MINISTRY OF DEFENCE
(Procurement Executive)

OFFICIAL DEFINITION OF CORAL 66

Prepared by the Inter-Establishment Committee on Computer Applications as a language standard for military programming

London Her Majesty's Stationery Office

ISBN 0 11 470221 7

Preface

Coral 66 is a general-purpose programming language based on Algol 60, with some features from Coral 64 and Jovial, and some from Fortran. It was originally designed in 1966 by I. F. Currie and M. Griffiths of the Royal Radar Establishment in response to the need for a compiler on a fixed-point computer in a control environment. In such fields of application, some debasement of high-level language ideals is acceptable if, in return, there is a worthwhile gain in speed of compilation with minimal equipment and in efficiency of object code. The need for a language which takes these requirements into account, even though it may not be fully machine-independent, is widely felt in industrial and military work. We have therefore formalized the definition of Coral 66, taking advantage of experience gained in the use of the language. Under the auspices of the Inter-Establishment Committee for Computer Applications, we have had technical advice from staff of the Royal Naval Scientific Service, the Royal Armament Research and Development Establishment, the Royal Radar Establishment, the Defence ADP Training Centre, from serving officers of all three services and from interested sections of industry, to all of whom acknowledgments are due.

The present definition is an inter-service standard for military programming, and has also been widely adopted for civil purposes in the British control and automation industry. Such civil usage is supported by RRE and by the National Computing Centre at Manchester, on behalf of the Department of Industry. The NCC has agreed to provide information services and training facilities, and enquiries about Coral 66 for industrial application should be directed to that organization.

Royal Radar Establishment
Malvern
Worcs.
June, 1974

P. M. WOODWARD
P. R. WETHERALL
B. GORMAN

Contents

1

Introduction

It is virtually impossible to design a standard language such that programs will run with equally high efficiency in all types of computer and in any applications. Much of the design of Coral 66 reflects this difficulty. For example, the language permits the use of non-standard 'code statements' for any parts of a program where it may be important to exploit particular hardware facilities. A special feature is scaled fixed-point arithmetic for use in small fixed-point machines; the floating point facilities of the language can be omitted when hardware limitations make the use of floating-point arithmetic uneconomical. Other features also may be dropped without reducing the power of the language to an unacceptably low standard. Some reduced levels of implementation are suggested in Appendix 3 to this definition.

1.1 Special-purpose languages

A clear distinction must be made between general-purpose languages for use by skilled programmers, and more limited languages designed to incorporate the inbuilt assumptions of specialized applications or to make direct computer access practical for the non-specialist user. Coral 66 belongs to the first category. Languages in this class are suitable for writing compilers and interpreters as well as for direct application. Special-purpose languages can therefore be implemented by means of software written in Coral 66, backed up as required with suites of specialized macros or procedures. It is largely for this reason that the facilities for using procedures have been kept as general as possible. The main differences between Coral 66 procedures and those of Algol 60 lie in the replacement of the Algol 60 dynamic 'name parameter' by the more efficient 'location' or reference parameter used in Fortran, and the requirement to declare recursive procedures explicitly as such, again in the interest of object-code efficiency.

1.2 Real time

The theory and structure of programming for real-time computer applications has not yet advanced to such a point that a particular choice of language facilities is inevitable. Further, the design of real-time languages is handicapped by the lack of agreed standard software interfaces for applications programmers or compiler writers. This does not imply that real-time programs cannot yet be written in high-level language. The use of Coral 66 in real-time applications implies the presence of a supervisory system for the control of communications, which may have been designed independently of the compiler. The programmer's control over external events, and the computer's reaction to them, is expressed by the use of procedures or macros which communicate with the outside world indirectly through the agency of the supervisory software. No fixed conventions are laid down for the names or action of such calls on the supervisor.

1.3 Syntax

The widespread use of syntax-driven methods of compilation lends increasing importance to syntax methods of language description. The present definition takes the form of a commentary on the syntax of Coral 66, and therefore starts with broad structure, working downwards to finer detail. For reasons of legibility, the customary Backus notation has been dropped in favour of a system relying on typographical layout. Each syntax rule has on its left-hand side a class name, such as Statement. Such names appear in lower case without spaces, and with an initial capital letter. On the right-hand side of a rule are found the various alternative expansions for the class. These alternatives are printed each on a new line. Where a single alternative spreads over more than one line of print, the continuation lines are inset in relation to the starting position of the alternatives. Each alternative expansion consists of a sequence of items separated by spaces. The items themselves are either further class names or 'terminal symbols' such as BEGIN. The class name Void is used for an empty class. For example, a typical pair of rules might be

```
Specimen = ALPHA Sign
           BETA Sign

Sign     = +
           —
           Void
```

Examples of legal specimens are ALPHA+ and BETA. The equals sign is used to separate the left-hand side from the right, except after its first appearance in a rule.

1.4 Implementation

Considerations of software engineering have been allowed to influence the design of Coral 66, principally to ensure the possibility of rapid compilation, loading and execution. Conceptually, Coral 66 compilation is a one-pass process. The insistence that identifiers are fully declared or specified before use simplifies the compiler by ensuring that all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analysers with no back-tracking to be employed. Features which require elaborate hardware in the object machine for efficient program execution, for example dynamic storage allocation, are not included in the language. Unless run in a special diagnostic mode, a Coral 66 compiler is not expected to generate run-time checks on subscript bounds. No run-time checking of procedure entries is necessary. The arrangements for separate compilation of program segments are designed to minimize load-time overheads, but the specification of the interface between a Coral 66 compiler and the loader is outside the scope of the present document.

2

The Coral 66 program

A distinction is made between *symbols* and *characters*. Characters, standing only for themselves, may be used in 'strings' or as literal constants. Apart from such occurrences, a program is regarded as a sequence of symbols, each visibly representable by a unique character or combination of characters. The symbols of the language are defined (Appendix 2), but the characters are not. For the purpose of the language definition, words in upper case letters are treated as single symbols. Lower case letters are reserved for use in *identifiers*, which may also include digits in non-leading positions. Except where they are used in strings, layout characters are ignored by a Coral 66 compiler.

2.1 Objects

A program is made up of symbols (such as BEGIN, =, 4) and arbitrary identifiers which, by declaration, specification or setting acquire the status of single symbols. Identifiers are names referring to *objects* which are classified as

data (numbers, arrays of numbers, tables)
places (labels and switches)
procedures (functions and processes)

2.2 Program

A program need not be compiled in one unit, but may be divided into *segments* for separate compilation. To make it possible to refer to chosen objects in different segments, the names and types of such objects are written outside the program segments in *communicators*. Objects fully defined within the program are rendered accessible to all segments by their mention in a COMMON communicator (sections 3.3 and 9.1). Objects whose full definition lies outside the program, for example library procedures, can be made

accessible to all segments by mention in forms of communicator whose definition will be implementation-dependent. A Coral 66 program will thus comprise

name of program
optional communicators
named segments

in some appropriate sequence. Each program segment is in the form of a *block* (section 3). The language definition does not specify how the program or its segments shall be named or how the segments are to be separated or terminated, but when a whole program is compiled together, a typical form might be:

```
name of program
COMMON etc ;
segment name 1
BEGIN . . . END ;
segment name 2
BEGIN . . . END
FINISH
```

The program starts running from the beginning of a segment, the choice of which will depend upon a convention or mechanism outside the definition of the language.

3

Scoping

A named object can be brought into existence for part of a program and may have no existence elsewhere (but see section 4.7). The part of the program in which it is declared to exist is known as its *scope*. One effect of scoping is to increase the freedom of choosing names for objects whose scopes do not overlap. The other effect is economy of computer storage space. The scope of an object is settled by the block structure of the program as described below.

3.1 Block structure

A block is a statement consisting, internally, of a sequence of *declarations* followed by a sequence of *statements* punctuated by semi-colons and all bracketed by a BEGIN and END. Formally,

```
Block    = BEGIN  Declist  ;  Statementlist  END

Declist  = Dec
           Dec  ;  Declist

Dec      = Datadec
           Overlaydec
           Switchdec
           Proceduredec

Datadec  = Numberdec
           Arraydec
           Tabledec
```

The declarations have the purpose of fully classifying new objects and providing them with names (identifiers). As a statement can be itself a block merely by having the right form, blocks may be nested to an arbitrary depth. Except for global objects (section 3.3), the scope of an object is the block in which it is declared, and within this block the object is said to be *local*. The scope penetrates inner blocks, where the object is said to be *non-local*.

3.2 Clashing of names

If two objects have the same name and their scopes overlap, the clash of definitions could give rise to ambiguity. Typically, a clash occurs when an inner block is opened and a local object is declared to have the same name as a non-local object which already exists. In this situation, the non-local object continues to exist through the inner block (e.g. a variable maintains its value), but it becomes temporarily inaccessible. The local meaning of the identifier always takes precedence.

