, but simple programs are driven only by the originating event: that is, the initiation of the program. In other words, simple programs, once started, run to completion, unless, of course, they contain errors. Event-driven programs, however, are dependent on the occurrence of events which are outwith the control of the program. We shall be examining later the kinds of event which can affect your programs if they read or write data.

9.1 Books, channels and files

In Algol 68 terms, external values are held in a **book**. Books have various properties. They usually have an identification string. Some books can be read, some written to and some permit both reading and writing. Some books allow you to browse: that is, they allow you to start anywhere and read (or write) from that point on. If browsing is allowed, you can restart at the beginning. Some books allow you to store external values in text form (human-readable form) only, while others allow you to store values in a compact internal form known as **binary**. In the latter form, values are stored more or less in the same form as they are held in the program. The values will not usually be human-readable, being more suited to fast access by computer programs.

In operating-system terms, Algol 68 books are called "files" (just to confuse you, of course), but a book has a wider meaning than an operating-system file.[1] When reading external values from the keyboard, your program is reading data from a read-only book. When printing data, your program is writing data to a write-only book. When accessing a device, such as `/dev/ttyS2`, to which you can attach a modem, your program can both read from and write to the book, but it cannot

[1] In Linux, a file has the mode, more-or-less, `REF BOOK`.

browse in it.

The **data** (as external values are called) in a book, or the data to be put in a book, travels between the book and your program via a **channel**. Three principal channels are provided in the standard prelude: `stand in channel`, `stand out channel` and `stand back channel`. The first is used for books which can only be read (they are "read-only"), the second for books which can only be written to (they are "write-only") and the last for books which permit both reading and writing. This classification is a little over-simplified. Many books permit both reading and writing, but you may only want your program to read it. The three standard channels mentioned are all "buffered". This means that when you, for example, write data to a book, the data is collected in memory until a fixed amount has been transput, when the collection is written to the book in its entirety. The standard channels use a buffer of 4096 bytes. The mode of a channel is `CHANNEL` and is declared in the standard prelude.

Your program keeps track of where you are in a book, which book is being accessed and whether you have come to the end of the book by means of a special structure which has the mode `FILE`. This is a complicated structure declared in the standard prelude. The internals of values of mode `FILE` are likely to change from time to time, but the methods of using them will remain the same.

9.2 Reading books

Before you can read the contents of an existing book, you need to connect the book to your program. The procedure `open` with the header

```
PROC open = (REF FILE f,
             STRING idf,
             CHANNEL chan)INT:
```

performs that function. `open` yields zero if the connection is established and non-zero otherwise. Here is a program fragment which establishes communication with a read-only book whose identification is `testdata`:

```
FILE inf;

IF open(inf,"testdata",stand in channel)/=0
THEN
   print(("Cannot open book testdata",
          newline));
   exit(1)
FI
```

Notice that the program displays a short message on the screen if for any reason the book cannot be opened and then terminates with a suitable error number. The procedure `exit` is not standard Algol 68, but is provided by a system routine whose declaration is in the standard prelude issued with the a68toc compiler.

After a book has been opened, data can be read from the book using the procedure `get` which transforms external values into internal values like `read` (you will meet `read` again shortly). It has the header

```
PROC get=(REF FILE f,[]SIMPLIN items)VOID:
```

The mode SIMPLIN is declared in the standard prelude as

```
MODE SIMPLIN=
   UNION(
    REF CHAR, REF[]CHAR,  REF STRING,
    REF BOOL, REF[]BOOL,

    REF LONG BITS,        REF[]LONG BITS,
    REF BITS,             REF[]BITS,
    REF SHORT BITS,       REF[]SHORT BITS,
    REF SHORT SHORT BITS,
                     REF[]SHORT SHORT BITS,

    REF LONG INT,         REF[]LONG INT,
    REF INT,              REF[]INT,
    REF SHORT INT,        REF[]SHORT INT,
    REF SHORT SHORT INT,
                     REF[]SHORT SHORT INT,

    REF REAL,             REF[]REAL,
    REF SHORT REAL,       REF[]SHORT REAL,

    REF COMPL,            REF[]COMPL,
    REF SHORT COMPL,      REF[]SHORT COMPL,

    STRAIGHT SIMPLIN
   ),
```

The mode BITS is covered in chapter 11 together with LONG and SHORT modes. As you can see, all the constituent modes of the union are the modes of names, except for the STRAIGHT SIMPLIN and the PROC(REF FILE)VOID. The PROC mode lets you use routines like newpage and newline as one of the parameters. The actual header of newline is

```
PROC newline = (REF FILE f)VOID:
```

and you can call it outwith get if you want. On input, the rest of the current line is skipped and a new line started. The position in the book is at the start of the new line, just before the first character of that line. Here is a program fragment which opens a book and then reads the first line and makes a name of mode REF STRING to refer to it. After reading the string, newline is called explicitly:

```
FILE inf;
open(inf,"book",stand in channel);
STRING line;  get(inf,line);  newline(inf)
```

This could equally well have been written

```
FILE inf;
open(inf,"book",stand in channel);
STRING line;  get(inf,(line,newline))
```

There is no reason why you should not declare your own procedures with mode `PROC(REF FILE)VOID`. Here is an example:

```
PROC nl3 = (REF FILE f)VOID:
   TO 3 DO newline(f) OD;
```

This procedure could then be used in `get`, for example:

```
STRING line1, line2;
get(inf,(line1,nl3,line2))
```

where `line2` would be separated by 2 skipped lines from `line1`.

The `STRAIGHT` operator converts any structure or multiple into a row of values of the constituent fields or elements. This means that `get` can read directly any structure or multiple (or even rows of structures or multiples).

There are four names of mode `REF FILE` declared in the standard prelude. One of these is identified by `stand in`. The procedure `read` which you have already used is declared as

```
PROC read=([]SIMPLIN items)VOID:
   get(stand in,items)
```

in the standard prelude. As you can see, it gets data from `stand in`. If you want to, you can use `get` with `stand in` instead of `read`. The file `stand in` is already open when your program starts and should not be closed[2]. Note that input from `stand in` is unbuffered, that is, it does not use the channel `stand in channel`.

When you have finished reading data from a book, you should sever the connection between the book and your program by calling the procedure `close`. This closes the book. Its header is

```
PROC close=(REF FILE f)VOID:
```

Exercises

9.1 Write a program called `list` which will read lines from a text book until a zero length line is read. The program should display each line on the screen on separate lines. Ans

9.2 Write a program which will read a positive integer from a text book and which will then read that number of numbers (integer or real) from the book and display their total. Ans

[2] Unless you know what you are doing!

9.3 Writing to books

You should use the `establish` procedure to create a new book. Here is a program fragment which creates a new book called `results`:

```
FILE outf;

IF establish(outf,
             "results",
             stand out channel,
             0,0,0)/=0
THEN
   print(("Cannot establish book results",
          newline));
   exit(1)
FI
```

As you can see, `establish` has a similar header to `open`. What are the integers used for? The header for `establish` is

```
PROC establish = (REF FILE f,
                  STRING idf,
                  CHANNEL chann,
                  INT p,l,c)INT:
```

The `p`, `l` and `c` in `establish` determine the maximum number of pages, lines and characters in the book which is being created. Values of `0` for all three integers mean that the file should be established with zero length. However, they are ignored by the `stand out channel` in the QAD standard prelude provided with the a68toc compiler.

The procedure used to write data to a book is `put`. Its header is

```
PROC put=(REF FILE f,[]SIMPLOUT items)VOID:
```

You can examine the source of the standard prelude to see how the mode `SIMPLOUT` is declared.

Again, `newline` and `newpage` can be used independently of `put` as in the following fragment:

```
FILE outf;
IF establish(outf,
             "newbook",
             stand out channel,
             0,0,0)/=0
THEN
   put(stand error,
       ("Cannot establish newbook",
        newline));
   exit(2)
ELSE
   put(outf,("Data for newbook",newline));
   FOR i TO 1000 DO put(outf,i) OD;
```

```
    newline(outf);
    close(outf)
FI
```

On output, the newline character is written to the book.

`newpage` behaves just like `newline` except that a form feed character is searched for on input, and written on output.

The procedure `establish` can fail if the disk you are writing to is full or you do not have write access (in a network, for example) in which case it will return a non-zero value.

When you have completed sending data to a book, you must close it with the `close` procedure. This is particularly important with books you write to because the channel is buffered as explained above. Using `close` ensures that any remaining data in the buffer is flushed to the book.

The procedure `print` uses the `REF FILE` name `stand out`. So

```
print(("Your name",newline))
```

is equivalent to

```
put(stand out,("Your name",newline))
```

Again, `stand out` is open when your program is started and it should not be closed. Transput via `stand out` is unbuffered. You cannot read from `stand out`, nor write to `stand in`. The procedure `write` is synonymous with `print`.

Exercises

9.3 Change the second program in the last set of exercises to put its total into a newly-created book whose identification is `result`. Ans

9.4 Adapt Eratosthenes' Sieve (see section 5.4.1) to output all the prime numbers less than 10,000 into a book called `primes`. Ans

---

9.4 String terminators

One of the really useful facilities available for reading data from books is that of being able to specify when the reading of a string should terminate. Usually, this is set as the end of the line only. However, using the procedure `make term`, the string terminator can be a single character or any one of a set of characters. The header of `make term` is

```
PROC make term=(REF FILE f,STRING term)VOID:
```

so if you want to read a line word by word, defining a word as any sequence of non-space characters, you can make the string terminator a space by writing

```
make term(inf,blank)
```

because `blank` (synonymous with `"␣"`) is rowed in the strong context of a parameter to `[]CHAR`. This will not remove the end-of-line as a terminator because the character `lf` is always added whatever characters you specify. You should remember that when a string is read, the string terminator is available for the next read—it has *not* been read by the previous read (but see 9.9).

Exercises

9.5 Write a program called `copy` which copies its input text book to its output text book, stopping when a blank line is read (all blanks or zero length). The input book is called `inbook` and the output book `outbook`.Ans

9.6 Rewrite the program from exercises in section 5.11 using `make term`. The data should be read from a book called `lines` and written to a book called `words`. Write one word to a line. Terminate the `lines` with an asterisk (`*`) on a line by itself. Ans

9.5 Events

Algol 68 transput is characterised by its use of events. In the limited transput supplied with the a68toc compiler, only four kinds of events are detected. These are:

1. The end of the file when reading. This is called the "logical file end".
2. The end of the file when writing. This is called the "physical file end".
3. A value error.
4. A character error.

The default action when an event occurs depends on the event. However, the default action can be replaced by a programmer-defined action using one of the "on"-procedures.

9.5.1 Logical file end

When the logical end of a file has been detected, the default action is to terminate the program immediately. All open files will be closed by the operating system, but any buffered output files will lose any data in the buffer. A programmer-supplied action must be a procedure with the header

```
(REF FILE f)BOOL:
```

The yield should be `TRUE` if some action has been taken to remedy the end of the file, in which case transput is re-attempted, or `FALSE`, when the default action will be taken.

The procedure `on logical file end` has the header

```
PROC on logical file end=
   (REF FILE f,
    PROC(REF FILE)BOOL p)VOID:
```

and an example will help explain its use. Here is a program which will display the contents of its text input file and print a message at its end.

```
PROGRAM readfile CONTEXT VOID
USE standard
IF FILE inf; []CHAR infn="textbook";
   open(inf,infn,stand in channel)/=0
THEN
   put(stand error,
       ("Cannot open ",
        infn,newline));
   exit(1)
ELSE
   on logical file end(inf,
      (REF FILE f)BOOL:
         (write(("End of ",
                 idf(f),
                 " read",newline));
          close(f);  FALSE));
   STRING line;
   DO
      get(inf,(line,newline));
      write((line,newline))
   OD
FI
FINISH
```

The anonymous procedure provided as the second parameter to `on logical file end` prints an informative message and closes the book before yielding `FALSE`. Since in this case all we want is for the program to end when the input has been read, the default action is fine. Notice also that the `DO` loop simply repeats the reading of a line until the `logical file end` procedure is called. The procedure `idf` is described in section 9.11.

You should be careful if you do any transput on the parameter `f` in the anonymous routine otherwise you could get an infinite loop (a loop which never ends). Also, because the `on logical file end` procedure assigns its procedure parameter to its `REF FILE` parameter, you should be wary of using `on logical file end` in limited ranges. The way out of this problem is to make a local copy of the `REF FILE` parameter as in:

```
BEGIN
   FILE loc f:=stand in;
   on logical file end(
        f,(REF FILE f)BOOL: ...);
     ...
END
```

Any piece of program which will yield an object of mode `PROC(REF FILE)BOOL` in a strong context is suitable as the second parameter of `on logical file end`.

If you want to reset the action to the default action, use the phrase

```
on logical file end(f,no file end)
```

9.5.2 Physical file end

The physical end of a file is met on writing if, for example, the disk is full. It can also occur when using the `mem channel` (see section 9.10). The default action closes all open files (but the buffers of buffered files will not be flushed to disk) and terminates the program with an exit value of 255.

A replacement procedure should have the mode

```
PROC(REF FILE)BOOL
```

and it should yield `TRUE` if the event has been remedied in some way, in which case transput will be re-attempted, and `FALSE` otherwise (when the default action will be elaborated).

The default procedure can be replaced with a procedure defined by the programmer using the procedure `on physical file end` which has the header:

```
PROC on physical file end =
   (REF FILE f,
    PROC(REF FILE)BOOL p)VOID:
```

9.5.3 Value error

This event is caused by the following circumstances:-

1. If an integer is expected, then the value read exceeds `max int`.
2. If a real number is expected, then the value read exceeds `max real`.
3. If a complex number is expected, then the value read for either the real part or the imaginary part exceeds `max real`.

The procedure `on value error` lets the programmer provide a programmer-defined procedure whose header must be

```
(REF FILE f)BOOL:
```

although any identifier could replace the `f`. Transput on the file `f` within the procedure should be avoided (but see `backspace` below), but any other transput is allowable, but try to ensure that a value error won't occur!

If the programmer-supplied routine yields `TRUE`, transput continues, otherwise an error message is issued to `stand err` and the program aborted with an exit value of 247.

9.5.4 Char error

This event can occur when reading a number if the number is entirely absent so that the first character is neither a sign nor a digit. In this case a default procedure is called having the header

```
(REF FILE f,REF CHAR c)BOOL:
```

The default procedure can be replaced with a programmer-defned procedure using the procedure `on char error`.

The char error procedure is called with the `c` referring to a suggested value for the next character. The replacement character must be a member of a particular set of characters. The default value is `0`. If the procedure returns `FALSE` the default suggested character will be used, otherwise the value referred to by `c` will be used. Thus the programmer-supplied procedure can not only change the default suggested character, but can also perform such other actions as are deemed necessary (such as logging the error).

The event can also occur when reading the digits before a possible `"."` for real numbers and the digits after the `"."`. For complex numbers, after the real part, an `i` or `I` is expected and its non-appearance will cause the `char error` procedure to be executed. The default suggestion is `i`, but can be replaced by another character and optional actions.

For a `BITS`[3] value, whenever a character which is neither `flip` nor `flop` is met, the `char error` procedure is called with `flop` as the suggested value. Thus the available suggested character sets are:-

1. For digits: 0 . . . 9
2. For exponent: `e E` \
3. For plus i times: `i I`
4. For flip or flop: `FT` (uppercase only) respectively

Exercises

9.7 Write a program whose input book has the identification `inbook` and which contains lines of differing length. Use `on logical file end` to specify a procedure which establishes the output book `outbook`, writes the average length and closes it and then yields `FALSE`. Ans

---

[3]This mode is described in section 11.2

9.6 The command line

When you execute a program at the command prompt, you type the identification of the program and then press return. You can specify parameters (sometimes called **arguments**) for the program after the program identification. These can then be accessed by the program to modify its activities.

Hitherto, the identifications of books have always been written into the actual code. In the last exercise, the input book was called `inbook` and the output book `outbook`. If your program could be given the identifications of the books whenever you executed the program, then it could have a much wider applicability.

The command line is available to the program via the channel `arg channel`. Here is a small program which reads its first argument and prints it on the screen:

```
PROGRAM arg1 CONTEXT VOID
USE standard
IF FILE args;  open(args,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access the command line",
        newline));
   stop
ELSE
  on logical file end(arg,
     (REF FILE f)BOOL:
        (put(stand error,
             ("No parameters",newline));
         FALSE));
  STRING id;
  get(arg,id);  write((id,newline))
FI
FINISH
```

Some points to note:

1. `stand error` is an output `FILE` which is usually used for error messages.
2. The identification field in the call to `open` is ignored by `arg channel`. In the example, it is written as the empty string.
3. `stop` is equivalent to `exit(0)`.
4. In Linux, the first parameter is always the full path of the identification of the program.

You can only read via the `arg channel` (using `get`). `make term` has already been set to make the string terminator `blank` (the last argument is always followed by a space) so you can read the individual parameters from the command line by reading strings. However, you should note that when you have read a string, the next character will be the terminator of the string. So when you have read a string, you will need to skip all characters which could possibly terminate the reading of a string (known as terminators) otherwise the next read of a string will yield the null string (denoted by `""`). The procedure `skip terminators` with header

```
PROC skip terminators=(REF FILE f)VOID:
```

is used for this purpose.

Exercises

9.8 Modify exercise `ex9.5` (see 9.4) to get the identifiers of its input and output books from the command line (remember that the first argument is always the program id, so use a `LOC STRING` for it). Remember to cater for the end of the input file. Ans

9.9 Write a program to replace all the spaces in its input book with the asterisk and write out the resulting lines to its output book, the book identifiers being given on the command line. Ans

---

9.7 Environment strings

In Linux, if, at the command prompt in a Bash shell, you key `set` followed by return, you will get a listing of the values of all the environment strings defined in your session. The value of the environment string `PATH` gives all the paths that the operating system will search when you try to execute a program.

Each string is identified by what is called an **environment variable** which behaves rather like a name of mode `REF STRING` except that each string is terminated with a `nul ch`. You can open a book containing the environment string using `env channel`. For example:

```
FILE p;  open(p,"PATH",env channel)
```

The open will fail if `PATH` has not been defined, so a plain `open` (as shown in the above example) would be better replaced by

```
FILE p;
IF open(p,"PATH",env channel)/=0
THEN #code to take emergency action#
ELSE #code to perform the usual actions#
FI
```

If you now use `make term` to make the colon `:` the string terminator, you can get the individual paths using `get`:

```
make term(p,":"+nul ch);
STRING path;
on logical file end(p,
 (REF FILE f)BOOL:
  (GOTO eof;  SKIP));
DO
   get(p,(skip terminators,path));
   IF UPB path >= LWB path
   THEN write((path,newline))
```

```
    FI
OD;
eof:
close(p);
```

You should close the book after using it. Notice the use of a `GOTO` followed by a label. The actual label, which looks just like an identifier. is followed by a colon.

Exercises

9.10 Write a program which will display the individual paths in the `PATH` environment string, one to a line, on the screen. Ans

9.11 Write a program which will read some arguments from its command line, each argument consisting of the identifier of an environment string terminated by "/" followed by a non-blank terminator. Using this data, read the environment string and display its constituent parts on the screen, one to a line. Allow for the possibility that the string might not end with the terminator (the code given in the answer caters for that possibility). Try an environment string which exists and one which doesn't. Ans

9.12 At the command line, by using the command

```
ABC="12 14 16"
```

you create (using `bash`) an environment string identified by `ABC`. Now write a program which will read the individual numbers from `ABC` and print their total. Try changing the value of `ABC` to give different numbers (not in the program). Include a test in your program to determine whether `ABC` is present in the environment (verb—open— will fail if it isn't) and terminate your program with a useful message if not. Ans

---

9.8 Writing reports

One of the problems of using the rather primitive facilities given so far for the output of real and integer numbers is that although they allow numbers to be printed in columns, the column widths are fixed. You might not always want 18 decimal places. To print reports where numbers must be neatly tabulated, it is necessary to have some other means of controlling the size of the resulting strings. The procedures `whole`, `fixed` and `float` provide this capability.

The procedure `whole` has the header

```
PROC whole = (NUMBER v, INT width)STRING:
```

and takes two parameters. The first is a real or integer value (modes `REAL` or `INT`)[4] and the second is an integer which tells `whole` the field width of the output number (the space in your output book required to accommodate a value is often called a **field**). If `width` is zero, then the number is printed with the minimum possible width and this will be wider for larger numbers. A positive value for `width` will give numbers preceded by a "+" if positive and a "-" if negative and the number output right-justified within the field with spaces before the sign. A negative value for `width` will only provide a minus sign for negative numbers and the width will be `ABS width`.

Try writing a program with the following fragment included:

```
[]INT ri = (0,99,-99,9 999,99 999);
[]CHAR wh pr = "Parameter for whole is";

FOR wi FROM -6 BY 3 TO 6
DO
   print((wh pr,wi,newline));
   FOR i TO UPB ri
   DO
      write((whole(ri[i],wi),newline))
   OD
OD
```

Notice that where the integer is wider than the available space, the output field is filled with the character denoted by `error char` (which is declared in the standard prelude as the asterisk (`*`) with mode `CHAR`), so it is wise to ensure that your output field is large enough to accommodate the largest number you might want to print.

If you give a real number to `whole`, it calls the procedure `fixed` with parameters `width` and `0`.

The procedure `fixed` has the header

```
PROC fixed = (NUMBER v,
              INT width, after)STRING:
```

and takes three parameters. The first two are the same as those for `whole` and the third specifies the number of decimal places required. The rules for `width` are the same as the rules for `width` for `whole`.

When you want to print numbers in scientific format (that is, with an exponent), you should use `float` which takes four parameters and has the header

[4] `NUMBER` is defined for more modes than `REAL` and `INT` which you will meet in chapter 11.

```
PROC float = (NUMBER v,
              INT width, after, exp)STRING:
```

The first three are the same as the parameters for `fixed`, while the fourth is the width of the exponent field. The version of `float` supplied in the transput library uses `e` to separate the exponent from the rest of the number. Thus the call

```
print(("[",float(pi*10,8,2,-2),"]"))
```

produces the output `[+3.14e 1]`. The parameter `exp` obeys the same rules as `width`.

Note that the transput of data in Algol 68 is so organised that values output by an Algol 68 program can be input by another (or the same) program.

Here are some exercises which test you on your understanding of `whole`, `fixed` and `float`.

Exercises

9.13 The monthly rainfall for a particular location is given by the following figures:

```
 6.54  12.3   10.1   13.83  5.04  9.15
14.34  16.38  13.84  10.45  8.49  7.57
```

Write a program which will print the figures vertically, each preceded by the name of the month. The months and the figures should line up vertically, the months left-justified, the figures with decimal points aligned. Ans

9.14 Write a program which will print the square roots of all the integers from 1 to 100 to 4 decimal places. Each number should be preceded by the corresponding integer. So, for example, the program should print ␣2␣␣1.4142 as a column-pair. Print the whole table in four columns with 25 entries in each column, the numbers 1–25 being in the first column. Ans

9.15 Write a program which will list the terms in the series π, π^2, π^3, ..., π^{10}. Each value should be written in scientific notation with 6 decimal places, and should be preceded by the value of the power (*i.e.*, the numbers 1 to 10). Use a field width of 12. Ans

---

9.9 Binary books

In section 9.1, it was mentioned that some books contain data in a compact form which is not usually human-readable. Most large books, especially those containing design figures in the engineering sciences as well as books containing the payroll data for a number of employees, will be stored in this form. They are called binary books.

Algol 68 allows you to write anything to binary books, just as for text books. Indeed, you can write an integer and a character to a binary book and then read back the data as a character followed by an integer. The results may not be particularly meaningful, but you can do it.

The only difference between transput to, or from, binary books is that instead of using the procedures `put` and `get`, you use the procedures `put bin` and `get bin`. The modes accepted by these procedures are identical with those accepted by `put` and `get` respectively except that you cannot transput procedures with mode

```
PROC(REF FILE)BOOL
```

Here is a program which will output the data produced by the program in the last exercise:

```
PROGRAM binary CONTEXT VOID
USE standard
BEGIN
  FILE f;

  IF establish(f,
               "pipowers",
               stand out channel)/=0
  THEN
    put(stand error,
        ("Cannot create pipowers",
         newline));
    stop
  FI;

  FOR i TO 10
  DO
    put bin(f,(i,pi**i))
  OD;
  close(f)
END
FINISH
```

Run the program and then look at the book it has produced. Compare it with the data produced by the program in the last exercise.

Values of mode `REF STRING` can be read by `get bin`, but you should remember to set the string terminator using the procedure `make term`. However, you should note that the string terminator will *always* include the character `lf`. Furthermore, if `set possible` is `FALSE` for the `REF FILE` on which transput is being performed, the

terminator will have been read when the routine `get bin` returns. If `set possible` is `TRUE` for that `REF FILE`, then the terminator will not have been read.

Another aspect of binary books is that of being able to browse. The principal procedure provided for this purpose is `set` which has the header

```
PROC set=(REF FILE f,INT p,l,c)VOID:
```

The last three parameters specify the position in the book where you want to start browsing, whether reading or writing. The QAD transput provided with the a68toc compiler ignores the `p` and `l` parameters because it regards a file as consisting of one page of one line.[5]

There are two other related procedures. One is `reset` which has the header

```
PROC reset=(REF FILE f)VOID:
```

and is equivalent to `set(f,0)`. One possible use of this procedure is to output data to a book, then use `reset` followed by `get` to read the data from the book. The sort of book used in this way is often called a **work file** (in operating system terms). Such a book contains data of use while a program is being elaborated, but is deleted at the end of the program. In fact, every program has such a book whose controlling `FILE` is called `stand back`. It uses the `stand back channel` and is deleted when the program has finished. However, you can write to it, reset it, then read the contents and copy them to another book. Note that the procedure `read bin` is equivalent to `get bin(stand back,...)` and the procedure `write bin` is equivalent to `put bin(standback,...)`.

The other related procedure is `logical end` which has the header

```
PROC logical end = (REF FILE f)INT:
```

and yields the value of the position at the end of the book, which is the size of the book. The position can be set to the end of the book by writing

```
set(f,logical end(f))
```

Here is a procedure which opens an existing book and sets the writing position to its end, then writes data to the end of the book:

```
PROC debug=(REF FILE dbg,[]SIMPLOUT st)VOID:
(
  open(dbg,idf(dbg),stand back channel);
  set(dbg,logical end(dbg));
  put(dbg,st);
  close(dbg)
)
```

We shall use this procedure in chapter 12 to store data about the running of a program while we are developing it. Notice that textual data is written to the book even though the procedures `set` and `logical end` are used. The point is that binary and textual data can be mixed in any book which allows binary transput.

In the QAD standard prelude, the current position in a book can be obtained from the procedure `current pos` which has the header

[5] The start of a book in the QAD transput is zero.

```
PROC current pos = (REF FILE f)POS:
```

This particular procedure is very useful if you want to store the book position of the beginning of a group of data in a book (such a group is often called a **record**). In the QAD standard prelude, `POS` is a synonym for `INT`.

Exercises

9.16 Write a program which creates a binary book containing the first 1000 whole numbers. Use `set` to read every 17th number and display them on the screen, one to a line. Ans

9.17 Write a program to read a book containing text and write each individual word to one book, and the position of the start of each word and the length of the word to another book. Both output books should be written using `put bin`. Ans

---

9.10 Internal books

Sometimes it is desirable to convert information from binary to text forms and then manipulate the resulting values. Conversely, when performing data entry (that is, reading data from the keyboard), it is usually better to perform the actual data entry in character format and then convert to internal values rather than converting the external data to internal values directly. The means of accomplishing this sort of specialised transput is provided by internal books.

Unfortunately, the QAD transput provided with the a68toc compiler does not provide the usual Algol 68 mechanism for internal books. However, a book consisting of a single line can be established using the `mem channel`. Here is an example:

```
PROGRAM memch CONTEXT VOID
USE standard
BEGIN
   FILE mf;
   establish(mf,"",mem channel,1,1,36);
   FOR i TO 3 DO put(mf,i**3) OD;
   print((file buffer(mf)[:current pos(mf)],
          newline));
   close(mf)
END
FINISH
```

When establishing a memory book using the `mem channel`, both the `p` and the `l` parameters should be `1` and the `c` parameter should be positive indicating the length of the line. All the transput procedures mentioned may be used on memory books. The procedure `file buffer` yields the internal buffer of a file, but uses a mode we have not yet met (see chapter 13: Standard Prelude). The procedure `current pos` gives the current position of its `REF FILE` parameter. For examples of files opened using the `mem channel`, see the example program `lf` described in sections 12.3 to 12.3.3.

9.11 Other transput procedures

The procedure `idf` has the header

```
PROC idf=(REF FILE f)[]CHAR:
```

and yields the identification of the book handled by the file `f`.

There are two other ways of closing a file. One is `scratch` and the other is `lock`. Here are their headers:-

```
PROC scratch=(REF FILE f)VOID:
PROC lock=(REF FILE f)VOID:
```

The procedure `scratch` deletes the file once it is closed. It is often used with work files. The procedure `lock` closes its file and then locks it so that it cannot be opened without some system action. In the QAD transput supplied with the a68toc compiler, `lock` removes all permissions from the file so that it cannot be accessed without first using the program `chmod`.

Exercises

9.18 Write a program to print the rainfall figures given in an earlier exercise. Start your report with a suitable heading. Ans

9.19 Write a program which will read a text file and print each line preceded by a line number. Ans

---

9.12 Summary

External values (usually called data) are stored in books. A program uses an internal structure, called a file (of mode `FILE`), to keep track of the process of transferring data to or from books. The link between them is controlled by a channel.

A number of procedures are provided in the standard prelude to facilitate the transfer of data to and from books, as well as changing the position recorded by a file within a book.

Books can be created and written to, or opened and read from, or both read from and written to. A file should be closed to sever the link between itself and its corresponding book, and to ensure that any data storage areas (usually called buffers) are flushed to the storage medium.

Formatting of numbers can be performed with the procedures `whole`, `fixed` and `float`. This facilitates the production of reports.

String terminators make it easier to read values of mode `STRING`. They are set with the procedure `make term`.

The command line can be read just like any other book (text only) and environment variables can be read.

Exercises

9.20 Write a program to read real numbers from the keyboard, and write them to the screen in scientific notation and 3 decimal places. Continue until zero is read. Ans

9.21 Using the mode `EMPLOYEE` declared in section 7.5, write a program to read the employee records from a binary book, and write a report of the name of each employee, her or his net pay for the current week and the total net pay and number of employees read. In the binary book, each string is preceded by the length of that string as an integer. Get the book idf and the week number from the command line. Ans

Chapter 10

Units

The aim of this chapter is to describe the grammar of units in a fairly rigorous manner. The chapter covers units, contexts and coercions, as well as a number of lesser, but still important, ideas such as casts and balancing. In describing some of the grammatical aspects of the language in previous chapters, it has been necessary to gloss over or distort some of the facts. The definitive truth about such matters is in this chapter.

An Algol 68 program consists of a closed `VOID` clause which means that any value yielded by the closed clause will be voided. Any closed clause can be used including conditional and loop clauses. It is unusual to write a program which starts other than with `BEGIN` (and ends other than with `END`), but there is nothing in the definition of the language to preclude it. On our round tour of units, we shall start at the bottom and work up.

10.1 Phrases

A phrase is a declaration or a unit. Declarations yield no value, even if they include an initial assignment. Units are the parts of the language which actually manipulate values. There are 22 different kinds of unit which can be subdivided into 5 classes arranged in a hierarchy:

```
Quaternaries
   Tertiaries
      Secondaries
         Primaries
            Enclosed clauses
```

where each class includes the lower class. For example, all enclosed clauses are primaries, but not all primaries are enclosed clauses.

The distinctions between different classes of units prevent the writing of ambiguous programs and help to provide the meaning you might expect.

The units in each class are as follows:

- Quaternaries
  - assignments

  - identity relations
  - routine denotations
  - `SKIP`
- Tertiaries
  - formulæ
  - `NIL`
- Secondaries
  - generators
  - selections
- Primaries
  - applied-identifiers
  - calls
  - casts
  - denotations (except routine denotations)
  - slices
- Enclosed clauses
  - case clauses
  - closed clauses
  - collateral clauses
  - conditional clauses
  - conformity clauses
  - loop clauses
  - parallel clauses
  - row-displays
  - structure-displays

10.2 Contexts

The circumstances which allow certain coercions are called contexts. Each context has an intrinsic strength. There are five contexts called **strong**, **firm**, **meek**, **weak** and **soft**. The places in a program which have these contexts are:

- Strong contexts
  - The actual-parameters of calls
  - The enclosed clauses of casts
  - The right-hand side of assignments
  - The right-hand side of identity declarations
  - The right-hand side of initialised name declarations

  - The units of routine denotations
  - `VOID` units
  - All constituents except one of a balanced clause
  - One side of an identity relation
- Firm contexts
  - Operands of formulæ
  - The actual parameters of transput calls
- Meek contexts
  - Enquiry-clauses (including `WHILE`)
  - Primaries of calls
  - The units following `FROM`, `BY` and `TO` in a loop clause
  - Trimmers, subscripts and bounds (must yield an `INT`)
- Weak contexts
  - Primaries of slices
  - Secondaries of selections
- Soft contexts
  - The left-hand side of assignments
  - The other side of an identity relation (see strong context)

10.3 Coercions

There are seven coercions in the language, namely

- **voiding**
- **rowing**
- **widening**
- **uniting**,
- **deproceduring**
- **dereferencing**
- **weakly-dereferencing**

Roughly speaking, the coercions can be arranged in a hierarchy within the hierarchy of contexts thus:

- Strong context
  - deproceduring
  - rowing
  - voiding

  - widening
- Firm context
  - uniting
- Meek context
  - dereferencing
- Weak context
  - weakly-dereferencing
- Soft context
  - deproceduring

The only coercion not yet met is weakly-dereferencing. However, it is useful to describe all the coercions here. Before we do so, it should be noted that one of the limitations of the language is that you cannot specify the kind of context. Thus if you have a weak context and you would like a firm context, you cannot specify it. However, in any context, you can use a **cast** (see the section on primaries below) which will always make a context strong and because all coercions are available in a strong context, you can use the cast to specify the mode you require.

10.3.1 Deproceduring

This coercion is available in all contexts. Deproceduring is the process by which a parameterless procedure is called. For example, the procedure `random`, declared in the standard prelude as having mode `PROC REAL`, when called yields a `REAL`. We can represent the coercion by

$$\texttt{PROC REAL} \Longrightarrow \texttt{REAL}$$

The `PROC` is "removed", which is why it is called deproceduring.

There are occasions when the identifier of a procedure can be written without the procedure being called. In the program fragment

```
PROC REAL rnd:=random
```

the right-hand side of the assignment requires the mode `PROC REAL` because the mode of the name identified by `rnd` is `REF PROC REAL`. Clearly, `random` is not called here.

The only possible ambiguities with deproceduring are those of assignments and casts. For example, having declared `rnd` above, the subsequent assignment

```
rnd:=random;
```

yields a value of mode `REF PROC REAL`, because the value of an assignment is the value of the left-hand side (see section 10.8). However, the following "go-on symbol" indicates that the assignment should now be voided. It is a rule of the language that voiding takes place before deproceduring if the unit being voided is an assignment. If, however, `rnd` had been used on its own, as in

```
rnd;
```

then it would have been dereferenced, then deprocedured and the resulting `REAL` value voided. This would ensure that any side-effects (see sections 6.1.6 and 6.2.3) would take effect.

Similarly, in the unit

```
PROC REAL(rnd);
```

`rnd` (with mode `REF PROC REAL`) will be dereferenced, but the resulting value of mode `PROC REAL` will be voided immediately since it is clear that a `REAL` value is not required. Note that all the code examples using a go-on symbol could have been written with `END` or `FI` etc, provided that the resulting context would have resulted in voiding.

When writing a program, it is common to make mistakes[1], and one mistake is to write the identifier of a procedure without its parameters (the primary of a call). This is not, strictly speaking, an error. At least, not a grammatical error. However, in such a case, the a68toc compiler will issue a warning:

```
Proc with parameters voided,
parameters of call forgotten perhaps
```

in which case the mistake should be obvious. Suppose you write the identifier of a procedure in a formula without its parameters, as in

```
PROC p1 = (INT n)INT:  n**2+3;
INT a:=4;  a:=4+p1;
```

then the a68toc compiler will issue the message

```
op + not declared for INT and PROC (INT)INT
```

The error message for a procedure identifier on the right-hand side of an assignment is

```
PROC (INT)INT cannot be coerced to INT
```

Deproceduring only occurs with parameterless procedures.

10.3.2 Dereferencing

This is the process of moving from a name to the value to which it refers (which could also be a name—see chapter 11). For example, if `x` has mode `REF REAL`, then in the formula

```
x * 3.5
```

the name `x` will be dereferenced to yield a new instance of the `REAL` referred to by `x`. The coercion can be represented by

$$\texttt{REF REAL} \Longrightarrow \texttt{REAL}$$

If `rx` has mode `REF REF REAL` (that is, `rx` can refer to a name of mode `REF REAL`), then the formula

[1] You should expect to make one mistake every 20 lines. Congratulate yourself if you do better!

```
rx * 3.5
```

will result in `rx` being dereferenced twice. In this case, the coercion could be represented as

$$\texttt{REF REF REAL} \Longrightarrow \texttt{REAL}$$

Dereferencing is available in all contexts except soft.

When a name, such as `rx`, is dereferenced twice, new instances of both the values referred to (in the case of `rx`, the `REF REAL` and the `REAL` values) are created. However, the new instance of the `REF REAL` value is discarded after the creation of the `REAL` value. This has no effect on the elaboration of the program.

10.3.3 Weakly-dereferencing

This is a variant of the dereferencing coercion in which any number of `REF`s can be removed except the last. Thus, in the case of `rx` above, weakly-dereferencing would yield a mode of `REF REAL` and could be represented by

$$\texttt{REF REF REAL} \Longrightarrow \texttt{REF REAL}$$

This coercion is only available in weak contexts. It is particularly useful in the selection of secondaries of structure modes which contain fields whose mode starts with `REF` (see section 10.6 and chapter 11).

10.3.4 Uniting

In this coercion, the mode of a value becomes a united-mode. For example, if `OO` is an operator both of whose operands are `UNION(INT,REAL)`, then in the formula

```
3.0 OO -2
```

both operands will be united to `UNION(INT,REAL)` before the operator is elaborated. These coercions can be represented by

$$\left.\begin{array}{l}\texttt{INT}\\ \texttt{REAL}\end{array}\right\} \Longrightarrow \texttt{UNION(INT,REAL)}$$

Uniting is available in firm and strong contexts and must precede rowing.

