Coral 66 Language Reference Manual

For mission-critical applications

www.xgc.com

Coral 66 Language Reference Manual

For mission-critical applications

Order Number: XGC-C66-1.7-RM-001120

XGC Technology

London
UK
<www.xgc.com>

Coral 66 Language Reference Manual: For mission-critical applications

by Ministry of Defence

Publication date November 2000

Acknowledgments

The information in this reference manual is based on the text of the HMSO publication *Official Definition of Coral 66*, first published in 1970 and reprinted with amendments in 1974. ISBN 0 11 470221 7.

This document was prepared using DocBook XML from docbook.org [http://www.docbook.org/].

Contents

Tables

Preface

This document describes the programming language supported by the XGC family of Coral 66 compilers. The text of this document is based on the 1974 edition of *Official Definition of Coral 66*, published by HMSO, London. The text includes details of the options and extensions that XGC Coral 66 offers along with additional examples.

1. Related Documents

Getting Started with Coral 66, which offers examples and advice for the new user.

The Coral 66 User's Guide, in three volumes, which describes the commands and options used to run the toolset.

The XGC Libraries, which documents the library functions available with the XGC compilers.

2. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC user manuals. You can send your comments in the following ways:

- Internet electronic mail: readers.comments@xgc.com

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

3. Preface to the 1974 Edition

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 whom all 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 the National Computing Centre in 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 P. M. Woodward

Chapter 1 Introduction

It is virtually impossible to design a standard language such that programs will run with equal 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 for use in small fixed-point machines; the floating point facilities of the language can be omitted where 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 C, *Levels of Implementation* [99] 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 the 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 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 language 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 a 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 the 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. The 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 equal 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 all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analyzers with no backtracking 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.

Chapter 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 may be regarded as a sequence of symbols, each visibly representable by a unique character or combination of characters. The symbols of the language are defined (see Appendix B, *List of Language Symbols* [97]), 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 as 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 name 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 (see Section 3.3, “Globals” [9] and see Section 9.1, “COMMON Communicators” [61]). 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* (see Chapter 3, *Scoping* [7]). 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 the 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.

Chapter 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, “Preservation of Values” [24]). 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 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 (see Section 3.3, “Globals” [9]), 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 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 format block surrounding the segments. In addition to objects that 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, “COMMON Communicators” [61], and are treated as local in *every* outermost block of the program. Global objects *declared* outside the segments are treated as non-local. With these rules of locality, questions of clashing are resolved in accordance with Section 3.2, “Clashing of Names” [8].

3.4. Labels

Any statement may be labeled by writing in front of it an identifier and a colon. The scope of the label is the smallest block embracing the statement which is labeled, 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, “COMMON Communicators” [61]). 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 the 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 is directly or indirectly calls itself, that procedure is said to be recursive and must be explicitly declared as such.

Chapter 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 (see Section 4.4.2.2, “Part-Word Table Elements” [18]), all three types are signed. Numeric type is indicated by the word FLOATING or INTEGER or by the word FIXED followed by the 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 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, “Presetting” [21]. The syntax for a number declaration is

```
    Numberdec ::=
Numbertype
    Idlist
    Presetlist
```

```
    Idlist ::=
Id
    Id , Idlist
```

4.3. Array References

An array reference 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[0:19];
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 array references 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, “Evaluation of Expressions” [37]. The syntax rules for array declaration, which include a presetting facility (Section 4.6.1, “Presetting of Simple References and Arrays” [21]), 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 for referring to packed data, one in which as unnamed field is selected from any computer word which holds data (see Section 6.1.1.2.2, “Part-Words” [33]), 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 a table 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 identifier, 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, “Table-Element Declaration” [16] below. Data-packing is implementation dependent, and the examples will be found to assume a word length 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, “Presetting of Tables” [23].

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 whole or part of a computer word, and the form of a declaration differs accordingly. The syntax for an element declaration, more fully developed in Section 4.4.2.2, “Part-Word Table Elements” [18], is

```
    Elementdec ::=
Id
    Numbertype
    Wordposition
    Id
    Partwordtype
    Wordposition , Bitposition
```

```
    Wordposition ::= Signedinteger
```

```
    Bitposition ::=  Integer
```

Word-position and bit-position are numbered from zero upward, and the least significant digits 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 text. The meaning of Bitposition is given in see Section 4.4.2.2, “Part-Word Table Elements” [18].