3.3 Globals

A program consists of a number of segments, each of which may be described as an *outermost block*, as there is no formal block surrounding the segments. In addition to objects which are local to inner blocks or outermost blocks, *global* objects may be defined. Such objects may be used in any segment, as their scope is the entire program. To become global, an object must be named in a communicator written outside the segments. For some types of object, such as COMMON data references, this takes the form of a declaration (and is the only declaration required). Other types of object, specifically COMMON labels, COMMON switches and COMMON procedures, must be fully defined within a segment. This means that COMMON labels must be set, and COMMON switches and procedures must be declared, in one of the outermost blocks of the program. Such objects are merely 'specified' in the COMMON communicator, as described in section 9.1, and are treated as local in *every* outermost block of the program. Global objects *declared* outside the segments are treated as non-local. All globals are non-local in all the inner blocks of any segment. With these rules of locality, questions of clashing are resolved in accordance with section 3.2.

3.4 Labels

Any statement may be labelled by writing in front of it an indentifier and a colon. The scope of a label is the smallest block embracing the statement which is labelled, extending from BEGIN to END. Thus labels can be used before they have been set. It also follows that the only means of entering an inner block is through its BEGIN. It is possible to jump into an outermost block from a different segment by the use of a COMMON label (or switch or procedure).

3.5 Restrictions connected with scoping

No identifier other than a label may be used before it has been declared or specified. Specification means that the type of object to which an identifier refers has been given, but not necessarily the full definition of the object (see section 9.1). Typically, a procedure identifier is specified as referring to a certain type of procedure with certain types of parameters by the heading of the procedure declaration, but the procedure is not fully defined until the end of the declaration as a whole. As an example of this, assume that two procedures f and g are declared in succession after the beginning of a segment. Then the body of g may call on itself or on the procedure f, but the body of f may not call on the procedure g unless g has been specified in a COMMON communicator. If a procedure is defined in a manner which directly or indirectly calls on itself, that procedure is said to be recursive and must be explicitly declared as such.

4

Reference to data

4.1 Numeric types

There are three types of number, floating-point, fixed-point and integer. Except in certain part-word table-elements (section 4.4.2.2), all three types are signed. Numeric type is indicated by the word FLOATING or INTEGER, or by the word FIXED followed by scaling constants which must be given numerically, e.g.

FIXED (13,5)

This specifies five fractional bits and a minimum of 13 bits to represent the number as a whole, including the fractional bits and a sign. The number of fractional bits may be negative, zero or positive, and may cause the binary point to fall outside the significant field of the number. It is assumed throughout this definition that a number is confined within a single computer word. If, in any implementation, a different system is adopted, e.g. two words for a floating-point number, a systematically modified interpretation of the language definition will be necessary. The syntax for numeric type is

```
Numbertype   = FLOATING
               FIXED Scale
               INTEGER

Scale        = ( Totalbits , Fractionbits )

Totalbits    = Integer

Fractionbits = Signedinteger
```

4.2 Simple references

The simplest objects of data are single numbers of floating, fixed-point or integer types. Identifiers may refer to such objects if suitably declared, e.g.

INTEGER i, j, k;
FIXED (13,5) x, y

and the declarations may optionally include assignment of initial values. This is known as presetting and is described in section 4.6. The syntax for a number declaration is

```
Numberdec = Numbertype Idlist Presetlist
Idlist    = Id
            Id , Idlist
```

4.3 Array references

An array is restricted to a one or two dimensional set of numbers all of the same type (including scale for fixed-point). An array is represented by an identifier, suitably declared with, for each dimension, a lower and upper index bound in the form of a pair of integer constants, e.g.

```
FIXED (13,5) ARRAY b[0:10];
FLOATING ARRAY c[1:3,1:3]
```

The lower bound must never exceed the corresponding upper bound. If more than one array is required with the same numeric type, and the same dimensions and bounds, a list of array identifiers separated by commas may replace the single identifiers shown in the above examples. Arrays with the same numeric type but different bounds or dimensions may also be included in a composite declaration, as shown below.

```
INTEGER ARRAY p, q, r[1:3], s[1:4], t, u[1:2, 1:3]
```

An array identifier refers to an array in its entirety, but its use in statements is confined to the communication of the array reference to a procedure. Elsewhere, an array identifier must be indexed so that it refers to a single array element. The index, in the form of an arithmetic expression enclosed in square brackets after the array identifier, is evaluated to an integer as described in section 6.1.3. The syntax rules for an array declaration, which include a presetting facility (section 4.6.1), are:

```
Arraydec   = Numbertype ARRAY Arraylist Presetlist
Arraylist  = Arrayitem
             Arrayitem , Arraylist
Arrayitem  = Idlist [ Sizelist ]
Sizelist   = Dimension
             Dimension , Dimension
```

```
Dimension  = Lowerbound : Upperbound
Lowerbound = Signedinteger
Upperbound = Signedinteger
```

4.4 Packed data

There are two systems of referring to packed data, one in which an unnamed field is selected from any computer word which holds data (see section 6.1.1.2.2), and one in which the data format is declared in advance. In the latter system, with which this section is concerned, the format is replicated to form a *table*. A group of *n* words is arbitrarily partitioned into bit-fields (with no fields crossing a word boundary), and the same partitioning is applied to as many such groups (*m* say) as are required. The total data-space for a table is thus *nm* words. Each group is known as a *table-entry*. The fields are named, so that a combination of field identifier and entry index selects data from all or part of one computer word, known as a *table-element*. The elements in an entry may occupy overlapping fields, and need not together fill all the available space in the entry.

4.4.1 Table declaration

A table declaration serves two purposes. The first is to provide the table with an indentifier, and to associate this identifier with an allocation of word-storage sufficient for the width and number of entries specified. For example,

```
TABLE   april [3,30]
```

is the beginning of a declaration for the table 'april' with 30 entries each 3 words wide, requiring an allocation of 90 words in all. The second purpose of the declaration is to specify the structure of an entry by declaring the elements contained within it, as defined in section 4.4.2 below. Data-packing is implementation dependent, and the examples will be found to assume a wordlength of 24 bits. The syntax for a table declaration is

```
Tabledec        = TABLE  Id [ Width , Length ] [Elementdeclist
                    Elementpresetlist ] Presetlist
Elementdeclist  = Elementdec
                  Elementdec ; Elementdeclist
Width           = Integer
Length          = Integer
```

Details of the two presetting mechanisms are given in section 4.6.2.

4.4.2 Table-element declaration

A table-element declaration associates an element name with a numeric type and with a particular field of each and every entry in the table. The field may be the whole or part of a computer word, and the form of declaration differs accordingly. The syntax for an element declaration, more fully developed in section 4.4.2.2, is

```
Elementdec   = Id Numbertype Wordposition
               Id Partwordtype Wordposition , Bitposition
Wordposition = Signedinteger
Bitposition  = Integer
```

Word-position and bit-position are numbered from zero upwards, and the least significant digit of a word occupies bit-position zero. Normally, table-elements will be located so that they fall within the declared width of the table, but a Coral 66 compiler does not check the limits. To improve program legibility, it is suggested that the word BIT be permitted as an alternative to the comma in the above syntax. The meaning of Bitposition is given in section 4.4.2.2.

4.4.2.1 Whole-word table-elements

As shown in the syntax of the previous section, the form of declaration for whole-word table-elements is

```
Id Numbertype Wordposition
```

For example,

```
tickets INTEGER 0
```

declares a pseudo-array of elements named 'tickets'. (True array elements are located consecutively in store, section 4.5.) Each element refers to a (signed) integer occupying word-position zero in an entry. Similarly,

```
weight FIXED (16,–4) 1
```

locates 'weight' in word-position 1 with a significance of 16 bits, stopping 4 bits short of the binary point. Floating-point elements are similarly permitted.

4.4.2.2 Part-word table-elements

Elements which occupy fields narrower than a computer word (and only such elements) are declared in forms such as

```
rain UNSIGNED (4,2) 2,0;
humidity UNSIGNED (6,6) 2,8;
temperature (10,2) 2,14
```

for *fixed-point elements*. The fixed-point scaling is given in brackets (total bits and fraction bits), followed by the word- and bit-position of the field within the entry. Word-position is the word within which the field is located, and bit-position is the bit at the least significant end of the field. The word UNSIGNED increases the capacity of the field for positive numbers at the expense of eliminating negative numbers. For example, (4,2) allows numbers from –2.00 to 1.75, whilst UNSIGNED (4,2) allows them from 0.00 to 3.75. If the scale contains only a single integer, e.g.

```
sunshine UNSIGNED (4) 2,4
```

the number in brackets represents the total number of bits for a *part-word integer*. Though (4,0) and (4) have essentially the same significance, the fact that (4,0) indicates fixed-point type and (4) indicates an integer should be borne in mind when such references are used in expressions. The syntax of Partwordtype, for substitution in the syntax of section 4.4.2, is

```
Partwordtype = Elementscale
               UNSIGNED Elementscale
Elementscale = ( Totalbits , Fractionbits )
               ( Totalbits )
```

The rules for Totalbits and Fractionbits are in section 4.1. The number of fraction bits may be negative, zero or positive, and it is permissible for the binary point to lie outside the declared field.