10.3.5 Widening

In a strong context, an integer can be replaced by a real number and a real number replaced by a complex number, depending on the mode required. This can be represented by

$$\begin{array}{rcl}\texttt{INT} & \Longrightarrow & \texttt{REAL}\\ \texttt{REAL} & \Longrightarrow & \texttt{COMPL}\end{array}$$

Widening is not available in formulæ (firm contexts).

10.3.6 Rowing

If, in a strong context, a multiple is required and a value is provided whose mode is the base mode of the multiple, then the value will be rowed to provide the required multiple. There are two cases to consider:

1. If the mode required is not a name and the base-mode of the multiple is the mode of the value given, then the value will be rowed to give `[]base-mode`. For example, if the required mode is `[]INT`, then the base-mode is `INT`. In the identity declaration

   ```
   []INT i = 3
   ```

   the value yielded by the right-hand side (an integer) will be rowed and the coercion can be expressed as

   $$\texttt{INT} \Longrightarrow \texttt{[]INT}$$

   If the value given is a row mode, such as `[]INT`, then there are two possible rowings that can occur.

   (a) In the identity declaration

   ```
   [,]INT a = i
   ```

   where `i` was declared above with mode `[]INT`, the coercion can be expressed as

   $$\texttt{[]INT} \Longrightarrow \texttt{[,]INT}$$

   In this case, an extra dimension is added to the multiple.

   (b) If the required mode is `[][]INT` as in

   ```
   [][]INT r = i
   ```

   then the value on the right-hand side is rowed to yield a one-dimensional multiple whose base-mode is `[]INT`. This coercion can be represented as

   $$\texttt{[]INT} \Longrightarrow \texttt{[][]INT}$$

2. If the multiple required is a name, then a name of a non-multiple can be supplied. For example, if the value supplied is a name with mode `REF INT`, then a name with mode `REF[]INT` will be created. In this identity declaration

   ```
   REF[]INT ni = LOC INT
   ```

   the local generator yields a name with mode `REF INT` and the rowing coercion yields a name with mode `REF[]INT` and bounds `[1:1]`. The coercion can be represented by

   $$\texttt{REF INT} \Longrightarrow \texttt{REF[]INT}$$

   The first kind of rowing could also occur. The identity-declaration

   ```
   []REF INT rri = LOC INT
   ```

produces the coercion represented by

$$\texttt{REF INT} \Longrightarrow \texttt{[]REF INT}$$

Likewise, a name of mode `REF[]INT` can be rowed to a name with mode `REF[,]INT` or a non-name with mode `[]REF[]INT`, depending on the mode required. Although `INT` has been taken as an example, any mode could have been used.

10.3.7 Voiding

In a strong context, a value can be thrown away, either because the mode `VOID` is explicitly stated, as in a procedure yielding `VOID`, or because the context demands it, as in the case of a semicolon (the go-on symbol). In this case, there are two exceptions to the rule that the value yielded depends only on the context. Casts and assignments are voided after the elaboration of the unit, but all other units are subjected to the usual coercions in a strong context. The following program illustrates this:

```
PROGRAM tproc CONTEXT VOID
USE standard
BEGIN
   PROC INT p;
   PROC pp = INT:
   (
      INT i=random int(6);
      print(i);
      i
   );
   p:=pp;
   print((" p:=pp",newline));
   pp;
   print((" pp",newline));
   p;
   print((" p",newline));
   PROC INT(p);
   print((" PROC INT(p)",newline))
END
FINISH
```

The output is

```
 p:=pp
         +6 pp
         +1 p
 PROC INT(p)
```

In the assignment `p:=pp`, the mode required on the right-hand side is `PROC INT` so `pp` is not deprocedured, and `p` is neither dereferenced nor deprocedured after the assignment has been elaborated. The cast `PROC INT(p)` is elaborated (that is, `p` is dereferenced) and then voided without the procedure `p` (or `pp`) being called.

10.3.8 Legal coercions

In any context, you have a unit which has, or yields, a value of some mode; and in that context you have a mode which you need. If the value of the mode you have can be coerced to a value of the mode you need (assuming that the two modes differ), then the coercion is legal.

For example, suppose you have a value of mode `PROC REF INT` in a strong context and the mode you want is `[]COMPL`. The required mode can be got via

- deproceduring to mode `REF INT`
- dereferencing to mode `INT`
- widening to mode `REAL`
- widening to mode `COMPL`
- rowing to mode `[]COMPL`

In practice, coercions are not usually as complicated as this.

Notice that deproceduring can take place before or after dereferencing, that widening must occur before rowing and that voiding can only take place after all other coercions. For example, you cannot coerce `[]INT` to `[]REAL`.

Exercises

10.1 Which coercions are available in a meek context? Ans

10.2 Which coercions are not available in a strong context? Ans

10.3 For each of the following, state whether the given mode can be coerced to the mode to the right of the arrow: Ans

(a) Weak context: `REF REF BOOL` $\Longrightarrow$ `REF BOOL`
(b) Firm context: `PROC INT` $\Longrightarrow$ `UNION(REAL,COMPL)`
(c) Soft context: `REF PROC CHAR` $\Longrightarrow$ `CHAR`
(d) Meek context: `PROC REF REAL` $\Longrightarrow$ `[]REAL`
(e) Weak context: `PROC REF BOOL` $\Longrightarrow$ `BOOL`
(f) Strong context: `PROC INT` $\Longrightarrow$ `UNION([]INT,[]REAL)`

10.4 Enclosed clauses

There are nine kinds of enclosed clause, most of which we have already met.[2]

1. The simplest is the closed clause which consists of a serial clause enclosed in parentheses (or `BEGIN` and `END`). The range of any identifiers declared in the closed clause is limited to the closed clause. The a68toc compiler limits the use of any identifiers declared in the closed clause to the closed clause at and after their declaration. Here are some examples of closed clauses:

```
(3)
BEGIN p + 3 END
(INT s;  read(s);  s)
(REAL q:=i+2;  sqrt(q))
```

2. Collateral clauses look like row-displays: there must be at least two units. Remember that declarations are not units. The units are elaborated collaterally. This means that the order is undefined and may well be in parallel. Examples of collateral clauses are[3]

```
(m:=3, n:=-2)
((INT m:=2; m),(CHAR a=REPR i; a))
```

   The second collateral clause has two units each of which is a closed clause.

   A parallel clause looks exactly like a collateral clause preceded by `PAR`. The constituent units (there must be at least two) are executed in parallel.[4]

The other enclosed clauses have already been discussed:

3. row-display in section 3.1.1
4. loop clause in section 3.7
5. conditional clause in section 4.5
6. case clause in section 4.6
7. structure-display in section 7.1
8. conformity clause in section 8.3

It should be noted that the enquiry clause (in a conditional- or case-clause) is in a meek context whatever the context of the whole clause. Thus, the context of the clause is passed on only to the final phrase (it must be a unit) in the `THEN`, `ELSE`, `IN` or `OUT` clauses.

[2] Note that a serial clause is *not* an enclosed clause.

[3] The a68toc compiler does not provide collateral clauses other than row- and structure-displays.

[4] The a68toc compiler does not provide parallel clauses.

Exercises

10.4 What kind of enclosed clause could each of the following be? Ans

(a) `((INT p:=ENTIER-4.7; p),37.5)`
(b) `PAR BEGIN 3, 15 END`
(c) `(i|3,-3|4)`
(d) `(si|(INT i): i,(STRING i): i)`
(e) `(a < 3|4|5)`
(f) `(a:=2; b:=-a)`

10.5 Primaries

Primaries are denotations, applied identifiers, casts, calls and slices. We have met denotations in chapters 1, 4 and 6. Only plain values, routines and a special name (`NIL`) have a denotation. `NIL` is dealt with in the section on tertiaries and the mode `BITS` is covered in chapter 11. Applied-identifiers means identifiers being used in a context, rather than in their declarations. We have met numerous examples of these. Routine denotations are not primaries.

A **cast** consists of a mode indicant followed by an enclosed clause, usually a closed clause. Here is a formula with a cast:

```
3.4 * REAL (i)
```

where `i` has mode `INT`. The cast puts the enclosed clause in a strong context. Thus, in the above formula, the normal context of an operand is firm (see chapter 2), but the cast causes the value of `i` to be widened to a `REAL`. Casts are usually used in formulæ and identity relations (see sections 10.8 and 11.6). Casts are sometimes used to coerce a conditional or case clause where balancing is insufficient to provide the mode required (see section 10.9 later in this chapter). The mode indicant can be any mode and can contain any of the mode-constructors such as `REF` or `PROC` or `[]` (but it should not be a generator, which is not a mode indicant). Care should be taken when using a structured mode. For example, in this formula,

```
3 * STRUCT(INT k)(4)
```

assuming that the operator has been declared for operands of modes `INT` and `STRUCT(INT k)`, the cast must include the field selector because it is part of the mode.

Calls were discussed in sections 6.3.1 and 6.3.2. Here is a simple example:

```
sqrt(0.7)
```

In this call, `sqrt` is itself a primary (it is an applied-identifier). In section 10.2, it was mentioned that the primary of a call is in a meek context. This applies even if the call itself (as a whole) is in a strong context. The primary of a call can be an enclosed clause. For example,

```
(a>4|sqrt|sin)(x)
```

which yields `sqrt(x)` if `a > 4` and `sin(x)` otherwise. In this case, the primary is

```
(a>4|sqrt|sin)
```

We discussed slices in section 3.2. They include simple subscripting. For example, given the declaration

```
[,]INT r = ((1,2,3),(4,5,6))
```

the units `r[1,]` and `r[2,3]` are both slices. Whatever the context of the slice, the context of the primary of the slice (`r` in these examples) is always weak. This means that only weak-dereferencing is allowed. Thus, given the phrases

```
[2,3]INT s:=r;  INT p:=s[2,1]
```

the slice `s[2,1]` is in a strong context, but the `s` is in a weak context, so the name that `s` identifies, which has the mode `REF[,]INT` will not be dereferenced, though the slice, which has mode `REF INT`, will be.

There is another consequence of the weak context of the primary of a slice: row-displays can only be used in a strong context. So if you want to change the bounds of a row-display, because the slicer will produce a weak context, the row-display must be enclosed in a cast.

The context of subscripts and bounds in trimmers is meek and they must be units.

All enclosed clauses are primaries, but not all primaries are enclosed clauses.

Exercises

10.5 What are the contexts of Ans

(a) `p` (mode `REF[]REAL`) in `[]REAL (p[3])`
(b) `q` (mode `PROC(REAL)INT`) in `REAL(q(0.5))`

10.6 How many primaries are there in each of the following units: Ans

(a) `3 * (1.4 + r)/2**6`
(b) `p:=sqrt(r) - 6`
(c) `num:=x[3,ENTIER r]`
(d) `i * []CHAR("e")`

10.6 Secondaries

We have discussed both kinds of secondary (selections and generators), but there are other points which need mentioning.

There are two kinds of generator (see section 5.1). Occasionally, when a procedure has a name parameter, the name may not be needed. Instead, therefore, of using an identifier of a name which is used for another purpose, which would be confusing, or declaring a name just for this purpose, which would be unnecessary, an anonymous name can be used. For example, a possible call of the procedure `char in string` could be

```
char in string(ch,LOC INT,str)
```

if you are only interested in whether the character is in the string and not in its position.

Another case where an anonymous name is useful is in the creation of odd-shaped multiples. Consider the program fragment:

```
[10]REF[]INT ri;  INT j;

FOR i TO UPB ri
DO
   read(j);
   ri[i]:=LOC[j]INT;  read(ri[i])
OD
```

Since there are no declarations in the loop clause, the scope of the name created by the generator is the enclosed clause surrounding the loop clause, which includes the declarations for `ri` and `j`. The mode of the slice `ri[i]` is `REF REF[]INT`. Thus the value of `ri[i]` is a name with two `REF`s in its mode, and it is made to refer to a name of mode `REF[]INT`, which has one `REF` less. Assignments of this type will be considered in detail in the next chapter. Note that the context of a parameter to `read` is firm so the parameter is dereferenced once before a value is read.

When discussing selections in section 7.2, you may have wondered about the peculiar rules of placing parentheses when talking about rows in structures, rows of structures and rows in rows of structures. Firstly, it should be mentioned that in the secondary

```
im OF z
```

where z has mode `COMPL` or `REF COMPL`, the `z` itself is not only a secondary, it is also a primary (it is an applied-identifier). This means that using the declarations

```
MODE AM = STRUCT(INT i,CHAR c),
     BM = STRUCT(INT i,AM a);
BM b
```

the selection

```
c OF a OF b
```

is valid because

```
a OF b
```

is also a secondary. We shall meet extended selections like this in chapter 11.

Secondly, a primary is a secondary, but not necessarily the other way round. Consider these declarations:

```
STRUCT(INT i,[3]REAL r)s1;
[3]STRUCT(INT i,REAL r)s2
```

The selection `r OF s1` has the mode `REF[]REAL`. If you want to slice it, to get one of the constituent names of mode `REF REAL` say, you cannot do so directly. The reason is that in a slice, as mentioned in the previous section, what is sliced must be a primary. To convert the secondary into a primary you have to enclose it in parentheses thus converting it into an enclosed clause; and enclosed clauses are also primaries (in section 10.1, it was said that the four classes of units are arranged in a hierarchy in which each class includes the lower classes). So the second name of `r OF s1` is yielded by `(r OF s1)[2]`.

On the other hand, considering the name identified by `s2`, the selection

```
r OF s2[2]
```

can be written without parentheses because `s2` is not only a secondary, it is also a primary (an applied-identifier) with mode `REF[]STRUCT...`. The phrase `s2[2]` is perfectly valid, it having mode `REF STRUCT(...)`. The selection `r OF s2` has the mode `REF[]REAL` and so it too can be sliced by writing `(r OF s2)[2]`. The effect is the same for both of the cases involving `s2`. Note that the a68toc compiler does not permit selection of a field from a row of structures. Doing so will yield the following error message:-

```
OPERATORS - select: []struct not implemented
FATAL ERROR (661) Compiler error:
   ENVIRONMENT (ASSERT) - assertion failure
```

To summarise, any primary can be regarded as a secondary, but not vice-versa.

Exercises

10.7 Give an example of a primary which is also a secondary. Ans

10.8 Give an example of a secondary which is not a primary. Ans

10.9 In this exercise, the following declarations hold:

```
MODE AM = STRUCT(CHAR a,b),
     BM = STRUCT(AM a,
                 STRUCT(CHAR a,AM b) c,
                 REF BM d);
BM p
```

How many secondaries are there in each of the following units? Ans

(a) `a OF p`
(b) `a OF a OF p`
(c) `a OF c OF p`
(d) `a OF a OF d OF p`

10.7 Tertiaries

Tertiaries are formulæ and `NIL`. Formulæ were covered in chapter two. All that needs to be said here is that a formula can consist solely of a single secondary or primary or enclosed clause although this is not usual. If a formula, containing at least one operator, is to be used as a primary or a secondary, it must be enclosed in parentheses (or `BEGIN` and `END`). For example, in the formula `next OF (H declarer)`, where `H = (INT)REF HAND` and `HAND = STRUCT(...,REF HAND next)`, the formula must be surrounded by parentheses to make it into a secondary.

The only name having a denotation is `NIL`. Its mode is `REF whatever`. In other words, it can have any mode which starts with `REF`. It does not refer to any value and, although it must only occur in a strong context, it cannot be coerced. Its uses are described in the next chapter.

10.8 Quaternaries

Quaternaries are assignments, routine denotations, identity relations and `SKIP`. Of the four, the assignment is the most common. An assignment consists of three parts. The left-hand side must be a tertiary. It is usually an applied identifier or, less commonly, an enclosed clause. Its value must be a name. Its context is soft, so no dereferencing is allowed unless a cast is used (see the next chapter), but deproceduring is allowed. The second part is the assignment token. The right-hand side (the third part) can be any quaternary (including, of course, another assignment). Its context is strong so any coercion is permitted. The mode of its value must contain one less `REF` than the mode of the left-hand side.

The right-hand side of an assignment is, most commonly, a formula which is a tertiary (all tertiaries are quaternaries, but not vice-versa). The left-hand side can also be a formula provided that the value yielded is a name (which is the case with the assigning operators—see section 5.1.2). If an assignment is to be used as a primary, a secondary or a tertiary, then it must be enclosed in parentheses(or `BEGIN` and `END`). The value of an assignment is the value of the left-hand side: that is, it is a name. Assignments were discussed in chapter 5.

Routine denotations were discussed in chapter 6.

`SKIP` yields an undefined value of any mode and can only occur in a strong context. It is particularly useful in the following case. Consider the procedure

```
PROC p=(REAL a,b)REAL:
IF b=0
THEN print(("Division by zero",newline)):
   stop;  SKIP
ELSE a/b
FI
```

Since the yield has mode `REAL`, and the `ELSE` part of the conditional clause yields a value of mode `REAL`, by the principle of balancing (see below) the `THEN` part also must yield a value of mode `REAL`. Now the construct `stop` yields a value of mode `VOID` which cannot be coerced to `REAL` in any context. If the procedure is going to compile successfully, the `THEN` part must yield `REAL` (or, at least, a value which can be coerced to `REAL` in the context of the body of the procedure which is strong) even though the value yielded will never be used (because the `stop` will terminate

the program). The `SKIP` will yield an undefined value of mode `REAL`. Although `SKIP` must occur in a strong context, it cannot be coerced.

Another use for `SKIP` is in row- or structure-displays where not all the units are known at the time of a declaration. For example:

```
[3]INT ii:=(4,?,5)
```

Before the multiple `ii` is used, the second element should be given a value. If no such value is assigned, and you try to print the value of `ii[2]` the a68toc compiler will generate code which will print whatever value was there at the time the multiple was generated, which may well be rubbish.

The identity relation is discussed in the next chapter, but its grammar has important consequences. The identity relation consists of two tertiaries separated by an identity relator (one of `:=:` or `:/=:`). Since a formula is a tertiary, it can safely be included in an identity relation. For example, given the declarations

```
INT x:=3, y:=1;
PROC x or y = (REAL r)REF INT: (r<0.5|x|y)
```

the identity relation

```
x or y(random) :=: x
```

is legal. However, if you want to include an identity relation in a formula then you must surround it with parentheses to make it into a tertiary, as in

```
IF (x or y(random) :=: x) AND x*y > 0
THEN
```

Since one side of an identity relation is in a soft context while the other is in a strong context, only one side of an identity relation can be strongly-dereferenced. The soft side can be weakly-dereferenced which means that one `REF` will always be left on that side. Balancing applies to identity relations (see the discussion in section 11.6).

This completes the general discussion of units.

Exercises

10.10 What kind of units are each of the following: Ans

(a) A cast.
(b) An applied-identifier.
(c) A selection.
(d) A multiple.
(e) A name.
(f) A formula.
(g) A loop clause.
(h) An assignment.
(i) A declaration.
(j) A procedure denotation.

10.11 Which units are to be found in each of the following: Ans

(a) `3.5 * (a - 2 * x)`
(b) `p OR q AND a = 4`
(c) `sin(x)`
(d) `a[3,2:4]`
(e) `x:=(c<"e"|2.4|-y)`
(f) `(i|x,y,z):=(p|2|-4)`
(g) `PAR(x:=1.2,y:=3.6)`

10.9 Balancing

In section 6.1, it was pointed out that the context of a routine denotation is passed on to the last unit in the denotation. In the example given, the body of the routine denotation was a closed clause. The yield of the routine was a value of mode `INT`, but the yield of the last unit was a name with mode `REF INT`. Since the context of the body of a routine denotation is strong, the name is dereferenced to get an `INT`. This principle is applicable to all enclosed clauses.

Now conditional clauses, case clauses and conformity clauses can yield one of a number of units, and so it is quite possible for the units to yield different values of different modes. The principle of balancing allows the context of all these units, except one, to be promoted to strong whatever the context of the enclosed clause. Balancing is also invoked for identity relations which are dealt with in the next chapter.

Considering, for example, the formula

```
x * (a > 0|3.0|2)
```

the context of the conditional clause is firm which means that widening is not allowed. Without balancing, the conditional clause could yield a `REAL` or an `INT`. In this example, the principle of balancing would promote the context of the `INT` to strong and widen it to `REAL`. Balancing thus means "making the modes the same".

In a balanced clause, one of the yielded units is in the context of the whole clause and all the others are in a strong context, irrespective of the actual context of the clause. Here is an example of a balanced case clause

```
INT i:=3,j:=4,a:=2;
PROC ij = REF INT: (random < 0.5|i|j);
print(2 + (a|i,ij|random))
```

where the `a` yields an `INT` in a meek context (that of the enquiry clause). In this example, the modes of the values that can be yielded by the case clause are `REF INT` (`i`), `PROC REF INT` (`ij`) and `PROC REAL` (`random`). In a firm context, the modes become `INT`, `INT` and `REAL`. Thus the context of `random` is taken to be firm, and the context of `i` and `ij` is promoted to strong and they are both dereferenced and widened to `REAL`. The result is that the case clause will yield a `REAL` value even though the clause as a whole is in a firm context (it is an operand of the operator `+`).

If instead, we had

```
PROC REAL r:=random;
(a|i,ij|j):=ENTIER r
```

using the declaration of `ij` in the previous example, then balancing would not be needed to produce the required mode. The modes of the yielded units are `REF INT`, `PROC REF INT` and `REF INT` respectively. In a soft context, these modes would yield `REF INT` (no dereferencing allowed), `REF INT` (deproceduring is allowed) and `REF INT`. Thus the case clause would yield `REF INT` on the left-hand side of the assignment.

Here is an example of a conditional clause which cannot be balanced:

```
INT i:=2, REAL a:=3.0;
(random > 0.5|i|r):=random
```

In this case, the two parts of the conditional clause yield `REF INT` and `REF REAL`. There is no coercion which will convert a `REF INT` into a `REF REAL`. When you try to compile this, the a68toc compiler gives the following error message:

```
lhs of assignment must be a reference
```

The balancing means that one of the yields is in a strong context and so is dereferenced which yields a value which is not a name.

The method of determining whether balancing is possible is as follows:

1. Determine the context of the choice clause.
2. In the context found in step 1, determine the mode yielded by each unit in the choice clause.
3. If there is a mode such that all the modes but that one can be strongly coerced to that mode, the clause can be balanced.

Exercises

10.12 In each of the following clauses, state whether balancing is possible, and if so, the mode yielded by the balanced clause. These declarations are in force:

```
INT i,j, REAL a,b:=random;
PROC ij = REF INT: (b>0.5|i|j);
PROC r = REAL: random * random;
UNION(INT,REAL) ri:=(random>0.6|i|b)
```

Ans

(a) `a:=2.0*(random<0.3|i|b)`
(b) `(j<2|ij|b):=r`
(c) `a:=((ri|(INT r):r,(REAL r):r)<1|2|3)`
(d) `b:=2.0*(j>3|4|SKIP)`

10.10 Well-formed modes

In chapter 6, the mode declaration was presented and it was pointed out that not all possible mode declarations are allowed. The rules for determining whether a mode declaration is well-formed are straightforward.

There are two reasons why a mode might not be well-formed:

1. the elaboration of a declaration using that mode would need an infinite amount of memory
2. the mode can be strongly coerced to a related mode

Let us look at some examples of modes which are not well-formed. Firstly, in the mode declaration

```
MODE WRONG = STRUCT(CHAR c,WRONG w)
```

the `WRONG` within the `STRUCT` would expand to a further `STRUCT` and so on *ad infinitum*. Even this declaration

```
MODE WRONGAGAIN = STRUCT(WRONGAGAIN wa)
```

will not work for the same reason. However, if the mode within the `STRUCT` is shielded by `REF` or `PROC`, then the mode declaration is legal:

```
MODE ALRIGHT = STRUCT(CHAR c,REF ALRIGHT a);
```

In the declaration

```
ALRIGHT ar = ("A",LOC ALRIGHT)
```

the second field of the structure is a name which is quite different from a structure. Likewise, the declaration

```
MODE OKP = STRUCT(CHAR c,PROC OKP po)
```

is well-formed because in any declaration, the second field is a procedure (or a name referring to such a procedure) which is not the original structure and so does not require an infinite amount of storage. It should be noted, however, that a `UNION` does not shield the mode sufficiently. Thus, the mode declarations

```
MODE MW1 = UNION(INT,MW1);
MODE MW2 = STRUCT(UNION(CHAR,MW2) u,CHAR c)
```

are not well-formed. In fact, the mode declaration of `MW1` fails on reason 2 above.

Secondly, a mode which could be strongly coerced to a related mode would lead to ambiguity in coercions. Thus the mode declarations

```
MODE WINT = PROC WINT;
MODE WREF = REF WREF;
MODE WROW = [5]WROW
```

are not well-formed.

All the above declarations have been recursive, but not mutually recursive. Is it possible to declare

```
MODE WA = STRUCT(WB wb,INT i),
     WB = STRUCT(WA wa,CHAR c)
```

Again, the elaboration of declarations using either mode would require an infinite amount of storage, so the modes are not well-formed. The following pair of mode declarations are all right:

```
MODE RA = STRUCT(REF RB rb,INT i),
     RB = STRUCT(PROC RA pra,CHAR c)
```

All non-recursive mode declarations are well-formed. It is only in recursive and mutually-recursive modes that we have to apply a test for well-formedness.

Determination of well-formedness

In any mutually-recursive mode declarations, or any recursive mode declaration, to get from a particular mode on the left-hand side of a mode declaration to the same mode indicant written on the right-hand side of a mode declaration, it is necessary to traverse various mode constructors such as `REF`, `PROC` or `UNION`. Above each `STRUCT` or set of procedure parameters write "yang". Above each `REF` or `PROC` write "yin". Now trace the path from the mode in question on the left-hand side of the mode declaration until you arrive at the same mode indicant on the right-hand side. If you have at least one "yin" and at least one "yang", the mode is well-formed.

Let us try this method on the recursive mode declarations given in this section. In the mode declaration for `WRONG`, write "yang" above the `STRUCT`. Thus to get from `WRONG` on the left to `WRONG` on the right, a single "yang" is traversed. Thus `WRONG` is not well-formed. Likewise, `WRONGAGAIN` is not well-formed. In mode `ALRIGHT`, you have to traverse a "yang" (`STRUCT`) and a "yin" (`REF`), so `ALRIGHT` is well-formed. Try it with the mode `OKP`.

Conversely, to get from `MW1` to `MW1` requires neither "yin" nor "yang", so `MW1` is not well-formed. To get from `MW2` to `MW2`, only a `STRUCT` is traversed (the `UNION`

does not count) so `MW2` is also not well-formed. Similar arguments hold for `WINT`, `WREF` and `WROW`.

Now consider the mutually-recursive mode declarations of `WA` and `WB`. At whichever mode we start, getting back to that mode means traversing two "yangs" (both `STRUCT`). Two "yangs" are all right, but there should be at least one "yin", so the modes are not well-formed. On the other hand, from `RA` to `RA` traverses a `STRUCT` and a `REF` and, via `RB`, a `STRUCT` and a `PROC` giving "yang-yin-yang-yin", so both `RA` and `RB` are well-formed.

Remember that if you want to declare modes which are mutually-recursive, the a68toc compiler requires that one of the modes should first be declared with a stub declaration.

Exercises

10.13 For each of the following mode declarations, determine whether the modes are well-formed: Ans

(a)
```
MODE MA = INT
```

(b)
```
MODE MB = PROC(MB)VOID
```

(c)
```
MODE MC =[3,2]MC
```

(d)
```
MODE MD = STRUCT(BOOL p,MD m)
```

(e)
```
MODE ME = STRUCT(STRING s,REF ME m)
```

(f)
```
MODE MF2,
     MF1 = STRUCT(REF MF2 f),
     MF2 = PROC(INT)MF1
```

(g)
```
MODE MGB,
     MGA = PROC(MGB)VOID,
     MGB = STRUCT(MGA a)
```

(h)
```
MODE B, C,
MODE A = PROC(B)A,
MODE B = STRUCT(PROC C c,
                STRUCT(B b,INT i)d),
MODE C = UNION(A,B)
```

(i)
```
C = PROC(C)C
```

10.11 Flexible names

Flexible names were introduced in section 5.5, but only one-dimensional names. What has not been made apparent in the text hitherto is that a multiple consists of two parts: a **descriptor** and the actual elements. The descriptor contains the lower and upper bounds of each dimension, the "stride" (that is, the number of bytes between two successive elements of the dimension in question), the address in memory of the first element of that dimension and whether the dimension is flexible. Consider the declaration

```
FLEX[1:0][1:3]INT flexfix
```

Because the mode of `flexfix` is `REF FLEX[][]INT`, when it is subscripted, the mode of each element is `REF[]INT` with bounds of `[1:3]`. Clearly, after the declaration, `flexfix` has no elements. In practice, because the first (and only) dimension is flexible, there must be some way of referring to a "ghost" element whose descriptor (it is a one-dimensional multiple) will give its properties. `flexfix` is quite different from

```
FLEX[1:0]FLEX[1:3]INT flexflex
```

each of whose elements (when it has any) have the mode `REF FLEX[]INT` with initial bounds `[1:3]`.

If the declaration of `flexfix` is followed by the assignment and slice

```
flexfix:=LOC[1:1][1:3]INT;
flexfix[1]:=(1,2,3)
```

then it is clear that the mode of `flexfix[1]` is `REF[]INT`. Note that after

```
flexfix:=LOC[1:4][1:3]INT
```

`flexfix` refers to a multiple of which each element has the mode `[]INT`. However, the single dimension of

```
flexfix[1]
```

is *not* flexible, which is why the assignment

```
flexfix:=LOC[1:4][1:4]REAL #this is wrong#
```

will fail[5].

10.12 Orthogonality

We have come a long way and introduced many new ideas, yet all these ideas are based on the primitive concepts of value, mode, context, coercion and phrase. These concepts are independent of each other, but their combination provides Algol 68 with a flexibility that few programming languages possess. For example, if a value of mode `INT` is required, such as in a trimmer or the bounds of the declaration of a

[5] The a68toc compiler will wrongly allow this last assignment both at compile-time and run-time.

multiple, then any unit which will yield an integer in that context will suffice. The consequence is that Algol 68 programs can be written in a wide variety of styles. Here is a simple example: given the problem of printing the sum of two numbers read from the keyboard, it could be programmed in two completely different ways. The conventional solution would be something like

```
INT a,b;  read((a,b));
print((a+b,newline))
```

but an equally valid solution is

```
print(((INT a,b;
        read((a,b));
        a+b),newline))
```

Provided that what you write is legal Algol 68, you can adopt any approach you please. Orthogonality refers to the independence of the basic concepts in that you can combine them without side-effects.

Another consequence of that independence is that there are very few exceptions to the rules of the language. This makes the language much easier to learn.

10.13 Summary

The grammar of Algol 68 is expressed in terms of a few primitive concepts: value, mode, context, coercion and phrase. A phrase is either a declaration or a unit. There are 5 contexts, 7 coercions, 22 different kinds of unit and potentially an infinite number of values and modes. The coercions available in each context have been described. Balancing is the means by which alternatives in conditional, case and conformity clauses and the two sides of an identity relation are coerced to a common mode, possibly making coercions available which would not normally be so in the context of the construct concerned.

No exercises are provided at this point.

Chapter 11

Advanced constructs

We have now covered most of Algol 68. All that remains is the identity relation, the parallel clause, the mode `BITS`, completers, different precisions of numbers and means of accessing operating system facilities. Most of these are deceptively simple. The identity relation is used with modes containing multiple `REF`s which take up the greater part of this chapter. The `BITS` mode and the parallel clause are introduced in later sections. Different numerical precisions and access to operating system facilities is mainly covered in chapter 12. We start with access to the machine word.

11.1 Bits, bytes and words

In our discussion of plain values (values of modes `CHAR`, `INT`, `REAL`, and `BOOL`), we have omitted saying how these values are stored in computer memory for one important reason: Algol 68 is a high-level programming language. A high-level programming language is one in which the concepts of computer programming are not expressed in terms of a computer, its instructions and its memory, but in terms of high-level concepts such as values and modes. Basically, the manner in which integers and characters and so on are stored in the computer are not our business. However, since programs written in Algol 68 need to access the operating system, it is useful to know something about memory, whether the main memory of the computer or the storage memory found on hard disks and other devices.

Computer memory consists of millions of **bits** (short for binary digits) which are grouped together as **bytes** or **words**. A bit can take two values: 0 and 1. A word is 16, 24, 32, 36, 60, 64 or 72 bits "wide", and a byte is 6, 8 or 9 bits "wide". Almost all microcomputers use 8-bit bytes. Microcomputers using the Intel Pentium processor (or compatibles) or later chips, use a 32-bit word and an 8-bit byte. Generally speaking, a byte is used to store a character and a word is used to store an integer. Real numbers are much more complicated than integers and we shall not describe how they are stored in memory. Before we can understand about the equivalences of values of mode `CHAR` and bytes, and values of mode `INT` and words, we need to say something about **radix arithmetic**. If this is something you already know, please skip the next section.

11.1.1 Radix arithmetic

Our ordinary arithmetic uses the ten digits 0, 1, ..., 9 and expresses numbers in powers of ten. Thus the number 1896 consists of 1 thousand, 8 hundreds, 9 tens and 6 units. This could be written

$$1896 = 1 \times 1000 + 8 \times 100 + 9 \times 10 + 6 \times 1$$

Remembering that 100 is ten squared (10^2) and 1000 is ten cubed (10^3), we could rewrite this equation as

$$1896 = 1 \times 10^3 + 8 \times 10^2 + 9 \times 10^1 + 6 \times 10^0$$

As you can see, the powers of ten involved are 3, 2, 1 and 0. When we write whole numbers, we understand that the digits we use represent powers of ten. We say that the base, or **radix**, of our arithmetic is ten, which is why it is frequently referred to as "decimal arithmetic" (decimal is derived from the Latin word for ten).

Now it is quite meaningful to develop an arithmetic having a different radix. For example, suppose we use two as the radix. We should express our numbers in terms of powers of two and they would be written using the digits 0 and 1 only. In an arithmetic of radix two, when we write a number, each digit would represent a power of two. For example, the number 101 would mean

$$101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

in an exactly analogous way to the number 1896 in decimal arithmetic. In fact, the decimal equivalent of 101 would be $4 + 0 + 1 = 5$ (in decimal). Here is another

example:

$$\begin{aligned} 1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13 \text{ (thirteen, in decimal)} \end{aligned}$$

We could then construct addition and multiplication tables as follows:

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

As you can see from the addition table, $1 + 1 = 10$ (take row 2 and column 2). When you read this equation, you must say "one-zero" for the number after the equals symbol. "Ten" means ten+zero units which this number definitely is not. The number 10 in radix 2 means "two+0 units" which is what you would expect for the sum of 1 and 1.

Two radices of particular use with computers are sixteen and two. Arithmetic with a radix of sixteen is called **hexadecimal** and arithmetic with a radix of two is called **binary**.

In hexadecimal arithmetic, the digits 0 to 9 are used, but digits are also required for the numbers ten to fifteen. The first six letters of the alphabet are used for the latter six numbers. They are commonly written in upper-case, but in Algol 68 they are written in lower-case for a reason which will become apparent in a later section. Thus the "digits" used for hexadecimal arithmetic are

```
0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
```

Addition and multiplication tables could be constructed for hexadecimal arithmetic on the same lines as those for radix two arithmetic. You should note that when writing a number containing more than one digit with a radix other than ten, the radix is commonly written (in decimal) as follows:

$$2 \times 3 = 12_4$$

Thus, in hexadecimal arithmetic, we could write

$$7 \times 9 = 3\mathrm{f}_{16}$$

and there are some exercises at the end of this section in which you can try your hand at hexadecimal and other arithmetics. Writing numbers in hexadecimal is sometimes called "hexadecimal notation".

A byte consists of eight binary digits and can take any value from 00000000_2 to 11111111_2. The equivalent decimal value can be obtained by rewriting it as the sum of descending powers of two:

$$\begin{aligned} 10011011_2 &= 1 \times 2^7 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 \\ &= 128 + 16 + 8 + 2 + 1 \text{ (in decimal)} \\ &= 155_{10} \end{aligned}$$

The exercises at the end of this section will give you some practice in this kind of conversion.

If you compare the number of digits used to express the same number, you will find that hexadecimal arithmetic uses the least. For example, the decimal number 135 can be written

$$\begin{aligned} 135 &= 87_{16} \\ &= 2013_4 \\ &= 10000111_2 \end{aligned}$$

When converting numbers written in binary to hexadecimal, the simplest way is to split the binary number into two groups of 4 bits and then convert each group into one hexadecimal digit. Thus `00101011` can be split into `0010` and `1011`, and their hexadecimal equivalents are `2` and `b`. If you intend accessing machine words, it would certainly be a good idea to learn the binary equivalents of the 16 hexadecimal digits `0-f`. To help you, here is the procedure `itostr` which converts a positive value of mode `INT` to a value of mode `STRING` (with minimum width) using any radix from 2 to 16:

```
[]CHAR digits="0123456789abcdef"[@0];

PROC itostr=(INT n#umber#,r#adix#)STRING:
IF n < r
THEN digits[n]
ELSE itostr(n%r,r)+digits[n MOD r]
FI
```

Notice how its recursive definition simplifies the code.

Exercises

11.1 Using the procedure `itostr`, write a program which will display the 16 integers between 0 and 15 (decimal) in decimal, hexadecimal and binary (the binary equivalent should be displayed as 4 bits) in three columns. Ans

11.2 For each of the following, rewrite the number in the given radix: Ans

(a) $94_{10} \Rightarrow 16$
(b) $13_{10} \Rightarrow 2$
(c) $1111\,1001_2 \Rightarrow 16$
(d) $3e1_{16} \Rightarrow 10$
(e) $2c_{16} \Rightarrow 2$
(f) $10101_2 \Rightarrow 10$

11.3 Express the value of each of the following using the radix of that exercise: Ans

(a) $101_2 + 110_2$
(b) $35_{16} + ae_{16}$
(c) $17_8 + 37_8$

11.2 The mode BITS

A value occupying a machine word has the mode `BITS`. The number of binary digits in one machine word is given by the environment enquiry (see section 13.2) `bits width` which, for the a68toc compiler is `32`. A `BITS` value can be denoted in four different ways using denotations written with radices of 2, 4, 8 or 16. Thus the declarations

```
BITS a = 2r 0000 0000 0000 0000
            0000 0010 1110 1101
BITS b = 4r 0000 0000 0002 3231
BITS c = 8r  000 0000 1355
BITS d = 16r 0000 02ed
```

are all equivalent because they all denote the same value. Notice that the radix precedes the `r` and is written in decimal. Notice also that the numbers can be written with spaces, or newlines, in the middle of the number. However, you cannot put a comment in the middle of the number. Since a machine word contains 32 bits, each denotation should contain 32 digits in radix 2, 16 digits in radix 4, 11 digits in radix 8 and 8 digits in radix 16, but it is common practice to omit digits on the left of the denotation whose value is zero. Thus the declaration for `d` could have been written

```
BITS d = 16r2ed
```

When discussing values of mode `BITS` where the values of more significant bits is important, full denotations like the above may be more appropriate.