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, see Section 4.5, “Storage

Allocation" [20].) Each element refers to a (signed) integer occupying a 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-position 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 -2.0 to 1.75 whilst UNSIGNED(4,2) allows them from 0.0 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, “Table-Element Declaration” [16], is

```
    Partwordtype ::=
Elementscale
    UNSIGNED
    Elementscale
```

```
    Elementscale ::=
( Totalbits , Fractionbits )
( Totalbits )
```

The rules for Totalbits and Fractionbits are in Section 4.1, “Numeric Types” [11].The number of fraction bits may be negative, zero, or positive, and it is for the binary point to lie outside the declared field.

4.4.3. Example of a 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, “Example of a Table Declaration” [19], the rain for april 6th would be written rain[5], for it should be noted than 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 equivalent 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 (see Section 4.8, “Overlay Declarations” [24]).

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 explicitly 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

1. the body of a recursive procedure, or
2. an inner block of the program, or
3. an inner block of a procedure body.

Procedure bodies do not count as blocks for the purposes 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, “Storage Allocation” [20]. 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, “Preservation of Values” [24]), 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, “Presetting of Tables” [23]. For the expansion of Constant, see Section 10.2, “Numbers” [66].

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 (see Section 4.4.4, “Reference to Tables and Table Elements” [19]), and preset as such (see Section 4.6.1, “Presetting of Simple References and Arrays” [21]). Alternatively the table elements may be preset after their declaration list, as shown at Elementpresetlist in the syntax of Section 4.4.1, “Table Declaration” [15]. 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 Section 4.6.1, “Presetting of Simple References and Arrays” [21]. Each entry of the table is preset in succession as a group of elements, taken in order of their declaration. Voids in the list imply an absence of assignment. This may be necessary to avoid duplication when fields overlap, as do "ratio" and "arc" in the above example. As in Section 4.6.1, “Presetting of Simple References and Arrays” [21], brackets used for grouping constants in the list of presets are ignored by the compiler. 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, “Presetting” [21], 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 then 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, on-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 Section 4.5, “Storage Allocation” [20]) 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  ]
```

Chapter 5

Place References–Switches

Place references refer to positions of program statements, and the simplest position market if the *label* (see Section 3.4, “Labels” [9]). 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
```

Chapter 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, “Conditions” [38]), 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, “Strings” [68].

6.1. Simple Expressions

Arithmetic is performed with the monadic and dyadic adding operators + and -, and with the dyadic 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 analyzed until no further reduction is possible an no expression brackets remain. When an expression contains no word-logical operators (see Section 6.1.1.2, “Typed Primaries” [32]), 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 may different ways within a computer word. The same applies to an expression, whose type is determined by context (see Section 6.1.3, “Evaluation of Expressions” [37]).

```
        Untypedprimary ::=
Real
```

```
( Expression )
```

A "real" (see Section 10.2, “Numbers” [66]) 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

```
    Typedprimary ::=
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 to 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 type declaration which may have been associated with that word with some other type. See also Section 6.1.1.2.3, “Locations” [34]. The syntax for a word reference is

```
        Wordreference ::=
Id
        Id [ Index ]
Id [ 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

1. a reference to a part-word table-element, or
2. 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[1] 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 all cases, however, the numeric type resulting from the application if BITS is INTEGER. The syntax for a part-word, which should be

[1]It is assumed that Totalbits will not be set equal to the full word length

distinguished from that of a *part-word reference* (see Section 7.1, “Assignments” [42]), 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 the value of type INTEGER. It may be notes 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 Numbertype as described in Section 6.1.3, “Evaluation of Expressions” [37].

6.1.1.2.5. Functions

The call of a typed procedure (see Chapter 8, *Procedures* [51]) 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, “Procedure Statements” [44].

6.1.1.2.6. Integers

An integer used in any expression (see Section 10.2, “Numbers” [66]) can be assumed to have the numeric type INTEGER before any necessary type-changes are enforced by context.

