BRITISH STANDARD

BS 5905:1980

Incorporating Amendment No. 1

Specification for

Computer programming language CORAL 66

UDC 681.3.06CORAL 66

Cooperating organizations

The Data Processing Standards Committee, under whose direction this British Standard was prepared, consists of representatives from the following Government departments and scientific and industrial organizations:

British Computer Society Ltd.*
British Paper and Board Industry Federation (PIF)
British Printing Industries Federation
Business Equipment Trade Association*
Central Computer Agency (Civil Service Department)*
Committee of London Clearing Bankers on behalf of the Committee of Scottish Clearing Bankers, Co-operative Bank, Central Trustee Savings Bank and Yorkshire Bank
Department of Industry (Computers Systems and Electronics)
Department of Industry (National Physical Laboratory)*
Electricity Supply Industry in England and Wales*
Government Communications Headquarters
HM Customs and Excise
Institute of Cost and Management Accountants
Institute of Purchasing and Supply
Institution of Electrical Engineers
Institution of Mechanical Engineers
Inter-university Committee on Computing
London Transport Executive
Ministry of Defence*
National Computer Users' Forum
National Computing Centre Ltd.*
National Research Development Corporation
Post Office*
Society of British Aerospace Companies Limited

The organizations marked with an asterisk in the above list, together with the following, were directly represented on the committee entrusted with the preparation of this British Standard:

Association for Literary and Linguistic Computing
Association of Computer Units in Colleges of Higher Education (ACUCHE)
British Gas Corporation
Computing Services Association
Control and Automation Manufacturer's Association (BEAMA)
Edinburgh Regional Computing Centre
Engineering Equipment Users' Association
Hatfield Polytechnic
University of London

This British Standard, having been prepared under the direction of the Data Processing Standards Committee, was published under the authority of the Executive Board and comes into effect on
31 October 1980

The following BSI references relate to the work on this standard:
Committee reference IST/5
Draft for comment 78/63605 DC

ISBN 0 580 11442 2

Amendments issued since publication

Amd. No.	Date of issue	Comments
5079	October 1986	Indicated by a sideline in the margin

Contents

Foreword

This British Standard has been prepared under the direction of the Data Processing Standards Committee and is based on the "Official definition of CORAL 66", first published in 1970 by Her Majesty's Stationery Office, and reproduces material taken from that publication. This standard follows some ten years after the "Official definition" and the BSI committee believes that the standard should not contain requirements incompatible with the very large number of existing implementations. For this reason, although a number of proposals for extensions of the language have been considered, their adoption has, in general, been reserved until a future revision of the standard.

Two exceptions have, however, been made in this first edition:

a) the number denotation HEX has been included (see **6.9.2**);

b) the language words CORAL and SEGMENT have been included to give a formal syntax to a multisegment program (see **6.1.3**).

Future revisions may make fuller provision for byte-addressable machines with a byte-oriented, but otherwise conformable, language definition. The current requirements assume a word-addressable virtual machine, but some provision for such byte-addressable machines is made in Appendix B.

This standard aims to achieve an overall economy, in terms of human effort, in the development of computer-based systems for real-time applications and to protect the interests of the user by encouraging the use of computers that comply with this standard.

CORAL 66 provides a means of increasing the implementation efficiency of computer applications in environments where the input/output communication requirements may not have been standardized and that are time-critical. CORAL 66 is a kernel high-level computer language intended to replace a high proportion of assembly code in each specific application.

Items not explicitly specified in this standard should be clearly understood as being unspecified. These unspecified items result from an original design objective that CORAL 66 should favour all computer architectures as equally as possible and should allow implementors to exploit hardware features as efficiently as possible. The objective is to promote the use of a common form of expression whenever it is expedient to do so. The language specification thus explicitly includes both the insertion of machine code statements and anonymous referencing, so as to ensure that special machine features and peripherals can be handled efficiently at the cost of reduced program portability. Furthermore, implementors are thus able to adopt the hardware conventions of the target machine with regard to numeric representation and computation. The parameterized macro facility enables the source text to retain a high level of readability. CORAL 66 therefore aims to minimize, rather than to eliminate, the consequent machine dependency of programs and to maximize the portability of programming and software maintenance staff.