Monadic operators for BITS

There are many operators for `BITS` values. Firstly, the monadic operator `BIN` takes an `INT` operand and yields the equivalent value with mode `BITS`. The operator `ABS` converts a `BITS` value to its equivalent with mode `INT`. The `NOT` operator which you first met in chapter 4 (section 4.2) takes a `BITS` operand and yields a `BITS` value where every bit in the operand is reversed. Thus

```
NOT 2r 1000 1110 0110 0101
       0010 1111 0010 1101
```

yields

```
      2r 0111 0001 1001 1010
         1101 0000 1101 0010
```

Notice that spaces have been used to make these binary denotations more comprehensible. `NOT` is said to be a **bit-wise operator** because its action on each bit is independent of the value of other bits.

Dyadic operators for BITS

AND and OR (both of which you also met in chapter 4) both take two BITS operands and yield a BITS value. They are both bit-wise operators and their actions are summarised as follows:

Left Operand	Right Operand	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

For OR, the yield of

```
2r 100110 OR 2r 10101
```

is 2r 110111. The priority of AND is 3 and the priority of OR is 2.

The AND operator is particularly useful for extracting parts of a machine word. For example, suppose you have a BITS value where the least-significant 8 bits are equivalent to a character. You could write

```
CHAR c = REPR ABS (b AND 16rff)
```

Here, the operators REPR and ABS do not generate machine-code instructions, but merely satisfy the compiler that the modes are correct. This sort of formula is, in fact, very efficient in Algol 68.

It is possible to extract a single bit from a word using the operator ELEM which has the header

```
(INT n,BITS t)BOOL:
```

For example, given the declaration

```
BITS bi = 16r 394a 2716
```

then each hexadecimal digit represents 4 bits: the 3 occupies bit positions 1–4, the 9 occupies bit positions 5–8, the 4, bit positions 9–12, and so on. Suppose we want the third bit (the leftmost bit is bit-1). The following declaration is valid:

```
BOOL bit3 = 3 ELEM bi
```

Thus, if the third bit is a 1, the declaration will give the value TRUE for bit 3. In fact, 3 written in binary is 0011_2, so bit 3 is 1. Thus

```
2 ELEM bi
```

would yield FALSE. The priority of ELEM is 7.

Incidentally, notice that in Algol 68 the most significant bit in a machine word is bit-1 and the least significant bit is bit-32. This strongly suggests that computers in the 1960's were "big-endian". Intel microprocessors and other compatible processors are "little-endian"[1]. Because the a68toc compiler translates Algol 68

[1]These terms come from the book by Jonathan Swift entitled "Gulliver's Travels" and they refer to the habit of some people of eating boiled eggs at the "big" end or the "little" end!

programs into C programs, it is quite possible for the Algol68toC system to be implemented on a "big-endian" microprocessor, such as the Motorola 68000-series. A "big-endian" processor stores a machine word as four bytes (each of 8-bits) with the most significant byte at the lowest memory address. "Little-endian" processors store the least significant byte at the lowest memory address. Whatever kind of microprocessor is used to elaborate your programs, the most significant bit of the word is bit-1 and the least significant bit is bit-32 in Algol 68.

The dyadic operators `SHL` and `SHR` shift a machine word to the left or to the right respectively by the number of bits specified by their right operand. Their priority is `8`. To illustrate their action we shall suppose that they all operate on the `BITS` value `16r 89ab cdef`. Both the shifts are by four bits which is equivalent to one hexadecimal digit (4 bits is half a byte and is commonly called a nibble: yes, software engineers do possess a sense of humour!).

The result of shifting the above value by 4 bits is given by the following table:

Original value = `16r 89ab cdef`		
Operator	Bits shifted	Yield
`SHL`	4	`9abc def0`
`SHL`	-4	`089a bcde`
`SHR`	4	`089a bcde`
`SHR`	-4	`9abc def0`

When shifting left (`SHL`), bits shifted beyond the most significant part of the word are lost. New bits shifted in from the right are always zero. When shifting right (`SHR`), the reverse happens. Note that the number of bits shifted should be in the range $[-32, +32]$. For `SHL`, if the number of bits to be shifted is negative, the `BITS` value is shifted to the right and likewise for `SHR`. The header for `SHL` is

```
OP SHL = (BITS b,INT i)BITS:
```

and correspondingly for `SHR`. The value `b` is the value to be shifted and the integer `i` is the number of bits to shift. `UP` and `DOWN` are synonyms for `SHL` and `SHR` respectively. The priorities of `SHL` and `SHR` are both 8.

As well as the operators `=` and `/=` (which have the usual meaning), the operators `<=` and `>=` are also defined for mode `BITS`. The formula

```
s >= t
```

yields `TRUE` only if for all bits in `t` that are `1`, the corresponding bits in `s` are also `1`. This is sometimes regarded as "s implies t". Contrariwise, the formula

```
s <= t
```

yields `TRUE` only if for all bits in `t` which are `0`, the corresponding bits in `s` are also `0`. Likewise, this is sometimes regarded as "`NOT t` implies `s`".

Exercises

11.4 Given the declarations

```
BITS a = 16r 1111 1111,
     b = 16r 89ab cdef
```

what is the value of each of the following: Ans

(a) `a AND b`
(b) `a OR b`
(c) `NOT a OR b` [Hint: convert each value to radix 2 and then combine]
(d) `a = b`

11.5 What is the value of Ans

(a) `16rab SHL 3`
(b) `16rba SHR 3`

11.3 Overlapping slices

Whathappens if two trimmed multiples overlap? For example, consider the program

```
PROGRAM slices CONTEXT VOID
USE standard
BEGIN
   [4]INT r;

   PROC res = VOID:
   FOR n FROM LWB r TO UPB r
   DO r[n]:=n OD;

   PROC mpr = ([]INT m)VOID:
   (
      FOR i FROM LWB m TO UPB m
      DO
         print((whole(m[i],0),blank))
      OD;
      print(newline)
   ); #mpr#

   res;
   print("Original contents:");  mpr(r);

   r[:UPB r-1]:=r[1+LWB r:];
   print((newline,"r[:3]:=r[2:]",newline,
          "Compiler does it: "));  mpr(r);
   res;
```

```
    FOR i FROM LWB r TO UPB r-1
    DO r[i]:=r[i+1] OD;
    print("Forwards loop:    "); mpr(r);
    res;

    FOR i FROM UPB r-1 BY -1 TO LWB r
    DO r[i]:=r[i+1] OD;
    print("Backwards loop:   "); mpr(r);
    res;  r[1+LWB r:]:=r[:UPB r-1];
    print((newline,"r[2:]:=r[:3]",newline,
           "Compiler does it: "));  mpr(r);
    res;

    FOR i FROM 1+LWB r TO UPB r
    DO r[i]:=r[i-1] OD;
    print("Forwards loop:    "); mpr(r);
    res;

    FOR i FROM UPB r BY -1 TO 1+LWB r
    DO r[i]:=r[i-1] OD;
    print("Backwards loop:   "); mpr(r)
END
FINISH
```

When compiled and executed, the program gives the following output:

```
Original contents:1 2 3 4

r[:3]:=r[2:]
Compiler does it: 2 3 4 4
Forwards loop:    2 3 4 4
Backwards loop:   4 4 4 4

r[2:]:=r[:3]
Compiler does it: 1 1 2 3
Forwards loop:    1 1 1 1
Backwards loop:   1 1 2 3
```

Notice that lines 5 and 8 of the results are definitely wrong, but that the compiler gets it right both times. The lesson is, do not worry about overlapping multiples: the compiler will ensure you get the effect you want.

A different matter is when you want to replace a column of a square multiple with a row. Here, the overlap is more of a "crossoverlap". In this case you need to be careful—see the next exercise.

Exercises

11.6 Given a square two-dimensional multiple of integers, write a procedure which uses trimmers (not necessarily overlapping) to convert its columns to rows and its rows to columns. For example:

```
((1,2,3),          ((1,4,7),
 (4,5,6),    =>     (2,5,8),
 (7,8,9))           (3,6,9))
```

Your procedure should cater for any size of square multiple. Ans

11.4 Completers

Sometimes it is desirable to have more than one possible end-point of a serial clause. This often happens when a loop needs to be prematurely terminated so that a surrounding serial clause can yield a value which is unexpected. A **completer** is so-called because it provides a completion point for a serial clause. A completer can be placed wherever a semicolon (the go-on symbol) can appear except in enquiry clauses (whether BOOL enquiry clauses or INT enquiry clauses). It consists of the construct `EXIT` followed by a **label** and a colon (`:`). A label is formed with the same rules as for an identifier and should not be the same as any identifier in the current range. Here is an example of a completer:

```
EXIT label:
```

The label must be referenced by a `GOTO` clause within the same serial clause in which the completer occurs, or in an inner clause (not necessarily serial). Here is an example of such a completer:

```
a:=(INT i;  read((i,newline));
    IF i < 0 THEN GOTO negative FI;
    sqrt(i) EXIT
negative:
    sqrt(-i)
   )
```

The example is artificial, but serves to illustrate the use of a completer.

A completer can sometimes save the declaration of a boolean name. For example, here is a procedure without a completer:

```
PROC is in str = (STRING t, CHAR c)BOOL:
(
   BOOL found := FALSE;

   FOR n FROM LWB t TO UPB t
   WHILE ~found
   DO
      found:=c = t[n]
```

```
   OD;

   found
);
```

Here is the procedure with a completer:

```
PROC is in str = (STRING t,CHAR c)BOOL:
(
   FOR n FROM LWB t TO UPB t
   DO
      IF c = t[n] THEN GOTO found
   OD;
   FALSE EXIT
found:
   TRUE
)
```

In fact, `GOTO` clauses are valid almost anywhere in Algol 68. They are particularly useful when it is required to jump out of nested clauses. Let us reconsider the program `echo` in section 9.5.1 with a `GOTO` clause:

```
PROGRAM echo CONTEXT VOID
USE standard
BEGIN
FILE args;
IF open(args,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access the arguments",
        newline));
   stop
ELSE
   FILE ff:=args;
   on logical file end(
      ff,
      (REF FILE f)BOOL:
         close(f);  GOTO end;  FALSE));
   DO
      STRING arg;
      get(ff,(skip terminators,arg));
      print((arg,newline))
   OD;
end:
   print(("End of arguments",newline))
FI
END
FINISH
```

Use of `GOTO` clauses should be confined to exceptions because otherwise they can destroy the natural structure of your programs making them much more difficult to understand and maintain.

11.5 References to names

The idea that a mode can contain more than one `REF`, or that a mode might be `REF[]REF[]CHAR` was broached at the start of chapter 5 and mentioned in section 10.3.2. The time has now come to address this topic fully.

Any mode which starts with `REF` is the mode of a name. The value to which a name refers has a mode with one `REF` less. Since names are values in their own right, there is no reason at all why a name should not refer to a name. For example, suppose we declare

```
INT x,y
```

then the mode of both `x` and `y` is `REF INT`. We could also declare

```
REF INT xx, yy
```

so that `xx` and `yy` both have the mode `REF REF INT`.

Now, according to the definition of an assignment (see section 10.8), it is perfectly legitimate to write

```
xx:=x
```

without any dereferencing because the identifier on the left has mode `REF REF INT` and the identifier on the right has mode `REF INT`. Leaving aside for the moment of how useful such declarations and assignments might be (and they are very useful, essential even), let us give our attention to the mechanics. We could assign `y` to `xx` and a value to `y` with the double assignment

```
xx:=y:=3
```

Again, no dereferencing is involved. Now, given that `xx` refers to `y` which refers to `3`, how could we make `y` refer to `4`, say? Simple. Assign `4` directly to `y`. However, if the assignment to `xx` was

```
xx:=(random>0.8|x|y)
```

we should not know which name `xx` referred to. Finding out which name `xx` refers to is the subject of the next section.

You may remember that the context of the left-hand side of an assignment is soft so no dereferencing is allowed. The way to coerce a name of mode `REF REF INT` to a name of mode `REF INT` is to use a cast:

```
REF INT(xx):=4
```

Note that the unit

```
print(xx)
```

will yield `4` with `xx` being dereferenced twice. There is nothing to stop us writing

```
REF REF INT xxx:=xx
```

with assignments like

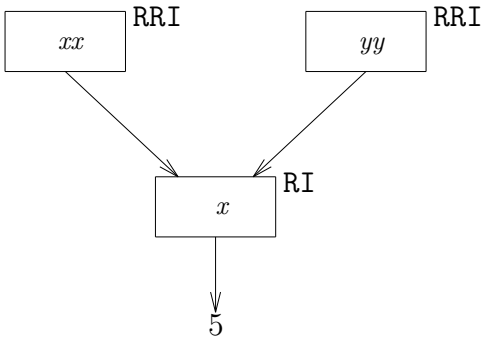
```
REF REF INT(xxx):=x
REF INT(xxx):=-2
```

and we shall see in a later section that names with modes `REF REF REF some-mode` have a use. Although you can use as many `REF`s as you like, there does not seem to be any need for more than three.

Now consider the assignments

```
xx:=yy:=x:=4
```

Both `xx` and `yy` refer to different instances of the name `x`, but when those instances are dereferenced, they both yield `4`. This means that if we assign `5` to `x`, when `xx` and `yy` are dereferenced twice, they will both yield `5`. We can represent this relationship by the diagram



where `RRI` and `RI` stand for `REF REF INT` and `REF INT` respectively. Thus, although strictly speaking `xx` and `yy` refer to different instances of the name identified by `x`, we shall regard them as both referring to `x`.

Exercises

11.7 Given the declaration

```
REF REAL xx:=LOC REAL
```

how would you make the anonymous name refer to `120.5`? Ans

11.8 Write a declaration for `rrq` which has the mode `REF REF REF[]CHAR` and make it refer to an anonymous name which refers to an anonymous name which refers to a multiple of 10 characters. Ans

11.9 Write the declaration of a name which can refer to a flexible name which can refer to a row of integers. In a separate assignment, assign the row-display `(3,-2,4)` to your name. Ans

---

11.6 Identity relations

Consider the declarations of the last section:

```
INT x,y; REF INT xx,yy
```

We had assigned a name to `xx` with the assignment

```
xx:=(random > 0.8|x|y)
```

and we wished to ascertain whether `xx` referred to `x` or to `y`. Unfortunately, we cannot use the equals operator `=` for this purpose because its operands would be completely dereferenced and the underlying integers would be compared. Instead, we use an **identity relation** which is used exclusively for comparing names. The identity relation

```
xx :=: x
```

yields `TRUE` if `xx` refers to `x`. The alternative representation of `:=:` is `IS`. The identity relation

```
xx :/=: x
```

yields `TRUE` if `xx` does not refer to `x`. The alternative representation of `:/=:` is `ISNT`. Here is a short program which illustrates the difference between `=` and `IS`:

```
PROGRAM test CONTEXT VOID
USE standard
BEGIN
   REF INT xx, INT x:=2,y:=3;
   TO 3
   DO
      xx:=(random>0.5|x|y);
      print(("xx :=: x =",
             (xx :=: x|"TRUE"|"FALSE"),
             newline,"xx = ",xx,newline))
   OD
END
FINISH
```

If you want to compare the names that both `xx` and `yy` refer to, it is no good writing

```
xx IS yy
```

This always yields `FALSE` because the names that `xx` and `yy` identify always differ (they were created using two local generators so the names are bound to be different). The point is that no automatic dereferencing takes place in an identity relation. To compare the names that both `xx` and `yy` refer to, you should place one side or both sides in a cast:

```
REF INT(xx) IS yy
```

This will ensure that the right-hand side (in this case) is dereferenced to yield a name of the same mode as the left-hand side. This is because an identity relation is subject to balancing: one side of the relation is in a soft context and the other side is in a strong context. Given the cast on the left-hand side, the two sides of the identity relation would yield `REF INT` and `REF REF INT`. Since no dereferencing is allowed in a soft context, it can be seen that the left-hand side is in the soft context and the right-hand side is in the strong context.

The `IS` and `ISNT` in the identity relation are not operators. Since the identity relation is a quaternary (see section 10.8), remember to enclose it in parentheses if you want to use it in a formula:

```
IF (field OF struct ISNT xx) & x>=-5
THEN field OF struct = 0
ELSE FALSE
FI
```

Exercises

11.10 The program fragment

```
REF STRING ff, ss; STRING f, s;
f:="Joan of Arc";
s:="Robert Burns";
ff:=(random<0.1|f|s);
ss:=(ff IS f|s|f)
```

applies to this and the following exercises.

What are the modes of `f` and `ss`? Ans

11.11 What does `f` refer to? Ans

11.12 Write a formula which compares the 3^{rd} and 4^{th} characters of the multiple `f` refers to with the 7^{th} and 8^{th} characters of the multiple `s` refers to. What are the modes of the operands of the operator? Ans

11.13 Write an expression which compares the name referred to by `ff` with the name referred to by `ss`. Ans

---

11.7 The value NIL

Sometimes it is desirable that a name of mode `REF REF whatever` should not refer to a definite name (see, for example, the discussion of queues below). This can be arranged by making it refer to `NIL` which is the only denotation of a name. The mode of `NIL` is `REF whatever`. For example, consider

```
REF[]CHAR rc=NIL;
REF INT ri=NIL
```

The first `NIL` has the mode `REF[]CHAR` and the second has the mode `REF INT`.

Given the declaration

```
REF INT xx:=NIL
```

the mode of `NIL` is `REF INT`. However, although `NIL` is a name, you cannot assign to it. That is, the assignment

```
REF INT(xx):=4
```

would cause the run-time error

```
Segmentation fault
```

and, very likely, a core dump, when using the a68toc compiler.

Nor can you use `NIL` in a formula if that would involve dereferencing. The only use of `NIL` is for determining, by using an identity relation, that a name refers to it. However, we shall see in the sections on queues and trees that this is a vital function.

Now consider the declaration

```
REF REF INT rrri;
```

where the mode of `rrri` is `REF REF REF INT`. We could make `rrri` refer to `NIL` directly using the assignment

```
rrri:=NIL
```

whence the mode of `NIL` is `REF REF INT`. Or we could use a `NIL` of mode `REF INT` by using an anonymous name:

```
rrri:=LOC REF INT:=NIL
```

whence the mode of the anonymous name is `REF REF INT`. In the identity relation

```
rrri IS NIL
```

how can we tell which `NIL` is in use? Of course, we could use a cast for `rrri`, but there is a simpler and more useful way. First we declare

```
REF INT nil ri = NIL
```

then balancing will ensure that the identity relation

```
rrri IS nil ri
```

gives the required answer with `rrri` being dereferenced twice. Alternatively, with the declaration

```
REF REF INT nil rri = NIL
```

we can ensure that the identity relation

```
rrri IS nil rri
```

will also be elaborated correctly. We shall see in the sections on queues and trees that the declaration of `nil ri` is more useful.

Now consider the declarations

```
INT x:=ENTIER(random * 6), y;
REF INT xx,yy;
PROC x or y = REF INT: (random>0.8|x|y)
```

and the identity relation

```
CASE randint(3) IN xx,x or y, NIL ESAC
                IS
CASE y IN x, SKIP, yy ESAC
```

The balancing of the identity relation includes balancing of the case clauses. The modes yielded are

```
xx REF REF INT
   x or y    PROC REF INT
   NIL       REF whatever
   x         REF INT
   SKIP      who knows?
   yy        REF REF INT
```

In a soft context, these modes become:

```
REF REF INT
                REF INT
                REF whatever
                REF INT
                who knows?
                REF REF INT
```

Thus the left-hand side is the soft context and the right-hand side (of the identity relation) is the strong context (remember that `SKIP` is only allowed in a strong context), and the final modes are all `REF INT`. In practice, it is rare that identity relations are so complicated.

Exercises

11.14 Given the declarations

```
FILE f1:=stand in, f2;
REF FILE cur file:=f2;
PROC p = REF FILE:
   (cur file IS f1|f1|f2)
```

what is the value of Ans

(a) `cur file:=f2`
(b) `cur file :/=: stand in`
(c) `p:=f1`
(d) `p:=:f1`

11.15 Given the declarations of exercise 1, what is the mode of `NIL` in Ans

(a) `cur file:=NIL`
(b) `REF REF FILE ff:=NIL`

11.8 Queues

Consider the problem of representing a queue. We shall suppose that the queue is at a football match and that each fan in the queue has a name, in the ordinary sense, and a ticket number. Rather than just present the solution to this problem, we shall discuss the problem in detail and show why the solution is what it is.

A suitable mode for the fan would be`FAN`:

```
MODE FAN = STRUCT(STRING name,INT ticket)
```

but what would be a suitable declaration for a queue? At first sight, it would appear that a flexible name which can refer to a multiple of fans would be suitable:

```
MODE QUEUE = FLEX[1:0]FAN
```

but there are difficulties. Firstly, the only way a new fan could be added to the queue would be to assign a whole new multiple to a name (in the Algol 68 sense) referring to the queue:

```
QUEUE q; q:=q+FAN("Jim",1)
```

assuming that the operator `+` has been declared with the header

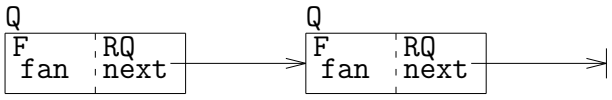
```
OP + = ([]QUEUE a,FAN b)[]QUEUE:
```

If the queue were long, this would be very inefficient. Secondly, given a particular fan, how would we find the fan behind him or her? Knowing the subscript of the fan would seem to be the answer, but what happens if someone joins the queue somewhere in front of the fan in question? Given that there might be several fans under consideration, the program would have to update all the relevant subscripts and keep a record of which subscripts would be relevant.

The solution is to represent a queue as a recursive structure:

```
MODE QUEUE=STRUCT(FAN fan,REF QUEUE next)
```

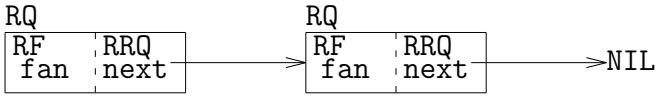
Then a queue with two fans in it could be represented by the diagram



where the mode of each box is `QUEUE` and `F` and `RQ` stand for `FAN` and `REF QUEUE` respectively. Notice that the `next` field of the first structure refers to the structure on its right. The `next` field of the second structure does not refer to anything.

From the declaration of the mode `QUEUE`, we can see that the `next` field of the structure is a name with mode `REF QUEUE`. It would appear that it is possible to construct a queue in the way depicted by the last diagram: each `next` field of each structure would refer to the next structure (of mode `QUEUE`) and the last `next` field would have the mode `REF QUEUE` and value `NIL`.

Now consider adding another `QUEUE` to the right-hand end of the queue. We immediately run into a difficulty. The value of the `next` field of the last `QUEUE` is `NIL` with mode `REF QUEUE`. However, we cannot assign to `NIL`, nor can we replace the name `NIL` with another name to make it refer to a new `QUEUE`. The reason is that a name of mode `REF QUEUE` can only be replaced by another name of mode `REF QUEUE` if the first name is referred to by a name of mode `REF REF QUEUE`. In other words, instead of making the structures have mode `QUEUE`, we should make them have mode `REF QUEUE`. In section 7.2, on field selection, we pointed out that the modes of the fields of a structure name are all preceded by a `REF`. This also applies to a recursively-defined structure. In this case, the mode of the `next` field becomes `REF REF QUEUE` and could refer to `NIL` (with mode `REF QUEUE`) or to another structure of mode `REF QUEUE`. We can depict this as



where `RQ`, `RRQ` and `RF` represent the modes `REF QUEUE`, `REF REF QUEUE` and `REF FAN` respectively.

Now let us consider how such a queue could be created. Since the length of the queue at the time the program is written is unknown (and will change as fans join or leave the queue), it is not possible to have an identifier for each structure forming the queue. Instead, we can create anonymous names using a generator. However, we must be able to refer to the queue in order to manipulate it. Let us declare a name, identified by `head`, to refer to the beginning of the queue:

```
REF QUEUE head:=NIL
```

We have made it refer to `NIL` (with mode `REF QUEUE`) because the queue is currently empty. Using the suggestion of the last section, we shall declare

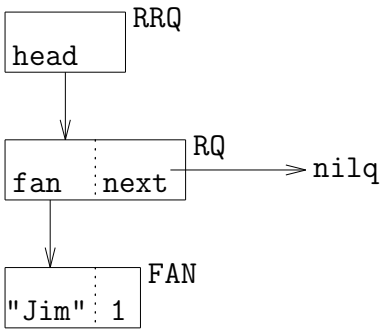
```
REF QUEUE nilq = NIL;
REF QUEUE head:=nilq
```

where `head` has the mode `REF REF QUEUE`.

Let us assign the first fan to the queue:

```
head:=LOC QUEUE:=(("Jim",1),nilq)
```

We can represent this by the diagram



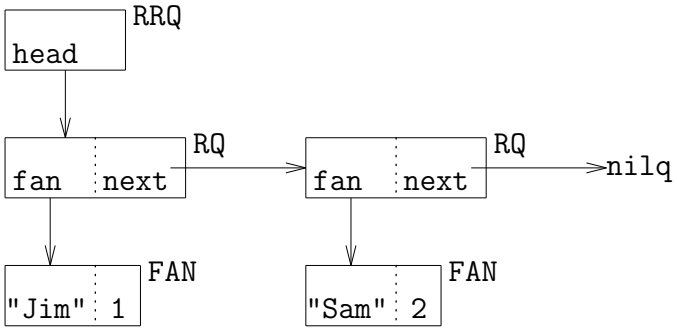
We can now assign another fan to the queue:

```
next OF head:=LOC QUEUE:=(("Fred",2),nilq)
```

Let us be quite clear what is happening here. The mode of `head` is `REF REF QUEUE`. It is a name which refers to a name so it has no fields. We can select fields only from a `QUEUE` or a `REF QUEUE`. However, the context of a selection is weak(see section 10.3) and so only weak-dereferencing is allowed. Thus in

```
next OF head
```

`head` is dereferenced to mode `REF QUEUE` and the `next` field selected (with mode `REF REF QUEUE`). The anonymous name `LOC QUEUE` has mode `REF QUEUE`, so the structure display (with mode `QUEUE`) is assigned to it, and it in turn is assigned to `next OF head` without dereferencing. This means that the `nilq` which `next OF head` referred to after the first fan `("Jim",1)` was added to the queue has been replaced by the second `LOC QUEUE` which is what we wanted. We can now depict the queue by



We could now extend the queue by writing

```
next OF next OF queue:=LOC QUEUE
```

but since we do not know how long the queue might become, clearly we cannot go on writing

```
next OF next OF ...
```

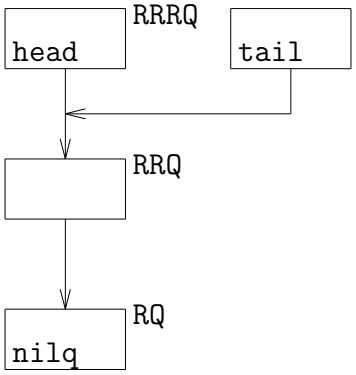
What we need is some way of referring to the tail of the queue without lots of selections. Because the tail of the queue always has mode `REF REF QUEUE` (it is the `next` field of a `REF QUEUE`), what we need is a name of mode `REF REF REF QUEUE` (yes, three `REF`s). Here it is:

```
REF REF QUEUE tail;
```

but again we run into a difficulty (the last one). When the queue is empty, `head` refers to `nilq`, but what does `tail` refer to since we cannot select from `nilq` (because it is `NIL`)? The solution is to make `head` have the mode `REF REF REF QUEUE` as well as `tail` and generate a name of mode `REF REF QUEUE` to refer to `nilq`! (bear with it, we're almost there):

```
tail:=head:=LOC REF QUEUE:=nilq
```

In this triple assignment, only `head` is dereferenced. We can depict this as



Now we can assign the first fan to the head of the queue:

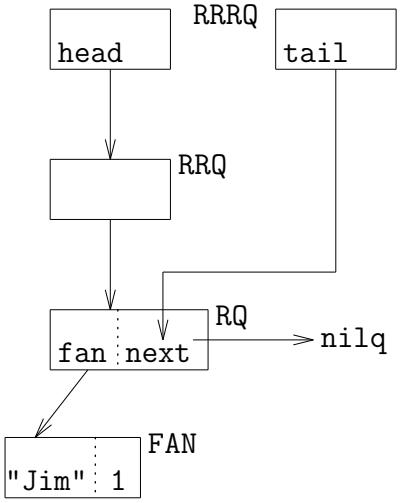
```
REF REF QUEUE(head):=LOC QUEUE:=
      (("Jim",1),nilq))
```

and make `tail` refer to the tail of the queue with

```
tail:=next OF head
```

in which `head` is dereferenced twice, but the selection is not dereferenced at all.

The queue can now be depicted as shown below.



A queue is one example of what is called a **linked-list**.

Exercises

11.16 Extend the queue by assigning another `REF QUEUE` to `tail`. Ans

11.17 Now make `tail` refer to the tail of the queue again. Ans

11.18 Has the name referred to by `head` changed after adding the new `REF QUEUE`? Ans

---

11.9 The procedure `add fan`

We are now ready to develop a procedure to add a fan to the end of the queue. Clearly, there are two different situations: an empty queue and a non-empty queue. Although we only need `tail` to extend the queue, we need `head` to determine whether the queue is empty. So here is the header:

```
PROC add fan = (REF REF REF QUEUE head,tail,
                REF FAN fan)VOID:
```

Firstly, we need to test whether the queue is empty:

```
IF head IS nilq
```

Remember that the mode of `head` is `REF REF REF QUEUE`, so in the identity relation `head` is dereferenced twice.

Secondly, if the queue is empty, we assign an anonymous `REF QUEUE` to the name `head` refers to and assign `(fan,nilq)` to the `REF QUEUE`:

```
THEN REF REF QUEUE(head):=
          LOC QUEUE:=(fan,nilq)
```

but this will not work because the scope of the `LOC QUEUE` is limited to the routine denotation. We must use a global generator:

```
THEN REF REF QUEUE(head):=
         HEAP QUEUE:=(fan,nilq)
```

Then we have to ensure that `tail` refers to the tail of the queue:

```
tail:=next OF head
```

If the queue is not empty, we assign an anonymous `REF QUEUE` to the name that `tail` points to:

```
ELSE REF REF QUEUE(tail):=
         HEAP QUEUE:=(fan,nilq)
```

and make `tail` refer to the new tail:

```
tail:=next OF tail
```

Here is the complete procedure:

```
PROC add fan = (REF REF REF QUEUE head,tail,
                REF FAN fan)VOID:
IF head IS nilq
THEN #the queue is empty#
   REF REF QUEUE(head):=
         HEAP QUEUE:=(fan,nilq);
   tail:=next OF head
ELSE
   REF REF QUEUE(tail):=
         HEAP QUEUE:=(fan,nilq);
   tail:=next OF tail
FI #add fan#
```

Exercises

11.19 It looks as though `add fan` could be optimised. Rewrite the body of `add fan` so that the overall structure is

```
tail:=next OF (REF REF QUEUE
   CO IF ... FI  plus two assignments CO
               )
```

Ans

11.20 Write a program containing the necessary declarations and loop to create a queue containing 1000 fans—alternate the names of the fans between `Iain` and `Fiona` and increment the ticket numbers by 1. Compile and run the program to check that there are no errors (no output will be produced). Ans

---

11.10 More queue procedures

We can now address three more procedures: how to insert a fan into a queue, how to remove a fan from the queue, and how to print the queue. Let us take the printing procedure first. Here it is:

```
PROC print queue = (REF REF QUEUE head)VOID:
IF head IS nilq
THEN print(("NIL",newline))
ELSE print((newline,
            "(",name OF fan OF head,",",
            whole(ticket OF fan OF head,0),
            ")=>"));
     print queue(next OF head)
FI
```

By not using the triple `REF` name for the head of the queue, we can use recursion to simplify the procedure. Recursion is common in procedures for linked-lists.

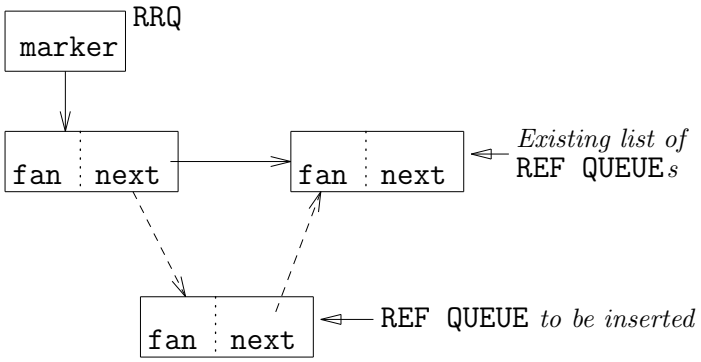
Inserting a fan is a little more difficult. There are several possibilities: the queue can be empty or non-empty. If it is non-empty, the fan can be inserted at the head of the queue, or if there are at least two fans in the queue, the fan could be inserted somewhere between the head and the tail. The question is, how many parameters are required for the procedure? Clearly, we need `head` to determine whether the queue is empty, `tail` to be updated in case it is or if the fan is to be added to the end of the queue. Here is a possible header:

```
PROC insert fan=(REF REF REF QUEUE head,tail,
                 REF FAN fan)VOID:
```

We need a criterion for determining where in the queue a fan should be inserted. Here is one: the fans should be inserted in the order of ticket number (using a queue is not an efficient way of doing this, but this criterion will do for our purposes). Here is `insert fan` with a diagram to help you understand it:

```
PROC insert fan=(REF REF REF QUEUE head,tail,
                 REF FAN fan)VOID:
IF   head IS nilq
THEN #the queue is empty#
   REF REF QUEUE(head):=
         HEAP QUEUE:=(fan,nilq);
   tail:=next OF head
ELIF ticket OF fan < ticket OF fan OF head
THEN
   #insert the fan at the head of the queue#
   REF REF QUEUE(head):=
         HEAP QUEUE:=(fan,head)
ELIF next OF head IS nilq
THEN #add the fan after the head#
   REF REF QUEUE(tail):=
         HEAP QUEUE:=(fan,nilq);
   tail:=next OF tail
```

```
ELIF REF QUEUE marker:=head;
   WHILE
      IF (next OF marker ISNT nilq)
      THEN
         ticket OF fan
               >
         ticket OF fan OF next OF marker
      ELSE FALSE
      FI
   DO marker:=next OF marker OD;
   next OF marker IS nilq
THEN
#add the fan to the end of the queue#
   REF REF QUEUE(tail):=
         HEAP QUEUE:=(fan,nilq);
   tail:=next OF tail
ELSE
CO insert the fan between 'marker'
   and 'next of marker' CO
   next OF marker:=
         HEAP QUEUE:=(fan,next OF marker)
FI
```



There are three lines where you need to look carefully at the modes and values involved:

- the line which ends in `(fan,head)`,
- the line which ends in `(fan,next OF marker)`,
- the line containing the `>` operator.

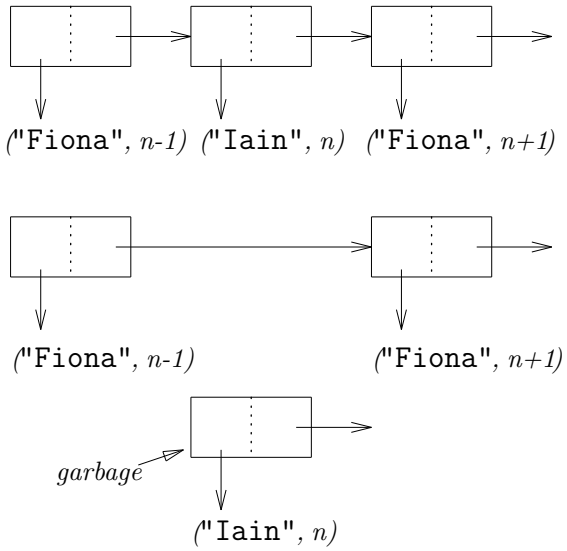
Discussion of this procedure completes our examination of queues.

Exercises

11.21 In the procedure `insert fan`, explain the circumstances in which the loop will terminate. Ans

11.22 Using the procedure `print queue`, confirm that the procedure `insert fan` works. Ans

11.23 Write the procedure `delete fan` which will delete a fan with a given ticket number from the queue. It should yield the fan if it has been deleted and `FALSE` if it cannot be found. This diagram should help you:



Include the procedure in a program and test it. Ans

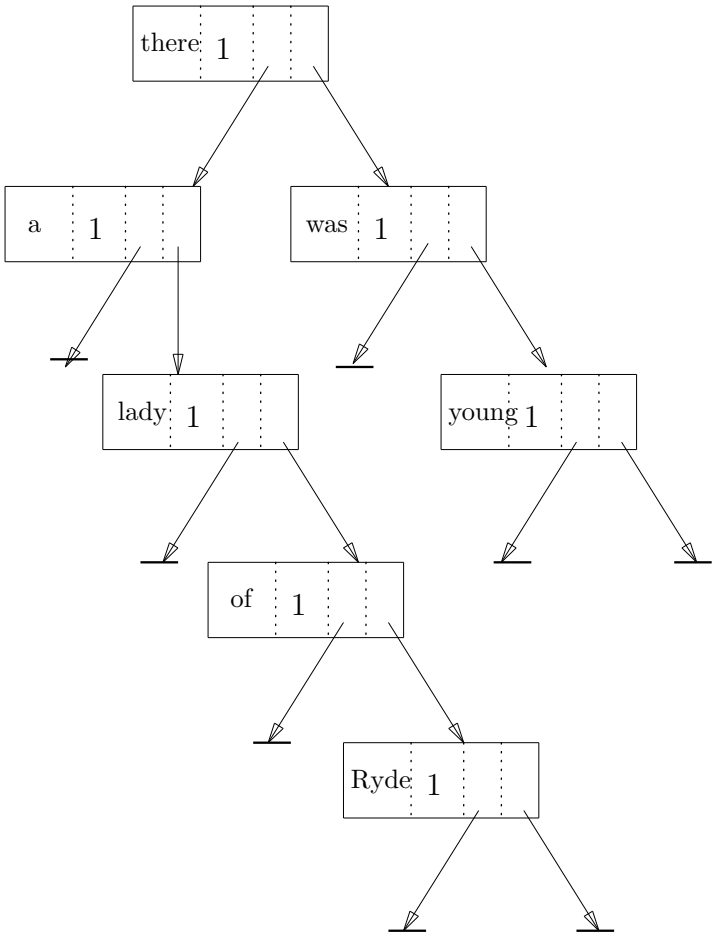
11.11 Trees

Both queues and trees are examples of recursive structures. Queues contain only one link between individual structures, trees contain at least two. Trees are another kind of linked-list and are interesting because they give more examples of how recursive procedures are used to manipulate recursively-defined data structures.