6.1.2. Word-Logic

Three dyadic 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 (see Section 6.2.1, “Conditions” [38]), a different terminology is used. The operators are

```
DIFFER        UNION         MASK
  0 1           0 1          0 1
 -----         -----        -----
0|0 1         0|0 1        0|0 0
1|1 0         1|1 1        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 Section 6.1.1, “Primaries” [31], 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 (see Section 6.1.1.2, “Typed Primaries” [32]), 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 the subtraction, whilst

```
x := INTEGER(i - j)
```

causes subtraction of integers before conversion to floating point. Although the order of evaluation of 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.

Chapter 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 were 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 the 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, “Preservation of Values” [24]). 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, “Word References” [32] 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 multiple assignment statement.

7.2. Goto Statements

The goto statement causes the next statement for execution to be the one having the 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 in the switch declaration. See also Section 3.4, “Labels” [9] and Chapter 5, *Place References–Switches* [27]. 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

1. 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, “Parameter Specification” [53]).
2. 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, “Parameter Specification” [53].

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 in 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 the 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 statement 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 go to a label which is set inside a compound statement. The syntax is

```
   Compoundstatement ::=
BEGIN
   Statementlist
   END
```

```
   Statementlist ::=
Statement
   Statement ; Statementlist
```

7.7. Blocks

See Chapter 3, *Scoping* [7].

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, 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
e1
UNTIL
e3
```

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

1. v1 is assigned to the control variable,
2. v1 is compared with v3. If (v1 - v3) * v2 > 0, then the for-element is exhausted, otherwise
3. the controlled statement is executed,
4. the value of v1 is set from the controlled variable, then incremented by v2 and the cycle is repeated from (a).

7.10.2. For-elements with WHILE

Let the element be denoted by

```
e1
WHILE
Condition
```

Then the sequence of operation is

1. e1 is evaluated and assigned to the control variable,
2. the condition is tested. If false, the for-statement is exhausted, otherwise
3. the controlled statement is executed and the cycle repeated from (i).

Unlike those in Section 7.10.1, “For-elements with STEP” [49], the expression e1 and those occurring in the condition are evaluated repeatedly.

Chapter 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 a declaration used for recursive procedures (see Section 3.5, “Restrictions Connected with Scoping” [9]). The statement following the procedure heading is the procedure body, which contains an answer statement (see Section 7.4, “Answer Statements” [45]) unless the answer specification is void (see Section 8.1, “Answer Specification” [52]), and is treated as a block whether or not it includes any local declarations (see Section 8.4, “The Procedure Body” [59]).

8.1. Answer Specification

The value of a typed procedure is given by an answer statement (see Section 7.4, “Answer Statements” [45]) 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.

```
    Answerspec ::=
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 8.1, “Parameters of Procedures” [54] 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 call. The equivalent syntax rules are:

```
   Parameterspec ::=
Specifier
   Idlist
   Tablespec
   Procedurespec
```

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

SWITCH

Table 8.1. Parameters of Procedures

Table 8.1. Parameters of Procedures

Object	Formal Specification	Actual Parameter
numerical value	VALUE Numbertype Id *	Expression

*Composite specification of similar parameters has Idlist in place of Id
[b]see Section 8.3.2.3, “Table Parameters” [55]
[c]see Section 8.3.4, “Procedure Parameters” [56]

*Composite specification of similar parameters has Idlist in place of Id
[b]see Section 8.3.2.3, “Table Parameters” [55]
[c]see Section 8.3.4, “Procedure Parameters” [56]

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 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 index table identifier, a whole-word table-element or an anonymous reference. Index expressions are evaluated on 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 (italicized for clarity) might be

```
        PROCEDURE q (LABEL b; FIXED(24,1) 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 (see Section 8.3, “Parameter Specification” [53]) 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 as the place where it appears on the program sheet (even though this might be an illegal position). Everything except the body of the procedure 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, “Restrictions Connected with Scoping” [9]. 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.

Chapter 9

Communicators

The segments of a program may communicate with each other through COMMON (see Section 9.1, “COMMON Communicators” [61] 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 (see Section 3.3, “Globals” [9]) 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 division of objects into data, places and procedures. A COMMON data item is a declaration of the identifiers listed within it, exactly as in Chapter 4, *Reference to Data* [11], storage being allocated as in Section 4.5, “Storage Allocation” [20], presets an overlays as in Section 4.6, “Presetting” [21] and Section 4.8, “Overlay Declarations” [24]. Communication of places and procedures takes the form of *specification*, as in the equivalent parameters of a procedure declaration (Section 8.3.3, “Place Parameters” [55] and Section 8.3.4, “Procedure Parameters” [56]). 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 the questions of implementation which falls 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 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.