4.4.3 Example of table declaration

```
TABLE april [ 3,30 ]
            [ tickets INTEGER 0;
              weight FIXED (16,–4) 1;
              rain UNSIGNED (4,2) 2,0;
              sunshine UNSIGNED (4) 2,4;
              humidity UNSIGNED (6,6) 2,8;
              temperature (10,2) 2,14 ]
```

It should be noted that all the numbers used to describe and locate fields must be constants.

4.4.4 Reference to tables and table-elements

A table-element is selected by indexing its field identifier. To continue from the example in section 4.4.3, the rain for april 6th would be written rain[5], for it should be noted that an entry always has the conventional lower bound of zero. In use, the names of table-elements are always indexed. On the other hand, a table identifier such as 'april' may stand on its own when a table reference is passed to a procedure. The use of an index with a table identifier does *not* (other than accidentally) select an entry of the table. It selects a computer word from the table data regarded as a conventional array of single computer words, with lower index bound zero. Thus the implied bounds of the array 'april' are 0 : 89. A word so selected is treated as a signed integer, from which it follows that april[6] in the example would be equivalents to tickets[2]. A table name is normally indexed only for the purpose of running through the table systematically, for example to set all data to zero, or to form a base for overlaying (section 4.8).

4.5 Storage allocation

Computer storage space for data is allocated automatically at compile time, one word for each simple reference, one for each array element, and as many as are declared for each table-entry. In any one composite declaration, a Coral 66 compiler is explicity required to perform allocation serially. For example, the declarations

```
INTEGER a , b , c ;
INTEGER p , q ;
```

will make the locations of a,b,c become n, n+1, n+2 respectively, and those of p, q become m, m+1 where n and m are undefined and unrelated. In two-dimensional arrays, the second index is stepped first: the declaration

```
INTEGER ARRAY   a[1:2], b[1:2, 1:2]
```

will locate the elements

```
a[1],  a[2],  b[1,1],  b[1,2],  b[2,1],  b[2,2]
```

in consecutive ascending locations.

4.6 Presetting

Certain objects of data may be initialized when the program is loaded into store by the inclusion of a presetting clause in the data declaration. Presetting

is not dynamic, and preset values which are altered by program are not reset unless the program or segment is reloaded. An object is not eligible for presetting if it is declared anywhere within

(a) the body of a recursive procedure, or

(b) an inner block of the program, or

(c) an inner block of a procedure body.

Procedure bodies do not count as blocks for the purpose of (b). For example, the integer i is eligible for presetting in a segment which begins as follows:

```
BEGIN PROCEDURE f;
      BEGIN PROCEDURE g;
            BEGIN INTEGER i
```

4.6.1 Presetting of simple references and arrays

The preset constants are listed at the end of the declaration after an assignment symbol, and are allocated in the order defined in section 4.5. As examples,

```
INTEGER a, b, c ← 1, 2, 3;
INTEGER ARRAY k[1:2, 1:2] ← 11, 12, 21, 22
```

If desired for legibility, round brackets may be used to group items of the presetlist, but such brackets are ignored by the compiler. The number of constants in the presetlist must not exceed, but may be less than, the number of words declared, and presetting ceases when the presetlist is exhausted. In special cases (see section 4.7), the preset assignment symbol may be the only part of the presetlist which is present. The syntax is

```
Presetlist    = ← Constantlist
                Void

Constantlist = Group
               Group , Constantlist

Group        = Constant
               ( Constantlist )
               Void
```

The main purpose of the final void will be seen in section 4.6.2. For the expansion of Constant, see section 10.2.

4.6.2 Presetting of tables

There are two alternative mechanisms. If the internal structure of a table is completely disregarded, the table can be treated as an ordinary one-dimensional array of whole computer words (4.4.4), and preset as such (4.6.1). Alternatively, all the table-elements may be preset after their declaration list, as shown at Elementpresetlist in the syntax of 4.4.1. For example,

```
        TABLE gears [1,3]
            [ teeth1 UNSIGNED (6) 0,0;
              teeth2 UNSIGNED (6) 0,6;
              ratio UNSIGNED (11,5) 0,12;
              arc UNSIGNED (5,5) 0,12
        PRESET (57,19,3.0, ), (50,25,2.0, ), (45,30,1.5, ) ]
```

For table-element presetting, the word PRESET is used instead of the assignment symbol of 4.6.1. Each entry of the table is preset in succession as a group of elements, taken in the order of their declaration. Voids in the list imply absence of assignment. This may be necessary to avoid duplication when fields overlap, as do 'ratio' and 'arc' in the above example. As in 4.6.1, brackets used for grouping constants in the list of presets are ignored by the complier. The syntax is

```
        Elementpresetlist = PRESET Constantlist
                            Void
```

The previous example could, with equal effect but less convenience, be expressed in the form

```
        TABLE gears [1,3]
            [  teeth1 UNSIGNED (6) 0,0;
               teeth2 UNSIGNED (6) 0,6;
               ratio UNSIGNED (11,5) 0,12;
               arc UNSIGNED (5,5) 0,12 ]
         ← OCTAL(1402371), OCTAL(1003162), OCTAL(603655)
```

4.7 Preservation of values

Objects of data may have no existence outside the scope of their declarations. The values to which local identifiers refer must in general be assumed undefined when a block is first entered and whenever it is subsequently re-entered. This is due to the fact that a block-structured language is designed for automatic overlaying of data. Local working space may therefore have been used for other purposes between one entry to a block and the next. In

Coral 66 this is not invariably the case. When a data declaration contains a presetlist as permitted by the rule given in section 4.6, the values of all the objects named in that declaration will remain undisturbed between successive entries to the block or procedure body, like 'own' variables in Algol 60. It is sufficient that a preset assignment symbol appears at the end of the declaration, even though the list of preset constants is void.

4.8 Overlay declarations

Overlaying may be found desirable when COMMON data is required in some segments and not in others, as it enables global data space to be re-used for other purposes. However, indiscriminate use of overlaying should be avoided, as it can lead to confusion and obscurity. The facility causes apparently different data references to refer simultaneously to the same objects of data, i.e. as alternative names for the same storage locations. To form an overlay declaration, an ordinary data declaration is preceded by a phrase of the form

```
                    OVERLAY Base WITH
```

where Base is a data reference which has previously been covered by a declaration in the same COMMON communicator or in the same segment. The base may be a simple reference, one-dimensional array reference or a table reference treated as a one-dimensional array of whole words. If the array or table identifier is not indexed, it refers to the location of its zero'th element (which may be conceptual). Storage allocated by the overlay declaration starts from the base, proceeds serially (as in 4.5) and will not be overlaid by succeeding declarations unless these are themselves overlay declarations. The syntax of an overlay declaration is

```
        Overlaydec = OVERLAY Base WITH Datadec

        Base       = Id
                     Id [ Signedinteger ]
```

5

Place references—Switches

Place references refer to positions of program statements, and the simplest position marker is the *label* (section 3.4). A *switch* is a preset and unalterable array of labels, which must be within scope at the switch declaration. Any use of the indexed switch name refers to the corresponding label. For example, the switch declaration

SWITCH s ← a, b, c

causes s[1] to refer to the label a, s[2] to b and s[3] to c. The syntax rules are

```
Switchdec = SWITCH Switch ← Labellist

Labellist = Label
            Label , Labellist

Switch    = Id

Label     = Id
```

6

Expressions

The term 'expression' is reserved for *arithmetic expressions.* Coral 66 has no designational expressions of Algol 60 type. As there are no Boolean variables and no bracketed Boolean expressions (see section 6.2.1), the expressions after IF are known as *conditions.* The syntax for expressions is

```
Expression              = Unconditionalexpression
                          Conditionalexpression

Unconditionalexpression = Simpleexpression
                          String
```

Strings are defined in section 10.4.

6.1 Simple expressions

Arithmetic is performed with the monadic and diadic adding operators + and −, and with the diadic multiplying operators * (multiply) and / (divide). The plus and minus operators work on *terms*, which are combinations of *factors* joined by multiplication or division. There is no exponentiation operator. The syntax for simple expression begins as follows

```
Simpleexpression = Term
                   Addoperator Term
                   Simpleexpression Addoperator Term

Term             = Factor
                   Term Multoperator Factor

Addoperator      = +
                   −

Multoperator     = *
                   /
```

6.1.1 Primaries

Primaries are the basic operands in expressions. For example, in the analysis of the expression

x + y * (a + b) − 4

we discover three terms, the primary x, the term y*(a + b) and the primary 4. The middle term is the product of two factors, the primary y and the primary (a + b). To complete the analysis, all expressions from within brackets are similarly analysed until no further reduction is possible and no expression brackets remain. When an expression contains no word-logical operators (see section 6.1.2), a factor must be a primary, which may or may not be of a defined type. Thus,

```
Factor  = Primary
          Booleanword
Primary = Untypedprimary
          Typedprimary
```

6.1.1.1 Untyped primaries

Untyped primaries are those operands which cannot be classed as integer, floating-point or fixed-point (of known scale) without reference to their context. For example, the number 3.1416 may be represented, with varying degrees of accuracy, in many different ways within a computer word. The same applies to an expression, whose type is determined by context (section 6.1.3).

```
Untypedprimary = Real
                 ( Expression )
```

A 'real' (section 10.2) is an unsigned numerical constant containing a decimal or octal point or a tens exponent.