There are two principal kinds of trees in common use: B-trees and binary trees. B-trees (sometimes called **balanced trees**) are too advanced to be described here.

A binary tree consists of a number of forks, usually called **nodes**, which are linked with two links per node.

Here is an example of a small tree:



The topmost node is called the **root** (trees are usually depicted upside-down[2]). Each node consists of three parts: the data which each node bears and left and right references which can refer to other nodes. In the small tree shown above, there are seven nodes on five levels. There are 4 nodes on the left branch of the root and 2 on the right, so that the tree is unbalanced.

A binary tree is particularly suitable for the ordering of data: that is, for arranging data in a predefined order[3]. In the previous section, in procedure `insert fan`, we considered inserting a fan into a queue in ascending order of ticket number. This is an inefficient way of ordering data. For example, suppose there are 100 fans in the queue. Then, on average, we can expect to insert a fan halfway down the queue; which means 50 comparisons of ticket numbers. If the fans were stored as a balanced binary tree, the maximum number of comparisons would be only 7 (because $2^6 < 100 < 2^7$). For larger numbers, the difference between the two kinds

[2]The remainder of the intriguing limerick runs as follows:-

> Who ate sour apples and died.
> The apples fermented inside the lamented
> and made cider inside 'er inside.

[3]but only if the tree is reasonably balanced

of linked-list is even more marked. For 1000 fans, a queue would need 500 comparisons on average, whereas a balanced binary tree would need 10 at most. While it is true that these figures are minima (they assume that the tree is balanced, that is, that there are as many nodes to the left of the root as to the right), nevertheless, on average, a binary tree is much more efficient than a queue for ordering data.

Here is a typical mode declaration for a binary tree:

```
MODE WORD = STRUCT(STRING wd,
                   INT ct,
                   REAL fq),
     TREE = STRUCT(REF WORD w,
                   REF TREE left,right);
```

The mode of the data in the declaration of `TREE` is `REF WORD` so that if an item of data is moved around, it is only the reference which is moved. This is more efficient than moving the data item itself.

We shall give two example tree procedures: adding an item of data to the tree and printing the tree. We need to check whether the tree at some node is empty. For this, we use the declaration

```
REF TREE leaf = NIL
```

Here is the procedure `add word`:

```
PROC add word = (REF REF TREE root,
                 REF WORD w)VOID:
IF   root IS leaf
THEN root:=HEAP TREE:=(w,leaf,leaf)
ELIF wd OF w < wd OF w OF root
THEN add word(left OF root,w)
ELIF wd OF w > wd OF w OF root
THEN add word(right OF root,w)
ELSE ct OF w OF root+:=1
FI
```

The ordering relation in `add word` is the alphabetical ordering of the string in each data item. When the string in the data item to be added to the tree has been found in the tree, the occurrence number is incremented by 1 (see the `ELSE` clause above). Note the use of recursion.

Printing the tree follows a similar pattern, but when the "root" under consideration is a leaf, nothing happens:

```
PROC print tree=(REF FILE f,
                 REF REF TREE root)VOID:
IF   root ISNT leaf
THEN print tree(f,left OF root);
     put(f,(wd OF w OF root,
            ct OF w OF root,
            newline));
     print tree(f,right OF root)
FI
```

As you can see, recursion is vital here. Although it is true that recursion can be avoided by using a loop, recursion is better because it clarifies the logic.

The allocation and release of memory for linked-lists (including trees) are quite transparent to the program. When a tree is read, and nodes possibly deleted, all the lost memory is collected every so often by a **garbage collector**. You do not have to worry about the details of memory maintenance—it is all done for you by the compiler and the run-time system. If you write a program which relies heavily on global generators, then you should allocate extra memory to the heap (see the on-line information for details of how to use the Algol 68 compilation system).

Exercises

11.24 Write a program which reads a text book and creates a binary tree containing the number of occurrences of each of the letters A–Z and a–z (that is, case is significant). Print a report with the frequency of occurrence represented by a percentage of the total number of letters in the book to 2 decimal places. You should print the letters going downwards with 13 lines for each column: first the upper case letters, then the lower case. Only print lines for those letters which occur in the book (use `mem channel` to build the complete table in memory before printing).Ans

---

11.12 Parallel programming

Unfortunately, the a68toc compiler does not provide parallel programming, so this section has been removed from the third edition of this book.

11.13 Summary

A machine word is accessed using the mode `BITS` and a number of operators. A value of mode `BITS` can be denoted using binary, quaternary (radix 4), octal or hexadecimal digits. Names which refer to names form the basis of self-referential modes (via `STRUCT` and `REF`) from which we can construct queues and trees. Some of the basic procedures were covered which manipulate these data structures.

Chapter 12

Program development

Of course, there is more to writing programs than learning a programming language. Although you will find many books on programming languages, you will not find many on computer programming as such. That is because it is very much a craft. Be aware that this book does not, and cannot, train you to become a professional programmer. Only on-the-job training and experience can do that—but after working through this chapter, you will have an idea of some of the activities a professional programmer does.

In the computer industry, there is a widespread attitude that program maintenance helps build good programmers. There are sound reasons for this. One is that reading other people's programs helps you learn how to lay out programs, how to organise the source, how to write structured code and how to solve the sort of problems that a programmer meets daily. Another reason is that program maintenance usually involves either removing errors (usually called **bugs**) or making small changes to the program to adapt it to changing requirements. You have to learn how a program works before you change it and reading someone else's program means that the philosophy of the program (the approach of the program to solving a problem) is already there—you do not have to create it.

However, there is no substitute for writing your own programs. The first section of this chapter is concerned with how to write your own programs, from problem analysis to documentation. The next topic discusses how to access operating system procedures. This introduces almost all those aspects of Algol 68 which involve direct machine access apart from the mode `BITS` and its associated operators which were covered in chapter 11.

Next, we turn to the first aspect of program maintenance: how to understand a program. A small utility (`lf`) is provided with the Algol68toC compilation system documentation. This section looks at `lf` and analyses its functioning.

12.1 Writing programs

The first stage in the development of a new program consists of analysing the problem that the program must solve. Unfortunately, there is no known method or methodology which will solve any kind of problem. However, a particularly good book on problem solving was written by George Pólya(see the Bibliography) and although the book is geared towards mathematical problems, it will help you solve

most technical problems.

Problem analysis is not usually taught to beginners at computer programming because, so far as we know, it is mainly an intuitive activity (it is a branch of Heuristics). Learning to analyse a problem with the intention of writing a computer program is largely accomplished by writing simple programs followed by programs of increasing sophistication—this is sometimes called "learning by doing". When we start analysing actual programs later in the chapter, each such analysis will be preceded by a problem analysis. You will be able to see how the program, as presented, accords with that analysis.

Nevertheless, even though no definitive method can be given, there are guidelines which help you to appreciate and analyse problems suitable for computer solution. In the field of systems analysis, you will find various methodologies (such as SSADM). These are usually geared towards large-scale systems and are designed to prevent systems designers from forgetting details. In the context of program design, knowing the data to be used by the program and the data to be produced by the program is the principal guide to knowing what manipulations the program must perform. Data knowledge specifies the books accessed by the program and usually constitutes a substantial part of the program's documentation.

Once you know the data your program operates on, you can determine the actual manipulations, or calculations, required. At this stage, you should be able to determine which data structures are suitable for the solution of your problem. The data structures in turn lead you to the mode declarations. The kind of data structure also helps to determine the kind of procedures required. Some examples: if your data structures include a queue, then queue procedures will be needed; or, if you are using multiples (repeated data), then you will almost invariably be using loops. Again, if an input book contains structured data, such as an item which is repeated many times, then again your program will contain a processing loop. The Jackson programming methodology is a useful way of specifying procedures given the data structures to be manipulated (see the bibliography).

12.1.1 Top-down analysis

After you have determined suitable modes and procedures, you need to analyse the problem in a top-down manner. Basically, **top-down analysis** consists of determining the principal actions needed to perform a given action, then analysing each of the principal actions in the same way. For example, suppose we wished to write a program to copy a book whose identifier is given on the command line. The topmost statement of the problem could be

```
copy an identified book
```

The next stage could be

```
get the book identifier
open the book
establish the output copy book
copy the input book to output
close both books
```

At this stage, the process "copy the input book to output" will depend on the structure of the input book. If it is text, with lines of differing length, you could

use a name of mode `REF STRING`. If the book contains similar groupings of data, called **records**, then it would be more appropriate to declare a structured mode and write appropriate input and output procedures:

```
DO
   get record from input book
   put record to output book
OD
```

The analysis is continued until each action can be directly coded.

12.1.2 Program layout

Before you start coding the program (writing the actual Algol 68 source program), you should be aware of various programming strategies besides the different means of manipulating data structures. The first to address is the matter of source program layout.

In the examples given in this book, code has been indented to reflect program structure, but even in this matter, there are choices. For example, some people indent the `THEN` and `ELSE` clauses of an `IF` clause:

```
IF ...
  THEN ...
  ELSE ...
FI
```

instead of

```
IF   ...
THEN ...
ELSE ...
FI
```

Others regard the parts of the `IF` clause as some kind of bracketing:

```
IF
   ...
THEN
   ...
ELSE
   ...
FI
```

Some people write a procedure as:

```
PROC ...
   BEGIN
   ...
   END
```

Others never use `BEGIN` and `END`, but only use parentheses.

Another point is whether to put more than one phrase on the same line. And what about blank lines—these usually improve a program's legibility. Whatever you decide, keep to your decision throughout the program (or most of the program) otherwise the format of the code may prove confusing. Of course, you will learn by your mistakes and usually you will change your programming style over the years.

12.1.3 Declarations

Another matter is whether to group declarations. Unlike many programming languages, Algol 68 allows you to place declarations wherever you wish. This does not mean that you should therefore sprinkle declarations throughout your program, although there is something to be said for declarations being as local as possible. There are also advantages in grouping all your global declarations so that they can be found easily. Generally speaking, it is a good idea to group all global names together (those in the outermost range) and within that grouping, to declare together all names which use the same base mode (for example, group declarations of modes `CHAR`, `[]CHAR` and `STRING`). Some of the exercises in this book only declare names when they are immediately followed by related procedures. If your program needs many global names, it makes sense to declare them near the beginning of the program, after mode declarations, so that if subsequent changes are required, you know that all the global name declarations are together and therefore you are unlikely to miss any.

12.1.4 Procedures

The next consideration is breaking your code into procedures. As you analyse the problem, you will find that some of the processing can be specified in a single line which must be analysed further before it can be directly coded. Such a line is a good indication that that process should be written as a procedure. Even a procedure which is used once only is worth writing if the internal logic is more than a couple of conditional clauses, or more than one conditional clause even.

You also have to decide between repeating a procedure in a loop, or placing the loop in the procedure. Deciding the level at which logic should be put in a procedure is largely the product of experience—yours and other people's—another reason for maintaining existing programs.

When you have decided where to use procedures, you should then consider the interface between the procedure and the code that calls it. What parameters should it have, what yield, should you use a united mode for the yield, and so on. Try to have as few parameters as possible, but preferably use parameters rather than assign to names global to the procedure. The design of individual procedures is similar to the design of a complete program.

When you are coding a procedure, be especially careful with compound Boolean formulæ. From experience, this is where most mistakes arise. If you are writing a procedure which manipulates a linked list, draw a diagram of what you are trying to do. That is much easier than trying to picture the structures in your head.

12.1.5 Monetary values

Problems can arise when dealing with money in computer programs because the value stored must be exact. For this reason, it is usually argued that only integers should be used. In fact, real numbers can be used provided that the precision of the mantissa is not exceeded. Real numbers are stored in two parts: the mantissa, which contains the significant digits of the value, and the exponent, which multiplies that value by a power of 2. In other words, using decimal arithmetic, the number $3 \cdot 14159 \times 10^{-43}$ has $3 \cdot 14159$ as a mantissa and -43 as an exponent. Because

real numbers are stored in binary (radix 2), the mantissa is stored as a value in the range $1 \leq \text{value} < 2$ with the exponent adjusted appropriately.

There are a number of identifiers declared in the standard prelude, known as **environment enquiries**, which serve to determine the range and precision of real numbers. The `real precision` is the number of bits used to store the mantissa, while the value `max exp real` is the maximum exponent which can be stored for a binary mantissa (*not* the number of bits, although it is a guide to that number). The `real width` and `exp width` say how many decimal digits can be written for the mantissa and the exponent. The values `max real` and `min real` are the maximum and minimum real numbers which can be stored in the computer. All these values are specified by the IEEE 754–1985 standard on "Binary Floating-Point Arithmetic" which is implemented by most microprocessors today.

The value of `real width` is 15 meaning that 15 decimal digits can be stored accurately. Leaving a margin of safety, we can say that an integer with 14 digits can be stored accurately, so that the maximum amount is

$$99,999,999,999,999$$

units. If the unit of currency is divided into smaller units, such as the sterling pound into pence, or the dollar into cents, then the monetary value should be stored in the smaller unit unless it is known that the smaller unit is not required. Thus the greatest sterling amount that can be handled would appear to be £999,999,999,999.99.

However, Algol 68 allows arithmetic values to be stored to a lesser or greater precision. The modes `INT`, `REAL`, `COMPL` and `BITS` can be preceded by any number of `SHORT`s or `LONG`s (but not both). Thus

```
LONG LONG LONG REAL r;
```

is a valid declaration for a name which can refer to an exceptionally precise real. When declaring identifiers of other precisions, denotations of the required precision can be obtained by using a cast with the standard denotation of the value as in

```
LONG REAL lr = LONG REAL(1);
```

One alternative is to use `LONG` with the denotation:

```
LONG REAL lr = LONG 1.0;
```

Another is to use the `LENG` operator, which converts a value of mode `INT` or `REAL` to a value of the next longer precision, as in

```
LONG REAL lr = LENG 1.0;
```

`SHORTEN` goes the other way.

```
SHORT SHORT INT ssi = SHORTEN SHORTEN 3;
```

All the arithmetic operators are valid for all the `LONG` and `SHORT` modes. Although you can write as many `LONG`s or `SHORT`s as you like, any implementation of Algol 68 will provide only a limited number. The number of different precisions available is given by some identifiers in the standard prelude called environment enquiries. They are

- `int lengths`
- `int shorths`
- `real lengths`
- `real shorths`
- `bits lengths`
- `bits shorths`

The values for complex numbers are the same as those for reals. For integers, where `int lengths` is greater than `1`, `long max int` and so on are also declared, and similarly for `short max int`. If `int lengths` is `1`, then only the mode `INT` is available.

For the a68toc compiler

```
int lengths=2
int shorths=3
```

Thus it is meaningful to write

```
LONG INT long int:=long max int;
INT int:=max int;
SHORT INT sh int:=short max int;
SHORT SHORT INT sh sh int:=
         short short max int;
```

The same applies to the mode `BITS`. Try writing a program which prints out the values of the environment enquiries mentioned in this section. The transput procedures `get`, `put`, `get bin` and `put bin` all handle the available `LONG` and `SHORT` modes.

Although you can still write

```
LONG LONG INT lli=LONG LONG 3;
```

the actual value created may not differ from `LONG INT` depending on the value of `int lengths`. Note that you cannot transput a value which is not covered by the available lengths/shorths. Use `LENG` or `SHORTEN` before trying to transput.

For monetary values, `LONG INT` is available with the value of `long max int` being

```
9,223,372,036,854,775,807
```

which should be big enough for most amounts.

12.1.6 Optimisation

There are two well-known rules about optimisation:

1. Don't do it.
2. Don't do it now.

However, often there is a great temptation to optimise code, particularly if two procedures are very similar. Using identity declarations is a good form of optimisation because not only do they save some writing, they also lead to more efficient code. However, you should avoid procedure optimisation like the plague because it usually leads to more complicated or obscure code. A good indicator of bad optimisation is the necessity of extra conditional clauses. In general, optimisation is never a primary consideration: you might save a few milliseconds of computer time at the expense of a few hours of programmer time.

12.1.7 Testing and debugging

When writing a program, there is a strong tendency to write hundreds of lines of code and then test it all at once. Resist it. The actual writing of a program rarely occupies more than 30% of the whole development time. If you write your overall logic, test it and it works, you will progress much faster than if you had written the whole program. Once your overall logic works, you can code constituent procedures, gradually refining your test data (see below) so that you are sure your program works at each stage. By the time you complete the writing of your program, most of it should already be working. You can then test it thoroughly. The added advantage of step-wise testing is that you can be sure of exercising more of your code. Your test data will also be simpler.

The idea behind devising test data is not just giving your program correct data to see whether it will produce the desired results. Almost every program is designed to deal with exception conditions. For example, the `lf` program has to be able to cope with blank lines (usually, zero-length lines) so the test data should contain not one blank line, but also two consecutive blank lines. It also has to be able to cope with extra-long lines, so the test data should contain at least one of those. Programs which check input data for validity need to be tested extensively with erroneous data.

It is particularly important that you test your programs with data designed to exercise boundary conditions. For example, suppose the creation of an output book fails due to a full hard disk. Have you tested it, and does your program terminate sensibly with a meaningful error message? You could try testing your program with the output book being created on a floppy disk which is full.

Sometimes a program will fault with a run-time error such as

```
Run time fault (aborting):
Subscript out of bounds
```

or errors associated with slicing or trimming multiples. A good way of discovering what has gone wrong is to write a monitor procedure on the lines of

```
PROC monitor=(INT a,
              []UNION(SIMPLOUT,
                      PROC(REF FILE)VOID)r
              )VOID:
BEGIN
   print(("*** ",whole(a,0)));
   print(r)
END
```

and then call `monitor` with an identifying number and string at various points in the program. For example, if you think a multiple subscript is suspect, you could write

```
monitor(20,("Subscript=",whole(subscript,0)))
```

By placing monitors at judicious points, you can follow the action of your program. This can be particularly useful for a program that loops unexpectedly: monitors will tell you what has gone wrong. If you need to collect a large amount of monitors, it is best to send the output to a book. The disadvantage of this is that the operating system does not register a book as having a size until it has been closed after creating. This means that if your program creates a monitoring book, writes a large amount of data to it and fails before the book is closed, you will not be able to read any of the contents because, according to most operating systems, there will not be any contents. A way round this problem is to open the book whenever you want to write to it, position the writing position at the end of the book, write your data to it and then close the book. This will ensure that the book will have all the executed monitors (unless, of course, it is a monitor which has caused the program to fail!). The procedure `debug` given in section 9.9 will do this.

An alternative method of tracing the action of a program at run-time is to use a source-level debugger. The `ddd` or `gdb` programs can help you debug the C source program produced by the a68toc compiler, but unless you understand the C programming language and the output of the a68toc compiler, you will not find it useful. Monitors, although an old-fashioned solution to program debugging, are still the best means of gathering data about program execution.

Another proven method of **debugging** (the process of removing bugs) is **dry-running**. This involves acting as though you are the computer and executing a small portion of program accordingly. An example will be given in the analysis of the `lf` program later.

Sometimes, no matter what you do, it just seems impossible to find out what has gone wrong. There are three ploys you can try. The first, and easiest, is to imagine that you are explaining your program to a friend. The second is to actually explain it to a friend! This finds most errors. Finally, if all else fails, contact the author.

12.1.8 Compilation errors

You can trust the compiler to find grammatical errors in your program if any are there. The compiler will not display an error message for some weird, but legal, construction. If your program is syntactically correct (that is, it is legal according to the rules of the language), then it will parse correctly.

When compiling a program of more than a hundred lines, say, you can use the parsing option (`-check`) which will more than double the speed of compilation. When your program parses without error, then it is worth doing a straight compilation (see the online documentation for program `mm` in the a68toc compilation system).

A definitive list of error messages can be found in the compiler source code file

```
algol68toc-1.xx/src/message.a68
```

You will find that most of the messages are easy to understand. Occasionally, you will get a message which seems to make no sense at all. This is usually because the actual error occurs much earlier in your program. By the time the compiler has discovered something wrong, it may well have compiled (or tried to compile) several hundred lines of code. A typical error of this sort is starting a comment and not finishing it, especially if you start the comment with an opening brace (`{`), which gives rise to the following error message:

```
ERROR (112) end of file inside
   comment or pragmat
```

If you start a comment with a sharp (`#`) and forget to finish it likewise, the next time a sharp appears at the beginning of another comment, the compiler will announce all sorts of weird errors.[1]

Another kind of troublesome error is to insert an extra closing parenthesis or `END`. This can produce lots of spurious errors. For example:

```
ERROR (118) FI expected here
               (at character 48)
ERROR (203) ELSE not expected here
               (at character 4)
ERROR (140) BOOL, INT or UNION required here,
               not VOID
ERROR (116) brackets mismatch
               (at character 2)
ERROR (159) elements of in-parts
               must be units
ERROR (117) FINISH expected here
               (at character 3)
```

Omitting a semicolon, or inadvertently inserting one will also cause the appearance of curious error messages. Messages about `UNION`s usually mean that you should use a cast to ensure that the compiler knows which mode you mean. If, for example, you have a procedure which expects a multiple of mode

```
[]UNION(STRING,[]INT)
```

and you present a parameter like

```
((1,2),(4,2),(0,4))
```

then the compiler will not know whether the display is a row-display or a structure-display. Either you should precede it with a suitable mode, or modify your procedure to take a single `[]INT` and loop through it in twos. Having to modify your program because the compiler does not like what you have written is rare however.

[1] One way of avoiding this sort of error is to use "lexical" highlighting with your favourite editor. A missing quote or sharp will cause large amounts of your program to be displayed as a string denotation or a comment.

12.1.9 Arithmetic overflow

Sometimes your program will fail at the time of elaboration or "run-time" due to arithmetic overflow. If, during a calculation, an intermediate result exceeds the capacity of an `INT`, no indication will be given other than erroneous results.

On some platforms, overflow of `REAL` numbers can be detected by the floating-point unit. The standard prelude contains the value `fpu cw algol 68 round` of mode `SHORT BITS` and the procedure

```
PROC set fpu cw = (SHORT BITS cw)VOID:
```

The small test program `testov` (to be found with the a68toc compilation system documentation) illustrates testing for overflow both with integers and real numbers.

12.1.10 Documentation

The most tedious aspect of writing a program is documenting it. Even if you describe what the program is going to do before you write it, but after you have designed it, documentation is not usually a vitally interesting task. Large programming teams often have the services of a technical writer whose job it is to ensure that all program documentation is completed.[2]

Existing programs are usually documented and there is no doubt that the best way of learning to document a program is to see how others have done it. There are several documentation standards in use, although most large companies have their own. Generally speaking, the documentation for a program should contain at least the following

- the program name
- the language used to write the program
- a short description of what the program does
- the details of all books used by, or produced by, the program, including the screen and the printer
- an analysis of how the program works, particularly any special algorithms or data structures (queues and trees are examples)
- who wrote the program, and when
- the location of the source code
- the latest listing of the source code

but not necessarily in the order given above. The aim of program documentation is to make it easy to amend the program, or to use it for a subsequent rewrite.

Lastly, it is worthwhile saying "don't be rigid in program design". If, as you reach the more detailed stages of designing your program, you discover that you have made a mistake in the high-level design, be willing to backtrack and revise it. Design faults are usually attributable to faulty analysis of the problem.

[2]Various schemes have been developed for documenting a program as it is written, They are often called "literate programming".

12.2 Non-canonical input

The `noncanon` program provides a means of entering data via the keyboard without echoing it to the screen. This is known as non-canonical input mode, the usual echoing of input being canonical input mode. The general details of terminal control are very complex, but simple access has been provided with the `kbd channel`.

Here is a sample program which may be used to test the effect of `kbd channel`:

```
PROGRAM noncanon CONTEXT VOID
USE standard
BEGIN
   STRING password;
   FILE kbd;  open(kbd,"",kbd channel);
   WHILE
      CHAR ch;  get bin(kbd,ch);
      ch /= REPR lf
   DO
      password+:=ch;
      print("*")
   OD;
   close(kbd);
   print(("You entered [",
          password,"]",
          newline))
END
FINISH
```

Notice that the program cannot be aborted by pressing `^C`. Ensure you close the `FILE` opened with the `kbd channel` after use otherwise you'll find all your commands at the command prompt unechoed. If that happens, issue the following command at the prompt:

```
$ stty sane
```

when normal echoing will be restored.

12.3 A simple utility

When you are writing computer programs, it is very useful to be able to copy your Algol 68 source programs to a printer with line numbers. Many editors, including `vim`, `Emacs` and `FTE`, use line numbers. When the Algol 68 compiler finds an error in your program, it displays the offending line together with its number and a descriptive message on the screen and the number of the character in the line where the error occurred. However, it is insufficient to merely copy the contents of a file to the printer (unless you are using the spooling facility of a header file) because the output will not contain any identifying information.

What is required is a small program which will optionally write line numbers and which will write the name of the file being printed together with the date and time at which the file was last modified. A page number is another useful item as it prevents pages being lost when the listing is made on separate sheets of paper.

It would also be very useful to be able to specify where in a file a listing should start and where it should finish. Such a program is called a **utility**. Notice that the program must be able to handle zero-length lines as well as lines which are too long to be printed on one line alone. Lastly, some editors allow you to insert tab characters into your document, so the utility must be able to print the file with the correct indentation.

The preceding problem analysis shows that we could write such a program if we knew how to obtain the date and time of last modification of a file from the operating system. In the directory `/usr/share/doc/algol68toc/`, you will find the source of the program `lf` which solves the problem described above for the Linux operating system. The source of `lf` is 520 lines long. Compile it and run it with the argument `-h`. The help information displayed by the program should be displayed by every program you write which is used at the command line: it prevents accidental use from causing damage to your operating system files or directories.

12.3.1 The source code

There are many ways of tackling the understanding of a program, but here is a method which does help with Algol 68 programs. In summary,

1. See what the program does.
2. Look at the principal processing.
3. Examine the mode declarations.
4. Examine the routines.
5. Repeat steps 2–4 for each routine.

Stage one of examining a program is to see what it does. Examples of its output, and possibly its input, help you to identify the actions of various parts of the program. Documentation of the input and output would suffice, but neither exists in this case because the input is a plain text file and the output is better seen than described. Compile the Algol 68 example program `lf` in

```
/usr/share/doc/algol68toc/
```

and use it to list the file `test.lf` (in the same directory) with line numbers on your printer using the command

```
lf -pg -n test.lf | lpr
```

to pipe the output to the printer unless you have a LaserJet 4 or 6L when you can omit the `-pg` argument. Notice that the time and date the file was last modified appears at the top of each page, together with the identifier of the file and the page number. If you used the `-n` parameter to print the test file, each line will be preceded by a line number and a colon. If you did not list the file with line numbers, do so now because the line numbers will highlight another feature of the program. The first line in `test.lf` is too long to be printed on one line, so the program breaks it into two parts. The second part does not have a line number since it is part of the same line in the input.

The second stage in understanding a program is to look at the principal processing. Since procedures and other values must be declared before use in the a68toc compiler, the last part of the program contains the main processing logic. Now print (or display) the source of `lf.a68` using the command

```
lf -n /usr/share/doc/algol68toc/lf.a68
```

In the source, the main processing logic is on lines 427–517. Examine those lines now.

Before processing any command line arguments, the program defines the actions to take when the last argument has been read. In other words, what should be done when the logical end of file has been reached for `comm line`. The default action is to terminate the program immediately with a suitable error message. In `lf`, no identification is given for `comm line` in the `open` procedure, because it isn't relevant, but if you insert such an identification, for example, `command line file`, then any error message issued by the transput system will include it. Notice that although the anonymous procedure used as the second parameter for `on logical file end` on line 448 occurs within the `IF ... FI` clause, because it is a denotation (a procedure denotation) it has global scope. That is one of the reasons why anonymous procedures are so useful. Also note the use of `SKIP` to yield a value of mode `BOOL`: in fact, it will never be used because `stop` is a synonym for `GOTO end of program`.

In lines 442–517, the program processes the command line argument by argument. If an argument starts with "-" it is assumed to be an option otherwise it is assumed to be a filename. Note the use of `skip terminators` to skip spaces in the command line. Options that require a number (`-s` and `-t`) expect it to follow the option directly (see lines 493 and 495). Lines 500–506 process a solitary `-` to mean "list the standard input". Lines 507–516 process a named file. As you examine the code, underline the identifiers of all procedure calls.

The next stage in understanding a program is to look at all the mode declarations. There are three in this program: `PRINTER`, `SEC` and `STAT`. You should scan the program to see what identifiers have that or a related mode and where they are used.

12.3.2 Routines

Finally, you need to examine the routines declared. It is a good idea, especially in a more complicated program, to list the identifiers of all procedures with nested declarations of procedures indented under their parent procedure identifiers. This helps to fix the structure of the program in your mind. Then you should examine the procedures used in the main processing loop. In `lf`, they are:

```
char in string      close
disp error          get
get mtime           get numeric arg
get sections        help
open                print
print file          process file name
reset parameters    skip terminators
```

When you examine each procedure, do the same as you did for the whole program: first the main logic, then the modes, then the procedures and operators. You will need to backtrack several times in a large program. If a lot of names are declared, prepare a list together with a description of what each name is used for, where it is declared and the places where it is used. A cross-reference program would be really useful, but it is not a simple program to write for Algol 68.

The principle processing is performed by the procedure `print file` on lines 258–322. Firstly, tab stops are set according to the current value of `tabs`, then `lines` is initialised and an initialisation string output to the printer. If letter quality has been chosen (option `-q`), a special string is sent to the printer accordingly. Then the `logical file end` event procedure is set. Each section specified on the command line (or the default section if no sections were specified) is then printed using the procedure `do line`. Each line is input using `get line` whose principal function is to expand tab characters to the required number of spaces (3 unless set by the `-t` option). Lines are not output until the `beg OF ss` line is reached (1 unless set by the `-s` option). Notice the code following `FROM` in the preamble to the inner `DO ... OD` loop (on lines 313–316) which ensures that the file is reset if the sections to be printed are not ordered (the definition of ordered is in the procedure `get sections` (lines 381–425).

Similar to your list of nested procedures, prepare a list of procedures where indented procedures identify procedures called by the parent procedure. Here is part of the list for `lf`:

```
fstat
   linux fstat
help
   exit, newline, put
reset parameters
lf print
   ODD, print
get mtime
   fstat, linux ctime
get sections
   +:=
   add section
      char in string
      get numeric arg
   char in string
```

12.3.3 Dry-running example

The procedure `get line` (lines 232–250) and its associated procedures `set tabs` (lines 220–224) and `tab pos` (lines 226–227) are worth examining in detail. The best way to see how they work is to dry-run them. Take a blank sheet of paper and make a vertical list of all the names, both local and global, used by the procedures. Opposite `in line`, write a piece of text containing tab characters (a piece of indented program, for example). Then work your way through the procedure, marking the value referenced by each name as you complete each step. You should also note the value of each non-name; for example, the loop identifier `i`. Here is what your list could look like after going 3 times round the outer loop (the inner loop is on lines 241–244):

tabstops	FFTFFTFFTFFTFFTFFTFFTFFT...
line(ln)	␣␣␣␣T
in line	→␣`THEN␣ch:="A"`
op	*1 2 3 4 5* 6

i	1 2 3
c	␣ T

Struck-out values have been superceded and ␣ denotes a space. Dry-running is a very useful method, if laborious and time-consuming, of finding bugs. `tab ch` is declared in the standard prelude.

This utility program (`lf`) is quite short, but we have analysed its working in detail so that you can see how it is done.

12.3.4 `ALIEN` procedures

The utility `lf` uses some of the extensions provided by the a68toc compiler, in particular, the `ALIEN` construct which provides access to procedures compiled by other compilers. In this section we shall look at the `get cwd` and the `fstat` procedures.

The procedure `fstat`

The procedure `fstat` is on lines 100–105. It depends on a call of the `linux fstat` procedure whose second parameter is a name referring to a value of mode `STAT`. The declaration of `STAT` is on lines 24–41.

If you investigate the file `/usr/include/bits/stat.h`, you will find the C definition of the `stat` structure therein. The `STAT` mode accurately reflects this structure using `LONG` or `SHORT` as appropriate. Briefly, a C `unsigned int` is equivalent to an Algol 68 `BITS`. For historical reasons, the C `unsigned long int` has the same meaning as an `unsigned int` so `BITS` could have been used for those fields as well. However, because the value is required as an integer (and is stored as a positive integer), it is possible to regard them as having mode `INT`. Some of the C modes[3] are hidden by further mode declarations[4], but if you hunt for `__dev_t` you will find it is an `unsigned long long int` which is equivalent to the Algol 68 `LONG BITS` or, as is used in `STAT`, `LONG INT`.

Now look at the declaration of `linux fstat` on lines 85–89. Most of this construction is C source code. The `ALIEN` construct may be written as

```
<mode> <identifier> = ALIEN "<symbol>"
   "<C source code>";
```

where the angle brackets denote items to be replaced. In the declaration for `linux fstat` we have

- `<mode> = PROC(INT,REF STAT)INT`
- `<identifier> = linux fstat`
- `<symbol> = FSTAT`

followed by three lines of C source code. It is not my intention to delve into the mysteries of C. If you don't understand that language, consult someone who does. However, the point of the declaration is to map the Algol 68 modes onto the C equivalents. The C procedure `fstat` takes two parameters: the first has mode `int` (equivalent to `INT`) and the second of mode `struct stat*` which is

[3] C people call them `types`.
[4] `typedef`s

equivalent to `REF STAT`. The cast in C consists of a mode in parentheses (compare with the Algol 68 cast in section 10.5) so the third line of C code ensures that the second parameter of the Algol 68 procedure `linux fstat` has the right mode. The `A_int_INT(...)` construct is a C language macro[5] for a cast which ensures that the yielded C integer is equivalent to the Algol 68 `INT`. If you want to see what the a68toc compiler generates, look for `FSTAT` in the file `lf.c`.

Reverting to line 102, the field `sys file OF f` has the correct mode for use as the "file descriptor" for `fstat`. You should check the manual page of `fstat` (in section 2 of the Linux Programming Manual) for details of its functioning and yield.

The procedure `get cwd`

The procedure `get cwd` is more complicated because it uses several facilities provided by the standard prelude as well as another extension provided by the a68toc compiler. Firstly, look at the `ALIEN` declaration of `linux getcwd` on lines 91–93. The mode `VECTOR[]CHAR` is similar to the mode `[]CHAR`, but the lower bound is always 1 and is omitted from the generated construct. In fact, a68toc translates this mode into the C equivalent of

```
STRUCT(REF CHAR data, INT gc, upb)
```

The `gc` field is an integer provided for the garbage-collector (the run-time memory management system which looks after the heap). The `data` field is a reference to the actual data (in fact it is a memory address)[6].

The C procedure `getcwd` requires two parameters: a reference to an area which it can use to return the full path of the current working directory and an integer which states how big that area is. The C source code in the declaration for `linux getcwd` contains the C macro

```
A_VC_charptr(buf)
```

which expands into `buf.data` (equivalent to the Algol 68 expression `data OF buf`) and the C macro `A_INT_int` which converts an Algol 68 `INT` into a C `int` (directly equivalent on Linux).

The yield of `linux getcwd` is a reference to the area in which the current working directory path has been put. Strictly speaking, this is identical to the first parameter of the C procedure `getcwd`, but the GNU C compiler complains if it is used as such. To get around this, the author used the cast `(void *)` which effectively causes the reference to be a reference to an anonymous piece of memory. The Algol 68 equivalent is `CPTR` which is defined in the standard prelude as `REF BITS`.

Now comes the clever bit. Look at line 98. The value of mode `CPTR` (`REF BITS`) is converted by the operator `CPTRTOCSTR` into a value of mode `CSTR` (declared in the standard prelude as `REF STRUCT 16000000 CHAR`). Now look at the definition of that operator (on line 95)! `BIOP` stands for "built-in operator" and `BIOP 99` is the only built-in operator implemented by the a68toc translator. `BIOP 99` maps its parameter (of one mode) onto its yield (of another mode). It effectively acts as a cast

[5] A synonym for another piece of text which is expanded by the C preprocessor

[6] The `VECTOR` mode is not limited to `CHAR`. You can use it for any mode. See section 13.5.1 for details

(in this case) from one `REF` mode to another `REF` mode. Have a look at the C source code in `lf.c` if you are interested in the details. Then the value of mode `CSTR` is converted using the operator `CSTRTORVC` to a value of mode `REF VECTOR[]CHAR` which is dereferenced and then coerced to a value of mode `STRING`. In fact, the a68toc compiler will silently coerce values of mode `REF STRUCT i MODE` to mode `REF VECTOR[]MODE` and thence to `REF[]MODE`. Notice that you cannot coerce a value of mode `REF VECTOR[]MODE` to `REF FLEX[]MODE`. The mode `STRING` has no flexibility (it is equivalent to `[]CHAR`).

Lastly, note that the parameter of `linux getcwd` is an anonymous `VECTOR[]CHAR` whose scope is limited to the scope of `get cwd` (the Algol 68 procedure).

If you want to examine the other macros used for the translated C source, have a look at the files either of the directories (depending on platform):

```
/usr/include/algol68/
/usr/local/include/algol68
```

12.4 Summary

In this chapter, we have covered most of the activities relating to program development, whether it be the maintenance of existing programs or the development of new programs. The constructor `ALIEN` is used to introduce procedures compiled by other compilation systems (such as C). We have described one program and have shown how to analyse the workings of a program.