Chapter 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:

1. *Constants* for presetting, optionally signed.
2. *Integers* and *reals* as primaries in expressions. A sign attached to a primary belongs syntactically to the expression and not to the number.
3. *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
    Octalist
```

```
    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, “Numbers” [66]). 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 B, *List of Language Symbols* [97]). Layout characters are not acceptable as arguments to 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 B, *List of Language Symbols* [97]). A string is classed as an unconditional expression (Chapter 6, *Expressions* [29]), 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 integers values as are defined for literal constants.

```
    String ::=
" sequence of characters with quotes matched "
```

Chapter 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 to 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 of canceled (Section 11.2.4, “Deletion and Redefinition of Macros” [74]. However the macro processor treats comments and character strings (see Section 10.4, “Strings” [68]) 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 statement could legally appear, and is removed from it by 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 the string quotes are matched (but see Appendix B, *List of Language Symbols* [97]). Care must be taken to include brackets, such as BEGIN and END, as part of the macro body wherever 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 Section 11.2.1, “String Replacement” [72]. 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 macros definitions for the formal parameters before the macro body is substituted. Thus, to continue the example from Section 11.2.2, “Parameters of Macros” [73], 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 of the macro name is redefined or deleted. The scope of a macro is independent of the blocks 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.

11.2.5. 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 A Syntax Summary

```
Actual ::=
  Expression
   Wordreference
   Destination
   Name
```

```
Actuallist ::=
  Actual
   Actual , Actuallist
```

```
Addoperator ::=
  +
  -
```

Alternative ::=
  Statement

Answerspec ::=
  Numbertype
   Void

Answerstatement ::=
  ANSWER
   Expression

Arraydec ::=
  Numbertype
   ARRAY
   Arraylist
   Presetlist

Arrayitem ::=
  Idlist [*Sizelist*]

Arraylist ::=
  Arrayitem
   Arrayitem , *Arraylist*

Assignmentstatement ::=
  Variable := *Expression*

Base ::=
  (*Id*)

```
  Id [ Signedinteger ]
```

```
Bitposition ::= Integer
```

```
Block ::=
  BEGIN
   Declist ; Statementlist
   END
```

```
Booleanword ::=
  Booleanword2
   Booleanword4
   DIFFER
   Booleanword5
```

```
Booleanword2 ::=
  Booleanword3
   Booleanword5
   UNION
   Booleanword6
```

```
Booleanword3 ::=
  Booleanword6
   MASK
   Typedprimary
```

```
Booleanword4 ::=
  Booleanword
   Typedprimary
```

```
Booleanword5 ::=
  Booleanword2
   Typedprimary
```

```
Booleanword6 ::=
  Booleanword3
   Typedprimary
```

```
Bracketedcomment ::=
  (
   any sequence of characters in which round brackets are matched
   )
```

```
Codesequence ::=
  defined in a particular implementation
```

```
Codestatement ::=
  CODE
   BEGIN
   Codesequence
   END
```

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

```
Commoncommunicator ::=
  COMMON ( Commonitemlist )
```

Commonitem ::=
Datadec
Overlaydec
Placespec
Procedurespec
Void

Commonitemlist ::=
Commonitem
Commonitem ; *Commonitemlist*

Comparator ::=
<
<=
=
>=
>
<>

Comparison ::=
Simpleexpression
Comparator
Simpleexpression

Compoundstatement ::=
BEGIN
Statementlist
END

Condition ::=
Condition
OR
Subcondition
Subcondition

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

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

```
Consequence ::=
  Simplestatement
   Label : Consequence
```

```
Constant ::=
  Number
   Addoperator
   Number
```

```
Constantlist ::=
  Group
   Group , Constantlist
```

Datadec ::=
  Numberdec
   Arraydec
   Tabledec

Dec ::=
  Datadec
   Overlaydec
   Switchdec
   Proceduredec

Declist ::=
  Dec
   Dec ; *Declist*

Destination ::=
  Label
   Switch [*Index*]

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

Digitlist ::=
  Digit
   Digit
   Digitlist

Dimension ::=
  Lowerbound : *Upperbound*

```
Dummystatement ::=
  void
```

```
Elementdec ::=
  Id
   Numbertype
   Wordposition
   Id
   Partwordtype
   Wordposition , Bitposition
```

```
Elementdeclist ::=
  Elementdec
   Elementdec ; Elementdeclist
```

```
Elementpresetlist ::=
  PRESET
   Constantlist
   Void
```

```
Elementscale ::=
  ( Totalbits , Fractionbits )
  ( Totalbits )
```

```
Endcomment ::=
  Id
```

```
Expression ::=
  Unconditionalexpression
   Conditionalexpression
```

Factor ::=
Primary
Booleanword

Forelement ::=
Expression
Expression
WHILE
Condition
Expression
STEP
Expression
UNTIL
Expression