6.1.1.2 Typed primaries

Typed primaries are classified as follows

```
Typed primary = Wordreference
                Partword
                LOCATION ( Wordreference )
                Numbertype ( Expression )
                Procedurecall
                Integer
```

6.1.1.2.1 Word references

A simple reference, or a reference to an array element or whole-word table-element, has a type defined in its declaration. Such references may be described as *word references* because they refer to items of data for which whole computer words are set aside. A further kind of word reference, the *anonymous reference*, takes the form

[Index]

where the index is any expression evaluated as an integer to give the actual location of a computer word. An anonymous reference possesses all the properties of an identified reference, except that it lacks an identifier. Just as a variable i, declared as INTEGER i, may be used in an expression to refer to the contents of the computer word allocated to i, so the use of an anonymous reference in an expression will refer to the contents of the address defined by Index. Such contents are taken to be of numeric type INTEGER, irrespective of any declaration which may have associated that word with some other type. See also section 6.1.1.2.3. The syntax for word reference is

```
Wordreference = Id
                Id [ Index ]
                Id [ Index , Index ]
                [ Index ]
Index         = Expression
```

6.1.1.2.2 Part-words

Any single item of packed data may act as a typed primary. Such an item is either

(a) a reference to a part-word table-element, or

(b) a specified field of any typed primary.

In case (a), the type is defined in the table declaration. In case (b), the desired field is selected by a prefix of the form

BITS [Totalbits , Bitposition]

in front of the primary to be operated upon. The result of this operation is a positive* integer value of width Totalbits and in units of the bit at 'Bitposition'. The value will in general be implementation-dependent, even though the operand must be typed, as no conventions are laid down for the internal representation of floating-point or fixed-point items of data. In

*It is assumed that Totalbits will not be set equal to the full wordlength.

all cases, however, the numeric type resulting from the application of BITS is INTEGER. The syntax for a part-word, which should be distinguished from that of a 'part-word reference' (section 7.1), is

```
Partword = Id [ Index ]
           BITS [ Totalbits , Bitposition ] Typedprimary
```

6.1.1.2.3 Locations

The computer location of any word reference is obtainable by the location operator which is written in the form

LOCATION (Wordreference)

and has a value of type INTEGER. It may be noted that if i and j refer to integers, [LOCATION(i)] is equivalent to i, and LOCATION ([j]) is equivalent to j. The reasoning is as follows. LOCATION (i) is the address of the computer word allocated to i. Enclosure in square brackets forms an entity equivalent to an identifier standing for this address, which by hypothesis is i. Similarly, [23] is equivalent to an identifier for the address 23, and LOCATION([23]) is the address for which this fictitious identifier stands, which is 23 by hypothesis.

6.1.1.2.4 Explicit type-changing

A typed primary may have its type changed, and an untyped primary may be typed, by enclosure within round brackets preceded by a specific Number-type as described in section 6.1.3.

6.1.1.2.5 Functions

The call of a typed procedure (section 8) may be treated as a function and used as a primary in any expression. For the syntax of a procedure call, see section 7.3.

6.1.1.2.6 Integers

An integer used in any expression (section 10.2) can be assumed to have the numeric type INTEGER before any necessary type-changes are enforced by context.

6.1.2 Word-logic

Three diadic logical operators are defined for use between typed primaries. The effect of these operators is implementation-dependent to the extent that the word-representation of data is not defined by the language. The *i*th bit of the result is a given logical function of the *i*th bits of the two operands, and the result as a whole has the numeric type INTEGER. To avoid confusion with Boolean operators in 'conditions' (section 6.2.1), a different terminology is used. The operators are

DIFFER	0	1
0	0	1
1	1	0

UNION	0	1
0	0	1
1	1	1

MASK	0	1
0	0	0
1	0	1

DIFFER is recognizable as 'not equivalent', UNION as 'inclusive or' and MASK as 'and'. The operators are shown in order of increasing tightness of binding. As bracketed expressions are untyped, the use of brackets to overcome binding priorities entails explicit integer scaling. For example,

a MASK INTEGER (b UNION c)

The Formal syntax, continued from 6.1.1, is

```
Booleanword  = Booleanword2
               Booleanword4 DIFFER Booleanword5

Booleanword2 = Booleanword3
               Booleanword5 UNION Booleanword6

Booleanword3 = Booleanword6 MASK Typedprimary

Booleanword4 = Booleanword
               Typedprimary

Booleanword5 = Booleanword2
               Typedprimary

Booleanword6 = Booleanword3
               Typedprimary
```

6.1.3 Evaluation of expressions

Expressions are used in assignment statements, as value parameters of procedures and as integer indexes, all of which contexts determine the numeric type finally required. Coral 66 expressions are automatically evaluated to this type, but in the process of calculation, data may be subjected by the compiler to various intermediate transformations. Although an algorithm for evaluating expressions does not form part of the official definition of the language, all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. In the simplest cases, this rule ensures predictable results, though it should be particularly noted that rounding-off errors will not be minimal, and overflow may occur. If an expression is enclosed in round brackets, its terms are not 'outermost', the rule no longer applies, and the algorithm for the particular compiler determines the sequence of events. The programmer can impose any desired system of evaluation by the use of Numbertype (Expression), which is a typed primary (section 6.1.1.2), any occurrence of which behaves like a variable, ref (say), declared as

```
        Numbertype ref;
```

and assigned a value by

```
        ref ← Expression
```

before it is used. For example, if i and j are integer references and x is a floating-point reference, the assignment statement

```
        x ← i − j
```

causes i and j to be converted to floating-point before subtraction, whilst

```
        x ← INTEGER ( i - j )
```

causes subtraction of integers before conversion to floating-point. Although the order of evaluation of an expression is undefined, the following rule concerning functions will apply. Value parameters of a function are necessarily evaluated before the function itself is computed, so that the order of evaluation of sin(cos(expn)) will be expn, cos, sin. Apart from this type of reversal, functions occurring in a simple expression will be evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets.

6.2 Conditional expressions

A conditional expression has the form

```
    Conditionalexpression = IF Condition
                               THEN Expression
                               ELSE Expression
```

with the usual interpretation.

6.2.1 Conditions

A condition is made up of arithmetic comparisons connected by Boolean operators OR and AND, of which AND is the more tightly binding. The permitted arithmetic comparisons are less than, less than or equal to, equal to, greater than or equal to, greater than, and not equal to. The syntax rules are

```
    Condition     = Condition OR Subcondition
                    Subcondition

    Subcondition  = Subcondition AND Comparison
                    Comparison

    Comparison    = Simpleexpression Comparator Simpleexpression
    Comparator    = <
                    ⩽
                    =
                    ⩾
                    >
                    ≠
```

The Boolean operators have their usual meanings, the OR being inclusive. Conditions are evaluated from left to right only as far as is necessary to determine their truth or falsity.

7

Statements

```
Statement        = Label : Statement
                   Simplestatement
                   Conditionalstatement
                   Forstatement

Simplestatement  = Assignmentstatement
                   Gotostatement
                   Procedurecall
                   Answerstatement
                   Codestatement
                   Compoundstatement
                   Block
                   Dummystatement
```

Statements are normally executed in the order in which they are written, except that a goto statement may interrupt this sequence without return, and a conditional statement may cause certain statements to be skipped.

7.1 Assignments

The left-hand side of an assignment statement is always a data reference, and the right-hand side an expression for procuring a numerical value. The result of assignment is that the left-hand side refers to the new value until this is changed by further assignment, or until the value is lost because the reference goes out of scope (but see section 4.7). The expression on the right-hand side is evaluated to the numeric type of the reference, with automatic scaling and rounding as necessary. The left-hand side may be a word reference as defined in section 6.1.1.2.1 or it may be a *part-word reference*, i.e. a part-word table-element or some selected field of a word reference. When assignment is made to a part-word reference, the remaining bits of the word are unaltered. As examples of assignment,

[LOCATION(i) + 1] ← 3.8

has the effect of placing the integer 4 in the location succeeding that allocated to i, and

BITS[2,6] x ← 3

has the effect of placing the binary digits 11 in bits 7 and 6 of the word allocated to x. This last assignment statement is treated in a similar manner to an assignment which has on its left-hand side an unsigned integer table-element. The statement

BITS[1,23] [LOCATION(i) + 1] ← 1

would in a 24-bit machine, force the sign bit in the indicated location to 'one'. The syntax of the assignment statement is

```
Assignmentstatement = Variable ← Expression

Variable            = Wordreference
                      Partwordreference

Partwordreference   = Id [ Index ]
                      BITS [ Totalbits , Bitposition ] Wordreference
```

There is no form of mutiple assignment statement.

7.2 Goto statements

The goto statement causes the next statement for execution to be the one having a given label. The label may be written explicitly after GOTO, or may be referenced by means of a switch whose index must lie within the range 1 to n where n is the number of labels specified in the switch declaration. See also section 3.4 and section 5. The syntax is

```
Gotostatement = GOTO Destination

Destination   = Label
                Switch [ Index ]
```

7.3 Procedure statements

A procedure identifier, followed in parentheses by a list of actual parameters (if any), is known generally as a *procedure call*. If the procedure possesses a value, it may be used as a primary in an expression, but whether it possesses a value or not, it may also stand alone as a statement. This causes

(i) the formal parameters in the procedure declaration to be replaced by the actuals in a manner which depends on the formal parameter specifications (see section 8.3),

(ii) the procedure body to be executed before the statement dynamically following the procedure statement is obeyed.