Chapter 13

Standard Prelude

The function of this chapter is to describe all the facilities in the standard prelude supplied with the Linux port of the a68toc compiler. The standard prelude contains both implicit declarations (facilities provided by the compiler) and explicit declarations (those defined in, and made available by, the QAD standard prelude[1]). They are classified and dealt with as follows:-

1. **Standard modes**
   These are the modes defined by the Algol 68 Revised Report, which defines the language, plus modes required by the transput.
2. **Environment enquiries**
   Some of these are defined in the Revised Report.
3. **Standard operators**
   There are a large number of these, all defined in the Revised Report and classified by the modes of their operands. They are preceded by a subsection giving their priorities.
4. **Other operators**
   Some operators are provided which are not in the Revised Report. They are described in this section. However, operators peculiar to the a68toc implementation are described in the section on a68toc extensions.
5. **Standard procedures**
   Only those procedures not used in transput and process control are defined here. They all appear in the Revised Report.
6. **Other procedures**
   Procedures which appear neither in the Revised Report nor in any other section appear here.
7. `ALIEN` **declarations** This section includes all the `ALIEN` declarations made available by the standard prelude.

[1] QAD stands for "quick-and-dirty" and was supposed to represent the provided standard prelude. While it is not entirely standard (as far as Algol 68 is concerned), it was certainly not implemented quickly!

8. **a68toc extensions**
   All the extensions to the language are described in this section including modes, constructs, operators and procedures.
9. **Process control**
   These declarations provide control over the working of the floating-point unit, integer overflow and signal handling. They include declarations for controlling the Algol 68 garbage collector.
10. **Transput**
    This very large section provides specifications for all the transput declarations available in the Standard Prelude, but omits those operators and procedures which are intended for internal use only.

See the bibliography for details of the Revised Report.

13.1 Standard modes

Many of the modes available in the standard prelude are built from the standard modes of the language which are all defined in the Revised Report.

1. `VOID`
   This mode has one value: `EMPTY`. It is mainly used as the yield of routines and in unions.
2. `BOOL`
   This mode has two values, namely `TRUE` and `FALSE`.
3. `INT`
   This is the basic arithmetic mode. Various precisions are available:-
   (a) `LONG INT` 64-bit integer
   (b) `INT` 32-bit integer
   (c) `SHORT INT` 16-bit integer
   (d) `SHORT SHORT INT` 8-bit integer
4. `REAL`
   This mode is used mainly for approximate calculations although exact values can be manipulated provided that the number of significant digits does not exceed the precision of the mantissa (see section 13.2.1). The following precisions are available:-
   (a) `REAL` 64-bit real
   (b) `SHORT REAL` 32-bit real
5. `COMPL`
   Strictly speaking, this is not a fundamental mode because it is regarded as a structure with two fields:-

   ```
   MODE COMPL = STRUCT(REAL re,im)
   ```

   However, the widening coercion will convert a `REAL` value into a `COMPL` value and `COMPL` values are not straightened (see section 13.7.6). Like `REAL`s, the following precisions are available:-

(a) `COMPL` 128-bit

(b) `SHORT COMPL` 64-bit

6. `CHAR`
   This mode is used for most character operations. See section 13.2.2 for further details.

7. `BITS`
   This mode is equivalent to a computer word regarded as a group of bits (binary digits) numbered 1 to `bits width` (see section 13.2.1). Various precisions are available:-

   (a) `LONG BITS` 64-bit

   (b) `BITS` 32-bit

   (c) `SHORT BITS` 16-bit

   (d) `SHORT SHORT BITS` 8-bit

8. `BYTES`
   The Revised Report describes the mode, but the a68toc compiler does not implement it.

9. `STRING`
   This mode is defined as

   ```
   MODE STRING = FLEX[1:0]CHAR
   ```

   and is provided with a shorthand construct for denotations of such values (see section 3.1).

13.2 Environment enquiries

Algol 68 was the first programming language to contain declarations which enable a programmer to determine the characteristics of the implementation. The enquiries are divided into a number of different groups. The actual values of the Linux port of the a68toc compiler are placed in square brackets. Those defined in the Revised Report are marked with (RR).

13.2.1 Arithmetic enquiries

These enquiries are so numerous that they are further subdivided.

Enquiries about precisions

Any number of `LONG` or `SHORT` can be given in the mode specification of numbers, but only a few such modes are distinguishable in any implementation. The following environment enquiries tell which modes are distinguishable. Note particularly that there are more distinguishable precisions available for `INT` and `BITS` than there are for `REAL` and `COMPL` in the a68toc implementation.

1. `INT int lengths` (RR) [2]
   1+ the number of extra lengths of integers.

2. `INT int shorths` (RR) [3]
   1+ the number of short lengths of integers.
3. `INT real lengths` (RR) [1]
   1+ the number of extra lengths of real numbers.
4. `INT real shorths` (RR) [2]
   1+ the number of short lengths of real numbers.
5. `INT bits lengths` (RR) [2]
   1+ the number of extra lengths of `BITS`.
6. `INT bits shorths` (RR) [3]
   1+ the number of short lengths of `BITS`.
7. `INT bytes lengths` (RR) [0]
   Bytes are not implemented by the a68toc compiler.
8. `INT bytes shorths` (RR) [0]
   Bytes are not implemented by the a68toc compiler.

Enquiries about ranges

1. `SHORT SHORT INT short short max int` (RR) [127]
   The maximum value of mode `SHORT SHORT INT`.
2. `SHORT INT short max int` (RR) [32 767]
   The maximum value of mode `SHORT INT`.
3. `INT max int` (RR) [2 147 483 647]
   The maximum value of mode `INT`.
4. `LONG INT long max int` (RR)
   [9 223 372 036 854 775 807]
   The maximum value of mode `LONG INT`.
5. `SHORT REAL short min real` [0.117 55$e-37$]
   The smallest representable short real. It should not be confused with `short small real`.
6. `SHORT REAL short max real` (RR) [0.340 28$e+39$]
   The largest short real value storable.
7. `SHORT REAL short small real` (RR)
   [1.192 09$e-7$]
   The smallest short real which, when added to 1.0 makes a discernible difference.
8. `REAL min real` [0.197 626 258 336 50$e-322$]
   The smallest representable real. It should not be confused with `small real`.
9. `REAL max real` (RR) [0.179 769 313 486 23$e+309$]
   The largest real value storable.
10. `REAL small real` (RR) [0.222 044 604 925 031$e-15$]
    The smallest real which, when added to 1.0, makes a discernible difference.
11. `INT bytes per bits` (RR) 0

Internal sizes of modes

1. `INT short short int width` [3]
   The maximum number of decimal digits expressible by a value of mode `SHORT SHORT INT`.
2. `INT short int width` [5]
   The maximum number of decimal digits expressible by a value of mode `SHORT INT`.
3. `INT int width` [10]
   The maximum number of decimal digits expressible by a value of mode `INT`.
4. `INT long int width` [19]
   The maximum number of decimal digits expressible by a value of mode `LONG INT`.
5. `INT short short bits width` (RR) [8]
   The number of bits required to hold a value of mode `SHORT SHORT BITS`.
6. `INT short bits width` (RR) [16]
   The number of bits required to hold a value of mode `SHORT BITS`.
7. `INT bits width` (RR) [32]
   The number of bits required to hold a value of mode `BITS`.
8. `INT long bits width` (RR) [64]
   The number of bits required to hold a value of mode `LONG BITS`.
9. `INT bytes width` (RR) [0]
   The mode `BYTES` is not implemented.
10. `INT short real precision` [24]
    The number of bits used for the mantissa of a short real.
11. `INT short real width` [6]
    The maximum number of significant decimal digits in a small real.
12. `INT short min exp` [−125]
    The minimum exponent of a short real.
13. `INT short max exp` [128]
    The maximum exponent of a short real.
14. `INT short exp width` [2]
    The maximum number of decimal digits in the exponent of a short real. This can be less than the number of digits occupied by `short max exp` because any decimal digit can be represented. For example, 99 but not 999.
15. `INT real precision` [53]
    The number of bits used for the mantissa of a real.
16. `INT real width` [15]
    The maximum number of significant decimal digits in a real.
17. `INT min exp` [−1021]
    The minimum exponent of a real.

18. `INT max exp` [1024]
    The maximum exponent of a real.
19. `INT exp width` [3]
    The maximum number of decimal digits in the exponent of a real. See also `short exp width`.

Sizes used in binary transput

These values give the sizes of each mode when transput using `put bin` or `get bin`.

1. `INT long bits bin bytes` [8]
2. `INT bits bin bytes` [4]
3. `INT short bits bin bytes` [2]
4. `INT short short bits bin bytes` [1]
5. `INT bool bin bytes` [4]
6. `INT long int bin bytes` [8]
7. `INT int bin bytes` [4]
8. `INT short int bin bytes` [2]
9. `INT short short int bin bytes` [1]
10. `INT real bin bytes` [8]
11. `INT short real bin bytes` [4]
12. `INT compl bin bytes` [16]
13. `INT short compl bin bytes` [8]
14. `INT char bin bytes` [1]

Particular arithmetic values

1. `REAL infinity`
   Defined by the C mathematics library as `HUGE_VAL`.
2. `SHORT REAL short pi` [3.14159]
3. `REAL pi` [3.141 592 653 589 79]
4. `REAL log2` [0.301 029 995 663 981]
   This is the value of $\log_{10} 2$.

13.2.2 Character set enquiries

The a68toc implementation of Algol 68 is bedevilled by the peculiar limitations of the C programming language in which a character is actually an integer and indistinguishable from such. Furthermore, a C 'character' is a signed integer, equivalent to a value of mode `SHORT SHORT INT`. Thus C 'characters' range from −128 to +127. Algol 68, on the other hand, has the mode `CHAR` which, at a high level, is distinct from values of both mode `INT` and mode `SHORT SHORT INT`. The absolute value of Algol 68 characters range from 0 to the value of `max abs char`. Furthermore, the operator `REPR` will convert any `INT` upto `max abs char` to a character. Be warned that the C value of `REPR 225`, for example, is −31! What character is represented by `REPR 225` will depend on the character set used by the displaying device. An ISO 8859-1 character set, for example, will display 'á'. The environment enquiries in this section are limited to a range enquiry and the values of commonly required characters.

1. `INT max abs char` (RR) [255]
   The largest positive integer which can be represented as a character.
2. `CHAR null character` (RR) [`REPR 0`]
3. `CHAR nul ch` [`REPR 0`]
   This is a synonym for `null character`.
4. `CHAR blank` (RR) [`REPR 32`]
   This is a space character.
5. `CHAR error char` (RR) [∗]
   This character is used by the conversion routines for invalid values.
6. `CHAR flip` (RR) [T]
   This character is used to represent `TRUE` as an external value.
7. `CHAR flop` (RR) [F]
   This character is used to represent `FALSE` as an external value.
8. `CHAR cr` [`REPR 13`]
   This character is sometimes used as a line terminator, usually in association with `lf`.
9. `CHAR lf` [`REPR 10`]
   This character terminates lines on Linux.
10. `CHAR ff` [`REPR 12`]
    This character is the "form-feed" character often used for continuous stationery.
11. `CHAR tab ch` [`REPR 9`]
    This character is used to provide crude formatting of text files, particularly those which mimic documents produced by typewriters.
12. `CHAR esc` [`REPR 27`]
    This character is mainly used to introduce "escape sequences" which control the format and colour of output on Linux virtual terminals (VTs) and *xterm* windows.[2]

[2] See the file `/usr/share/doc/xterm/ctlseqs.txt.gz` for the latter.

13. `CHAR eof char` [`REPR 26`]
    This character was used to denote the end of a plain text file in the MS-DOS operating system.

13.3 Standard operators

The number of distinct operators is vastly increased by the availability of `SHORT` and `LONG` modes. Thus it is imperative that some kind of shorthand be used to describe the operators. Following the subsection on the method of description are sections devoted to operators with classes of operands. The end of this section contains tables of all the operators.

13.3.1 Method of description

Where an operator has operands and yield which may include `LONG` or `SHORT`, the mode is written using `L`. For example,

```
OP + = (L INT, L INT)L INT:
```

is a shorthand for the following operators:-

```
OP + = (LONG INT,LONG INT)LONG INT:
OP + = (INT,INT)INT:
OP + = (SHORT INT,SHORT INT)SHORT INT:
OP + = (SHORT SHORT INT,SHORT SHORT INT)
             SHORT SHORT INT:
```

Ensure that wherever `L` is replaced by `SHORT`s or `LONG`s, it should be replaced by the same number of `SHORT`s or `LONG`s throughout the definition of that operator. This is known as "consistent substitution". Note that any number of `SHORT`s or `LONG`s can be given in the mode of any value whose mode accepts such constructs (`INT`, `REAL`, `COMPL` and `BITS`), but the only modes which can be distinguished are those specified by the environment enquiries in section 13.2.1. However, you should note that even though values of modes `LONG REAL` and `LONG LONG REAL` cannot be distinguished internally, the Algol 68 compiler still regards them as having unique modes and you will need to use the `LENG` operator to convert a value of mode `LONG REAL` to a value of mode `LONG LONG REAL`.

The priority of an operator is independent of the mode of the operator and so is given in a separate subsection. Each operator is accompanied by a short description of its function.

13.3.2 Standard priorities

The priority of declarations of the standard operators can be changed in subsidiary ranges using the `PRIO` declaration (see section 6.2.3). Each of the following enumerated nine sections contains a list of those operators which have that priority. Operators in parentheses are not defined in the Revised Report. See section 13.6 for their details.

1. `+:=`, `-:=`, `*:=`, `/:=`, `%:=`, `%*:=`, `+=:`
   `PLUSAB`, `MINUSAB`, `TIMESAB`, `DIVAB`, `OVERAB`, `MODAB`, `PLUSTO`
2. `OR`
3. `&`, `AND`
4. `=`, `/=`, `EQ`, `NE`
5. `<`, `<=`, `>=`, `>`
   `LT`, `LE`, `GE`, `GT`
6. `-`, `+`,
7. `*`, `/`, `%`, `%*`,
   `OVER`, `MOD`, `ELEM`,
8. `**`, `UP`, `DOWN`, `SHL`, `SHR`, `LWB`, `UPB`
9. `+*`, `I`, (`MIN`, `MAX`)

13.3.3 Operators with row operands

Both monadic and dyadic forms are available. We shall use the mode `ROW` to denote the mode of any multiple.

1. Monadic.
   `OP LWB = (ROW)INT:`
   `OP UPB = (ROW)INT:`
   Yields the lower or upper bound of the first or only dimension of its operand.
2. Dyadic.
   `OP LWB = (INT n,ROW r)INT:`
   `OP UPB = (INT n,ROW r)INT:`
   Yields the lower or upper bound of the n-th dimension of the multiple `r`.

13.3.4 Operators with `BOOL` operands

1. `OP OR = (BOOL a,b)BOOL:`
   Logical `OR`.
2. `OP & = (BOOL a,b)BOOL:`
   Logical `AND` (synonym `AND`).
3. `OP NOT = (BOOL a)BOOL:`
   Logical `NOT`: `TRUE` if `a` is `FALSE` and vice versa.
4. `OP = =(BOOL a,b)BOOL:`
   `TRUE` if `a` equals `b` (synonym is `EQ`).
5. `OP /= =(BOOL a,b)BOOL:`
   `TRUE` if `a` not equal to `b` (synonym is `NE`).
6. `OP ABS = (BOOL a)INT:`
   `ABS TRUE` is 1 and `ABS FALSE` is 0.

13.3.5 Operators with `INT` operands

Most of these operators take values of any precision. The `L` shorthand is used for those that can.

Monadic operators

Consistent substitution applies to all those operators in this section which use the `L` shorthand: apart from `LENG` and `SHORTEN`, the precision of the yield is the same as the precicion of the operand.

1. `OP + = (L INT a)L INT:`
   The identity operator. Does nothing.
2. `OP - = (L INT a)L INT:`
   The negation operator.
3. `OP ABS = (L INT a)L INT:`
   The absolute value. `ABS -3 = +3`
4. `OP SIGN = (L INT a)INT:`
   Yields -1 for a negative operand, $+1$ for a positive operand and 0 for a zero operand.
5. `OP ODD = (L INT a)BOOL:`
   Yields `TRUE` if the operand is odd.
6. `OP LENG = (L INT a)LONG L INT:`
   `OP LENG = (SHORT L INT a)L INT:`
   Converts its operand to the next longer precision. Note that you cannot use both `SHORT` and `LONG` in the same mode.
7. `OP SHORTEN = (L INT a)SHORT L INT:`
   `OP SHORTEN = (LONG L INT a)L INT:`
   Converts its operand to the next shorter precision. If the value exceeds `l max int` for the next shorter precision, the value will be truncated. This can lead to erroneous results. See also `LENG`.

Dyadic operators

In this section, consistent substitution is used wherever the `L` shorthand is used. For operators with mixed operands, see section 13.3.8.

1. `OP + = (L INT a,L INT b)L INT:`
   Arithmetic addition: $a + b$. No check is made for integer overflow.
2. `OP - = (L INT a,L INT b)L INT:`
   Arithmetic subtraction: $a - b$. No check is made for integer overflow.
3. `OP * = (L INT a,L INT b)L INT:`
   Arithmetic multiplication: $a \times b$. No check is made for integer overflow.
4. `OP / = (L INT a,L INT b)L REAL:`
   Arithmetic fractional division. Even if the result is a whole number (for example, 6/3), the yield always has mode `L REAL`. Where a result of mode `L REAL` needs to be output, but cannot be output due to the limitations built

into the definition of the mode `SIMPLOUT`, the operators `LENG` or `SHORTEN` should be used. Floating-point overflow can be checked (see section 13.6.1).

5. `OP % = (L INT a,L INT b)L INT:`
   Arithmetic integer division. Division by zero in the a68toc implementation produces a floating-point exception paradoxically (synonym `OVER`).
6. `OP ** = (L INT a,INT b)L INT:`
   Computes a^b for $b \geq 0$.
7. `OP %* = (L INT a,L INT b)L INT:`
   Arithmetic modulo (synonym `MOD`). For example

   ```
   5 MOD 3 = 2
   ```

8. `OP +* = (L INT a,L INT b)L COMPL:`
   Converts two integers into a complex number of the same precision (synonym `I`).
9. `OP < = (L INT a,L INT b)BOOL:`
   Arithmetic "less than": $a < b$ (synonym `LT`).
10. `OP <= = (L INT a,L INT b)BOOL:`
    Arithmetic "less than or equals": $a \leq b$ (synonym `LE`).
11. `OP >= = (L INT a,L INT b)BOOL:`
    Arithmetic "greater than or equals": $a \geq b$ (synonym `GE`).
12. `OP > = (L INT a,L INT b)BOOL:`
    Arithmetic "greater than": $a > b$ (synonym `GT`).
13. `OP = = (L INT a,L INT b)BOOL:`
    Arithmetic equality: $a = b$ (synonym `EQ`).
14. `OP /= = (L INT a,L INT b)BOOL:`
    Arithmetic inequality: $a \neq b$ (synonym `NE`).

13.3.6 Operators with `REAL` operands

Most of these operators can have operands of any precision. The `L` shorthand is used for them.

Monadic operators

1. `OP + = (L REAL a)L REAL:`
   Arithmetic identity. Does nothing.
2. `OP - = (L REAL a)L REAL:`
   Arithmetic negation: $-a$.
3. `OP ABS = (L REAL a)L REAL:`
   The absolute value. `ABS -3.0 = +3.0`
4. `OP SIGN = (L REAL a)INT:`
   Yields -1 for negative operands, $+1$ for positive operands and 0 for a zero operand (0.0).

5. `OP ROUND = (REAL a)INT:`
   Rounds its operand to the nearest integer. If the value ends with .5, it is rounded to the nearest even number. This is contrary to normal Linux C library practice, but is an internationally accepted standard which ensures that rounding errors do not accumulate. The operator checks for integer overflow (see section 13.6.1 for details).

6. `OP ROUND = (L REAL a)L INT:` (for any precision except `REAL`)
   Rounds its operand to the nearest integer. Does not check integer overflow. If its operand exceeds `l max int`, an erroneous result will ensue. `ROUND` should be used for a `REAL` operand if you want to check for integer overflow (see section 13.6.1 for details of floating-point overflow checking).

7. `OP ENTIER = (REAL a)INT:`
   Truncates its operand to the next lowest integer. The operator checks for integer overflow (see section 13.6.1 for details).

8. `OP ENTIER = (L REAL a)L INT:` (for any precision except `REAL`)
   Truncates its operand to the next lowest integer. The operator does not check integer overflow. If its operand exceeds `l max int`, an erroneous result will ensue. Use `ENTIER` for a `REAL` operand if you want to check for integer overflow (see section 13.6.1 for details of floating-point overflow checking).

9. `OP LENG = (L REAL a)LONG L REAL:`
   `OP LENG = (SHORT L REAL a)L REAL:`
   Converts its operand to the next longer precision. Note that you cannot use both `SHORT` and `LONG` in the same mode.

10. `OP SHORTEN = (L REAL a)SHORT L REAL:`
    `OP SHORTEN = (LONG L REAL a)L REAL:`
    Converts its operand to the next shorter precision. If a value exceeds `l max real` for the next shorter precision, the value will be truncated leading to an erroneous result. The mantissa will always be truncated.

Dyadic operators

In this section, consistent substitution is used wherever the `L` shorthand appears. For operators with mixed operands, see section 13.3.8.

1. `OP + = (L REAL a,L REAL b)L REAL:`
   Floating-point addition: $a+b$. Floating-point overflow will cause a trappable signal (see section 13.6.1).

2. `OP - = (L REAL a,L REAL b)L REAL:`
   Floating-point subtraction: $a-b$. Floating-point overflow will cause a signal which can be trapped (see section 13.6.1).

3. `OP * = (L REAL a,L REAl b)L REAL:`
   Floating-point multiplication: $a \times b$. Floating-point overflow will cause a signal which can be trapped (see section 13.6.1).

4. `OP / = (L REAL a,L REAL b)L REAL:`
   Floating-point divison: a/b. Floating-point overflow and divide-by-zero will cause a trappable signal (see section 13.6.1). Where a result of mode `L REAL`

needs to be output, but it cannot be output due to the limitations built into the definition of the mode `SIMPLOUT`, the operators `LENG` or `SHORTEN` should be used.

5. `OP +* = (L REAL a,L REAL b)L COMPL:`
   Converts two reals into a complex number of the same precision (synonym `I`).
6. `OP < = (L REAL a,L REAL b)BOOL:`
   Floating-point "less than": $a < b$ (synonym `LT`).
7. `OP <= = (L REAL a,L REAL b)BOOL:`
   Floating-point "less than or equals": $a \leq b$ (synonym `LE`).
8. `OP >= = (L REAL a,L REAL b)BOOL:`
   (synonym `GE`)
   Floating-point "greater than or equals": $a \geq b$.
9. `|OP > = (L REAL a,L REAL b)BOOL:`
   Floating-point "greater than": $a > b$ (synonym `GT`).
10. `OP = = (L REAL a,L REAL b)BOOL:`
    Floating-point equality: $a = b$ (synonym `EQ`).
11. `OP /= = (L REAL a,L REAL b)BOOL:`
    Floating-point inequality: $a \neq b$ (synonym `NE`).

13.3.7 Operators with `COMPL` operands

Algol 68 is one of the few programming languages which have a built-in mode for complex numbers. It is complemented by a rich set of operators, some of which are only available for values of mode `COMPL`. Again, consistent substitution is applicable to all operators using the `L` shorthand.

Monadic operators

1. `OP RE = (L COMPL a)L REAL:`
   Yields the real component: `re OF a`.
2. `OP IM = (L COMPL a)L REAL:`
   Yields the imaginary component: `im OF a`.
3. `OP ABS = (L COMPL a)L REAL:`
   Yields $\sqrt{\texttt{RE}\, a^2 + \texttt{IM}\, a^2}$.
4. `OP ARG = (L COMPL a)L REAL:`
   Yields the argument of the complex number.
5. `OP CONJ = (L COMPL a)L COMPL:`
   Yields the conjugate complex number.
6. `OP + = (L COMPL a)L COMPL:`
   Complex identity. Does nothing.
7. `OP - = (L COMPL a)L COMPL:`
   Complex negation.

8. `OP LENG = (L COMPL a)LONG L COMPL:`
   `OP LENG = (SHORT L COMPL a)L COMPL:`
   Converts its operand to the next longer precision. Note that you cannot use both `SHORT` and `LONG` in the same mode. Unfortunately, although a68toc will translate a program containing this operator apparently without errors, the resulting C file will not compile. The error produced will be "conversion to non-scalar type requested". You should use the following code instead:-

   ```
   (LENG RE z,LENG IM z)
   ```

9. `OP SHORTEN = (L COMPL a)SHORT L COMPL:`
   `OP SHORTEN = (LONG L COMPL a)L COMPL:`
   Converts its operand to the next shorter precision. Note that you cannot use both `SHORT` and `LONG` in the same mode. Unfortunately, the a68toc translator will generate incorrect code (see the note for the operator `LENG`). Use the following code instead:-

   ```
   (SHORTEN RE z,SHORTEN IM z)
   ```

   If either of the components of the complex number exceeds `l max real` for the next shorter precision, an erroneous result will ensue, but no error will be generated.

Dyadic operators

The remarks in section 13.3.6 concerning floating-point overflow apply doubly here.

1. `OP + = (L COMPL a,L COMPL b)L COMPL:`
   Floating-point addition for both components.
2. `OP - = (L COMPL a,L COMPL b)L COMPL:`
   Floating-point subtraction for both components.
3. `OP * = (L COMPL a,L COMPL b)L COMPL:`
   Standard complex multiplication with floating-point arithmetic.
4. `OP / = (L COMPL a,L COMPL b)L COMPL:`
   Standard complex division with floating-point arithmetic.
5. `OP = = (L COMPL a,L COMPL b)BOOL:`
   Complex equality with floating-point arithmetic (synonym `EQ`).
6. `OP /= = (L COMPL a,L COMPL b)BOOL:`
   Complex inequality with floating-point arithmetic (synonym `NE`).

13.3.8 Operators with mixed operands

Consistent substitution is applicable to all operators using the `L` shorthand. Additional shorthands are used as follows:-

- The shorthand `P` stands for +, `-`, `*` or `/`.
- The shorthand `R` stands for `<`, `<=`, `=`, `/=`, `>=`, `>`,
  or `LT`, `LE`, `EQ`, `NE`, `GE`, `GT`.

- The shorthand `E` stands for `= /=`,
  or `EQ` or `NE`.

1. `OP P = (L INT a,L REAL b)L REAL:`
2. `OP P = (L INT a,L COMPL b)L COMPL:`
3. `OP P = (L REAL a,L COMPL b)L COMPL:`
4. `OP P = (L REAL a,L INT b)L REAL:`
5. `OP P = (L COMPL a,L INT b)L COMPL:`
6. `OP P = (L COMPL a,L REAL b)L COMPL:`
7. `OP R = (L INT a,L REAL b)BOOL:`
8. `OP R = (L REAL a,L INT b)BOOL:`
9. `OP E = (L INT a,L COMPL b)BOOL:`
10. `OP E = (L REAL a,L COMPL b)BOOL:`
11. `OP E = (L COMPL a,L INT b)BOOL:`
12. `OP E = (L COMPL a,L REAL b)BOOL:`
13. `OP ** = (L REAL a,INT b)L REAL:`
    `OP ** = (L COMPL a,INT b)L COMPL:`
    Exponentiation: a^b (synonym `UP`).
14. `OP +* = (L INT a,L REAL b)L COMPL:`
    `OP +* = (L REAL a,L INT b)L COMPL:`
    (synonym `I`)

13.3.9 Operators with `BITS` operands

Consistent substitution applies to all operators using the `L` shorthand.

Monadic operators

1. `OP BIN = (L INT a)L BITS:`
   Mode conversion which does not change the internal value.
2. `OP ABS = (L BITS a)L INT:`
   Mode conversion which does not change the internal value.
3. `OP NOT = (L BITS a)L BITS:`
   Yields the bits obtained by inverting each bit in the operand. That is, 0 goes to 1, 1 goes to 0.
4. `OP LENG = (L BITS a)LONG L BITS:`
   `OP LENG = (SHORT L BITS a)L BITS:`
   Converts a bits value to the next longer precision by adding zero bits to the more significant end. Note that you cannot use both `SHORT` and `LONG` in the same mode.
5. `OP SHORTEN = (L BITS a)SHORT L BITS:`
   `OP SHORTEN = (LONG L BITS a)L BITS:`
   Truncates a bits value to a value of the next shorter precision. The more significant bits are simply ignored.

Dyadic operators

1. `OP & = (L BITS a,L BITS b)L BITS:`
   (synonym `AND`)
   The logical "and" of each pair of binary digits in *a* and *b*.
2. `OP OR = (L BITS a,L BITS b)L BITS:`
   The logical "or" of each pair of binary digits in *a* and *b*.
3. `OP SHL = (L BITS a,INT b)L BITS:`
   The left operand shifted left by the number of bits specified by the right operand. New bits shifted in are all zero. If the right operand is negative, shifting is to the right (synonym `UP`).
4. `OP SHR = (L BITS a,INT b)L BITS:`
   (synonym `DOWN`)
5. `OP ELEM = (INT a,L BITS b)BOOL:`
   Yields `TRUE` if bit *a* is 1.
6. `OP = = (L BITS a,L BITS b)BOOL:`
   Logical equality (synonym `EQ`).
7. `OP /= = (L BITS a,L BITS b)BOOL:`
   Logical inequality (synonym `NE`).
8. `OP <= = (L BITS a,L BITS b)BOOL:`
   Yields `TRUE` if each bit in the left operand implies the corresponding bit in the right operand (synonym `LE`).
9. `OP >= = (L BITS a,L BITS b)BOOL:`
   Yields `TRUE` if each bit in the right operand implies the corresponding bit in the left operand (synonym `GE`).

13.3.10 Operators with `CHAR` operands

The shorthands in section 13.3.8 apply here.

1. `OP ABS = (CHAR a)INT:`
   The integer equivalent of a character.
2. `OP REPR = (INT a)CHAR:`
   The reverse of `ABS`. The operand should be in the range `[0:max abs char]`.
3. `OP + = (CHAR a,CHAR b)STRING:`
   The character *b* is appended to the character *a* (concatenation).
4. `OP E = (CHAR a,CHAR b)BOOL:`
   Comparison of the arithmetic equivalents of *a* and *b*.

13.3.11 Operators with `STRING` operands

1. `OP + = (STRING a,STRING b)STRING:`
   String *b* is appended to string *a* (concatenation).
2. `OP + = (CHAR a,STRING b)STRING:`
   String *b* is appended to character *a*.

3. `OP + = (STRING a,CHAR b)STRING:`
   Character b is appended to string a.
4. `OP * = (INT a,STRING b)STRING:`
   Yields a lots of string b concatenated.
5. `OP * = (STRING a,INT b)STRING:`
   Yields b lots of string a concatenated.
6. `OP * = (INT a,CHAR b)STRING:`
   Yields a lots of character b concatenated.
7. `OP * = (CHAR a,INT b)STRING:`
   Yields b lots of character a concatenated.
8. `OP < = (STRING a,STRING b)BOOL:`
   The absolute value of each character of a is compared with the absolute value of the corresponding character in b (for the purpose of the comparison, the lower bounds of both strings are regarded as equal to 1). If the strings are equal upto the end of the shorter of the strings, then the longer string is the greater (synonym `LT`).
9. `OP <= = (STRING a,STRING b)BOOL:`
   (synonym `LE`)
   The text for the operator `<` in this section applies.
10. `OP >= = (STRING a,STRING b)BOOL:`
    (synonym `GE`)
    The text for the operator `<` in this section applies.
11. `OP > = (STRING a,STRING b)BOOL:`
    (synonym `GT`)
    The text for the operator `<` in this section applies.
12. `OP = = (STRING a,STRING b)BOOL:`
    If the strings differ in length, they are unequal, else they are compared as for the operator `<` in this section (synonym `EQ`).
13. `OP /= = (STRING a,STRING b)BOOL`
    (synonym `NE`)
    If the strings differ in length, they are unequal, else they are compared as for the operator `<` in this section.
14. `OP E = (STRING a,CHAR b)BOOL:`
    `OP E = (CHAR a,STRING b)BOOL:`
    The shorthand `E` as described in section 13.3.8 applies for these cases.

13.3.12 Assigning operators

Consistent substitution applies to all operators containing the `L` shorthand.

1. `+:=` (synonym `PLUSAB`)
   The operator is a shorthand for `a:=a+b`.

Left operand	Right operand	Yield
REF L INT	L INT	REF L INT
REF L REAL	L INT	REF L REAL
REF L COMPL	L INT	REF L COMPL
REF L REAL	L REAL	REF L REAL
REF L COMPL	L REAL	REF L COMPL
REF L COMPL	L COMPL	REF L COMPL
REF STRING	CHAR	REF STRING
REF STRING	STRING	REF STRING

2. `+=:` (synonym `PLUSTO`)
   The operator is a shorthand for `b:=a+b`.

Left operand	Right operand	Yield
STRING	REF STRING	REF STRING
CHAR	REF STRING	REF STRING

3. `-:=` (synonym `MINUSAB`)
   The operator is a shorthand for `a:=a-b`.

Left operand	Right operand	Yield
REF L INT	L INT	REF L INT
REF L REAL	L INT	REF L REAL
REF L COMPL	L INT	REF L COMPL
REF L REAL	L REAL	REF L REAL
REF L COMPL	L REAL	REF L COMPL
REF L COMPL	L COMPL	REF L COMPL

4. `*:=` (synonym `TIMESAB`)
   The operator is a shorthand for `a:=a*b`.

Left operand	Right operand	Yield
REF L INT	L INT	REF L INT
REF L REAL	L INT	REF L REAL
REF L COMPL	L INT	REF L COMPL
REF L REAL	L REAL	REF L REAL
REF L COMPL	L REAL	REF L COMPL
REF L COMPL	L COMPL	REF L COMPL
REF STRING	INT	REF L COMPL

5. `/:=` (synonym `DIVAB`)
   The operator is a shorthand for `a:=a/b`.

Left operand	Right operand	Yield
REF L REAL	L INT	REF L REAL
REF L REAL	L REAL	REF L REAL
REF L COMPL	L INT	REF L COMPL
REF L COMPL	L REAL	REF L COMPL
REF L COMPL	L COMPL	REF L COMPL

6. `OP %:= = (REF L INT a,L INT b)REF L INT:`
   (synonym `OVERAB`)
   The operator is a shorthand for `a:=a%b`.

7. `OP %*:= = (REF L INT a,L INT b)REF L INT:`
   (synonym `MODAB`)
   The operator is a shorthand for `a:=a%*b`.

13.3.13 Other operators

This section contains those operators which appear neither in the Revised Report nor in the section concerning a68toc extensions (section 13.5).

1. `OP &* = (REAL r,INT e)REAL:`
   Multiply `r` by 2^e. The routine does not use multiplication, but simply increments the exponent of `r` accordingly.
2. `OP ELEM = (INT a)BITS:`
   The operator yields a value with all bits zero except the bit specified by the operand.
3. `OP MIN = (L INT a,L INT b)L INT:`
   `OP MIN = (L REAL a,L REAL b)L REAL:`
   `OP MIN = (L INT a,L REAL b)L REAL:`
   `OP MIN = (L REAL a,L INT b)L REAL:`
   The lesser of the two operands.
4. `OP MAX = (L INT a,L INT b)L INT:`
   `OP MAX = (L REAL a,L REAL b)L REAL:`
   `OP MAX = (L INT a,L REAL b)L REAL:`
   `OP MAX = (L REAL a,L INT b)L REAL:`
   The greater of the two operands.
5. `OP VALID = (REAL r)BOOL:`
   Whether the real value `r` is a valid real in terms of the IEEE standard.

13.4 Standard procedures

These mainly consist of mathematical procedures. All the procedures associated with interfacing with alien procedures appear in the a68toc section and all the transput procedures appear in the transput section. Procedures associated with floating-point, process and garbage-collector control appear in section 13.6.

13.4.1 Mathematical procedures

Strictly speaking, there are as many precisions of each of the mathematical functions as there are for real numbers. However, in the standard prelude provided with the a68toc compiler, the only extra precision implemented is that for `short real`. The `L` shorthand is used to simplify the list of procedures. All these procedures depend on the corresponding C library functions, so consult the manual pages for details.

1. `PROC l sqrt = (L REAL x)L REAL:`
   Yields the square root of x provided that $x \geq 0$.
2. `PROC l exp = (L REAL x)L REAL:`
   Yields e^x if such a value exists.

3. `PROC l ln = (L REAL x)L REAL:`
   Yields the natural (or Napierian) logarithm of x provided that $x > 0$, otherwise the procedure fails and `errno` is set (see section 13.6 for details).
4. `PROC l log = (L REAL x)L REAL:`
   Yields the logarithm of `x` to base 10.
5. `PROC l cos = (L REAL x)L REAL:`
   Yields the cosine of x, where x is in radians.
6. `PROC l arccos = (L REAL x)L REAL:`
   Yields the inverse cosine of x as a value between `L 0` and `2 l pi` inclusive. If `ABS x > 1` then the procedure yields an erroneous result, but `errno` is set (see section 13.6 for details).
7. `PROC l sin = (L REAL x)L REAL:`
   Yields the sine of x, where x is in radians.
8. `PROC l arcsin = (L REAL x)L REAL:`
   Yields the inverse sine of x as a value between `L 0` and `2 l pi` inclusive. If `ABS x > 1` then the procedure yields an erroneous result, but `errno` is set (see section 13.6 for details).
9. `PROC l tan = (L REAL x)L REAL:`
   Yields the tangent of x, where x is in radians.
10. `PROC l arctan = (L REAL x)L REAL:`
    Yields the inverse tangent of x as a value between `L 0` and `2 l pi` inclusive.
11. `PROC next random = (REF INT a)REAL:`
    The next `INT` value after a in a pseudo-random sequence uniformly distributed in the range `L 0` to `max int` is assigned to a. The yield x is in the range $0 \leq x < 1$ obtained by a uniform mapping of a.
12. `INT last random`
    `LONG INT long last random`
    Names initialised to fixed values and used by other random procedures as a seed.
13. `PROC random = REAL:`
    A call of `next random(last random)`.
14. `PROC short random = SHORT REAL:`
    As for `random` with the yield shortened.
15. `PROC l random int = (L INT n)L INT:`
    Yields a pseudo-random sequence of integers in the range $1 \leq x \leq n$.

13.4.2 Other procedures

The procedures `whole`, `fixed` and `float` are dealt with in the transput section (13.7).

1. `PROC l bits pack = ([]BOOL a)L BITS:`
   Packs `l bits width` booleans into a value of mode `L BITS`.

2. `PROC char in string =`
   `(CHAR c,REF INT i,STRING s)BOOL:`
   If the character c occurs in the string s, the procedure yields `TRUE` and assigns the position of c in s to i, otherwise it yields `FALSE` when no assignment to i takes place.