It is virtually impossible to design a standard language such that programs will run with equally high efficiency in all types of computer and in any applications. Much of the design of CORAL 66 reflects this difficulty. For example, the language permits the use of non-standard "code statements" for any parts of a program where it may be important to exploit particular hardware facilities. A special feature is scaled fixed-point arithmetic for use in small fixed-point machines; the floating-point facilities of the language can be omitted when hardware limitations make the use of floating-point arithmetic uneconomical. Other features can also be omitted without reducing the power of the language to an unacceptably low level. Major features that may be omitted are listed in **6.12**.

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

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

Editorial note. It is normal convention in British Standards to use italic type for algebraic quantities. Since the status of such quantities contained in this standard may or may not directly represent true variable quantities, this convention has not been adopted in this standard.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, pages i to iv, pages 1 to 34, an inside back cover and a back cover.

This standard has been updated (see copyright date) and may have had amendments incorporated. This will be indicated in the amendment table on the inside front cover.

blank

1 Scope

This British Standard specifies the semantics and syntax of the computer programming language CORAL 66 by specifying requirements for a compiler and for a conforming program.

NOTE 1 Any feature that is not explicitly specified in this standard has been intentionally left unspecified. Appendix A lists some unspecified features.

Appendix B gives additional information regarding implementation.

NOTE 2 The examples given among the specification requirements are not intended to add to or detract from the requirements, but are included purely to aid understanding.

2 References

The titles of the publications referred to in this standard are listed on the inside back cover.

3 Definitions

For the purposes of this British Standard, the following definitions, together with those for other terms given in BS 3527, apply.

3.1
layout characters

the six "layout characters" and the space character specified in BS 4730

3.2
printing characters

the graphic characters specified in BS 4730

NOTE To draw attention to the language concepts, some terms are printed in italics on their first mention in this standard.

4 Syntactic metalanguage[1)]

The widespread use of syntax-driven methods of compilation lends increasing importance to syntax methods of language description. This standard specifies 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 abandoned 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 the various alternative expansions for the class name. These alternatives are either each printed on a new line or separated by "*or*", as appropriate. Where a single alternative spreads over more than one line of print, the additional lines are inset in relation to the starting position of the alternative. 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 specimens complying with these rules are ALPHA + and BETA.

The equals sign is used to separate the left-hand side from the right, except after its first appearance in a rule.

Certain constructs are defined informally in this standard; these are underlined.

The syntax rules are listed in Table 1.

[1)] A British Standard syntactic metalanguage is at an early stage of preparation.

5 Compliance

5.1 Implementations

5.1.1 Any implementation of CORAL 66 that can accept all of the features of the language specified in clause **6** and with the meanings defined in clause **6** complies with the requirements of this standard.

5.1.2 Except as specifically allowed by **6.12**, an implementation that omits any of the features of the language specified in clause **6** or that includes such a feature, but with a meaning altered in any way from that specified in clause **6**, does not comply with the requirements of this standard.

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Absolutecommunicator	=	defined in a particular implementation to conform to the style of the commoncommunicator	**6.8.5**
Actual	=	Expression	**6.6.4**
		Wordreference	
		Destination	
		Name	
Actuallist	=	Actual	**6.6.4**
		Actual, Actuallist	
Addoperator	=	+	**6.5.2**
		–	
Alternative	=	Statement	**6.6.10**
Answerspec	=	Numbertype	**6.7.2**
		Void	
Answerstatement	=	ANSWER Expression	**6.6.5**
Arraydec	=	Numbertype ARRAY Arraylist Presetlist	**6.3.3**
Arrayitem	=	Idlist [Sizelist]	**6.3.3**
Arraylist	=	Arrayitem	**6.3.3**
		Arrayitem, Arraylist	
Assignmentstatement	=	Variable ← Expression	**6.6.2**
Base	=	Id	**6.3.8**
		Id [Signedinteger]	
Bitposition	=	Integer	**6.3.4.3.1**
Block	=	BEGIN Declist; Statementlist END	**6.2.1**
Booleanword	=	Booleanword2	**6.5.4**
		Booleanword4 DIFFER Booleanword5	
Booleanword2	=	Booleanword3	**6.5.4**
		Booleanword5 UNION Booleanword6	
Booleanword3	=	Booleanword6 MASK Typedprimary	**6.5.4**
Booleanword4	=	Booleanword	**6.5.4**
		Typedprimary	
Booleanword5	=	Booleanword2	**6.5.4**
		Typedprimary	
Booleanword6	=	Booleanword3	**6.5.4**
		Typedprimary	
Bracketedcomment	=	(any sequence of characters in which round brackets are matched)	**6.10.3**
Codesequence	=	defined in a particular implementation	**6.6.6**
Codestatement	=	CODE BEGIN Codesequence END	**6.6.6**