The syntax for a procedure call is

```
Procedurecall = Id
                Id ( Actuallist )

Actuallist    = Actual
                Actual , Actuallist

Actual        = Expression
                Wordreference
                Destination
                Name

Name          = Id
```

The purposes of the four types of actual parameter are defined in section 8.3.

7.4 Answer statements

An answer statement is used only within a procedure body, and is the means by which a value is given to the procedure. It causes an expression to be evaluated to the numeric type of the procedure, followed by automatic exit from the procedure body. The syntax is

```
Answerstatement = ANSWER Expression
```

7.5 Code statements

Any sequence of code instructions enclosed by CODE BEGIN and END may be used as a Coral 66 statement and it is recommended that code statements provide for the inclusion of nested Coral text. The form of the code is not defined; it may be the assembly code for a particular computer, or it may be at a higher level enabling available compiler features to be exploited. The code should, above all, enable the Coral programmer to exploit all available hardware facilities of the computer. For communication

between code and other statements, it must be possible to use any identifier of the program within the code statement, provided such identifiers are in scope. In some implementations, a code statement may be said to possess a value. The 'statement' may then be used as a primary in an expression, like a call of a typed procedure. Though not prohibited, this is not a standard feature of Coral 66, and may not be extended to other forms of statement. The syntax for a code statement is

Codestatement = CODE BEGIN Codesequence END

Codesequence = *defined in a particular implementation*

7.6 Compound statements

A compound statement is a sequence of statements grouped to form a single statement, for use where the syntactic structure of the language demands. Compound statements are transparent to scopes. It is therefore permitted to goto a label which is set inside a compound statement. The syntax is

```
Compoundstatement = BEGIN Statementlist END

Statementlist     = Statement
                    Statement ; Statementlist
```

7.7 Blocks

See section 3.

7.8 Dummy statements

A dummy statement is a void whose execution has no effect. For example, a dummy statement follows the colon in

; label: END

The syntax rule is

Dummystatement = Void

7.9 Conditional statements

The two forms of conditional statement are

```
Conditionalstatement = IF Condition THEN Consequence
                       IF Condition THEN Consequence ELSE
                           Alternative

Consequence          = Simplestatement
                       Label : Consequence

Alternative          = Statement
```

If the condition is true, the consequence is obeyed. If the condition is false and ELSE is present, the alternative is obeyed. If the condition is false and no ELSE is present, the conditional statement has no effect beyond evaluation of the condition.

7.10 For statements

The for-statement provides a means of executing repeatedly a given statement, the 'controlled statement', for different values of a chosen variable, the 'control variable', which may (or may not) occur within the controlled statement. A typical form of for-statement is

```
FOR i ← 1 STEP 1 UNTIL 4,
        6 STEP 2 UNTIL 10,
        15 STEP 5 UNTIL 30
        DO Statement
```

Other forms are exemplified by

```
FOR i ← 1, 2, 4, 7, 15 DO Statement
```

which is self-explanatory, and

```
FOR i ← i+1 WHILE x < y DO Statement
```

In the latter example, the clause 'i+1 WHILE x < y' counts as a single for-element and could be used as one element in a list of for-elements (the 'for-list'). As each for-element is exhausted, the next element in the list is taken. The syntax is

```
Forstatement = FOR Wordreference ← Forlist DO Statement

Forlist      = Forelement
               Forelement , Forlist
```

```
Forelement   = Expression
               Expression WHILE Condition
               Expression STEP Expression UNTIL Expression
```

The controlled variable is a word reference, i.e. either an anonymous reference or a declared word reference.

7.10.1 For-elements with STEP

Let the element be denoted by

```
e1 STEP e2 UNTIL e3
```

In contrast to Algol 60, the expressions are evaluated once only. Let their values be denoted by v1, v2 and v3 respectively. Then

(i) v1 is assigned to the control variable,

(ii) v1 is compared with v3. If (v1 – v3) * v2 > 0, then the for-element is exhausted, otherwise

(iii) the controlled statement is executed,

(iv) the value v1 is set from the controlled variable, then incremented by v2 and the cycle is repeated from (i).

7.10.2 For-elements with WHILE

Let the element be denoted by

```
e1 WHILE Condition
```

Then the sequence of operation is

(i) e1 is evaluated and assigned to the control variable,

(ii) the condition is tested. If false, the for-element is exhausted, otherwise

(iii) the controlled statement is executed and the cycle repeated from (i).

Unlike those in section 7.10.1, the expression e1 and those occurring in the condition are evaluated repeatedly.

8 Procedures

A procedure is a body of program, written out once only, named with an identifier, and available for execution anywhere within the scope of the identifier. There are three methods of communication between a procedure and its program environment.

(a) The body may use formal parameters, of types specified in the heading of the procedure declaration and represented by identifiers local to the body. When the procedure is called, the formal parameters are replaced by *actual* parameters, in one-to-one correspondence.

(b) The body may use non-local identifiers whose scopes embrace the body. Such identifiers are also accessible outside the procedure.

(c) An answer statement within the procedure body may compute a single value for the procedure, making its call suitable for use as a function in an expression. A procedure which possesses a value is known as a *typed procedure.*

The syntax for a procedure declaration is

```
Proceduredec = Answerspec PROCEDURE Procedureheading ;
                 Statement
               Answerspec RECURSIVE Procedureheading ;
                 Statement
```

The second of the above alternatives is the form of declaration used for recursive procedures (see section 3.5). The statement following the procedure heading is the procedure body, which contains an answer statement (section 7.4) unless the answer specification is void (8.1), and is treated as a block whether or not it includes any local declarations (8.4).

8.1 Answer specification

The value of a typed procedure is given by an answer statement (section 7.4) in its body; and its numeric type is specified at the front of the procedure

declaration. An untyped procedure has no answer statement, possesses no value, and has no answer specification in front of the word PROCEDURE.

```
Answerpec = Numbertype
            Void
```

8.2 Procedure heading

The procedure heading gives the procedure its name. It also describes and lists any identifiers used as formal parameters in the body. On a call of the procedure, the compiler sets up a correspondence between the actual parameters in the call and the formal parameters specified in the procedure heading. The syntax of the heading is

```
Procedureheading  = Id
                    Id ( Parameterspeclist )

Parameterspeclist = Parameterspec
                    Parameterspec ; Parameterspeclist
```

8.3 Parameter specification

Any object can be passed to a procedure by means of a parameter, whether it be an object of data, a place in the program, or a process to be executed. For data, there are two distinct levels of communication, *numerical values* (for input to the procedure) and *data references* (for input or output). Table I lists all the types of object which can be passed, the syntactic form of specification, and the corresponding form of the actual parameter which must be supplied in the procedure call. The equivalent syntax rules are:

```
Parameterspec = Specifier Idlist
                Tablespec
                Procedurespec

Specifier     = VALUE Numbertype
                LOCATION Numbertype
                Numbertype ARRAY
                LABEL
                SWITCH
```

TABLE I

Parameters of procedures

Object	Formal specification	Actual parameter
numerical value	VALUE Numbertype Id*	Expression
location of data word	LOCATION Numbertype Id*	Wordreference
name of array	Numbertype ARRAY Id*	Id
name of table	Tablespec †	Id
place in program	LABEL Id*	Destination
name of switch	SWITCH Id*	Id
name of procedure	Procedurespec‡	Id

*Composite specification of similar parameters has Idlist in place of Id.
†See section 8.3.2.3.
‡See section 8.3.4.

8.3.1 Value parameters

The formal parameter is treated as though declared in the procedure body; upon entry to the procedure, the *actual* expression is evaluated to the type specified (including scaling if the numeric type is FIXED), and the value is forthwith *assigned* to the formal parameter. The formal parameter may subsequently be used for working space in the body; if the actual parameter is a variable, its value will be unaffected by assignments to the formal parameter.

8.3.2 Data reference parameters

Location, array and table parameters are all examples of data references. Upon entry to the procedure, these formals are made to refer to the same computer locations as those to which the actual parameters already refer. Operations upon such formal parameters within the procedure body are therefore operations on the actual parameters. For example, the values of the actual parameters may be altered by assignments within the procedure.

8.3.2.1 Word location parameters

The actual parameter must be a word reference, i.e. a simple data reference, an array element, an indexed table identifier, a whole-word table-element or an anonymous reference. Index expressions are evaluated upon entry to the procedure as part of the process of obtaining the location of the actual parameter. *The numeric type of the actual parameter must agree exactly with the formal specification.* Part-word references, such as table-elements are not allowed as word location parameters. An example of a procedure heading and a possible call of the same procedure is

heading f(VALUE INTEGER n; LOCATION INTEGER m)

call f(LOCATION (u [i]), [j])

8.3.2.2 Array parameters

As in an array declaration, the specified numeric type applies to all the elements of the array named. The numeric type of the *actual* array name must agree with this formal specification. By indexing within the body, the procedure can refer to any element of the actual array.