13.4.3 `ALIEN` declarations

This section contains all those values which are declared as `ALIEN` values, but which are not mentioned in the Revised Report.

1. `erange err` is one Linux system error used in the QAD standard prelude.
2. The following integer values are used in the transput.
   (a) `INT posix seek cur`
   Used in `posix lseek` to specify the current position.
   (b) `INT posix seek end`
   Used in `posix lseek` to specify the end of the file.
   (c) `INT posix seek set`
   Used in `posix lseek` to specify a direct offset.
3. These values are used in the manipulation of the `kbd channel`.
   (a) `INT termios vtime`
   The offset in the `termios` structure.
   (b) `INT termios vmin`
   The offset in the `termios` structure.
   (c) `INT tcsanow`
   Used in the call to `linux tc set attr`.
   (d) `INT isig`
   Used in the call to `linux tc set attr`.
   (e) `INT icanon`
   Used in the call to `linux tc set attr`.
   (f) `INT echo`
   Used in the call to `linux tc set attr`.
4. `nil func ptr` has mode `CPTR` and is used to provide a `NIL` pointer to C functions.
5. `null c charptr` is equivalent to the C value `NULL`.
6. `prelude` is a dummy declaration used to access the parameters provided to a program when called. It should not be used.
7. The signal names in Linux are given in the following table:

sighup	sigstkflt	sigint	sigchld
sigquit	sigcont	sigill	sigstop
sigtrap	sigtstp	sigabrt	sigttin
sigbus	sigttou	sigfpe	sigurg
sigkill	sigxcpu	sigusr1	sigxfsz
sigsegv	sigvtalrm	sigusr2	sigprof
sigpipe	sigwinch	sigalrm	sigio
sigterm	sigpwr		

See `signal(7)` for details of each signal.

8. `errno` identifies a value of mode `REF INT` and contains the error number of the latest Linux system error.

13.4.4 `ALIEN` routines

All the `ALIEN` routines made available by the QAD standard prelude can be found in this section. No attempt has been made to give details of their function: consult the "man" pages on your Linux system for that (the appropriate page is given after the header of each routine with the section of the Linux programming manual given in parentheses). The `ALIEN` declarations have been classified by the first word of their identifiers which gives the standard which specifies them. For example, the routine `bsd mkstemp` is specified by the BSD4.4 standard.

Many of the routines are used in the definition of other, higher-level routines. They are made available so that implementers of other preludes will not have to redefine them.

Routines conforming to Ansi C

1. `PROC(INT)INT ansi raise`
   See `raise(3)`.
2. `PROC(INT,CPTR)CPTR ansi signal`
   See `signal(2)`.
3. `PROC(VECTOR[]CHARJ)REAL ansi strtod`
   See `strtod(3)`.

Routines conforming to BSD4.4

1. `PROC(INT,VECTOR[]CHAR,REF INT)INT bsd accept`
   See `accept(2)`.
2. `PROC(INT,VECTOR[]CHAR,INT)INT bsd bind`
   See `bind(2)`.
3. `PROC(VECTOR[]CHAR,INT)INT bsd chmod`
   See `chmod(2)`.
4. `PROC(INT,VECTOR[]INT,INT)INT bsd connect`
   See `connect(2)`.
5. `PROC(INT,INT)INT fchmod`
   See `fchmod(2)`.

6. `PROC(VECTOR[]CHAR)CCHARPTRPTR bsd gethostbyname`
   See `gethostbyname(3)`.
7. `PROC(VECTOR[]CHAR,REF BITS)INT bsd inet aton`
   See `inet_aton(3)`.
8. `PROC(INT)INT bsd is a tty`
   See `isatty(3)`.
9. `PROC(INT,INT)INT bsd listen`
   See `listen(2)`.
10. `PROC(VECTOR[]CHAR)INT bsd mkstemp`
    See `mkstemp(3)`.
11. `PROC(VECTOR[]CHAR,VECTOR[]CHAR,REAL,INT,INT)INT bsd real snprintf`
    Although the underlying routine can be used for the transput of any plain value, it is used here for the transput of a `REAL` only. See `snprintf(3)`.
12. `PROC(INT,INT)INT bsd shutdown`
    See `shutdown(2)`.
13. `PROC(INT,INT,INT)INT bsd socket`
    See `socket(2)`.

Routines conforming to an ISO standard

1. `PROC(CPTR)INT iso at exit`
   See `atexit(3)`.

Routines peculiar to Linux

1. `PROC(CPTR,CSTR)INT linux on exit`
   See `on_exit(3)`.
2. `PROC(INT,CCHARPTR)INT linux tc get attr`
   See `tcgetattr(3)`.
3. `PROC(INT,BITS,CCHARPTR)INT linux tc set attr`
   See `tcsetattr(3)`.

Routines conforming to POSIX

1. `PROC(INT)INT posix close`
   See `close(2)`.
2. `PROC(VECTOR[]CHAR,INT)INT posix creat`
   See `creat(2)`.
3. `PROC(INT)VOID posix exit`
   See `exit(2)`.
4. `PROC(VECTOR[]CHAR)CSTR posix get env`
   See `getenv(3)`.
5. `PROC INT posix getpid`
   See `getpid(2)`.

6. `PROC(INT,INT,INT)INT posix lseek`
   See `lseek(2)`.
7. `PROC(VECTOR[]CHAR,INT,INT)INT posix open`
   See `open(2)`.
8. `PROC(INT,VECTOR[]CHAR,INT)INT posix read`
   See `read(2)`.
9. `PROC(VECTOR[]CHAR,VECTOR[]CHAR)INT posix rename`
   See `rename(2)`.
10. `PROC(INT)CSTR posix strerror`
    See `strerror(3)`.
11. `PROC(CSTR)INT posix strlen`
    See `strlen(3)`.
12. `PROC(REF INT)INT posix time`
    See `time(2)`.
13. `PROC(VECTOR[]CHAR)INT posix unlink`
    See `unlink(2)`.
14. `PROC(INT,VECTOR[]CHAR,INT)INT posix write`
    See `write(2)`.

Local routines

1. `PROC(REF SHORT BITS)VOID get fpu cw`
   Gets the control word of the floating point unit.
2. `PROC(SHORT BITS)VOID set fpu cw`
   Sets the control word of the floating point unit.
3. `PROC(REAL,REF INT)VOID ph round`
   Rounds a `REAL` to an `INT`.

13.5 a68toc extensions

The a68toc manual describes the language restrictions of the translator. Chapter 3 contains details of the `FORALL` construct. This section is intended to document those extensions used in the QAD standard prelude.

13.5.1 Modes peculiar to a68toc

The principal extensions to Algol 68 modes are the introduction of multiple modes whose housekeeping overhead is less than the standard row modes.

1. `STRUCT n MODE`
   This mode is called an "indexable structure". The n, a non-negative integer, is built into the mode and must be an integer denotation. The base mode can be any mode. It is equivalent to a C language "array". Here is a list of modes used in the QAD standard prelude which are either indexable structures or references to such:-

(a) `CSTR=REF STRUCT 16000000 CHAR`
This is a reference mode and is equivalent to the C type `char *`. It is used in the `ALIEN` (see section 12.3.4) definitions of `linux getenv`, for example, to reference data.

(b) `CCHARPTRPTR=REF STRUCT 16000000 CSTR`
Again, this is a reference mode and is equivalent to the C type `char **`. It is used to access the program's arguments.

A considerable number of operators use indexable structures for converting values of one mode to another using memory mapping (see section 13.5.3).

2. `VECTOR[n]MODE`
The `VECTOR` mode has less overhead than a row mode because its lower bound is always one. It is commonly used to provide strings to C procedures. The following modes are defined using `VECTOR`:-

(a) `STR=VECTOR[0]CHAR`
Due to the way in which C multiples are defined (without bounds), the mode `STR` can be used for any length `VECTOR`.

(b) `RVC=REF STR`
This mode is used in a number of operators, such as

```
OP MAKERVC = ([]CHAR s)RVC:
```

It is also used to construct other modes such as `BOOK` (see section 13.7).

3. Coercions provided by a68toc
A value of mode `STRUCT n MODE` can be coerced directly to a value of mode `VECTOR[]MODE`. Likewise, a value of the latter mode can be coerced to a value of mode `[]MODE`. Therefore, preferably use the mode `[]MODE` for a parameter to a procedure.

4. Other modes used by a68toc
Some modes are provided to make interfacing with C library procedures easier. Here are the ones provided by the QAD standard prelude:-

(a) `CPTR=REF BITS`
This mode is equivalent to the C type `void *`.

(b) `CINTPTR=REF INT`
Equivalent to the C type `int *`.

(c) `CCHARPTR=REF CHAR`
Equivalent to the C type `char *`.

(d) `GCPARAM=STRUCT(STR name,INT value)`
Used to access parameters of the garbage-collector (see section 13.6.3 below).

(e) `PDESC=STRUCT(CPTR cp,CSTR env)`
This represents the structure created by a68toc for every Algol 68 procedure. The field `cp` contains the actual memory address of the procedure and the field `env` contains data used by the procedure.

(f) `VDESC=STRUCT(CCHARPTR data,BITS gc,`
`INT upb)`
This mode represents the housekeeping overhead of a `VECTOR`. The `data` field is the actual memory address of the start of the data and the `upb` field is the upper bound of the vector. The `gc` field is used by the garbage-collector (the heap manager).

13.5.2 a68toc constructs

This section describes those constructs which are either peculiar to a68toc or which are in some way different from standard Algol 68.

1. `FORALL`
`FORALL` is described in section 3.10.

2. `ALIEN` and `CODE`
Both `ALIEN` and `CODE` are described in the a68toc manual. `ALIEN` is also described in section 12.3.4. All `ALIEN` declarations used in the QAD standard prelude appear in the file `spaliens.a68` which you should consult for further details. You should note that the `ALIEN` declarations were established by trying various modes until a definition was found which a68toc translated to a compilable C source file. The declarations for `get fpu cw`, `set fpu cw` and the like, use the `__asm__` construct of the GNU C compiler: this provides a means of incorporating short sequences of assembler instructions into a C program. This is platform-specific. Consult the node `Extended Asm` in info file `gcc.info` for details.

   As described in the a68toc manual, source files may contain either a `PROGRAM` module or a `DECS` module. The latter may contain declarations and `CODE` clauses only. See the file `transput.a68`, lines 1185–92, for an example of how to execute code when a `DECS` module is being elaborated.

3. `USE` lists
The `USE` list of a `DECS` or a `PROGRAM` module generates calls to the relevant initialiser `PROC`s (see the generated C file for `standard.a68` for an example) in the reverse order of the given modules. Therefore, if the order matters, ensure that the `USE` clause mentions each module in the proper order.

4. The default case in a `CASE` clause
If in a `CASE` clause, whether a simple `CASE` or a conformity `CASE` clause (one which determines the mode of the value in its enquiry clause), the default clause can occur, then you must include at least `OUT SKIP`, otherwise you will get a run-time fault.

5. `BIOP 99`
In Algol 68, a `UNION` (see section 8.1) is a well-defined mode composed of constituent modes. A value of one of the constituent modes may be assigned to a name of the united mode and only that value (with its original mode) can be extracted. In the C language, however, a "free union" or just "union" is a piece of memory which can have different interpretations. The `BIOP 99` construct enables the operand of an operator using it to be re-interpreted as a value of the mode given in the yield. for example, the operator `FLAT` declared as

```
OP(REAL)STRUCT 8 CHAR FLAT = BIOP 99;
```

accepts a `REAL` parameter which, as the yield, is regarded as an indexable structure of 8 characters each of which can be accessed separately. See section 13.5.3 for operators using this construct.

13.5.3 Operators

These are largely operators using the `BIOP 99` construct, but there are a number of other operators which ease the task of interfacing with C library procedures.

Operators using `BIOP 99`

Most of the operators used in the QAD standard prelude which are defined using the `BIOP 99` construct are for internal use only. In the following list, the full declaration

```
OP(CPTR)CSTR TOCSTR = BIOP 99;
```

is abbreviated to

```
OP TOCSTR=(CPTR)CSTR:
```

Here is a list of operators using the `BIOP 99` construct which are made available by the QAD standard prelude:-

1. `OP CCHARPTRTOCSTR=(CCHARPTR)CSTR:`
   This operator is used to define the `on exit` procedure.
2. `OP CSTRTOCCHARPTR=(CSTR)CCHARPTR:`
   This operator converts in the opposite direction.
3. `OP TOCPTR=(INT)CPTR:`
4. `OP TOCSTR=(CPTR)CSTR:`
5. `TOPDESC=(PROC VOID)PDESC:`
   This operator provides a means of getting the address of a procedure and is used to provide the identifier of an Algol 68 procedure which must be elaborated by an `ALIEN` procedure (such as a C library routine).
   `OP TOPDESC=(PROC(INT,CPTR)VOID)PDESC:`
   `OP TOPDESC=(PROC(INT)VOID)PDESC:`
   `OP TOPDESC=(PROC(INT,RVC)PDESC:`
   You can define as many `TOPDESC` operators as you wish with operands of procedures you will need. You will certainly need more definitions of `TOPDESC` if you write wrapper procedures for X Window System procedures which have procedural parameters. See the file `transput.a68` for details of how this operator is used.
6. `OP TORPDESC=(REF PROC VOID)RPDESC:`
   This operator converts a reference to a `PROC VOID` to a reference to a value of mode `PDESC`.
7. `OP TOVDESC=(STR)VDESC:`
   This operator provides a means of getting the address of the `STR` in a form suitable as a parameter to C library routines.

8. `OP TOVBDESC=(VECTOR[]BITS)VBDESC`
   This operator provides a means of getting the address of the `VECTOR[]BITS`.
9. `OP TOVIDESC=(VECTOR[]INT)VIDESC`
   Exactly as for the previous routine, but for a `VECTOR[]INT`.

Other operators

Here is a list of operators not using the `BIOP 99` construct:-

1. `OP CPTRTORVC=(CPTR)RVC:`
   Used to cast the C type `void *` to the type `char *`.
2. `OP CSTRTORVC=(CSTR)RVC:`
   Converts a C string to an `RVC` using the standard RS Algol 68 coercion

   ```
   REF STRUCT n CHAR => REF VECTOR[]CHAR
   ```

   It is mainly used to access C strings yielded by C library routines. The parameter must be terminated by a `null character`.
3. `OP FLATRVB=(RVC)BITS:`
   Converts a `VECTOR[4]CHAR` into a `BITS`.
4. `OP FLATRVLB=(RVC)LONG BITS:`
   Converts a `VECTOR[8]CHAR` into a `LONG BITS`.
5. `OP FLATRVSB=(RVC)SHORT BITS:`
   Converts a `VECTOR[2]CHAR` into a `SHORT BITS`.
6. `OP FLATRVSSB=(RVC)SHORT SHORT BITS:`
   Converts a `BECTOR[1]CHAR` into a `SHORT SHORT BITS`.
7. `OP FLATRVR=(RVC)REAL:`
   Converts a `VECTOR[8]CHAR` into a `REAL`.
8. `OP FLATRVSR=(RVC)SHORT REAL:`
   Converts a `VECTOR[4]CHAR` into a `SHORT REAL`.
9. `OP MAKERVC=(CHAR)RVC:`
   `OP MAKERVC=(STR)RVC:`
   `OP MAKERVC=([]CHAR)RVC:`
10. `OP VBTOCPTR=(VECTOR[]BITS)CPTR:`
    This operator gets the address of the `VECTOR[]BITS` in a form suitable as a parameter to a C library routine.
11. `OP VCTOCHARPTR=(STR)CCHARPTR:`
    Yields the C pointer from a a68toc descriptor. If a C string is expected, a `null character` must be appended to the data before the routine is called. This need not be done for string denotations. This routine may be used to yield a C pointer from an `RVC`, as the C representation is the same.
12. `OP VITOINTPTR=(VECTOR[]INT)CINTPTR:`
    Yields the address of the `VECTOR[]INT` in a form suitable for use as a parameter of a C library routine.

13. `OP STRTOCSTR=(STR)CSTR:`
    The operator combines the action of the operators `CCHARPTRTOCSTR` and `VCTOCHARPTR`.

14. `OP Z=(STR str)STR:`
    Yields a null-terminated `STR` from a `STR` for use with the C library.

13.6 Control routines

Three groups of procedures and operators are provided to control various aspects of the run-time environment. These are floating-point control, process termination control and garbage-collector control.

13.6.1 Floating-point unit control

The Intel Pentium microprocessors all have a floating-point unit (FPU) as an integral part of the microprocessor. The action of the FPU is determined by the contents of a 16-bit register called the "control word register". Details of the register can be found in the file

```
/usr/include/fpu_control.h
```

Details of the working of the FPU, as controlled by the control word register can be found in the three volumes of the "Intel Architecture Software Developer's Manual". The control word contains bits which control rounding, precision and whether floating-point errors should cause an exception. The QAD standard prelude provides two procedures which enable you to get and set the control word register:-

1. `PROC get fpu cw = (REF SHORT BITS cw)VOID:`
   After calling `get fpu cw`, the current value of the FPU control word will be assigned to the parameter.

2. `PROC set fpu cw = (SHORT BITS cw)VOID:`
   After calling `set fpu cw`, the current value of the FPU control word will have been set to the value of the parameter.

The QAD standard prelude provides three values of mode `SHORT BITS` which enable you to control how rounding is performed. They are:-

1. `fpu cw ieee`
   This value enables you to reset the FPU control word to the standard value for the C library.

2. `fpu cw algol68 round`
   This value ensures that the FPU will perform rounding to the nearest number. A `REAL` value ending in $0 \cdot 5$ will be rounded to the nearest even number. This ensures that rounding errors in random values will not accumulate.

3. `fpu cw algol68 entier`
   This value ensures that the FPU will truncate `REAL` numbers towards $-\infty$ when converting to an integer of the equivalent precision.

These values are used as masks. Here, for example, is the source code for the operator `ROUND`:-

```
OP ROUND = (REAL r)INT:
(
   INT n;
   SHORT BITS ocw;  get fpu cw(ocw);
   set fpu cw(ocw & fpu cw algol68 round);
   ph round(r,n);
   set fpu cw(ocw);
   n
)
```

Notice how the FPU control word is reset to its original value before the end of the operator.

The FPU control word is also used to control whether overflow should be detectable. The standard mode of operation is to ignore integer overflow. The control word masks mentioned above ensure that integer overflow can be detected using a **signal**. The procedure `on signal` is declared as follows:-

```
PROC on signal=(INT sig,
                PROC(INT)VOID handler)VOID:
```

The example program `testov.a68` shows how `on signal` can be used. The Algol 68 identifiers for all the Linux signals are the same as the Linux signal identifiers, but in lower case. For example, the signal used in FPU control is `sigfpu`. The signal generated by keying `Ctrl-C` (sometimes depicted as `^C`) on program input is `sigint`. Here is a short program which illustrates signal trapping:-

```
PROGRAM sig CONTEXT VOID
USE standard
BEGIN
   on signal(sigint,
             (INT sig)VOID:
              (write(("sigint trapped",
                      newline));
               exit(1)));
   write("Please key ^C: "); read(LOC CHAR);
   write(("No signal trapped",newline))
END
FINISH
```

Usually, when you trap a signal such as `sigint`, your program could close down processing in an orderly manner: files could be closed properly, a message to the user could be issued, and so on. You can do anything you want in the procedure provided as a parameter to `on signal`. You can also predeclare the procedure and simply provide its identifier in the `on signal` call.

Integer overflow is ignored by the microprocessor. So the formula `max int + 3` simply yields an incorrect value.

The procedure `ansi raise` will cause any specified signal to occur. Here is the mode of `ansi raise`:

```
PROC ansi raise = (INT sig)INT:
```

13.6.2 Terminating a process

As well as raising and trapping signals, it is sometimes useful to specify procedures to be elaborated when your program has finished, for whatever reason. Four procedures are provided for process termination:-

1. `PROC iso at exit=(PROC VOID p)INT:`
   The procedure `p` is registered to be elaborated when the program terminates normally or when the procedure `exit` (see procedure 3) is called. Registered procedures are elaborated in the reverse order of being registered, so that the procedure registered last is elaborated first. The procedure `at exit` yields 0 for success, −1 for an error.

2. `PROC on exit=(PROC(INT,CPTR)VOID p,`
   `[]CHAR arg)INT:`
   Unlike the procedure `at exit` (see above), `on exit` allows you to register a procedure which takes two parameters. The first is the integer parameter given to the `exit` procedure (or 0 for normal termination) and the second is a `[]CHAR` which the procedure `p` can use. `on exit` yields 0 for success and −1 for an error.

3. `PROC exit = (INT status)VOID:`
   This procedure terminates the program immediately. Any procedures registered using `at exit` or `on exit` will be elaborated in the reverse order of registration. Any open files will be closed, but Algol 68 buffers will not be flushed. The value of `status` will be passed to the parent process of the program.

4. `PROC stop = VOID:`
   This is a synonym for `exit(0)`.

The example program `testexit.a68` shows one way in which `at exit` and `on exit` may be used.

13.6.3 Garbage-collector control

The garbage-collector manages the run-time heap. The term "garbage" is used to designate memory on the heap which is no longer referenced. Although the garbage-collector is usually called whenever space on the heap is required, a number of routines are provided for explicit garbage collection or for fine control of the garbage-collector.

1. `PROC garbage_collect = VOID:`
   The garbage-collector can be called explicitly by an Algol 68 program using this procedure.

2. `PROC disable_garbage_collector = VOID:`
   Disables the garbage-collector.

3. `PROC enable_garbage_collector = VOID:`
   Enables the garbage-collector.

4. `PROC gc_param = (VECTOR[]CHAR cmd,INT v)INT:`
   This routine is used to set or get the values of a number of parameters which control the garbage-collector. The `cmd` should consist of `GET␣` or `SET␣` followed

by the string identifying the required parameter followed by a `nul ch`. The available strings are

(a) `COLLECTION THRESHOLD` The number of bytes in use before the next garbage collection is allowed.

(b) `HEAP INCREMENT` The number of bytes by which the heap should be increased in size whenever the heap is grown.

(c) `MAX HEAP SIZE` The maximum size of the heap in bytes.

(d) `MIN HEAP SIZE` The minimum size of the heap in bytes.

(e) `MAX SEGMENT SIZE` The maximum size of a memory segment acquired for the heap.

(f) `MIN SEGMENT SIZE` The minimum size of a memory segment acquired for the heap.

(g) `POLICY` The heap policy. Three values are provided for setting the heap policy:-

i. `INT always collect`
The garbage-collector will always be called if space is required.

ii. `INT always grow heap`
The garbage-collector will never be called even if space is required.

iii. `INT default policy`
The garbage-collector will be called if there is insufficient space in the heap for the memory required. Extra space will be acquired if garbage-collection does not yield the spaced needed.

Whether the heap is grown or whether garbage-collection takes place depends on the current policy which is usually specified by the environment string `A68_GC_POLICY`.

5. `PROC get_gc_param = (VECTOR[]CHAR param)INT:`
Gets the current value of the garbage-collector parameter (any one of the strings given in this section).

6. `PROC set_gc_params = (VECTOR[]GCPARAM gcpar)VOID:`
Sets the value of the garbage-collector parameter (any one of the strings given in this section).

For further details about the garbage-collector, consult the code in the `library` directory in the a68toc source tree.

13.7 Transput

If you are unclear about the working of Algol 68 transput, consult chapter 9. The function of this section is to document all the transput declarations so that you can use it as a reference manual.

The declarations will be covered in the following order:-

1. Modes
2. Standard channels
3. Standard files

4. Opening files
5. Closing files
6. Transput routines
7. Interrogating files
8. File properties
9. Event routines
10. Conversion routines
11. Layout routines

In the sequel, transput errors are mentioned using identifiers whose values appear in the following table:-

`physical file end not mended`	255
`logical file end not mended`	254
`stand in redirected`	253
`environment string unset`	252
`environment string estab err`	251
`estab invalid parameters`	250
`open invalid parameters`	249
`no program args`	248
`value error not mended`	247

Identifiers for transput errors

13.7.1 Transput modes

Only five modes are available:-

`FILE` A structure containing details of a a book accessed by the program.

`CHANNEL` A structure containing procedures for accessing books.

`SIMPLIN` A union of names of all plain modes, rows of plain modes, structures of plain modes and their combinations.

`SIMPLOUT` A union of all plain modes, rows of plain modes, structures of plain modes and their combinations.

`BUFFER` A synonym for `RVC`. It is used as the yield of the procedure `file buffer` (see section 13.7.7).

The mode `NUMBER` is used as a parameter of the procedures `whole`, `fixed` and `float`, but because it is the union of all number modes, it is unnecessary to specify it and so has not been made available for general use.

13.7.2 Standard channels

For each channel in this section, the general properties are first given, followed by a table giving the properties of books opened on the channel and then a list of specific properties for the following procedures where applicable:-

establish
open
create
close
lock
scratch
set
logical end
reidf

1. `CHANNEL stand in channel`
   `CHANNEL stand out channel`
   `CHANNEL stand back channel`
   These three channels have similar properties because they use the same access procedures. The standard buffered input channel is `stand in channel`. Books on this channel have the following properties:-

stand in channel	
bin possible	`TRUE`
put possible	`FALSE`
get possible	`TRUE`
set possible	`TRUE`
reidf possible	`FALSE`

   and are available as the `stand in book`.

   The `stand out channel` is the standard buffered output channel. Books on this channel have the following properties:-

stand out channel	
bin possible	`TRUE`
put possible	`TRUE`
get possible	`FALSE`
set possible	`TRUE`
reidf possible	`FALSE`

   and are available as the `stand out book`.

   The `stand back channel` is the standard buffered input/output channel. Books on this channel have the following properties:-

stand back channel	
bin possible	`TRUE`
put possible	`TRUE`
get possible	`TRUE`
set possible	`TRUE`
reidf possible	`TRUE`

and are available as the `stand back book`.

The channels have the following properties:-

`establish` You must have write access to the file. If it already exists, it will be truncated to zero length. The default mode is `8r644`. If the file cannot be established, the routine will return the value `errno` (the system error name) refers to.

`open` the file will be opened with a default mode of `8r444`. If the file cannot be opened, the routine will return the value `errno` refers to.

`create` A zero length file with a unique identification will be created using the default mode of `8r644`.

`close` The file will be closed. For the `stand out channel` and the `stand back channel`, the buffer will be flushed.

`lock` The file will be closed and then all permissions will be removed from the file provided you have write access to the directory containing the file.

`scratch` The file will be closed and then unlinked.

`set` The current position will be set to any legal position in the book (non-negative positions only). If the position is set beyond the current logical end, a sparse file will be created.

`logical end` The position will be set to just beyond the last byte in the file.

`reidf` For the `stand back channel` only. When the file is closed, it will be renamed. If the rename fails (an already existing file with that name, for example), an error message will be output on the `stand error` file giving a description of the error and identifying the file.

2. `CHANNEL arg channel`
The `arg channel` gives access to the program arguments including the actual call of the program which precedes the program arguments. Arguments are separated by a single space. A name of mode `REF FILE` opened with this channel has `blank` as the string terminator. The arguments, as a book, have the following properties:-

`arg channel`	
bin possible	`FALSE`
put possible	`FALSE`
get possible	`TRUE`
set possible	`TRUE`
reidf possible	`FALSE`

The channel has the following properties:-

`establish` Same as `open`.

`open` The program arguments will be made available. If the arguments are unavailable, the routine will return `no program args`.

`create` Same as `open`.

`close` No action.

`lock` No action.

`scratch` No action.

`set` Provided that the required position lies between the beginning and the end of the arguments, the position will be set accordingly.

`logical end` The position will be set to just beyond the last character of the last argument.

`reidf` Inapplicable.

The procedure `make term` can be used to set the string delimiter to any required value to facilitate searching for quote-delimited or otherwise delimited arguments.

3. `CHANNEL env channel`
The `env channel` gives read-only access to environment strings (referred to in Linux documentation as "environment variables"). The environment string, as a book, has the following properties:-

`env channel`	
bin possible	`FALSE`
put possible	`FALSE`
get possible	`TRUE`
set possible	`TRUE`
reidf possible	`FALSE`

The channel has the following properties:-

`establish` Yields an error of value
`environment string estab err`.

`open` If the environment string is the null string or is unset, `open` yields an error of value
`environment string unset`. Otherwise, the string is available as a book.

`create` Yields an error of value
`environment string estab err`.

`close` No action.

`lock` The routine will attempt to remove all permissions from a file of the same identification as the environment string identification.

`scratch` The routine will attempt to unlink a file of the given identification.

`set` Provided that the required position lies between the beginning and the end of the string, the position will be set accordingly.

`logical end` The position will be set to just beyond the last character.

`reidf` Inapplicable.

The default string terminator is `nul ch`. You should set the string terminator using `make term`.

4. `CHANNEL kbd channel`
   The `kbd channel` provides access to unechoed keystrokes (also referred to as "non-canonical input"). Be warned that if a file opened with this channel is not closed and the program ends prematurely, none of your keystrokes will be echoed! You can reset to canonical input using the command

   ```
   stty sane
   ```

   The keyboard is made available as a book having the following properties:-

`kbd channel`	
bin possible	`TRUE`
put possible	`FALSE`
get possible	`TRUE`
set possible	`FALSE`
reidf possible	`FALSE`

   The channel is usually used to access the characters input by control and function keys as well as normal keystrokes, so it is advisable to use `get bin` rather than `get`. The channel has the following properties:-

   `establish` Same as `open`.

   `open` You should use the null string `""` for the identification. The routine checks to see whether stand in has been redirected and yields the error `stand in redirected` if so. Otherwise, the characteristics of `stand in` are changed to wait for a single character with no minimum waiting time and with no echo of the input.

   `create` Same as `open`.

   `close` The routine resets `stand in` to the condition it was in previously.

   `lock` Same as `close`.

   `scratch` Same as `close`.

   `set` Inapplicable.

   `logical end` Inapplicable.

   `reidf` Inapplicable.

5. `CHANNEL mem channel`
   The `mem channel` provides a memory buffer with access to all transput facilities. It is similar to the standard Algol 68 `associate` except that binary transput is also allowed. The buffer behaves as a book with the following properties:-

`mem channel`	
bin possible	`TRUE`
put possible	`TRUE`
get possible	`TRUE`
set possible	`TRUE`
reidf possible	`FALSE`

   The channel has the following properties:-

`establish` If the values of `p` and `l` are both `1` and the value of `c` is a positive integer then `c` is taken to be the size of the buffer. Otherwise, the routine yields `estab invalid parameters` as error value. The identification should be `""`.

`open` The routine should be called with an identification of mode `RVC` (see section 2b). The identification will be used as the memory buffer.

`create` The value `estab invalid parameters` will be returned.

`close` No action.

`lock` Inapplicable.

`scratch` Inapplicable.

`set` Provided the position lies in or just beyond the end of the buffer, the position will be set.

`logical end` The position will be set to just beyond the end of the buffer.

`reidf` Inapplicable.

The channel can be used to access individual characters of integers and reals. `make term` can also be used.

6. `CHANNEL client socket channel`
`CHANNEL server socket channel`
These channels provide UNIX- or Internet-type sockets in the form of standard Algol 68 files. Sockets behave as books with the following properties:-

`client/server socket channel`	
bin possible	`TRUE`
put possible	`TRUE`
get possible	`TRUE`
set possible	`FALSE`
reidf possible	`FALSE`

The channels have the following properties:-

`establish` The `p` should be the family of socket (either `af unix` or `af inet`). If the latter, the `l` should be the port. The `c` should be the MTU (maximum transport unit). This governs the size of the buffer associated with the socket. No checks are performed on its value. If `p` is neither `af unix` nor `af inet`, the routine returns `estab invalid parameters` as error value. The server socket should be established before the client socket.

`open` Yields an error of `open invalid parameters` for both channels.

`create` Inapplicable.

`close` The buffer is flushed and the socket closed.

`lock` The buffer is flushed, the socket closed and then all access permissions removed (provided that write access is available to the directory containing the socket).

`scratch` The buffer is flushed, the socket closed and then unlinked.

`set` Inapplicable.

`logical end` Inapplicable.

`reidf` Inapplicable.

An extra procedure `accept`, which mirrors the C library procedure, has the following header

```
PROC accept = (REF FILE socket)REF FILE:
```

and is used by the server to accept a `client socket`, thereby yielding a `REF FILE` name which can be used to communicate with the client.

The example programs `client1`, `server1`, `client2` and `server2` (whose source can be found in the `examples` directory) demonstrate simple use of sockets.

13.7.3 Standard files

Four standard files are provided:-

1. `REF FILE stand in`
   This file corresponds to the C `stdin`. Books connected via `stand in` differ from those connected via the channel `stand in channel`: `set possible` returns `FALSE`. Thus this file must be regarded as a simple stream of bytes. When the `kbd channel` is being used, `stand in` is unavailable.
2. `REF FILE stand out`
   This file corresponds to the C `stdout`. Books connected via `stand out` differ from those connected via the channel `stand out channel`: `set possible` returns `FALSE`. Thus this file must be regarded as a simple stream of bytes.
3. `REF FILE stand error`
   This file corresponds to the C `stderr` and behaves like the file `stand out`.
4. `REF FILE stand back`
   This file accesses a workbook which is deleted on termination of the program. All kinds of transput are allowed on this file.

13.7.4 Opening files

Three procedures are available for opening files:-

1. ```
   PROC establish=
       (REF FILE f,STRING idf,
        CHANNEL chann,INT p,l,c)INT:
   ```
   In standard Algol 68, this procedure creates a new file with `p` pages, each page containing `l` lines, each line containing `c` characters. In the QAD standard prelude, the `mem channel` (see section 5) takes notice of `p`, `l` and `c` and both `p` and `l` must be `1`. The socket channels (see section 6) use `p` for the socket family, `l` for the port if the family is `af inet` and `c` for the size of the MTU. For other channels, the values of `p`, `l` and `c` are ignored. The procedure yields zero on success, otherwise an integer denoting an error (see section 13.7.2).
   ```

2. `PROC open=(REF FILE f,`
   `UNION(CHAR,STRING,RVC) idf,`
   `CHANNEL chann)INT`
   In standard Algol 68, the second parameter of this procedure has mode `STRING`. The above union ensures that an `RVC` parameter can be used to open an existing memory buffer with the memory channel. This is particularly useful for performing transput on buffers obtained from C library routines. The procedure yields zero on success, otherwise an integer denoting an error (see section 13.7.2).

3. `PROC create=(REF FILE f,CHANNEL chann)INT:`
   Creates a work file with a unique identification in the directory `/tmp` using the given channel.

13.7.5 Closing files

Three procedures are provided:-

1. `PROC close=(REF FILE f)VOID:`
   This is the standard procedure for closing a file. It is standard practice to close every opened file. The procedure checks to see whether the file is open. If the `reidf` procedure has been called, after closing the file, the procedure renames the file to the identification given in the `reidf` field.
2. `PROC lock=(REF FILE f)VOID:`
   The Algol 68 Revised Report requires that the file be closed in such a manner that some system action is required before it can be reopened. In this case, the file is closed and then all access permissions removed. Before the file can be reopened, the user will have to use `chmod`.
3. `PROC scratch=(REF FILE f)VOID:`
   The file is closed and then unlinked.

13.7.6 Transput routines

The procedures in this section are responsible for the transput of actual values. Firstly, formatless transput is covered and then binary transput. The a68toc compiler does not support formatted transput. In each section, the shorthand `L` is used for the various precisions of numbers and bits values.

Straightening

The term **straightening** is used in Algol 68 to mean the process whereby a complex mode is separated into its constituent modes. For example, the mode

```
MODE X=STRUCT(INT a,
              CHAR b,
              UNION(REAL,VOID) u)
```

would be straightened into values of the following modes:-

- `INT`
- `CHAR`
- `UNION(REAL,VOID)`

The mode `REF[]X` would be straightened into a number of values each having the mode `REF X`, and then each such value would be further straightened into values having the modes

- `REF INT`
- `REF CHAR`
- `REF UNION(REAL,VOID)`

However, a value of mode `COMPL` is not straightened into two values both of mode `REAL`. Instead, the real part is transput, then an `"I"` read or written followed by the imaginary part.

Formatless transput

Formatless transput converts internal values into strings of characters or strings of characters into internal values.

1. `PROC write=([]UNION(SIMPLOUT,`
   `PROC(REF FILE)VOID) x)VOID:`
   This is equivalent to `put(stand out,x)` (synonym `print`).

2. `PROC put=(REF FILE f,`
   `[]UNION(SIMPLOUT,`
   `PROC(REF FILE)VOID) x)VOID:`
   The parameter `x` is straightened and the resulting values are output. Each plain mode is output as follows:-

   `CHAR` Output a character to the next logical position in the file. For `[]CHAR`, output all the characters on the current line.

   `BOOL` Output `flip` or `flop` for `TRUE` or `FALSE` respectively. For `[]BOOL`, output `flip` or `flop` for each `BOOL`.

   `L BITS` Output `flip` for each bit equal to one and `flop` for each bit equal to zero. `l bits width` characters are output in all. No newline or newpage is output. For `[]L BITS`, each `BITS` value is output as above with no intervening spaces.

   `L INT` Output a space character if the logical position is not at the start of a line. Then output the integer using the call

   ```
   whole(i,1+l int width)
   ```

   which will right-justify the integer in

   ```
   1+l int width
   ```

   positions with a preceding sign. For `[]L INT`, each integer is output as described above, preceded by a space if it is not at the beginning of the line. No newlines or newpages are output.

   `L REAL` A space is output if the logical position is not at the start of a line and then the number is output space-filled right-justified in

   ```
   l real width+l exp width+3
   ```

   positions in floating-point format and preceded by a sign. For a value of mode `[]L REAL`, each `REAL` value is output as described above.

   `L COMPL` The complex value is output as two real numbers in floating-point format separated by `␣i␣`. For `[]L COMPL`, each complex value is output as described above.

   `PROC(REF FILE)VOID:` An `lf` character is output if the routine is `newline` and an `ff` character if the routine is `newpage`. User-defined routines with this mode can be used.

3. `PROC read=([]UNION(SIMPLIN,`
   `PROC(REF FILE)VOID) x)VOID:`
   This is equivalent to `get(stand in,...)`.

4. `PROC get=(REF FILE f,`
   `[]UNION(SIMPLIN,`
   `PROC(REF FILE)VOID x)VOID:`

   This procedure converts strings of characters into internal values. Inputting data is covered for each plain mode and `REF STRING`. In each case, if the end of the file is detected while reading characters, the logical file end procedure is called:-

   `REF CHAR`
   : Any characters `c` where `c < blank` are skipped and the next character is assigned to the name.
   If a `REF[]CHAR` is given, then the above action occurs for each of the required characters of the multiple.