Forlist ::=
Forelement
Forelement , *Forlist*

Formalpair ::=
Id : *Id*

Formalpairlist ::=
Formalpair
Formalpair , *Formalpairlist*

Forstatement ::=
FOR
Wordreference := *Forlist*
DO
Statement

Fractionbits ::= *Signedinteger*

Gotostatement ::=
  GOTO
   Destination

Group ::=
  Constant
  (*Constantlist*)
  Void

Id ::=
  Letter
   Letterdigitstring

Idlist ::=
  Id
   Id , *Idlist*

Index ::=
  Expression

Integer ::=
  Digitlist
   OCTAL (*Octallist*)
    LITERAL (*printing character*)

Label ::= *Id*

Labellist ::=
  Label
   Label , *Labellist*

Length ::= *Integer*

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

Letterdigitstring ::=
  Letter
   Letterdigitstring
   Digit
   Letterdigitstring
   Void

Lowerbound ::=
  Signedinteger

Macrobody ::=
    any sequence of characters in which string quotes are matched

Macrocall ::=
  Macroname
   Macroname (*Macrostringlist*)

Macrodefinition ::=
    DEFINE
   Macroname " *Macrobody* " ;
    DEFINE
   Macroname (*Idlist*) " *Macrobody* " ;

Macrodeletion ::=
    DELETE
  Macroname ;

Macroname ::=
  Id

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

Macrostringlist ::=
  Macrostring , *Macrostringlist*
  Macrostring

Multoperator ::=
  *
  /

Name ::=
  Id

Number ::=
  Real
  Integer

Numberdec ::=
  Numbertype
   Idlist
   Presetlist

Numbertype ::=
  FLOATING
   FIXED
   Scale
   INTEGER

Octaldigit ::= 0 1 2 3 4 5 6 7

Octallist ::=
  Octaldigit
   Octaldigit
   Octalist

Overlaydec ::=
  OVERLAY
   Base
   WITH
   Datadec

Parameterspec ::=
  VALUE
   Formalpairlist
   LOCATION
   Formalpairlist
   Specifier
   Idlist
  etc

Parameterspec ::=
  Specifier
   Idlist
   Tablespec
   Procedurespec

Parameterspeclist ::=
  Parameterspec
   Parameterspec ; *Parameterspeclist*

Partword ::=
  Id [*Index*]
  BITS [*Totalbits* , *Bitposition*] *Typedprimary*

Partwordreference ::=
  Id [*Index*]
  BITS [*Totalbits* , *Bitposition*] *Wordreference*

Partwordtype ::=
  Elementscale
   UNSIGNED
   Elementscale

Placespec ::=
  LABEL
   Idlist
   SWITCH
   Idlist

Presetlist ::=
  := *Constantlist*

Void

Primary ::=
  Untypedprimary
   Typedprimary

Procedurecall ::=
  Id
   Id (*Actuallist*)

Proceduredec ::=
  Answerspec
   PROCEDURE
   Procedureheading ; *Statement*
   Answerspec
   RECURSIVE
   Procedureheading ; *Statement*

Procedureheading ::=
  Id
   Id (*Parameterspeclist*)

Procedurespec ::=
  Answerspec
   PROCEDURE
   Procparamlist

Procparameter ::=
  Id
   Id (*Typelist*)

Procparamlist ::=
  Procparameter
  Procparameter , *Procparamlist*

Real ::=
  Digitlist . *Digitlist*
  Digitlist
  10
  Signedinteger
  10
  Signedinteger
  Digitlist . *Digitlist*
  10
  Signedinteger
  OCTAL (*Octallist* . *Octallist*)

Scale ::= (*Totalbits* , *Fractionbits*)

Sign ::=
  +
  -
  Void

Signedinteger ::=
  Integer
  Addoperator
  Integer

Simpleexpression ::=
  Term
   Addoperator
   Term
   Simpleexpression
   Addoperator
   Term

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

Sizelist ::=
  Dimension
   Dimension , *Dimension*

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

Specimen ::=
  ALPHA
   Sign
   BETA

```
  Sign
```

```
Statement ::=
  Label : Statement
   Simplestatement
   Conditionalstatement
   Forstatement
```

```
Statementlist ::=
  Statement
   Statement ; Statementlist
```

```
String ::=
  " sequence of characters with quotes matched "
```

```
Subcondition ::=
  Subcondition
   AND
   Comparison
   Comparison
```

```
Switch ::= Id
```

```
Switchdec ::=
  SWITCH
   Switch := Labellist
```

```
Tabledec ::=
  TABLE
   Id [ Width , Length ]
```