8.3.2.3 Table parameters

The specification of a table parameter is identical in form to a table declaration except that presetting is not allowed. The syntax rule is

Tablespec = TABLE Id [Width , Length] [Elementdeclist]

The element declaration list need include only such fields as are used in the procedure body.

8.3.3 Place parameters

8.3.3.1 Label parameters

The actual parameter must be a 'destination', i.e. a label *or a switch element.* In the latter case, the index is evaluated once upon entry to the procedure. The actual parameter must be in scope at the call, even if it is out of scope where the formal parameter is used in the procedure body.

8.3.3.2 Switch parameters

The actual parameter is a switch identifier. By indexing within the procedure body, the procedure can refer to any of the individual labels which form the elements of the switch.

8.3.4 Procedure parameters

Within the body of a procedure, it may be necessary to execute an unknown procedure, i.e. a procedure whose name is to be supplied as an actual parameter. The features of the unknown procedure must be formally specified in the heading of the procedure within which it is called. As an example, suppose that a procedure g has been declared as

FIXED (24,2) PROCEDURE g(VALUE INTEGER i, j; INTEGER ARRAY a); Statement

and further suppose that a procedure q has a formal parameter f for which it may be required to substitute g. A declaration of q, illustrating the necessary specification (italicised for clarity) might be

PROCEDURE q(LABEL b; *FIXED (24, 2) PROCEDURE f(VALUE INTEGER, VALUE INTEGER, INTEGER ARRAY)*); Statement

A typical call of q would be q(lab,g). At the inner level of parameter specification, no formal identifiers are required, no composite specifications are allowed (as for i and j in g) and the specifications are separated by commas. To pursue the example to a deeper level of nesting, suppose that a procedure c66 has a parameter p for which it may be required to substitute q. A declaration of c66 might then be

PROCEDURE c66(PROCEDURE p(LABEL, *FIXED (24,2) PROCEDURE*); SWITCH s); Statement

A typical call of c66 would be c66(q,sw). At the level of specification shown in italics in the latter example, no further parameter specifications are required. The syntax rules for a procedure specification are

```
Procedurespec  = Answerspec PROCEDURE Procparamlist

Procparamlist  = Procparameter
                 Procparameter , Procparamlist

Procparameter  = Id
                 Id ( Typelist )
```

```
Typelist       = Type
                 Type , Typelist

Type           = Specifier
                 TABLE
                 Answerspec  PROCEDURE
```

8.3.5 Non-standard parameter specification

The need to specify numeric type for formal value and location parameters places an undesirable constraint on the designer of input and output procedures. For such procedures it is desirable that the procedure should adapt itself to the numeric type and scale of the actual parameters. The following extension of the syntax for Parameterspec (section 8.3) is regarded as an acceptable device in Coral 66 implementations:

```
Parameterspec  = VALUE Formalpairlist
                 LOCATION Formalpairlist
                 Specifier Idlist
                 etc

Formalpairlist = Formalpair
                 Formalpair , Formalpairlist

Formalpair     = Id : Id
```

At the call of the procedure, each formal pair corresponds to a single actual parameter. The first identifier is used within the procedure body, with numeric type integer, as a reference to the value of, or as the location of, the actual parameter. The compiler arranges that the second identifier passes the numeric type and scale of the actual parameter, represented in the form of an integer by some implementation-dependent convention. For example, the declaration of an output procedure might begin

PROCEDURE out(VALUE u:v)

If x is a variable of numeric type FIXED (24,12), the procedure statement out(x) would take account of this known scale.

8.4 The procedure body

For purposes of scoping, a procedure declaration may be regarded as a block at the place where it appears on the program sheet (even though this might be an illegal position). Everything except the body can be disregarded, and the formal parameters treated as though declared within the body, labels included. Identifiers which are non-local to the procedure body are those in scope at the place of the procedure declaration, subject to the restrictions given in section 3.5. Actual parameters must, of course, be in scope at the procedure call. For example, the block:

```
BEGIN INTEGER i;
      INTEGER PROCEDURE p; ANSWER i;
      i ← 0;
      BEGIN INTEGER i;
            i ← 2;
            print(p)
      END
END
```

has the effect of printing 0.

9

Communicators

The segments of a program may communicate with each other through COMMON (section 9.1 below), and with objects external to the program by means of communicators such as LIBRARY, EXTERNAL or ABSOLUTE, as defined in particular implementations.

9.1 COMMON communicators

Global objects declared within a program (section 3.3) are communicated to all segments through a COMMON communicator. This consists of a list of COMMON items separated by semi-colons all within round brackets following the word COMMON. Such items are of three kinds, corresponding to the division of objects into data, places and procedures. A COMMON data item is a declaration of the identifiers listed within it, exactly as in section 4, storage being allocated as in section 4.5, presets and overlays as in sections 4.6 and 4.8. Communication of places and procedures takes the form of *specification*, as in the equivalent parameters of a procedure declaration (sections 8.3.3 and 8.3.4). For each identifier specified in a COMMON communicator, there must correspond an appropriate declaration (or for labels a setting) in one and only one outermost block of the program. The syntax is

```
Commoncommunicator = COMMON ( Commonitemlist )

Commonitemlist     = Commonitem
                     Commonitem ; Commonitemlist

Commonitem         = Datadec
                     Overlaydec
                     Placespec
                     Procedurespec
                     Void

Placespec          = LABEL  Idlist
                     SWITCH  Idlist
```

9.2 LIBRARY communicators

To make provision for the use of library procedures (and possibly also data references used by such procedures), programs may include LIBRARY communicators. These should begin with the word LIBRARY and be styled to conform with the rest of the language. The relative importance attached to COMMON and LIBRARY as means of inter-segment communication borders on questions of implementation which fall outside the scope of the present language definition.

9.3 EXTERNAL communicators

It may be desirable to refer to an object external to a Coral 66 program by means of an identifier. Provided the loader permits, this may be achieved by an EXTERNAL communicator similar in form to a COMMON communicator.

9.4 ABSOLUTE communicators

Coral 66 programs may refer to objects having absolute addresses in the computer by the use of ABSOLUTE communicators which associate an identifier with a specification of the 'absolute' object, including its address. The form recommended is that of a COMMON communicator, except that each identifier to be associated with an absolute location takes the syntactic form Id / Integer.

10

Names and constants

10.1 Identifiers

Identifiers are used for naming objects of data, labels and switches, procedures, macros and their formal parameters. An identifier consists of an arbitrary sequence of lower case letters and digits, starting with a letter. It carries no information in its form, e.g. single-letter identifiers are not reserved for special purposes. It may be of any length, though it is permissible for compilers to disregard all but the first twelve printing characters. As layout characters are ignored, spaces may be used in identifiers without acting as terminators.

```
Id                 = Letter  Letterdigitstring

Letterdigitstring  = Letter  Letterdigitstring
                     Digit  Letterdigitstring
                     Void

Letter = a b c d e f g h i j k l m n o p q r s t u v w x y z

Digit  = 0 1 2 3 4 5 6 7 8 9
```

An obvious liberty is taken with the layout of alternatives in the above rules.

10.2 Numbers

Numerical constants appearing in other sections of this definition are of the following types:

(a) '*Constants* for presetting, optionally signed.

(b) *Integers* and *reals* as primaries in expressions. A sign attached to a primary belongs syntactically to the expression and not to the number.

(c) *Integers* and *signed integers* used in declarations or specifications, typically for defining fixed scales, bit-fields and array bounds.

The syntactic classification is as follows:

Constant	=	Number
		Addoperator Number
Number	=	Real
		Integer
Signedinteger	=	Integer
		Addoperator Integer
Real	=	Digitlist . Digitlist
		Digitlist $_{10}$ Signedinteger
		$_{10}$ Signedinteger
		Digitlist . Digitlist $_{10}$ Signedinteger
		OCTAL (Octallist . Octallist)
Integer	=	Digitlist
		OCTAL (Octallist)
		LITERAL (*printing character*)

The further expansions are

Digitlist	=	Digit
		Digit Digitlist
Octallist	=	Octaldigit
		Octaldigit Octallist
Octaldigit	=	0 1 2 3 4 5 6 7

where 0 to 7 are alternatives.

10.3 Literal constants

A printing character is assumed to have a unique integer representation within the computer, dependent on some hardware or software convention. The integer value may be referred to within the program by the LITERAL operator. For example,

LITERAL(a)

has an integer value uniquely representative of ‘a’. The form is included within the syntax of integer (section 10.2). The printing characters will be implementation-dependent, but it must be assumed that the set includes one 26-letter alphabet and a set of 10 digits (see Appendix 2). Layout characters are not acceptable as arguments of LITERAL.

10.4 Strings

A string is any succession of characters (printing or layout) enclosed in quotation marks (string quotes). Assuming that the hardware representations of the opening and closing quote symbols are distinguishable, occurrence of such marks must be properly paired within the string (but see Appendix 2). A string is classed as an unconditional expression (section 6), and its value is its location, but it may not be used as a LOCATION parameter. Procedures capable of selecting individual characters from a string should be designed so that characters are represented by the same integer values as are defined for literal constants.