   `REF STRING`
   : All characters, including any control characters, are assigned to the name until any character in the character set specified by the `string term` field of `f` is read. The file is then backspaced so that the string terminator will be available for the next `get`.

   `REF BOOL`
   : The next non-space character is read and, if it is neither `flip` nor `flop`, the char error procedure is called with `flop` as the suggestion. For `REF[]BOOL`, each `REF BOOL` name is assigned a value as described above.

   `REF L BITS`
   : The action for `REF BOOL` is repeated for each bit in the name. For `REF[]L BITS`, each `REF L BITS` name is assigned a value as described above.

   `REF L INT`
   : If the next non-control character (*ie*, a character which is neither a space, a tab character, a newline or newpage character or other control character) is not a decimal digit, then the char error procedure is called with `"0"` as the suggestion. Reading of decimal digits continues until a character which is not a decimal digit is encountered when the file is backspaced. If during the reading of decimal digits, the value of `l max int` would be exceeded, reading continues, but the input value is not increased. For `REF[]L INT`, each integer is read as described above.

   `REF L REAL`
   : A real number consists of 3 parts:-
     - an optional sign possibly followed by spaces
     - an optional integral part
     - a `"."` followed by any number of control characters (such as newline or tab characters) and the fractional part
     - an optional exponent preceded by a character in the set `"Ee\"`. The exponent may have a sign. Absence of a sign is taken to mean a positive exponent

     The number may be preceded by any number of control characters or spaces. For `REF[]L REAL`, each `REAL` value is read as described above.

`REF L COMPL`
: Two real numbers separated by `"i"` are read from the file. Newlines or newpages or other control characters can precede each real. The first number is regarded as the real part and the second the imaginary part. For `REF[]L COMPL`, each `REF L COMPL` is read as described above.

Binary transput

Binary transput performs no conversion on internal values, thus providing a means of storing internal values in a compact form in books or reading such values into a program.

1. `PROC write bin=([]SIMPLOUT x)VOID:`
   This is equivalent to `put bin(stand back,x)`.
2. `PROC put bin=(REF FILE f,`
   `[]SIMPLOUT x)VOID:`
   This procedure outputs internal values in a compact form. Then external size is the same as the internal size.
3. `PROC read bin=([]SIMPLIN x)VOID:`
   This procedure is equivalent to

   ```
   get bin(stand back,x)
   ```

4. `PROC get bin=(REF FILE f,[]SIMPLIN x)VOID:`
   This procedure transfers external values in a compact form directly into internal values.

In all the above procedures, the transput is direct with no code conversion. It should also be noted that the procedure `make term`, although usually used with formatless transput, can also be used with binary transput in the QAD standard prelude for inputting a `STRING` terminated by any of a number of characters. You should note that if set possibleor the channel concerned, then the terminator (which will *always* include the lfharacter) will not have been read when get bineturns. However, if not `set possible` for the channel (and neither `stand in` nor `stand out` can be set), then no backspace is possible for binary transput and so the terminating character will have been read.

Other procedures

A number of miscellaneous procedures fall into this category.

1. `PROC file buffer = (REF FILE f)BUFFER:`
   Yields the buffer of a `REF FILE` value.
2. `PROC flush buffer = (REF FILE f) VOID:`
   This procedures empties the buffer if it has been changed by a `put` or a `put bin`.
3. `PROC no file end=(REF FILE f)BOOL:`
   One of the default procedures in `default io procs`.
4. `PROC ignore value error = (REF FILE f)BOOL:`
   One of the default procedures in `default io procs`.

5. `PROC ignore char error = (REF FILE f, REF CHAR ch)BOOL:`
   One of the default procedures in `default io procs`.

13.7.7 Interrogating files

A number of procedures are available for interrogating the properties of files:-

1. Properties of the book:-
   (a) `PROC bin possible=(REF FILE f)BOOL:`
   Yields `TRUE` if binary transput is possible.
   (b) `PROC put possible=(REF FILE f)BOOL:`
   Yields `TRUE` if data can be sent to the book.
   (c) `PROC get possible=(REF FILE f)BOOL:`
   Yields `TRUE` if data can be got from the book.
   (d) `PROC set possible=(REF FILE f)BOOL:`
   Yields `TRUE` if the book can be browsed: that is, if the position in the book for further transput can be set.
   (e) `PROC reidf possible=(REF FILE f)BOOL:`
   Yields `TRUE` if the identification of the book can be changed.
2. `PROC current pos=(REF FILE f)INT:`
   The standard Algol 68 procedure yields a triple giving the page, line and character number. However, the QAD standard prelude does not use pages, lines and characters, so this procedure yields the current character position within the book for the next transput operation.
3. `PROC file buffer=(REF FILE f)BUFFER:`
   Yields the memory buffer associated with the file `f`.
4. `PROC idf=(REF FILE f)RVC:`
   Yields the current identification of the book.
5. `PROC logical end=(REF FILE f)INT:`
   The current length of the book connected to `f`.

13.7.8 File properties

Three procedures are provided for altering the properties of files:-

1. `PROC make term=(REF FILE f,`
   `STRING term)VOID:`
   Makes `term` the current string terminator.
2. `PROC reidf=(REF FILE f,STRING new idf)VOID:`
   Changes the `reidf` field of `f` to the given value so that when the file is closed, the book will be renamed.
3. `PROC set flush after put=(REF FILE f)VOID:`
   Ensures that the buffer of a file is flushed whenever data is written to the file.

13.7.9 Event routines

Four event routines are provided. For each routine, the default behaviour will be described. In each case, if the user routine yields `FALSE`, the default action will be elaborated. If it yields `TRUE`, the action depends on the event.

1. `PROC on char error=(REF FILE f,`
   `PROC(REF FILE,`
   `REF CHAR)BOOL p)VOID:`
   This procedure assigns the procedure `p`, which may be an identifier or an anonymous procedure, to the `char error mended` field of `f`. The actions on character error are:-

   Default action Use the default character for the particular situation.

   User action A character may be assigned to the `REF CHAR` parameter and will be used if it is in the particular character set involved.

   The relevant situations are:-

   (a) When reading an integer of any precision, first character, possibly following an optional sign with following spaces, is not a digit. Any decimal digit can be substituted. The default is `"0"`.

   (b) When reading a real of any precision, the first non-space character, optionally preceded by a decimal point `"."`, is not a digit. Any decimal digit can be substituted. The default is `"0"`.

   (c) When reading a real of any precision, an exponent is present (the character `"e"` or `"E"` or `"\"` has been read), and the first non-space character is not a digit. Any decimal digit can be substituted. The default is `"0"`.

   (d) When reading a complex number, the first non-space character following the first real is not in the set `iI`. The default is `"i"`.

2. `PROC on logical file end=`
   `(REF FILE f,PROC(REF FILE)BOOL p)VOID:`
   This procedure assigns the procedure `p`, which may be an identifier or an anonymous procedure, to the `logical file mended` field of `f`. The actions on logical file end are:-

   Default action On any channel, if the end of the file has been reached or, in unformatted character transput, an `eof char` is read then the error message `logical file end not mended` will be issued and the program terminated with the exit value `logical file end not mended`.

   User action Any action as specified. Care should be taken if transput is performed on the file in question as an infinite loop could occur.

3. `PROC on physical file end=`
   `(REF FILE f,PROC(REF FILE)BOOL p)VOID:`
   This procedure assigns the procedure `p`, whether an identifier or an anonymous procedure, to the

   `physical file mended`

   field of `f`. The actions on physical file end are:-

Default action On any channel, if there is no more room on the physical medium, the program issues the error message

```
physical file end not mended
```

and then terminates the program with the exit value `physical file end not mended`.

User action Any action as specified. Care should be taken if transput is performed on the file in question as an infinite loop could occur.

4. `PROC on value error=`
   `(REF FILE f,PROC(REF FILE)BOOL p)VOID:`
   This procedure assigns procedure `p` to the `value error mended` field of `f`. The actions taken on a value error are:-

   Default action The program issues the error message `value error not mended` and then terminates with the same exit value.

   User action Transput continues.

   The error occurs in the following situations:-

   (a) When an integer on input exceeds `max int` for the precision concerned.

   (b) The size of the exponent of a real number exceeds `max int`.

   (c) An input real number is $\pm\infty$ or greater than `max real` or is less than `min real` for the precision concerned.

13.7.10 Conversion routines

The conversion routines consist of three procedures conversion of numbers to strings of characters, one operator and the procedure `char in string`. All the procedures `whole`, `fixed` and `float` will return a string of `error char` if the number to be converted is too large for the given width, or, if the number is a real, if it is infinite or otherwise invalid.

1. `PROC char in string=`
   `(CHAR c,REF INT p,STRING s)BOOL:`
   If the character `c` occurs in the string `s`, its index is assigned to `p` and the procedure yields `TRUE`, otherwise no value is assigned to `p` and the procedure yields `FALSE`.

2. `PROC whole=(NUMBER v,INT width)STRING:`
   The procedure converts integer values. Leading zeros are replaced by spaces and a sign is included if `width>0`. If `width` is zero, the shortest possible string is yielded. If a real number is supplied for the parameter `v`, then the call `fixed(v, width, 0)` is elaborated.

3. `PROC fixed=(NUMBER v,`
   `INT width,after)STRING:`
   The procedure converts real numbers to fixed point form, that is, without an exponent. The total number of characters in the converted value is given by the parameter `width` whose sign controls the presence of a sign in the converted value as for `whole`. The parameter `after` specifies the number of

required digits after the decimal point. From the values of `width` and `after`, the number of digits in front of the decimal point can be calculated. If the space left in front of the decimal point is insufficient to contain the integral part of the value being converted, digits after the decimal point are sacrificed.

4. `PROC float=(NUMBER v,`
   `INT width,after,exp)STRING:`
   The procedure converts reals to floating-point form ("scientific notation"). The total number of characters in the converted value is given by the parameter `width` whose sign controls the presence of a sign in the converted value as for `whole`. Likewise, the sign of `exp` controls the presence of a preceding sign for the exponent. If `exp` is zero, then the exponent is expressed in a string of minimum length. In this case, the value of `width` must not be zero. Note that `float` always leaves a position for the sign. If there is no sign, a blank is produced instead. The values of `width`, `after` and `exp` determine how many digits are available before the decimal point and, therefore, the value of the exponent. The latter value has to fit into the width specified by `exp` and so, if it cannot fit, decimal places are sacrificed one by one until either it fits or there are no more decimal places (and no decimal point). If it still doesn't fit, digits before the decimal place are also sacrificed. If no space for digits remains, the whole string is filled with `error char`.

5. `OP HEX=(L BITS v)[]CHAR:`
   The operator `HEX` converts a value of mode `L BITS` into a row of hexadecimal digits. The total number of digits equals `l bits width OVER 4`. For example, `HEX 4r3` yields `00000003`.

13.7.11 Layout routines

These routines provide formatting capability on both input and output.

1. `PROC space=(REF FILE f)VOID:`
   The procedure advances the position in file `f` by 1 byte. It does *not* read or write a blank.

2. `PROC backspace=(REF FILE f)VOID:`
   The procedure advances the position in file `f` by -1 bytes. It does not read or write a backspace. Note that not every channel permits backspacing more than once consecutively.

3. `PROC newline=(REF FILE f)VOID:`
   On input, skips any remaining characters on the current line and positions the file at the beginning of the next line. This means that all characters on input are skipped until a linefeed character `lf` is read. On output, emits a linefeed character. This is non-standard behaviour.

4. `PROC newpage=(REF FILE f)VOID:`
   On input, skips any remaining characters on the current page and positions the file at the beginning of the next page. This means that all characters on input are skipped until a formfeed character `ff` is read. Note that if the character following a number is a formfeed character, then that character will have been read during the read of the number. Hence, the skip to formfeed

character will skip the whole of the *following* page. On output, a formfeed character is emitted immediately.

5. `PROC skip terminators=(REF FILE f)VOID:`
   Any `STRING` terminators are skipped on input and the file positioned at the next non-terminating character. The procedure is usually called after a `STRING` has been read.

13.8 Summary

The whole of the standard prelude has been described in the above sections. Apart from the built-in operators implemented by the a68toc compiler, the whole of the standard prelude is implemented by Algol 68 source code.

Appendix A

Answers

A.1 Chapter 1

Ex 1.1

(a) Yes, it contains lower-case letters.
(b) Yes, it starts with a digit.
(c) No.
(d) Yes, a space is included.
(e) Yes, a full stop is included.

Ex 1.2 It starts with a capital letter and continues with capital letters, digits or underscores with no intervening spaces, tab characters or newline characters.

Ex 1.3 33

Ex 1.4

(a) It contains commas.
(b) It contains a decimal point.
(c) It is not a denotation: it is a formula (see chapter 2).

Ex 1.5

(a) It is not an identifier: it is a mode-indicant.
(b) Nothing—it's all right.
(c) It contains a minus symbol.
(d) It contains upper-case letters.

Ex 1.6

(a) The `>` symbol should be `=`.
(b) The integer denotation is larger than the largest integer that the compiler can handle.

Ex 1.7 `INT max int = 2 147 483 647`

Ex 1.8 `"." "," "8"`

Ex 1.9 `CHAR question mark = "?"`

Ex 1.10 The `5.` should be `5.0`. Either the semicolon should be replaced by a comma, or `z` should be preceded by `REAL` or `INT`.

Ex 1.11 `REAL light year = 9.454 26 e15`
(assuming 365 days per year).

Ex 1.12 The print phrase has one opening parenthesis and two closing ones and there is no `CONTEXT VOID USE standard` preceding the `BEGIN`.

Ex 1.13 The first displays `20` at the start of the line. The second displays ␣␣␣␣␣␣␣␣`+20`␣␣␣`+48930767` on one line.

Ex 1.14 It should display your name without quote symbols on the screen. Here is an example program:-

```
PROGRAM ex1 14 CONTEXT VOID
USE standard
BEGIN
   CHAR s="S", i="i", a="a", n="n";
   CO Letters of my first name CO
   print(s); print(i);
   print(a); print(n)
END
FINISH
```

which will display `Sian` on the screen.

Ex 1.15

(a) `1996`

(b) `"e"`

(c) `0.142857`

Ex 1.16

(a) Yes, it contains spaces.

(b) Yes, it contains a decimal point.

(c) No.

(d) Yes, it starts with a digit.

Ex 1.17

(a) `INT fifty five = 55`

(b) `REAL three times two point seven = 8.1`

(c) `CHAR colon=":"`

Ex 1.18 Yes, you cannot guarantee that the declaration for `x` will be elaborated before the declaration of `y`. The declarations should be written

```
REAL x = 1.234;
REAL y = x
```

Ex 1.19 `0` denotes an integer with mode `INT`, `0.0` denotes a real number with mode `REAL`.

Ex 1.20

```
PROGRAM ex1 20 CONTEXT VOID
USE standard
BEGIN
   print(0.5);  print(blank);
   print("G");  print(1);
   print(blank);print(":");
   print(34 000 000)
END
FINISH
```

A.2 Chapter 2

Ex 2.1 `INT minus thirty five = -35`

Ex 2.2

(a) `1`
(b) `1.0`
(c) `5.0`
(d) `0`
(e) `5`

Ex 2.3

(a) `6`
(b) `-6`
(c) `13.5`
(d) `4.5`

Ex 2.4

(a) `5`
(b) `-45.0`
(c) `-61`

Ex 2.5

(a) `20 INT`

(b) `1 INT`

(c) `1.25 REAL`

(d) `1 INT`

(e) `17.0 REAL`

Ex 2.6 Your answer should be something like this:-

```
PROGRAM ex2 6 CONTEXT VOID
USE standard
BEGIN
   print(-7 MOD 3);
   print( 7 MOD -3);
   print(-7 MOD -3)
END
FINISH
```

This will display

␣␣␣␣␣␣␣␣␣␣+2␣␣␣␣␣␣␣␣␣␣␣+1␣␣␣␣␣␣␣␣␣␣␣+2

on your screen.

Ex 2.7 `REAL two pi = 2 * pi`

Ex 2.8

(a) `4 INT`

(b) `3.25 REAL`

(c) `12 INT`

Ex 2.9

(a) `-3 INT`

(b) `-9 REAL`

(c) `2.0 REAL`

Ex 2.10 `1.5`

Ex 2.11

(a) `5`
(b) `2`
(c) `345`
(d) `32`
(e) `"1"`
(f) `8`
(g) `0.0`

Ex 2.12 The first `print` phrase displays

```
$0.0000000000000000$
```

(16 zeros) and the second displays `+infinity`.

Ex 2.13 The compiler detects the error and rejects it.

Ex 2.14

(a) The brackets should be replaced with parentheses.
(b) There are more opening than closing parentheses. The first opening parenthesis should be deleted.
(c) The operator `ROUND` has not been declared to use an operand with mode `CHAR`.
(d) The operator `ENTIER` has not been declared for use with an operand with mode `INT`.

A.3 Chapter 3

Ex 3.1

(a) Strictly speaking, the definition of Algol 68 allows parentheses wherever brackets (`[` and `]`) are allowed. The a68toc compiler does not support this and will flag it as an error.
(b) The apostrophes should be replaced by quote symbols.
(c) The value `2.0` in the row-display cannot be coerced to a value of mode `INT` in a strong context (or any context, for that matter).

Ex 3.2 `[]INT first 4 odd numbers = (1,3,5,7)`

Ex 3.3

(a) `8`
(b) `1`
(c) `3`

Ex 3.4

(a) `1 LWB a`, `1 UPB a`, `2 LWB a`, `2 UPB a`, `3 LWB a`, `3 UPB a`
(b) `LWB b`, `UPB b`

Ex 3.5

(a) `6`

(b) `(9,10,11,12)`

(c) `(4,8,12,16)`

Ex 3.6

(a) `r[3,2]`

(b) `r[2,]`

(c) `r[,3]`

Ex 3.7

```
[][]CHAR months=
   ("January","February","March",
    "April","May","June",
    "July","August","September",
    "October","November","December")
```

Ex 3.8

(a) `30`

(b) `(0.0,-5.4)`

(c) `11.4`

(d) `(6,7,8)`

(e) `"pqrst"`

Ex 3.9 This exercise is self-marking, but here is a program to print the answer to the first exercise:-

```
PROGRAM ex3 5 CONTEXT VOID
USE standard
BEGIN
  [,]INT r = ((1,2,3,4),(5,6,7,8),
           (9,10,11,12),(13,14,15,16));

  print(("r[2,2]=",r[2,2],newline,
       "r[3,]=",r[3,],newline,
       "r[,2 UPB r]=",r[,2 UPB r],
       newline))
END
FINISH
```

Ex 3.10

(a) `Man bites dog`

(b) `bbbii`

Ex 3.11

```
PROGRAM ex3 11 CONTEXT VOID
USE standard
BEGIN
   FOR num TO 25
   DO
      print((num^3,newline))
   OD
END
FINISH
```

Ex 3.12

```
PROGRAM ex3 12 CONTEXT VOID
USE standard
BEGIN
   FOR c FROM ABS "Z" BY -1 TO ABS "A"
   DO
      print(REPR c)
   OD
END
FINISH
```

Ex 3.13 The main difficulty lies in computing the letter to print. The first solution uses numbers and `REPR`:-

```
PROGRAM ex3 13a CONTEXT VOID
USE standard
BEGIN
   FOR row TO 5
   DO
      FOR letter TO 4
      DO
         print((REPR((row-1)*5
                  +letter+ABS"@"),",")
      OD;

      print((REPR(row*5 + ABS "@"),
             newline))
   OD
END
FINISH
```

The second solution uses an actual alphabet and a modified inner loop. Note that the formulæ in the `FROM` and `TO` constructs are elaborated once only: before the inner loop is elaborated for the first time in each elaboration of the outer loop:-

```
PROGRAM ex3 13b CONTEXT VOID
USE standard
```

```
BEGIN
   []CHAR alphabet =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[@1];

   FOR row TO 5
   DO
      INT row5 = row*5;

      FOR letter FROM row5-4 TO row5-1
      DO
         print((alphabet[letter],","))
      OD;

      print((alphabet[row5],newline))
   OD
END
FINISH
```

Ex 3.14 The most difficult part is in declaring the multiple. `print` will quite happily take the 3-dimensional multiple as its parameter:

```
PROGRAM ex3 14 CONTEXT VOID
USE standard
BEGIN
   [,,]REAL m=(((1e-7,1e-6),
                (1e-5,1e-4)),
               ((1e-3,1e-2),
                (1e-1,1.0)));
   print(m)
END
FINISH
```

Ex 3.15

(a) The brackets for the row-display should be replaced by parentheses.

(b) The number of integers in each row should be the same.

(c) Nothing. The denotation of an apostrophe is not doubled.

Ex 3.16

(a) `[1:2,1:3]`

(b) `[1:3]`

(c) `[1:2]`

Ex 3.17

(a) `(6,5,4) []INT`

(b) `(8,5,2) []INT`

(c) `(7,4) []INT`

(d) `((6,5),(3,2)) [,]INT`

Ex 3.18 `"abcabcabcdefg"`

Ex 3.19

```
PROGRAM ex3 19 CONTEXT VOID
USE standard
BEGIN
   []CHAR alphabet =
         "abcdefghijklmnopqrstuvwxyz";
   []INT by = (1,6,11,16,21,26);

   FOR c BY 5 TO UPB alphabet
   DO
      print(alphabet[c])
   OD
END
FINISH
```

A.4 Chapter 4

Ex 4.1

(a) `0`

(b) `4.4`

(c) `FALSE`

(d) `TRUE`

(e) `TRUE`

Ex 4.2

(a) `TRUE`

(b) `TRUE`

(c) `FALSE` (the `UPB t[2:]` is `5`

Ex 4.3

(a) `TRUE`

(b) `TRUE`

(c) `TRUE`

(d) `TRUE`. It is inadvisable to created compuound conditions with this sort of complexity simply because the condition is so difficult to understand. You should particularly avoid compound conditions with `NOT` in front of the various parts.

(e) `FALSE`

Ex 4.4

(a) `TRUE`

(b) `4 <= 2`

(c) `a <= b OR b <= c`

(d) `x /= y AND x /= z`

Ex 4.5

```
IF x < pi
THEN print("Yes")
ELSE print("No")
FI
```

Ex 4.6

```
FOR i TO 96
DO
   print(i*3);

   IF i MOD 16 = 0
   THEN print(newline)
   FI
OD
```

Ex 4.7 The second operand of `OREL` is only elaborated if the first yields `FALSE`.

```
PROGRAM p CONTEXT VOID
USE standard
BEGIN
   INT a = 3, b = 5, c = 4;

   IF
      IF a > b
      THEN TRUE
      ELSE b > c
      FI
   THEN print("Ok")
```

```
   ELSE print("Wrong")
   FI
END
FINISH
```

Ex 4.8 The right-hand side of the identity declaration is clearly an abbreviated case clause, so `p` must yield `INT`, not `BOOL`.

Ex 4.9

```
PROGRAM ex4 9 CONTEXT VOID
USE standard
CASE SIGN x + 2
IN
   print("x < 0.0"),
   print("x = 0.0"),
   print("x > 0.0")
ESAC
FINISH
```

Ex 4.10 `TRUE` and `FALSE`

Ex 4.11

(a) `TRUE`

(b) `TRUE`

(c) `TRUE`

(d) `FALSE`

(e) `FALSE`

(f) `FALSE`

Ex 4.12 You cannot mix full and abbreviated conditional clauses. Replace the vertical bar with `THEN`. Also replace the `ESAC` with `FI`.

Ex 4.13 `TFTFTFTFTF`

Ex 4.14

```
IF m < 10
THEN print("Units")
ELIF m < 100
THEN print("Tens")
ELIF m < 1000
THEN print("Hundreds")
ELSE print("Too big")
FI
```

Ex 4.15

```
print((card|"Ace","two","three",
           "four","five","six",
           "seven","eight","nine",
           "ten","Jack","Queen",
           "King"))
```

A.5 Chapter 5

Ex 5.1 `REF INT`

Ex 5.2 `REAL`

Ex 5.3 The right-hand side of the identity declaration should yield a value of mode `INT`. Insert `ENTIER` or `ROUND` before the `r`.

Ex 5.4 No value has been assigned to `x` when the second assignment is elaborated.

Ex 5.5

(a) A name with mode `REF REAL`.

(b) The real number denoted by `2.5` with mode `REAL`.

Ex 5.6 `1.166666...`

Ex 5.7 A name with mode `REF[,]REAL`.

Ex 5.8

(a) The bounds of the slice on the left-hand side of the assignment are `[-2:0]`, but the bounds of `n` are `[1:3]`. The assignment will cause a run-time error.

(b) You could write `m[5,]:=m[,-1]`, but it is unlikely that you would get what you wanted because the second column overlaps the third row. Here is a solution guaranteed to work:-

```
[]INT temp = m[,-1];
m[5,]:=temp[@-2]
```

Ex 5.9 There is no known formula which will tell you how big the sieve must be to find the 365^{th} prime; you just have to guess. A sieve with `size` equal to 5000 suffices. You need a counter for the primes. The complete program is:-

```
PROGRAM sieve CONTEXT VOID
USE standard
BEGIN
   INT size=5000;
   REF[]BOOL flags = LOC[2:size]BOOL;

   FOR i FROM LWB flags TO UPB flags
   DO
```

```
      flags[i]:=TRUE
   OD;

   FOR i FROM LWB flags TO UPB flags
   DO
      IF flags[i]
      THEN
         FOR k FROM i*2 BY i TO UPB flags
         DO
            flags[k]:=FALSE
            CO Remove multiples of i CO
         OD
      FI
   OD;

   REF INT count = LOC INT:=0;

   FOR i FROM LWB flags TO UPB flags
   DO
      IF flags[i] ANDTH (count+:=1)=365
      THEN print(i)
      FI
   OD
END
FINISH
```

Ex 5.10

(a) A name of mode `REF FLEX[]CHAR`.

(b) 1 and 5.

Ex 5.11

```
PROGRAM ex5 11 CONTEXT VOID
USE standard
BEGIN
   REF STRING ss = LOC STRING;
   FOR c FROM ABS "a" TO ABS "z"
   DO
      ss:="a"-REPR c;  print((ss,newline))
   OD
END
FINISH
```

Ex 5.12

```
REF FLEX[,]REAL f=
   LOC FLEX[1:0,1:0]REAL;
f:=(5.0,10.0,15.0,20.0);
```

```
print((1 LWB f,1 UPB f,
       2 LWB f,2 UPB f))
```

Ex 5.13

```
PROGRAM ex5 13 CONTEXT VOID
USE standard
BEGIN
   REF REAL a = LOC REAL,
         b = LOC REAL;

   print(Enter two real numbers->");
   read((a,b,newline));
   print(("Their sum is",a+b,newline,
        "Their product is",a*b))
END
FINISH
```

Ex 5.14

```
PROGRAM ex5 14 CONTEXT VOID
USE standard
BEGIN
   REF STRING line = LOC STRING;

   DO
     read((line,newline));
     IF UPB line = 0
     THEN stop #terminate the program#
     ELSE
       FOR i
       FROM UPB line BY -1 TO LWB line
       DO
         print(line[i])
       OD;
       print(newline)
     FI
   OD
END
FINISH
```

Ex 5.15

```
REF[]REAL r=
   LOC[(REF INT s=LOC INT; read(s); s)]REAL
```

Ex 5.16

```
PROGRAM ex5 16 CONTEXT VOID
USE standard
BEGIN
   REF INT number=LOC INT;
   read(number);

   REF[]INT multiple=LOC[number]INT;
   read(multiple);
   REF INT sum=LOC INT:=0;

   FOR i TO number
   DO
     sum+:=multiple[i]
   OD;
   print(sum)
END
FINISH
```

Ex 5.17

```
PROGRAM ex5 17 CONTEXT VOID
USE standard
BEGIN
   REF INT neg = LOC INT:=0,
           pos = LOC INT:=0;

   WHILE
      REF INT i=LOC INT;
      read((i.newline));
      i /= 0
   DO
      (i < 0|neg|pos) +:= i
   OD;

   print(("Sum of negative integers =",
          neg,newline,
          "Sum of positive integers =",
          pos,newline))
END
FINISH
```

Ex 5.18

```
PROGRAM ex5 18 CONTEXT VOID
USE standard
BEGIN
   REF STRING line = LOC STRING;

   WHILE
      read((line,newline));
      UPB line /= 0
   DO
     REF INT v=LOC INT:=0;

     FOR i TO UPB line
     DO
       v+:=ABS line[i]*i
     OD;

     print((line,v,newline))
   OD
END
FINISH
```

Ex 5.19

(a) `[100]CHAR rc`

(b) `FLEX[1:0]INT fi`

(c) `BOOL b:=TRUE`

Ex 5.20

(a) `REF INT a=LOC INT, b=LOC INT, c=LOC INT`

(b)
```
REF REAL x=LOC REAL;
 REF[]CHAR y=LOC[5]CHAR;
 REF[,]REAL z=LOC[3,3]REAL
```

(c) `REF FLEX[]CHAR s=LOC FLEX[1:0]CHAR`

Ex 5.21 `REF[]INT m=LOC[1000]INT; [1000]INT m`

Ex 5.22

```
PROGRAM ex5 22 CONTEXT VOID
USE standard
BEGIN
   REAL sum:=0.0, salary, INT num:=0;

   WHILE read(salary);  salary /= -1.00
   DO
     sum+:=salary;  num+:=1
   OD;

   print(("Average salary=",sum/num))
END
FINISH
```

Ex 5.23 When writing a program as involved as this, do not expect to get it right first time. In practice, a programmer adds fine details to a program after she has designed the main structure.

```
PROGRAM ex5 23 CONTEXT VOID
USE standard
BEGIN
   BOOL in word:=FALSE,
   STRING line;
   INT line start, line finish;
   INT word start, word finish;

   read((line,newline));
   line start:=LWB line;
   line finish:=UPB line;

   WHILE line[line start]=blank
                &
         line start<=UPB line
   DO
     line start+:=1
   OD;

   WHILE line[line finish]=blank
                &
         line finish>=line start
   DO
     line finish-:=1
   OD;

   line:=line[line start:line finish]
                   +blank;
```

```
    FOR c FROM LWB line
    WHILE c <= UPB line
    DO
      CHAR lc = line[c];

      IF   lc /= blank & NOT in word
      THEN word start:=c;  in word:=TRUE
      ELIF lc = blank & NOT in word
      THEN SKIP
      ELIF lc /= blank & in word
      THEN SKIP
      ELSE #lc = blank & in word#
        word finish:=c-1;
        in word:=FALSE;
        print((line[
                word start:word finish],
               newline))
      FI
    OD
END
FINISH
```

Notice that both `word start` and `word finish` are made to refer to new values before being used. This is a good check that you are writing the program properly. Notice also that the four possible states of the compound condition on line 26 are carefully spelled out on lines 28, 30 and 32.

A.6 Chapter 6

Ex 6.1 An identity declaration is

```
<formal-mode-param> = <actual-mode-param>
```

Ex 6.2 Because it is an identifier with a mode, but no associated value.

Ex 6.3

(a) `REAL`

(b) `INT`

(c) Strong

(d) `-5`

Ex 6.4 Using a loop:-

```
([]INT i)[]CHAR:
(
   [LWB i:UPB i]CHAR s;

   FOR n FROM LWB i TO UPB i
```

```
        DO
        s[n]:=REPR ii
        OD;
        s
    )
```

Ex 6.5 `(REF REAL id)REAL:`

Ex 6.6 `(REF CHAR a,b)REF CHAR:`

Ex 6.7

```
    (STRING s)[]STRING:
    BEGIN
       FLEX[1:0]STRING r:="";
          #rowing coercion#
       BOOL in word:=FALSE;
       INT st:=LWB s,fn:=UPB s;

       WHILE s[st]=blank & st<=UPB line
       DO
         st+:=1
       OD;

       WHILE s[fn]=blank & fn>=st
       DO
        fn-:=1
       OD;

       STRING ss:=s[st:fn]+blank;

       FOR c FROM LWB ss UNTIL c > UPB ss
       DO
          CHAR ssc=ss[c];

          IF   ssc/=blank & NOT in word
          THEN st:=c;  in word:=TRUE
          ELIF ssc=blank & NOT in word
          THEN SKIP
          ELIF ssc/=blank & in word
          THEN SKIP
          ELSE #ssc=blank & in word#
             fn:=c-1;  in word:=FALSE;
             [UPB r+1]STRING rr;
             rr[:UPB r]:=r;
             rr[UPB rr]:=ss[st:fn];
             r:=rr
         #The word has been added to r#
          FI
       OD;
```

```
    r[2:]  #Omit the null string#
END
```

Ex 6.8 `REAL:`

Ex 6.9 `VOID: print("Hi, there")`

Ex 6.10 This table summaries the occurrences:-

	Occurrences	
Line	Applied	Defining
5	`T p`	2
5	`T c`	3
8	`T 4`	2
8	`T q`	7
10	`T REPR 2`	3
12	`T c`	3
12	`T q`	2

Ex 6.11

(a) A name of mode `REF INT`.
(b) The integer denoted by `16` of mode `INT`.
(c) The integer nine of mode `INT`.
(d) The integer four of mode `INT`.

Ex 6.12 The two declarations are firmly related because, in a firm context, a name of mode `REF[]INT` can be dereferenced to a multiple of mode `[]INT`.

Ex 6.13

(a) 1.
(b) 1.
(c) 2.
(d) 2.

Ex 6.14 These reasons are the most important:-

1. Because their actions are not clear from the program code.
2. They can cause indeterminate states to occur.

Ex 6.15

(a) You cannot mix letters and symbols.
(b) The symbol should start with `+` which has already been declared as a monadic operator.
(c) This symbol is used for the identity relation (see section 11.6) and is not an operator.

Ex 6.16 `OP PP = (REF INT a)REF INT:  a+:=1`

Ex 6.17 `PROC p = VOID:  a:=3`

Ex 6.18

```
PROC p = INT:
BEGIN
   [(INT i; read((i,newline)); i)]INT a;
   read(a);
   INT sum:=0;

   FOR i TO UPB a DO sum+:=a[i] OD;
   sum
END
```

Ex 6.19

```
PROC p = REF[,]CHAR:
(
   HEAP[3,20]CHAR n;
   read((n,newline));
   n
)
```

Ex 6.20

```
PROC p=(REF REAL r)REF REAL:
   r:=r/pi*180
```

Ex 6.21

```
PROC p = (STRING s,INT i)VOID:
(
   INT ii = IF i < 0
            THEN print(newline); ABS i
            ELSE i
            FI;
   TO ii DO print(s) OD
)
```

Ex 6.22

```
PROC num in multiple=(INT i,
                      []INT m,
                      REF INT p)BOOL:
(
   INT pos:=LWB m - 1;

   FOR j FROM LWB m TO UPB m
   WHILE pos < LWB m
   DO
      (i=m[j]|pos:=j)
   OD;
```

```
   IF pos < LWB m
   THEN FALSE
   ELSE p:=pos; TRUE
   FI
)
```

Ex 6.23

(a) `10.0`

(b) `0.3`

(c) `0.0`.

Ex 6.24

```
PROC reverse = ([]CHAR s)[]CHAR:
(SIZE s=1|s|s[UPB s]+reverse(s[:UPB s-1]))
```

Ex 6.25

```
PROC(INT)INT cube;

PROC square=(INT p)INT:
   (ODD p|cube(p)|p^2);
cube:=(INT p)INT: (ODD p|p^3|square(p))
```

Ex 6.26 They form the two sides of an identity declaration.

Ex 6.27

```
OP LARGEST = ([,]REAL a)REAL:
(
   REAL largest:=a[1 LWB a,2 LWB a];

   FOR i FROM 1 LWB a TO 1 UPB a
   DO
      FOR j FROM 2 LWB a TO 2 UPB a
      DO
         largest:=largest MAX a[i,j]
      OD
   OD;
   largest
)
```

Ex 6.28

```
PROC pr = (INT n)REF[]INT: HEAP[n]INT
```

Ex 6.29

```
PROC leng = INT:
(STRING s;
 read((s,newline));
 UPB s)
```

A.7 Chapter 7

Ex 7.1 `STRUCT(INT i,j,k) s1 = (1,2,3)`

Ex 7.2 `STRUCT(INT i,REAL r,BOOL b)s2`

Ex 7.3

(a) `REF STRUCT(CHAR a,INT b)`

(b) `REF CHAR`

(c) `REF CHAR`

(d) `INT`, provided that a procedure had been assigned to `p OF st`.

(e) `INT`

(f) `REF STRUCT(CHAR a,INT b)`

Ex 7.4

```
PROC p1=(STRUCT(CHAR a,INT b)s)INT:
   ABS a OF s * b OF s
```

Ex 7.5

```
MODE EX_7_3_1=STRUCT(REAL r,
                     PROC(REAL)REAL p)
```

Ex 7.6

```
MODE EX_7_3_2=
   STRUCT(EX_7_3_1 e,
          PROC(EX_7_3_1)VOID p,
          CHAR c)
```

Ex 7.7 One of the `BMODE` and `AMODE` structures is insufficiently shielded. You will get an error for `BMODE` saying it is not a legal mode and another error for the declaration of a `REF AMODE` saying that the mode `AMODE` has not been declared.

Ex 7.8

(a) `(2.0,3.0)`
(b) `-12.0`
(c) Write a short program to get

```
3.6055512754639891
```

(d) `0.982 793 723 247 329 1`

Ex 7.9 The value denoted by `(12.0,-10.0)`.

Ex 7.10

(a) `REF REAL`, a name.
(b) `REAL  -3.0`
(c) `REAL  3.0`
(d) `REAL  3.0`

Ex 7.11

(a) `REF[]STRING`
(b) `REF REAL`
(c) `REF REAL`
(d) `REF[]REAL`

Ex 7.12 `[1:3]`.

Ex 7.13

(a) `REF CHAR`
(b) `REF[]STRING`
(c) `REF STRING`
(d) `REF[]REAL`
(e) `REF[]REAL`

Ex 7.14

```
MODE TEAM=STRUCT([11]STRING name,
                 STRING team,
                 INT played, won, drawn,
                     for, against)
```

Ex 7.15 Slicing binds more tightly than selecting, so the selection must be enclosed in parentheses (see section 10.6 for the full explanation).

Ex 7.16 The slicing takes place before the selection so no parentheses are needed.