[*Elementdeclist*
Elementpresetlist] *Presetlist*

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

Term ::=
Factor
Term
Multoperator
Factor

Totalbits ::= *Integer*

Type ::=
Specifier
TABLE
Answerspec
PROCEDURE

Typedprimary ::=
Wordreference
Partword
LOCATION (*Wordreference*)
Numbertype (*Expression*)
Procedurecall
Integer

Typelist ::=
Type

```
  Type , Typelist
```

```
Unconditionalexpression ::=
  Simpleexpression
   String
```

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

```
Upperbound ::=
  Signedinteger
```

```
Variable ::=
  Wordreference
   Partwordreference
```

```
Width ::= Integer
```

```
Wordposition ::= Signedinteger
```

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

Appendix B List of Language Symbols

The following 45 keywords are specified in the Official Definition Appendix 2.

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

The following keywords extend the official syntax.

BINARY	FINISH	PROGRAM	SRA
CONSTANT	HEX	SEGMENT	SRL
CORAL	LONG	SLL	

Appendix C

Levels of Implementation

The language requirements for a particular machine or 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
- 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.

Appendix D

Implementation-Defined Characteristics

The Coral 66 language allows for certain machine dependences in a controlled manner. Each implementation must document all implementation-defined characteristics:

D.1. Language Profiles

Using compile time switches, the user may select one of several language profiles.

The profiles supported are:

- Official Definition Profile
- XGC Profile
- Custom Profile

D.1.1. Official Definition Profile

This profile corresponds to the features of the official definition, and inlcudes all features that are described as optional. That is, the profile includes:

- RECURSIVE procedures
- TABLE facility
- FIXED numbers
- BITS, DIFFER, UNION and MASK
- FLOATING numbers
- COMMON Communicators
- EXTERNAL Communicators
- ABSOLUTE Communicators
- CODE statements

D.1.2. The XGC Profile

This profile includes all the features of the Official Definition Profile, and the following extensions:

- The BYTE Numbertype
- BYTE arrays from the Blandford Extension
- LONG Numbertypes
- The LIBRARY communicator
- BINARY Numbers
- HEX Numbers
- CONSTANT Declarations
- Shift operators
- ANSI C compatible strings

D.1.3. The Custom Profile

The custom profile is reserved for users who require the language features to be the same as those of some other compiler. There is no default custom profile, and each custom profile requires changes to the compiler to implement the necessary language features.

D.2. Implementation Details

Numeric Types. See Section 4.1, “Numeric Types” [11].

XGC Coral supports 16[1], 32 and 64 bit integer types, 16, 32 and 64 bit fixed point types, and 32 and 64 bit floating point types. The implementation defined keyword LONG is used to form LONG INTEGER, LONG FIXED and LONG FLOATING.

Meaning of Word-position. See Section 4.4.2, “Table-Element Declaration” [16].

XGC Coral allows table elements to extend over more than one word. Word-position always refers to the word in which the least significant bit is located.

Format of Code Statements. See Section 7.5, “Code Statements” [45].

In XGC Coral 66, a code statement follows the practice established for other programming languages. Each line of the code statement is an assembly language instruction, where the operands are references to Coral source objects. A full description will be found in Section 7.5, “Code Statements” [45].

Support for the COMMON Communicator. See Section 9.1, “COMMON Communicators” [61].

COMMON Communicators are implemented as recommended. They are the primary means by which program segments communicate. Where the source text for a program is located across many files, it is usual for the COMMON Communicators to have their own files, which are then compiled along with the program segments by giving the file name on the compiler command line. Note: there is no *include* feature in Coral 66.

Support for the LIBRARY Communicator. See Section 9.2, “LIBRARY Communicators” [62].

In XGC Coral, the library communicator is used to include a Coral source file that contains an external communicator that defines the

[1]16-bit types are only supported on 16-bit computers

data and procedures in a program library. It is typically used with standard libraries (an I-O library for example), but may be used with user-defined libraries too. The communicator is written LIBRARY ("filename"). The convention is for files used in this way to have the suffix ".h66" to indicate that they are Coral 66 files. For example, the file for the math library is called "math.h66", and is included by the statement LIBRARY ("math.h66").

Support for the EXTERNAL Communicator. See Section 9.3, “EXTERNAL Communicators” [63].