String = $\nless$ *sequence of characters with quotes matched* $\ngtr$

11

Text processing

11.1 Comment

A program may be annotated by the insertion of textual matter which is ignored by the compiler.

11.1.1 Comment sentences

A comment sentence may be written wherever a declaration or statement can appear. It consists of the word COMMENT followed by text and terminated by a semi-colon. For obvious reasons, the text must not contain a semi-colon. The entire comment sentence is ignored by the compiler.

11.1.2 Bracketed comment

Bracketed comment is any textual matter enclosed within round brackets immediately after a semi-colon of the program. The text may contain brackets provided that they are matched. Bracketed comment (including the brackets) is ignored by the compiler.

11.1.3 END comment

Annotation may be inserted after the word END provided that it takes the form of an identifier only. The 'identifier' is ignored by the compiler.

11.2 Macro facility

A Coral 66 compiler embodies a macro processor, which may be regarded as a self-contained routine which processes the text of the Coral program before passing it on to the compiler proper. Its function is to enable the

programmer to define and use convenient macro names, in the form of identifiers, to stand in place of cumbersome or obscure portions of text, typically code statements. Once a macro name has been defined, the processor expands it in accordance with the definition wherever it is subsequently used, until the definition is altered or cancelled (11.2.4). However, the macro processor treat comments and constant character strings (section 10.4) as indivisible entities, and does not expand any identifiers within these entities. No character which could form part of an identifier may be written adjacent to the use of a macro name or formal parameter, as this would inhibit the recognition of such names. A macro definition may be written into the source program wherever a declaration or a statement could legally appear, and is removed from it by the action of the macro processor.

11.2.1 String replacement

In the simplest use, a macro name stands for a definite string of characters, the macro body. For example, the (fictitious) code statement

```
CODE BEGIN 123,45,6 END
```

might be given the name 'shift6'. The macro definition would be written

```
DEFINE shift6 ≮ CODE BEGIN 123,45,6 END ≯ ;
```

The expansion, or body, can be any sequence of characters in which string quotes are matched (but see Appendix 2). Care must be taken to include brackets, such as BEGIN and END, as part of the macro body whenever there is the possibility that the context of the expansion may demand them.

11.2.2 Parameters of macros

A macro may have parameters, as in the following example,

```
DEFINE shift(n) ≮ CODE BEGIN 123,45,n END ≯ ;
```

Subsequent occurrences of shift(6) would be expanded to the code statement in 11.2.1. A formal parameter, such as n above, must be written as an identifier. An actual parameter (e.g. 6) is any string of characters in which string quotes are matched, all round and square brackets are nested and matched, and all occurrences of a comma lie between round or square brackets. This rule enables commas to be used for separating actual parameters. The number of actual parameters must be the same as the number of formals, which are also separated by commas.

11.2.3 Nesting of macros

A macro definition may embody definitions or uses of other macros to any depth. When a macro is defined, the body is kept but not expanded. When the macro is used, it is as though the body were substituted into the program text, and it is during this substitution that any other macros encountered are processed. The use of a macro with parameters may be regarded as introducing virtual macro definitions for the formal parameters before the macro body is substituted. Thus, to continue the example from 11.2.2, the occurrence of shift(6) is equivalent to

```
DEFINE n ≮ 6 ≯ ;
CODE BEGIN 123,45,n END
```

followed immediately by deletion of the virtual macro n. Throughout the scope of the macro 'shift', the formal parameter n may not be defined as a macro name. A formal parameter may not be used in any inner nested macro definition; neither in its body nor as a macro name nor as a formal parameter. Furthermore, no identifier in an actual parameter string, or its subsequent expansions, may be the same as any formal parameter of the calling macro.

11.2.4 Deletion and redefinition of macros

Macro definitions are valid from the point of definition until either the end of the program text is reached or the macro name is redefined or deleted. The scope of a macro is independent of the block structure of the program. To delete a macro, the command

```
DELETE Macroname ;
```

is used wherever a declaration or statement could appear. Alternatively, a macro name can be redefined. Macro definitions which have the same name are stacked, so that the most recent is the one which applies when the name is used. If a redefined macro is deleted, it is the most recent definition which is deleted, and the previous one is reinstated. 'Recent' and 'previous' refer to the sequence as processed by the macro processor.

11.3 Syntax of comment and macros

Commentsentence	= COMMENT *any sequence of characters not including a semi-colon* ;

Bracketedcomment	= (*any sequence of characters in which round brackets are matched*)
Endcomment	= Id
Macrodefinition	= DEFINE Macroname ≮ Macrobody ≯ ;
	DEFINE Macroname (Idlist) ≮ Macrobody ≯ ;
Macroname	= Id
Macrobody	= *any sequence of characters in which string quotes are matched*
Macrodeletion	= DELETE Macroname ;
Macrocall	= Macroname
	Macroname (Macrostringlist)
Macrostringlist	= Macrostring , Macrostringlist
	Macrostring
Macrostring	= *any sequence of characters in which commas are protected by round or square brackets and in which such brackets are properly matched and nested*

Appendix - 1 **Syntax Rules In Alphabetical Order**

```
Actual = Expression                                                  7.3
         Wordreference
         Destination
         Name
Actuallist = Actual                                                  7.3
             Actual , Actuallist
Addoperator = +                                                      6.1
              −
Alternative = Statement                                              7.9
Answerspec = Numbertype                                              8.1
             Void
Answerstatement = ANSWER Expression                                  7.4
Arraydec = Numbertype ARRAY Arraylist Presetlist                     4.3
Arrayitem = Idlist [ Sizelist ]                                      4.3
Arraylist = Arrayitem                                                4.3
            Arrayitem , Arraylist
Assignmentstatement = Variable ← Expression                          7.1


Base = Id                                                            4.8
       Id [ Signedinteger ]
Bitposition = Integer                                                4.4.2
Block = BEGIN Declist ; Statementlist END                            3.1
Booleanword = Booleanword2                                           6.1.2
              Booleanword4 DIFFER Booleanword5
Booleanword2 = Booleanword3                                          6.1.2
               Booleanword5 UNION Booleanword6
Booleanword3 = Booleanword6 MASK Typedprimary                        6.1.2
Booleanword4 = Booleanword                                           6.1.2
               Typedprimary
Booleanword5 = Booleanword2                                          6.1.2
               Typedprimary
Booleanword6 = Booleanword3                                          6.1.2
               Typedprimary
Bracketedcomment = ( any sequence of characters in which round       11.3
                         brackets are matched )


Codesequence = defined in a particular implementation                7.5
Codestatement = CODE BEGIN Codesequence END                          7.5
Commentsentence = COMMENT any sequence of characters not             11.3
                          including a semi-colon ;
Commoncommunicator = COMMON ( Commonitemlist )                       9.1
```

```
Commonitem = Datadec                                                 9.1
             Overlaydec
             Placespec
             Procedurespec
             Void
Commonitemlist = Commonitem                                          9.1
                 Commonitem ; Commonitemlist
Comparator = < or ⩽ or = or ⩾ or > or ≠                              6.2.1
Comparison = Simpleexpression Comparator Simpleexpression            6.2.1
Compoundstatement = BEGIN Statementlist END                          7.6
Condition = Condition OR Subcondition                                6.2.1
            Subcondition
Conditionalexpression = IF Condition                                 6.2
                        THEN Expression
                        ELSE Expression
Conditionalstatement = IF Condition THEN Consequence                 7.9
                       IF Condition THEN Consequence
                             ELSE Alternative
Consequence = Simplestatement                                        7.9
              Label : Consequence
Constant = Number                                                    10.2
           Addoperator Number
Constantlist = Group                                                 4.6.1
               Group , Constantlist


Datadec = Numberdec                                                  3.1
          Arraydec
          Tabledec
Dec = Datadec                                                        3.1
      Overlaydec
      Switchdec
      Proceduredec
Declist = Dec                                                        3.1
          Dec ; Declist
Destination = Label
              Switch [ Index ]                                       7.2
Digit = 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9               10.1
Digitlist = Digit                                                    10.2
            Digit Digitlist
Dimension = Lowerbound : Upperbound                                  4.3
Dummystatement = Void                                                7.8


Elementdec = Id Numbertype Wordposition                              4.4.2
             Id Partwordtype Wordposition , Bitposition
Elementdeclist = Elementdec                                          4.4.1
                 Elementdec ; Elementdeclist
```