Ex 7.17

(a) `REF PROC S2`
(b) `REF PROC(S1)S2`
(c) `REF[]CHAR`

A.8 Chapter 8

Ex 8.1 `MODE BINT = UNION(BOOL,INT)`

Ex 8.2 `BINT b = TRUE`

Ex 8.3 One of the constituent modes of the union is firmly-related to the united mode. In other words, in a firm context, `REF UB` can be dereferenced to `UB`.

Ex 8.4 `UNION(INT,[]INT,[,]INT) mint`

Ex 8.5 The first parameter is deprocedured to mode `CHAR` before being united. The second is dereferenced to mode `[]CHAR` and then united. The two values of the united mode are regarded as a row-display and the procedure is then called. The second parameter is an example of an anonymous name—no identifier is attached.

Ex 8.6

```
PROC ucis=(CHAR ch,[]CHAR s)
               UNION(INT,VOID):

IF   INT p = ch FIND s;  p >= LWB s
THEN p
ELSE EMPTY
FI
```

Ex 8.7

```
PROC p = (MIRC m)IRC:
CASE m IN
   ([]INT i): (INT sum:=0;
               FOR j FROM LWB i TO UPB i
               DO sum+:=i[j] OD;
               sum),
   ([]REAL r):(REAL sum:=0;
               FOR j FROM LWB r TO UPB r
               DO sum+:=r[j] OD;
               sum),
   ([]COMPL c):(COMPL sum:=0;
               FOR j FROM LWB c TO UPB c
               DO sum+:=c[j] OD;
               sum)
ESAC
```

Ex 8.8

```
OP * = (IRC a,b)IRC:
CASE a IN
   (INT i):  CASE b IN
                (INT j):   i*j,
                (REAL j):  i*j,
```

```
                    (COMPL j): i*j
                ESAC,
    (REAL i): CASE b IN
                    (INT j):   i*j,
                    (REAL j):  i*j,
                    (COMPL j): i*j
                ESAC,
    (COMPL i):CASE b IN
                    (INT j):   i*j,
                    (REAL j):  i*j,
                    (COMPL j): i*j
                ESAC
ESAC
```

Ex 8.9 `MODE CRIB = UNION(CHAR,REAL,INT,BOOL)`

Ex 8.10

```
OP UABS = (CRIB c)UNION(INT,REAL):
CASE c IN
   (CHAR a):   ABS a,
   (REAL a):   ABS a,
   (INT a):    ABS a,
   (BOOL a):   ABS a
ESAC
```

Ex 8.11 `UABS "c"; UABS -4.0; UABS -3; UABS TRUE`

A.9 Chapter 9

Ex 9.1

```
PROGRAM list CONTEXT VOID
USE standard
BEGIN
   FILE f;
   IF   open(f,
             "textbook",
             stand in channel)/=0
   THEN
      print("Cannot open textbook");
      exit(1)
   FI;

   STRING s;
   WHILE get(f,(s,newline));  UPB s /= 0
   DO
      print((s,newline))
   OD;
```

```
   close(f)
END
FINISH
```

Ex 9.2

```
PROGRAM ex9 2 CONTEXT VOID
USE standard
BEGIN
   FILE f;

   IF   open(f,
             "textbook",
             stand in channel)/=0
   THEN
      print("Cannot open textbook");
      exit(1)
   FI;

   REAL r, sum:=0, INT n;  get(f,n);

   TO n DO get(f,r); sum+:=r OD;
   print(sum);  close(f)
END
FINISH
```

Ex 9.3

```
PROGRAM ex9 3 CONTEXT VOID
USE standard
BEGIN
   FILE inf,outf;

   IF   open(inf,
             "textbook",
             stand in channel)/=0
   THEN
      print("cannot open textbook");
      exit(1)
   ELIF establish(outf,
                  "result",
                  stand out channel,
                  0,0,0)/=0
   THEN
      print("Cannot create result");
      exit(2)
   FI;

   REAL sum:=0, r, INT n;
```

```
   get(inf,n);
   TO n
   DO
      get(inf,r);  sum+:=r
   OD;
   put(outf,sum);
   close(inf);  close(outf)
END
FINISH
```

Ex 9.4

```
PROGRAM ex9 4 CONTEXT VOID
USE standard
BEGIN
   INT size = 10 000;
   [2:size]BOOL flags;

   FOR i
   FROM LWB flags TO UPB flags
   DO flags[i]:=TRUE OD;

   FOR i
   FROM LWB flags TO UPB flags
   DO
      IF flags[i]
      THEN
         FOR k
         FROM i+i BY i TO UPB flags
         DO
            flags[k]:=FALSE
         OD
      FI
   OD;

   #Now the file is needed#
   FILE f;
   IF   establish(f,
                  "primes",
                   stand out channel,
                   0,0,0)/=0
   THEN
      print("Cannot create primes");
      exit(1)
   FI;

   FOR i FROM LWB flags TO UPB flags
   DO
      IF flags[i]
```

```
      THEN put(f,(i,newline))
      FI
   OD;

   close(f)
END
FINISH
```

Ex 9.5 Notice that the processing of a line is done entirely within the `WHILE` clause.

```
PROGRAM ex9 5 CONTEXT VOID
USE standard
BEGIN
   FILE inf, outf;

   IF   open(inf,
             "inbook",
             stand in channel)/=0
   THEN
      print("Cannot open inbook");
      exit(1)
   ELIF establish(outf,
                  "outbook",
                  stand out channel,
                  0,0,0)/=0
   THEN
      print("Cannot create outbook");
      exit(2)
   FI;

   STRING line;

   WHILE
      get(inf,(line,newline));
      put(outf,(line,newline));
      IF UPB line = 0
      THEN FALSE
      ELSE line /= UPB line * blank
      FI
   DO SKIP OD;

   close(inf);  close(outf)
END
FINISH
```

Ex 9.6

```
PROGRAM ex9 6 CONTEXT VOID
USE transput
BEGIN
```

```
   FILE inf, outf;

   IF   open(inf,
             "lines",
             stand in channel)/=0
   THEN
      print("Cannot open book lines");
      exit(1)
   ELIF establish(outf,
                  "words",
                  stand out channel,
                  0,0,0)/=0
   THEN
      print("Cannot create book words");
      exit(2)
   FI;

   []CHAR terminators=" *"+cr+lf;
   make term(inf,terminators);

   STRING word, CHAR ch:=blank;

   WHILE
      get(inf,word);
      IF ch/=blank
      THEN ch PLUSTO word
      FI;

      WHILE
         get(inf,ch);
      CO String terminator,
         but cr/lf ignored CO
         ch = blank
      DO SKIP OD;  #Skip spaces#
      put(outf,(word,newline));
      ch /= "*"
   DO SKIP OD;

   close(inf);  close(outf)
END
FINISH
```

Ex 9.7 If the `on logical file end` procedure yields `FALSE`, the standard prelude causes an error message to be displayed and the program itself exits with an equivalent error number. Here is the code for the program:-

```
PROGRAM tt CONTEXT VOID
USE standard
IF FILE inf;
   STRING line;  INT n,sum:=0;
   open(inf,
        "inbook",
         stand in channel)/=0
THEN
   print(("Cannot open inbook",
          newline));
   exit(1)
ELSE
   on logical file end(inf,
    (REF FILE f)BOOL:
      IF FILE ouf;
         establish(ouf,
                   "outbook",
                   stand out channel,
                   0,0,0)/=0
      THEN
         print(("Cannot establish ",
                "outbook",newline));
         exit(2); SKIP
      ELSE
         put(ouf,(sum/n,newline));
         close(ouf);  FALSE
      FI);

   FOR i
   DO
      get(inf,(line,newline));
      n:=i;  sum+:=UPB line
   OD
FI
FINISH
```

Ex 9.8 In the following solution, note how `skip terminators` is called immediately after reading the first argument (the full path of the program):-

```
PROGRAM ex9 8 CONTEXT VOID
USE standard
IF FILE arg, inf, ouf;
   STRING line, infn, oufn;
   INT n,sum:=0;

   open(arg,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access the ",
        "program arguments",
        newline));
     exit(1)
ELIF
   on logical file end(arg,
    (REF FILE f)BOOL:
     (put(stand error,
          ("Insufficient arguments",
           newline));
      stop;  SKIP));
   get(arg,(LOC STRING,skip terminators,
            infn,skip terminators,
             oufn));
   open(inf,infn,stand in channel)/=0
THEN
   print(("Cannot open ",infn,newline));
   exit(2)
ELSE
   on logical file end(inf,
    (REF FILE f)BOOL:
     IF establish(ouf,
                  oufn,
                  stand out channel,
                  0,0,0)/=0
     THEN
        print(("Cannot establish ",
               oufn,
               newline));
        exit(3); SKIP
     ELSE
        put(ouf,("Average=",sum/n,
                 newline));
        close(ouf);
        FALSE
     FI);

   FOR i
```

```
   DO
      get(inf,(line,newline));
      n:=i;  sum+:=UPB line
   OD
FI
FINISH
```

Ex 9.9 Notice that the physical file end of the output file has also been covered:-

```
PROGRAM ex9 9 CONTEXT VOID
USE standard
IF FILE arg, inf, ouf;
   STRING line, infn, oufn;
   open(arg,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access the arguments",
        newline));
   exit(1)
ELIF
   on logical file end(arg,
    (REF FILE f)BOOL:
     (put(stand error,
          ("Insufficient arguments",
           newline));  stop;  SKIP));
   get(arg,(LOC STRING,
            skip terminators,
            infn,skip terminators,
            oufn));
   open(inf,infn,stand in channel)/=0
THEN
   print(("Cannot open ",infn,newline));
   exit(2)
ELIF
   establish(ouf,
             oufn,
             stand out channel,
             0,0,0)/=0
THEN
   print(("Cannot establish ",oufn,
          newline));
          exit(3)
ELSE
   on logical file end(inf,
    (REF FILE f)BOOL:
     (close(ouf);  close(inf);
      stop;  SKIP));
   on physical file end(ouf,
    (REF FILE f)BOOL:
     (put(stand error,
```

```
            ("Write error on ",idf(ouf),
             newline));
         exit(4); SKIP));
      DO
         get(inf,(line,newline));
         FOR i FROM LWB line TO UPB line
         DO
            REF CHAR li=line[i];
            IF li=blank THEN li:="*" FI
         OD;
         put(ouf,(line,newline))
      OD
   FI
   FINISH
```

Ex 9.10

```
PROGRAM ex9 10 CONTEXT VOID
USE standard
IF FILE env;
   open(env,"PATH",env channel)=0
THEN
   on logical file end(env,
    (REF FILE e)BOOL: (stop; SKIP));
   make term(env,":"+nul ch);
   STRING s;
   DO
      get(env,s);
      IF UPB s >= LWB s
      THEN print((s,newline))
      FI;
      skip delimiters(env)
   OD;
   close(env)
FI
FINISH
```

Ex 9.11

```
PROGRAM ex9 11 CONTEXT VOID
USE standard
IF FILE arg;
   open(arg,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access arguments",
        newline));
   exit(1)
ELSE
   on logical file end(arg,
```

```
 (REF FILE a)BOOL: (stop;  SKIP));
get(arg,(LOC STRING,
         LOC CHAR,
         skip terminators));
DO
   make term(arg,"/");
   STRING env name;
   CHAR terminator:=nul ch;
   get(arg,
       (env name,
        skip terminators,
        terminator));
   IF FILE env;
      open(env,
           env name,
           env channel)/=0
   THEN
      print((env name," undefined",
             newline))
   ELSE
      make term(env,
                terminator+nul ch);
      STRING s;
      on logical file end(
         env,
         (REF FILE f)BOOL:
           (GOTO continue; SKIP));
      DO
         get(env,s);
         IF UPB s >= LWB s
         THEN print((s,newline))
         FI;
         skip terminators(env)
      OD;
      continue:
      close(env)
   FI;
   make term(arg,blank);
   skip terminators(arg)
OD
FI
FINISH
```

Notice the addition of `nul ch` to cater for the lack of a specific terminator in the environment string.

Ex 9.12

```
PROGRAM ex9 12 CONTEXT VOID
USE standard
IF FILE abc;
```

```
    open(abc,"ABC",env channel)/=0
THEN
    print(("Environment string ABC",
          "is undefined",newline));
    stop
ELSE
    INT sum:=0, n;
    on logical file end(
     abc,
     (REF FILE f)BOOL:
      (close(f);
       print(("Total=",sum,newline));
       stop;  SKIP));
   DO
      get(abc,n);
      sum+:=n
   OD
FI
FINISH
```

Ex 9.13 Notice how the size of the month denotation is used to ensure that the rainfall is aligned appropriately.

```
PROGRAM ex9 13 CONTEXT VOID
USE standard
BEGIN
[]STRING months=
    ("January","February","March",
     "April","May","June","July",
     "August","September",
     "October","November",
     "December");

[]REAL rainfall=
    ( 6.54, 12.3,  10.1,  13.83,
      5.04, 9.15,  14.34, 16.38,
      13.84, 10.45, 8.49, 7.57);

FOR m TO UPB months
DO
   STRING mm=months[m];
   print((mm,(12-UPB mm)*blank,
          fixed(rainfall[m],-5,2),
          newline))
OD
END
FINISH
```

Ex 9.14 The difficult part is calculating which number to print at each position.

```
PROGRAM ex9 14 CONTEXT VOID
USE standard
BEGIN
   print(("Table of square roots ",
          "1 to 100",
          newline,newline));

   FOR i TO 25
   DO
      FOR j TO 4
      DO
         INT number = (j-1)*25+i;
         print((whole(number,-6),
                fixed(sqrt(number),
                      -8,4)))
      OD;
      print(newline)
   OD
END
FINISH
```

Ex 9.15

```
PROGRAM ex9 15 CONTEXT VOID
USE standard
BEGIN
   REAL pi power:=1;
   print(("Table of powers of pi",
          " 1 to 10",
        newline,newline));

   FOR i TO 10
   DO
     pi power*:=pi;
     print((whole(i,-3),"  ",
            float(pi power,
                  12,6,2),
            newline))
   OD
END
FINISH
```

Ex 9.16 To write this program, you need to know how many bytes Algol 68 uses to store an integer in a binary book. In the program below, that number is presumed to be identified by `int bin bytes`. You will need to write a short program to output a couple of integers to a binary book and then see how long it is (and you might find its contents of interest).

```
PROGRAM ex9 16 CONTEXT VOID
USE standard
BEGIN
FILE work;

IF   establish(work,
               "ex9 16.tmp",
               stand back channel,
               0,0,0)/=0
THEN
   print("Cannot create workbook");
   exit(1)
FI;

FOR i TO 1000 DO put bin(work,i) OD;

INT int bin bytes=?;
CO Your value replaces ? CO

FOR i FROM 17 BY 17 TO 1000
DO
   set(work,0,0,(i-1)*int bin bytes);
   INT n;  get bin(work,n);
   print((n,newline))
OD;

close(work)
END
FINISH
```

Ex 9.17 Reading the words should not present any problems to you. The only new bit is the output. However, for the sake of completeness, here is the whole program.

```
PROGRAM ex9 17 CONTEXT VOID
USE standard
BEGIN
   FILE inf, out1, out2;

   IF open(inf,
           "inbook",
           stand in channel)/=0
   THEN
      print("Cannot open inbook");
```

```
      exit(1)
   ELIF establish(out1,
                  "outbook1",
                  stand out channel,
                  0,0,0)/=0
   THEN
      print("Cannot create outbook1");
      exit(2)
   ELIF establish(out2,
                  "outbook2",
                  stand out channel,
                  0,0,0)/=0
   THEN
      print("Cannot create outbook2");
      exit(3)
   FI;

   make term(inf, blank+cr+lf);

   STRING word; CHAR ch:=blank;

   on logical file end(inf,
    (REF FILE f)BOOL:
     (close(out1);
      close(out2);
      close(f);
      stop;  SKIP));

   DO
      get(inf,(word,
               skip terminators));

      IF UPB word > 0
      THEN
         put bin(out2,
                 (current pos(out1),
                  UPB word));
         put bin(out1,word)
      FI
   OD
END
FINISH
```

Ex 9.18 A useful wrinkle is to end your report with the words `END OF REPORT` so that your reader knows that there are no pages of the report which could have been lost. In a professionally written program, you would put a page number and the date of the report, but we have not yet covered how that can be done (see chapter 12).

```
PROGRAM ex9 18 CONTEXT VOID
USE standard
IF []STRING
     months =
      ("January","February","March",
       "April","May","June",
       "July","August","September",
       "October","November","December");
   []REAL
     rainfall =
      ( 6.54, 12.30, 10.10, 13,83,
        5.04,  9.15, 14.34, 16.38,
       13.84, 10.45, 8.49,   7.57);
   FILE prn;
   establish(prn,
             "rainfall.out",
             stand out channel,
             0,0,0)/=0
THEN
   put(stand error,
       ("Cannot establish rainfall.out",
        newline));  stop
ELSE
   put(prn,
       ("Rainfall figures in 1995",
        newline,newline,
        "Month",7*blank,
        "Rainfall in mm",
        newline));

   FOR m TO UPB months
   DO
     STRING mm = months[m];
     put(prn,
         (mm,(12-UPB mm)*blank,
          fixed(rainfall[m],-5,2),
          newline))
   OD;
   put(prn,
       (newline,
        "END OF REPORT",
        newline));
   close(prn)
END
```

```
FINISH
```

Ex 9.19 You will need to get the identification of the file from the argument line.

```
PROGRAM ex9 19 CONTEXT VOID
USE standard
IF STRING in idf;  FILE arg, inf, prn;
   open(arg,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access arguments",
        newline));
   exit(1)
ELIF
   on logical file end(arg,
    (REF FILE f)BOOL:
     (put(stand error,
          ("Usage: tt idf",
           newline));
      stop;  SKIP));
   get(arg,(LOC STRING,skip terminators,
            in idf));
   close(arg);
   open(inf,in idf,stand in channel)/=0
THEN
    put(stand error,
        ("Cannot open ",in idf,
         newline));
   exit(2)
ELIF
   establish(prn,"tt.out",
             stand out channel,
             0,0,0)/=0
THEN
   put(stand error,
       ("Cannot establish tt.out",
        newline));
   exit(3)
ELSE
   STRING line;
   on logical file end(inf,
    (REF FILE f)BOOL:
     (close(f);  close(prn);
      stop;  SKIP));

   FOR i
   DO
      get(inf,(line,newline));
      put(prn,(whole(i,-6),": "));
      IF UPB line > 0
```

```
      THEN put(prn,line)
      FI;
      newline(prn)
   OD
FI
FINISH
```

Ex 9.20

```
PROGRAM ex9 20 CONTEXT VOID
USE standard
BEGIN
   REAL r;

   WHILE read(r); r/=0.0
   DO
      print((float(r,-12,3,-2),newline))
   OD
END
FINISH
```

Ex 9.21 This program is not all that difficult. Take it slowly, step by step. Although reading an employee record only appears once in the program, it is better to write it as a procedure so as not to obscure the main logic. Likewise, printing each line of the report is also declared as a procedure. Notice how the given solution checks for errors.

```
PROGRAM ex9 21 CONTEXT VOID
USE standard
BEGIN
   FILE arg, emp, prn;
   STRING emp idf;
   INT week:=0;

   IF open(arg,"",arg channel)/=0
   THEN
     put(stand error,
        ("Cannot access the arguments",
         newline));
     exit(1)
   ELIF
     on logical file end(arg,
      (REF FILE f)BOOL:
      (put(stand error,
          ("Usage: tt emp-book week-no",
           newline));
       exit(2); SKIP));
     get(arg,
        (LOC STRING,LOC CHAR,
         emp idf,week));
```

```
  week < 1 OR week > 53
THEN
   put(stand error,
       ("Invalid week number",
        newline));
   exit(3)
ELIF open(emp,
          emp idf,
          stand in channel)/=0
THEN
  put(stand error,
     ("Cannot open ",emp idf,
      newline));
  exit(4)
ELIF
  establish(prn,
            "report",
            stand out channel,
            0,0,0)/=0
THEN
  put(stand error,
     ("Cannot establish report",
      newline));
  exit(5)
FI;

MODE
  EMPLOYEE=STRUCT(STRING name,
                  [2]STRING address,
                  STRING dept,
                         ni code,
                         tax code,
                  REAL basic,
                       overtime,
                  [52]REAL
                     net pay,tax);

PROC get emp=(REF FILE f,
              REF EMPLOYEE e)VOID:
BEGIN
  [80]CHAR s;

  PROC get str=[]CHAR:
  (
     INT len;  get bin(f,len);
     [len]CHAR s;
     get bin(f,s);
     s
  ); \#get str\#
```

```
  IF (name OF e:=get str) /= ""
  THEN
    FOR i TO UPB address OF e
    DO
      (address OF e)[i]:=get str
    OD;
    dept OF e:=get str;
    ni code OF e:=get str;
    tax code OF e:=get str;
    get bin(f,(basic OF e,
                overtime OF e,
                net pay OF e,
                tax OF e))
  FI
END #get emp#;

PROC put emp=(REF FILE f,
              REF EMPLOYEE e)VOID:
  put(f,(name OF e,
         (40-UPB name OF e)*blank,
          fixed((net pay OF e)[week],
                 -8,2),
          newline));

INT line:=60, page:=0;

PROC heading = (REF FILE f)VOID:
IF line = 60
THEN line:=0; #reset the line count#
  put(f,
      (newpage,
       "Report of net pay for week ",
       whole(week,0),
       40*blank,"Page ",
       whole(page+:=1,0),
       newline,newline,
       "Employee name",
       28*blank,"Net pay",
       newline,newline))
FI #heading#;

EMPLOYEE employee;
REAL total pay:=0; INT n:=0;

on logical file end(emp,
 (REF FILE f)BOOL:
  (put(prn,
       ("Total net pay for ",
```

```
            whole(n,0),
            " employees =",
            fixed(total pay,-11,2),
            newline,
            newline,
            "End of report",newline));
      close(f);  close(prn);  stop;
      SKIP));

   DO
      heading(prn);
      get emp(emp,employee);

      IF name OF employee /= ""
      THEN
         total pay+:=
          (net pay OF employee)[week];
         n+:=1;
         #count of total employees#
         put emp(prn,employee);
         line+:=1
      FI
   OD
END
FINISH
```

A.10 Chapter 10

Ex 10.1 Deproceduring and dereferencing (not weakly-dereferencing).

Ex 10.2 None.

Ex 10.3

(a) Yes.

(b) No (cannot widen).

(c) No (cannot dereference).

(d) No (cannot row).

(e) No (cannot dereference).

(f) No (cannot unite after rowing).

Ex 10.4

(a) Row-display, structure-display, collateral clause.
(b) Parallel clause.
(c) Case clause.
(d) Conformity clause.
(e) Conditional clause.
(f) Closed clause or enclosed clause.

Ex 10.5

(a) Weak.
(b) Meek.

Ex 10.6

(a) 6 (4 denotations, 1 applied-identifier, 1 closed clause).
(b) 5 (1 denotation, 3 applied-identifiers, 1 call).
(c) 5 (1 denotation, 3 applied-identifiers, 1 slice).
(d) (1 denotation, 1 closed clause, 1 cast, 1 applied-identifier).

Ex 10.7 The identifier of a structure or a name referring to a structure.

Ex 10.8 A selection.

Ex 10.9

(a) 2.
(b) 3.
(c) 3.
(d) 4.

Ex 10.10

(a) A primary.

(b) A primary.

(c) A secondary.

(d) A primary.

(e) A primary.

(f) Tertiary.

(g) Enclosed clause.

(h) A quaternary.

(i) It is not a unit.

(j) A quaternary.

Ex 10.11

(a) 2 denotations + 2 applied-identifiers = 4 primaries. 1 closed clause. 3 formulæ = 3 tertiaries.

(b) 1 denotation + 3 applied-identifiers = 4 primaries. 3 formulæ = 3 tertiaries.

(c) 2 applied-identifiers + 1 call = 3 primaries.

(d) 3 denotations + 1 applied-identifier + 1 slice = 5 primaries.

(e) 2 denotations + 3 applied-identifiers = 5 primaries; 1 conditional clause = 1 enclosed clause, 2 formulæ = 2 tertiaries, 1 assignment = 1 quaternary.

(f) 2 denotations + 5 applied-identifiers = 7 primaries, 1 formula = 1 tertiary, 1 assignation = 1 quaternary, 1 case clause + 1 conditional clause = 2 enclosed clauses.

(g) 2 denotations + 2 applied-identifiers = 4 primaries, 2 assignments = 2 quaternaries, 1 parallel clause = 1 enclosed clause.

Ex 10.12

(a) The conditional clause can yield a value of mode `REF INT` or `REF REAL`. In a firm context, these can be coerced to `INT` and `REAL`. Thus the `INT` is widened to `REAL` and the balanced clause yields a value of mode `REAL`.

(b) The conditional clause in a soft context will yield `REF INT` or `REF REAL`. Neither can be coerced to the other in a strong context, so the clause cannot be balanced. The error message from the compiler arises from the coercions applied in a strong context for the attempted balancing.

(c) The conformity clause yields `INT` or `REAL`. In a strong context, `INT` can be widened to `REAL`. Thus the balanced clause will yield `REAL`.

(d) The conditional clause yields `INT` or whatever. In a strong context, `SKIP` will yield `INT`. Thus the balanced clause yields `INT`. However, the result will be undefined if the `SKIP` is used in the assignment.

Ex 10.13

(a) Yes.
(b) Yes.
(c) No.
(d) No.
(e) Yes.
(f) Yes.
(g) Yes.
(h) No.
(i) Yes! It's an example in the "Revised Report".

A.11 Chapter 11

Ex 11.1

```
PROGRAM ex11 1 CONTEXT VOID
USE standard
BEGIN
   []CHAR digits =
         "0123456789abcdef"[@0];

   PROC itostr = (INT n,r#adix#)STRING:
   IF n<r
   THEN digits[n]
   ELSE itostr(n%r,r)+digits[n%*r]
   FI;

   print(("Table of numbers 0--15",
          newline,newline,
          "Dec. Hex. Binary",newline));

   FOR i FROM 0 TO 15
   DO
      STRING bin = itostr(i,2),
             dec = itostr(i,10),
             hex = digits[i];
             #only one digit#
      print(((4-UPB dec)*blank,
             dec,3*blank,hex,
             4*blank,(4-UPB bin)*"0",
             bin,newline))
   OD
END
FINISH
```

Ex 11.2

(a)

$$\begin{aligned} 94_{10} &= 5 \times 16^1 + 14 \times 16^0 \\ &= 5\mathrm{e}_{16} \end{aligned}$$

(b)

$$\begin{aligned} 13_{10} &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1101_2 \end{aligned}$$

(c)

$$1111\,1001_2 = \mathrm{f}9_{16}$$

(d)

$$\begin{aligned} 3\mathrm{e}1_{16} &= 3 \times 16^2 + \mathrm{e} \times 16^1 + 1 \times 16^0 \\ &= 3 \times 256 + 14 \times 16 + 1 \\ &= 768 + 224 + 1 \\ &= 993_{10} \end{aligned}$$

(e) $2\mathrm{c}_{16} = 0010\,1100_2$.

(f)

$$\begin{aligned} 10101_2 &= 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 \\ &= 16 + 4 + 1 \\ &= 21_{10} \end{aligned}$$

Ex 11.3

(a) 1011_2

(b) $\mathrm{e}3_{16}$

(c) 56_8

Ex 11.4

(a) `16r 0101 0101`

(b) `16r 99bb ddff`

(c) `16r 6745 2301`

(d) `FALSE`

Ex 11.5

(a) `16r 558`

(b) `16r 17`

Ex 11.6

```
PROC transpose=(REF[,]INT m)VOID:
IF   1 UPB m - 1 LWB m
            =
     2 UPB m - 2 LWB m
THEN #m is square#
   REF[,]INT mm=m[@1,@1]; #a precaution#

   FOR i TO 1 UPB mm - 1
   DO
      REF[]INT mr=mm[i,i+1:],
               mc=mm[i+1:,i];
      []INT temp=mr;
      mr:=mc;  mc:=temp
   OD
FI
```

Ex 11.7 Use a cast: `REF REAL(xx):=120.5`

Ex 11.8

```
REF REF[]CHAR rrq;
[]CHAR m = "ABCDEFGHIJ";
rrq:=LOC REF[]CHAR:=LOC[10]CHAR:=m[@1];
```

Ex 11.9

```
REF FLEX[]INT rfi;
rfi:=FLEX[1:0]INT:=(3,-2,4)
```

Ex 11.10 `f` has the mode `REF STRING` and `ss` has the mode `REF REF STRING`.

Ex 11.11 The multiple of mode `STRING` whose value is `"Joan of Arc"`.

Ex 11.12 `f[3:4]=s[7:8]`. The modes are both `STRING`.

Ex 11.13 Here are three possible answers:

```
REF STRING(ff) IS ss
ff IS REF STRING(ss)
REF STRING(ff) IS REF STRING(ss)
```

You could also use `ISNT`, `:=:` or `:/=:`.

Ex 11.14

(a) A name of mode `REF REF FILE`.

(b) `TRUE  BOOL`.

(c) A name of mode `REF FILE`.

(d) `FALSE  BOOL`.

Ex 11.15

(a) `REF FILE`

(b) `REF REF FILE`

Ex 11.16
```
REF REF QUEUE(tail):=
    LOC QUEUE:=(("Barbara",3),nilq)
```

Ex 11.17 `tail:=next OF tail`

Ex 11.18 No.

Ex 11.19
```
PROC add fan=(REF REF REF QUEUE
                 head,tail,
              REF FAN fan)VOID:
   tail:=next OF (REF REF QUEUE
          (head IS nilq|head|tail):=
             HEAP QUEUE:=(fan,nilq))
```

Ex 11.20
```
PROGRAM ex11 20 CONTEXT VOID
USE standard
BEGIN
   MODE FAN = STRUCT(STRING name,
                     INT ticket),
      QUEUE = STRUCT(FAN fan,
                     REF QUEUE next);

   REF QUEUE nilq = NIL;

   PROC add fan=(REF REF REF QUEUE
                     head,tail,
                  REF FAN fan)VOID:
      tail:=next OF
             (REF REF QUEUE
               (head IS nilq|head|tail)
                  :=HEAP QUEUE
                  :=(fan,nilq);

   REF REF QUEUE head,tail;
```

```
   head:=tail:=LOC REF QUEUE:=nilq;

   FOR q TO 1000
   DO
      add fan(head,tail,
              LOC FAN:=(IF ODD q
                        THEN "Iain"
                        ELSE "Fiona"
                        FI,
                        q))
   OD
END
FINISH
```

The generator `LOC FAN` is used because `add fan` requires a parameter of mode `REF FAN`. The scope of the generated name is from the declarations of `head` and `tail` to the end of the program because there are no identity declarations in the `FOR` loop clause (therefore it is not a range).

Ex 11.21 Because `marker` has mode `REF REF QUEUE`, it is made to refer to each `REF QUEUE` name in the linked-list. The condition

```
next OF marker ISNT nilq
```

ensures that `marker` is not currently referring to the last `REF QUEUE` in the list. The loop will terminate when `marker` refers to the last `REF QUEUE` in the list or the number of the ticket of the fan to be inserted in the queue does not exceed the number of the ticket of the fan referred to by `marker`. If the operator `AND` had been used, both operands would have been elaborated before the operator; in which case, if the left operand had yielded `FALSE`, elaboration of the right operand would have caused the run-time error `"Selection from NIL"`.

Ex 11.22 This can best be done by writing a program. Here is a possible solution:-

```
PROGRAM ex11 22 CONTEXT VOID
USE standard
BEGIN
   MODE FAN = STRUCT(STRING name,
                     INT ticket),
      QUEUE = STRUCT(FAN fan,
                     REF QUEUE next);
   REF QUEUE nilq = NIL;

   PROC insert fan = . . .
   . . .
   PROC print queue = . . .
   . . .
   REF REF QUEUE head,tail;
   head:=tail:=LOC REF QUEUE:=nilq;
```

```
   INT max ticket = 1000;
   INT tickets issued:=0;
   [max ticket]BOOL ticket issued;

   FOR i
   FROM LWB ticket issued
   TO UPB ticket issued
   DO FALSE OD;

   WHILE tickets issued < max ticket
   DO
      INT i=random int(max ticket);
      IF REF BOOL ti=ticket issued[i];
         NOT ti
      THEN
         ti:=TRUE;
         insert fan(head,tail,HEAP FAN:=
                    ((ODD i
                     |"Iain"
                     |"Fiona"
                     ),i));
         tickets issued+:=1
      FI
   OD #fans added to the queue#;

   print queue(head)
END FINISH
```

Instead of sending the output to `stand out`, it would be better to direct it to an output book so that the results could be examined at leisure. Alternatively, command line redirection could be used. The use of `ticket issued` ensures that unique ticket numbers are added to the queue since `insert fan` does not cater explicitly for duplicate ticket numbers.

Ex 11.23 The procedure has to find the fan concerned and must keep track of the reference to that fan.

```
PROC delete fan=(REF REF QUEUE q,
                 INT t#icket#
                )UNION(REF FAN,BOOL):
IF   q IS nilq
THEN FALSE #empty queue#
ELIF next OF q IS nilq
THEN #last fan in the queue#
   IF   ticket OF q = t
   THEN REF FAN rf = q;
      q:=nilq;  #delete last fan#
      rf
   ELSE FALSE
   FI
ELIF ticket OF next OF q < t
```

```
THEN delete fan(next OF q,t)
ELIF ticket OF next OF q > t
THEN #not found# FALSE
ELSE REF FAN rf = next OF q;
   next OF q:=next OF next OF q;
   rf
FI #delete fan#;
```

In the assignment, the mode of `next OF q` is `REF REF QUEUE`, so the mode of `next OF next OF q` must be `REF QUEUE`. Look at the required dereferencing to see what is assigned to `next OF q`.

Ex 11.24

```
PROGRAM ex11 24 CONTEXT VOID
USE standard
BEGIN
   MODE
   LETTER=STRUCT(CHAR c,INT o),
   TREE=STRUCT(REF LETTER l,
               REF TREE left,right);

   REF TREE leaf=NIL;
   REF TREE root:=leaf;

   PROC get letter=(REF FILE f)
                    REF LETTER:
   IF   CHAR ch;  get(f,ch);
        ch>="A" & ch<="Z"
               OR
        ch>="a" & ch<="z"
   THEN HEAP LETTER:=(ch,1)
   ELSE get letter(f) #skip non-letters#
   FI #get letter#;

   PROC add letter=
         (REF REF TREE root,
          REF LETTER let)VOID:
   IF   root IS leaf
   THEN root:=HEAP TREE:=(let,leaf,leaf)
   ELIF c OF l OF root > c OF let
   THEN add letter(left OF root,let)
   ELIF c OF l OF root < c OF let
   THEN add letter(right OF root,let)
   ELSE o OF l OF root+:=1
   FI #add letter#;

   FILE inf, arg;
   STRING in bk;
```

```
INT max row=13;
[max row,81]CHAR out page;
INT row:=max row, col:=0;

FOR i TO max row
DO
   out page[i,:80]:=80*blank;
   out page[i,81]:=lf
OD #initialise out page#;

INT num letters:=0;

PROC put letter=(REF LETTER let)VOID:
BEGIN
   IF   row=max row
   THEN col+:=1;  row:=1
   ELSE row+:=1
   FI;
   FILE f;
   establish(f,
             "",
             mem channel,
             1,1,20);
   put(f,(c OF let,
          fixed(o OF let/
                  num letters*100,
                -7,2),blank*12));
   out page[row,(col-1)*20+1:col*20]
     :=file buffer(f);
   close(f)
END #put letter#;

PROC print tree=
      (REF REF TREE root)VOID:
IF   root ISNT leaf
THEN
   print tree(left OF root);
   IF o OF l OF root > 0
   THEN put letter(l OF root)
   FI;
   print tree(right OF root)
FI #print tree#;

IF open(arg,"",arg channel)/=0
THEN
   put(stand error,
       ("Cannot access arguments",
        newline));
   stop
```

```
ELIF
   on logical file end(arg,
    (REF FILE f)BOOL:
     (put(stand error,
       ("Usage: tt in-book",
        newline)); stop; SKIP));
   get(arg,(LOC STRING,
            LOC CHAR,
            in bk));
   open(inf,
        in bk,
        stand in channel)/=0
THEN
   put(stand error,
       ("Cannot open book ",in bk,
        newline));
   stop
ELSE
   on logical file end(inf,
    (REF FILE f)BOOL:
     (
      print tree(root);
      print((
         "Frequency of occurrence ",
         "of letters in the book ",
         idf(f),newline,
         newline,out page,newline,
         "Total letters read: ",
         whole(num letters,0),
         newline));
      stop;  SKIP
     ))
FI;

FOR i TO 26 #letters in the alphabet#
DO
   add letter(
      root,
      HEAP LETTER:=
       (REPR(ABS("A")-1+i),0));
   add letter(
      root,
      HEAP LETTER:=
       (REPR(ABS("a")-1+i),0))
OD #all letters are now in the tree#;

DO
   add letter(root,get letter(inf));
   num letters+:=1
```

```
   OD
END
FINISH
```

Bibliography

For a thorough treatment of the language from a more old-fashioned point of view, I can recommend this book:-

- Lindsey, C. H. and van der Meulen, S. G., Informal Introduction to Algol 68, North-Holland (1977).

The original report is not for the faint-hearted, but it is the final arbiter of what constitutes Algol 68. Do not make the mistake of the many detractors of Algol 68 who confused the method of description (a two-level grammar) with the language itself. If you have read as far as here, you will know that Algol 68 is easier to learn than to describe:-

- van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, S., Lindsey, C. H., Meertens, L. G. L. T. and Fisker, R. G. (eds), Revised Report on the Algorithmic Language Algol 68, Springer-Verlag (1976).

This little book contains much wisdom about solving problems. It *is* geared towards mathematical problems, but you should not find it too difficult to apply to a whole range of other problems. It used to be the set book for the *Foundation Course* in Mathematics at the Open University:-

- Pólya, G., How to Solve It, 2nd ed., Penguin Books (1985).

Jackson's original book is well worth reading if you are considering taking up programming seriously or even if you are already a professional programmer:-

- Jackson, M. A., Principles of Program Design, Academic Press (1975).

Details of the floating-point processor within the Intel Pentium microprocessor were taken from the following books:-

- Intel Architecture Software Developer's Manual, *Volume I, Basic Architecture*, Intel Corporation, 1999.
- Intel Architecture Software Developer's Manual, *Volume II, Instruction Set Reference*, Intel Corporation, 1999.

Index