The EXTERNAL Communicator is supported as recommended. The form is similar to that of the COMMON communicator, but using the keyword EXTERNAL rather than COMMON. As an implementation-defined extension, the syntactic form Id / String is permitted so that linkage may be made to external symbols that are not acceptable as Coral identifiers.

Support for the ABSOLUTE Communicator. See Section 9.4, “ABSOLUTE Communicators” [63].

ABSOLUTE Communicators are implemented as recommended. The syntactic form Id / Integer gives the address of the object identified by Id. On a byte-addressed computer, this is a byte address.

Length of identifiers. See Section 10.1, “Identifiers” [65].

XGC Coral permits identifiers of any length up to the length of a line. A compile-time option is provided to specify whether just the first twelve or all the characters are significant.

Binary Numbers. See Section 10.2, “Numbers” [66].

Integers and floating point numbers may also be given in binary notation. The implementation defined keyword BINARY is used as a prefix, as in the following example: BINARY (1.0001). Note: binary floating point numbers cannot have an exponent.

Hexadecimal Numbers. See Section 10.2, “Numbers” [66].

Integers and floating point numbers may also be given in hexadecimal notation. The additional digits that represent values 10 to 15 are represented using the letters 'a' to 'f', in lower case or in upper case. The implementation defined keyword HEX is used as a prefix, as in the following example: HEX (fff.8). Note: neither the octal nor hexadecimal number formats permit an exponent.

Literal Constants. See Section 10.3, “Literal Constants” [68].

The numeric representation of characters in the LITERAL form, is determined by the host computer, and is assumed to be 7-bit ASCII. Non-printing characters, with the exception of the space character, are not accepted as literals. Note also the only one character is permitted between the parentheses.

The layout of strings. See Section 10.4, “Strings” [68].
The first character of each string contains the length of the string. The maximum length of a string is 255 characters.

The Macro Facility. See Section 11.2, “Macro Facility” [72].
Macros are supported as described in Section 11.2, “Macro Facility” [72]. The macro body may be written over as many lines as necessary, and there is no limit on its size.

Support for formal pairs. See Section 8.3.5, “Non-Standard Parameter Specification” [58].
Formal pairs are supported as suggested in the Official Definition with the address of the actual parameter passed as the first of the pair, and the encoded type of the parameter passed as the second. The encoded type is represented as follows:

Table D.1. Encoded Type in a Formal Pair

Table D.1. Encoded Type in a Formal Pair

Encoded Type	Meaning
LITERAL (s)	BYTE

Constant Declarations. This is an XGC extension.
The keyword CONSTANT may be used to define a compile-time value that may be later used where a constant value is required. The value may be the result of an expression that includes literal numbers and references to other constants. For example:

```
                    CONSTANT Pi := 3.14159263589793;
CONSTANT Pi over 2 := Pi / 2;
```

Appendix E

Format of Code Statements

E.1. Format of Code Statements

In XGC Coral 66 the format of a code statement is:

```
CODE
BEGIN  instructions : output operands : input operands END
```

For example, here is how to use the 68881's “fsinx” instruction:

```
CODE
BEGIN  "fsinx   %1,%0" : "=f" (result) : "f" (angle) END
```

Here `angle` is the Coral expression for the input operand while `result` is that of the output operand. Each operand has `"f"` as its operand constraint, saying that a floating point register is required. The `=` in `=f` indicates that the operand is an output; all output operands' constraints must use `=`.

Each operand is described by an operand-constraint string followed by the Coral expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be locations; the compiler can check this. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. Code statements are most often used for machine instructions that the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit field), your constraint must allow a register. In that case, the compiler will use the register as the output of the code statement, and then store that register into the output location.

The output operands must be write-only; the compiler will assume that the values in these operands before the instruction are dead and need not be generated. Code statements do not support input-output or read-write operands. For this reason, the constraint character `+`, which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same Coral expression for both operands, or different expressions. For example, here we write the (fictitious) `combine` instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
CODE
```

```
BEGIN  "combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar) END
```

The constraint "0" for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
CODE
BEGIN  "combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar) END
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; the compiler knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but the compiler can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
   CODE
   BEGIN
"movc3 %0,%1,%2"
: COMMENT  no outputs ;
: "g" (from), "g" (to), "g" (count)
: "r0", "r1", "r2", "r3", "r4", "r5" END
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with %; to produce one % in the assembler code, you must write %% in the input.

If your assembler instruction can alter the condition code register, add cc to the list of clobbered registers. the compiler on some machines represents the condition codes as a specific hardware register; cc serves to name this register. On other machines, the condition code is handled differently, and specifying cc has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add memory to the list of clobbered registers. This will cause the compiler to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single code statement, separated either with newlines (written as \n) or with semicolons. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine _foo accepts arguments in registers 9 and 10:

```
CODE
BEGIN  "movl %0,r9;movl %1,r10;call _foo"
: COMMENT  no outputs;
: "g" (from), "g" (to)
: "r9", "r10" END
```

Unless an output operand has the & constraint modifier, the compiler may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use & for each output operand that may not overlap an input.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the code statement, as follows:

```
CODE
BEGIN
```

```
"clr %0;frob %1;beq 0f;mov #1,%0;0:"
  : "g" (result)
  : "g" (input) END
```

Speaking of labels, jumps from one code statement to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

If a code statement has output operands, the compiler assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

E.2. Constraints for Operands

Here are specific details on what constraint letters you can use with code statement operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

E.2.1. Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

“ `m` ”

A memory operand is allowed, with any kind of address that the target computer supports in general.

“ `o` ”

A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an auto-increment or auto-decrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter “ `o` ” is valid only when accompanied by both “ `<` ” (if the target machine has pre-decrement addressing) and “ `>` ” (if the target machine has pre-increment addressing).

“ `V` ”

A memory operand that is not offsettable. In other words, anything that would fit the “ `m` ” constraint but not the “ `o` ” constraint.

“ `<` ”

A memory operand with auto-decrement addressing (either pre-decrement or post-decrement) is allowed.

“ `>` ”

A memory operand with auto-increment addressing (either pre-increment or post-increment) is allowed.

“ `r` ”

A register operand is allowed provided that it is in a general register.

“ `d` ”, “ `a` ”, “ `f` ”, ...
Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. “ `d` ”, “ `a` ” and “ `f` ” are defined on the 68000/68020 to stand for data, address and floating point registers.

“ `i` ”

An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

“ `n` ”

An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use “ `n` ” rather than “ `i` ”.

“ `I` ”, “ `J` ”, “ `K` ”, ... “ `P` ”

Other letters in the range “ `I` ” through “ `P` ” may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, “ `I` ” is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

“ `E` ”

An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

“ `F` ”

An immediate floating operand (expression code `const_double`) is allowed.

“ `G` ”, “ `H` ”

“ `G` ” and “ `H` ” may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

“ `s` ”

An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an instruction allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use “ `s` ” instead of “ `i` ”? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a “ `moveq` ” instruction. We arrange for this to happen by defining the letter “ `K` ” to mean "any integer outside the range -128 to 127", and then specifying “ `Ks` ” in the operand constraints.

“ `g` ”

Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

“ `X` ”

Any operand whatsoever is allowed.

“ `0` ”, “ `1` ”, “ `2` ”, ... “ `9` ”

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which code statements distinguish. For example, an add instruction uses two input operands and an output operand, but in many computers an add instruction really has only two operands, one of them an input-output operand:

```

addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

“ `p` ”

An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

“ `p` ” in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

“ `Q` ”, “ `R` ”, “ `S` ”, ... “ `U` ”

Letters in the range “ `Q` ” through “ `U` ” may be defined in a machine-dependent fashion to stand for arbitrary operand types.

E.2.2. Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the “ `?` ” and “ `!` ” characters:

`?`

Disparage slightly the alternative that the “ `?` ” appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each “ `?` ” that appears in it.

`!`

Disparage severely the alternative that the “ `!` ” appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

E.2.3. Constraint Modifier Characters

Here are constraint modifier characters.

“ `=` ”

Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

“ `+` ”

Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. “ `=` ” identifies an output; “ `+` ” identifies an operand that is both input and output; all other operands are assumed to be input only.

“ `&` ”

Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

“ `&` ” applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires “ `&` ” while others do not. See, for example, the “ `movdf` ” instruction of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the “ `mulsi3` ” instruction of the ARM.

“ `&` ” does not obviate the need to write “ `=` ”.

“ `%` ”

Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints.

“ `#` ”

Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

E.2.4. M68000 Constraints

These additional constraints apply to the M68000 family.

“ `d` ”
: A data register, %d0 to %d7

“ `a` ”
: An address register, %a0 to %a7

“ `f` ”
: A MC68881 floating point register, %fp0 to %fp7

Index