Elementpresetlist	= PRESET Constantlist	4.6.2
	Void	
Elementscale	= (Totalbits , Fractionbits)	4.4.2.2
	(Totalbits)	
Endcomment	= Id	11.3
Expression	= Unconditionalexpression	6
	Conditionalexpression	
Factor	= Primary	6.1.1
	Booleanword	
Forelement	= Expression	7.10
	Expression WHILE Condition	
	Expression STEP Expression UNTIL Expression	
Forlist	= Forelement	7.10
	Forelement , Forlist	
Forstatement	= FOR Wordreference ←Forlist DO Statement	7.10
Fractionbits	= Signedinteger	4.1
Gotostatement	= GOTO Destination	7.2
Group	= Constant	4.6.1
	(Constantlist)	
	Void	
Id	= Letter Letterdigitstring	10.1
Idlist	= Id	4.2
	Id , Idlist	
Index	= Expression	6.1.1.2.1
Integer	= Digitlist	10.2
	OCTAL (Octallist)	
	LITERAL (*printing character*)	
Label	= Id	5
Labellist	= Label	5
	Label , Labellist	
Length	= Integer	4.4.1
Letter	= a *or* b *or* c *or* . . . . . *or* z	10.1
Letterdigitstring	= Letter Letterdigitstring	10.1
	Digit Letterdigitstring	
	Void	
Lowerbound	= Signedinteger	4.3

Macrobody	= *any sequence o) cnaracters n which string quotes are matched*	11.3, Appx. 2
Macrocall	= Macroname	11.3
	Macroname (Macrostringlist)	
Macrodefinition	= DEFINE Macroname ≮ Macrobody ≯ ;	11.3
	DEFINE Macroname (Idlist) ≮ Macrobody ≯ ;	
Macrodeletion	= DELETE Macroname ;	11.3
Macroname	= Id	11.3
Macrostring	= *any sequence of characters in which commas are protected by round or square brackets and in which such brackets are properly matched and nested*	11.3
Macrostringlist	= Macrostring	11.3
	Macrostring , Macrostringlist	
Multoperator	= *	6.1
	/	
Name	= Id	7.3
Number	= Real	10.2
	Integer	
Numberdec	= Numbertype Idlist Presetlist	4.2
Numbertype	= FLOATING	4.1
	FIXED Scale	
	INTEGER	
Octaldigit	= 0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7	10.2
Octallist	= Octaldigit	10.2
	Octaldigit Octallist	
Overlaydec	= OVERLAY Base WITH Datadec	4.8
Parameterspec	= Specifier Idlist	8.3
	Tablespec	
	Procedurespec	
Parameterspeclist	= Parameterspec	8.2
	Parameterspec ; Parameterspeclist	
Partword	= Id [Index]	6.1.1.2.2
	BITS [Totalbits , Bitposition] Typedprimary	
Partwordreference	= Id [Index]	7.1
	BITS [Totalbits , Bitposition] Wordreference	
Partwordtype	= Elementscale	4.4.2.2
	UNSIGNED Elementscale	
Placespec	= LABEL Idlist	9.1
	SWITCH Idlist	
Presetlist	= ← Constantlist	4.6.1
	Void	

Primary =	Untypedprimary	6.1.1
	Typedprimary	
Procedurecall =	Id	7.3
	Id (Actuallist)	
Proceduredec =	Answerspec PROCEDURE Procedureheading ; Statement	8
	Answerspec RECURSIVE Procedureheading ; Statement	
Procedureheading =	Id	8.2
	Id (Parameterspeclist)	
Procedurespec =	Answerspec PROCEDURE Procparamlist	8.3.4
Procparameter =	Id	8.3.4
	Id (Typelist)	
Procparamlist =	Procparameter	8.3.4
	Procparameter , Procparamlist	
Real =	Digitlist . Digitlist	10.2
	Digitlist $_{10}$ Signedinteger	
	$_{10}$ Signedinteger	
	Digitlist . Digitlist $_{10}$ Signedinteger	
	OCTAL (Octallist . Octallist)	
Scale =	(Totalbits , Fractionbits)	4.1
Signedinteger =	Integer	10.2
	Addoperator Integer	
Simpleexpression =	Term	6.1
	Addoperator Term	
	Simpleexpression Addoperator Term	
Simplestatement =	Assignmentstatement	7
	Gotostatement	
	Procedurecall	
	Answerstatement	
	Codestatement	
	Compoundstatement	
	Block	
	Dummystatement	
Sizelist =	Dimension	4.3
	Dimension , Dimension	
Specifier =	VALUE Numbertype	8.3
	LOCATION Numbertype	
	Numbertype ARRAY	
	LABEL	
	SWITCH	
Statement =	Label : Statement	7
	Simplestatement	
	Conditionalstatement	
	Forstatement	

Statementlist =	Statement	7.6
	Statement ; Statementlist	
String =	≮ *sequence of characters with quotes matched* ≯	10.4, Appx. 2
Subcondition =	Subcondition AND Comparison	6.2.1
	Comparison	
Switch =	Id	5
Switchdec =	SWITCH Switch ← Labellist	5
Tabledec =	TABLE Id [Width , Length]	4.4.1
	[Elementdeclist Elementpresetlist]	
	Presetlist	
Tablespec =	TABLE Id [Width , Length] [Elementdeclist]	8.3.2.3
Term =	Factor	6.1
	Term Multoperator Factor	
Totalbits =	Integer	4.1
Type =	Specifier	8.3.4
	TABLE	
	Answerspec PROCEDURE	
Typedprimary =	Wordreference	6.1.1.2
	Partword	
	LOCATION (Wordreference)	
	Numbertype (Expression)	
	Procedurecall	
	Integer	
Typelist =	Type	8.3.4
	Type , Typelist	
Unconditionalexpression =	Simpleexpression	6
	String	
Untypedprimary =	Real	6.1.1.1
	(Expression)	
Upperbound =	Signedinteger	4.3
Variable =	Wordreference	7.1
	Partwordreference	
Width =	Integer	4.4.1
Wordposition =	Signedinteger	4.4.2
Wordreference =	Id	6.1.1.2.1
	Id [Index]	
	Id [Index , Index]	
	[Index]	

Appendix 2 - List of language symbols

ABSOLUTE	9.4	EXTERNAL	9.3	OVERLAY	4.8
AND	6.2.1				
ANSWER	7.4	FIXED	4.1	PRESET	4.6.2
ARRAY	4.3	FLOATING	4.1	PROCEDURE	8
		FOR	7.10	RECURSIVE	8
BEGIN	3.1, 7.6				
BIT	4.4.2	GOTO	7.2	STEP	7.10.1
BITS	6.1.1.2.2			SWITCH	5, 8.3, 9.1
		IF	6.2, 7.9		
CODE	7.5	INTEGER	4.1	TABLE	4.4.1
COMMENT	11.1.1			THEN	6.2, 7.9
COMMON	9.1	LABEL	8.3, 9.1		
		LIBRARY	9.2	UNION	6.1.2
DEFINE	11.2.1	LITERAL	10.3	UNSIGNED	4.4.2.2
DELETE	11.2.4	LOCATION	6.1.1.2, 8.3	UNTIL	7.10.1
DIFFER	6.1.2				
DO	7.10	MASK	6.1.2	VALUE	8.3
ELSE	6.2, 7.9	OCTAL	10.2	WHILE	7.10.2
END	3.1, 7.6	OR	6.2.1	WITH	4.8

0 1 2 3 4 5 6 7 8 9	digits, 10.1
abcdefghijklmnopqrstuvwxyz	letters, 10.1
+ −	adding operators, 6.1
* /	multiplying operators, 6.1
< ⩽ = ⩾ > ≠	comparators, 6.2.1
( )	expression brackets, etc.
[ ]	index brackets, etc.
≮ ≯	string quotes, 10.4, 11.2
, ;	separators for lists
:	separator for bounds, 4.3 terminator for label setting, 7
←	assignment symbol, 4.6.1, 5, 7.1
.	point, 10.2
$_{10}$	'times ten to the power of', 10.2

The above symbols are too numerous for representation by single characters on most printing peripheral equipment. It is recommended that, where necessary, the two alphabets be distinguished by enclosure of language words between primes or accents, and that the following representations be adopted:

Official definition	*Representation*
⩽	<=
⩾	>=
≠	<>
←	:=
≮	"
≯	"

The use of the ISO quote-character (") is a desirable representation, but as this makes the opening and closing symbols indistinguishable, it is recommended that the Algol 68 system be adopted (Numerische Mathematik, 14, 79–218 (1969), para. 5.1.4), in which a quote-symbol within a string is represented by a pair of quote-characters ("").

Appendix 3 - **Levels of implementation**

The language requirements for a particular machine or for particular classes of work, or generally for both, are not easily assessed. The richer the language, the larger the compiler may become, and the more difficult it may be to compile into efficient object-code. The balance between code efficiency and the human effort needed to attain it is not easy to strike. The objective of Coral 66 development has been to permit latitude, not in details, where there is little merit in diversity of expression, but in the presence or absence of major features such as RECURSIVE procedures, which may or may not be considered worth having. Other such major features are:

- TABLE facility
- FIXED numbers
- BITS, DIFFER, UNION and MASK
- OVERLAY of data
- FLOATING numbers

A full Coral 66 compiler handles all these features, but it would not normally be expected that a compiler for an object machine lacking floating point hardware should handle the FLOATING type of number. The use of additional features, not officially within the Coral 66 language, and not clashing with the official definition or with each other, may be approved for specific fields of defence work.

Index of terms

For syntax words, see Appendix 1; for language words, see Appendix 2.

Printed in England by Her Majesty's Stationery Office Press, Manor Farm
01016 Dd698730 K10 2/81