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Commentsentence	=	COMMENT any sequence of characters not including a semicolon;	**6.10.3**
Commoncommunicator	=	COMMON (Commonitemlist)	**6.8.2**
Commonitem	=	Datadec	**6.8.2**
		Overlaydec	
		Placespec	
		Procedurespec	
		Void	
Commonitemlist	=	Commonitem	**6.8.2**
		Commonitem; Commonitemlist	
Comparator	=	< *or* ⩽ *or* = *or* ⩾ *or* > *or* ≠	**6.5.6.2**
Comparison	=	Simpleexpression Comparator Simpleexpression	**6.5.6.2**
Compoundstatement	=	BEGIN Statementlist END	**6.6.7**
Condition	=	Condition OR Subcondition	**6.5.6.2**
		Subcondition	
Conditionalexpression	=	IF Condition	**6.5.6.1**
		THEN Expression	
		ELSE Expression	
Conditionalstatement	=	IF Condition THEN Consequence	**6.6.10**
		IF Condition THEN Consequence ELSE Alternative	
Consequence	=	Simplestatement	**6.6.10**
		Label: Consequence	
Constant	=	Number	**6.9.2**
		Addoperator Number	
Constantlist	=	Group	**6.3.6.2**
		Group, Constantlist	
Datadec	=	Numberdec	**6.2.1**
		Arraydec	
		Tabledec	
Dec	=	Datadec	**6.2.1**
		Overlaydec	
		Switchdec	
		Proceduredec	
Declist	=	Dec	**6.2.1**
		Dec; Declist	
Destination	=	Label	**6.6.3**
		Switch [Index]	
Digit	=	0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7 *or* 8 *or* 9	**6.9.1**
Digitlist	=	Digit	**6.9.2**
		Digit Digitlist	
Dimension	=	Lowerbound : Upperbound	**6.3.3**
Dummystatement	=	Void	**6.6.9**
Elementdec	=	Id Numbertype Wordposition	**6.3.4.3.1**
		Id Partwordtype Wordposition, Bitposition	

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Elementdeclist	=	Elementdec	**6.3.4.2**
		Elementdec; Elementdeclist	
Elementpresetlist	=	PRESET Constantlist	**6.3.6.3**
		Void	
Elementscale	=	(Totalbits, Fractionbits)	**6.3.4.3.3**
		(Totalbits)	
Endcomment	=	Id	**6.10.3**
Expression	=	Unconditionalexpression	**6.5.1**
		Conditionalexpression	
Externalcommunicator	=	defined in a particular implementation to conform to the style of the commoncommunicator	**6.8.4**
Factor	=	Primary	**6.5.3.1**
		Booleanword	
Forelement	=	Expression	**6.6.11.1**
		Expression WHILE Condition	
		Expression STEP Expression UNTIL Expression	
Forlist	=	Forelement	**6.6.11.1**
		Forelement, Forlist	
Forstatement	=	FOR Wordreference ← Forlist DO Statement	**6.6.11.1**
Fractionbits	=	Signedinteger	**6.3.1**
Gotostatement	=	GOTO Destination	**6.6.3**
Group	=	Constant	**6.3.6.2**
		(Constantlist)	
		Void	
Hexdigit	=	Digit	**6.9.2**
		A *or* B *or* C *or* D *or* E *or* F	
Hexlist	=	Hexdigit	**6.9.2**
		Hexdigit Hexlist	
Id	=	Letter Letterdigitstring	**6.9.1**
Idlist	=	Id	**6.3.2**
		Id, Idlist	
Index	=	Expression	**6.5.3.4**
Integer	=	Digitlist	**6.9.2**
		HEX (Hexlist)	
		OCTAL (Octallist)	
		LITERAL (printing character)	
Label	=	Id	**6.4**
Labellist	=	Label	**6.4**
		Label, Labellist	
Length	=	Integer	**6.3.4.2**
Letter	=	a *or* b *or* c *or* d *or* e *or* f *or* g *or* h *or* i *or* j *or* k *or* l *or* m *or* n *or* o *or* p *or* q *or* r *or* s *or* t *or* u *or* v *or* w *or* x *or* y *or* z	**6.9.1**
Letterdigitstring	=	Letter Letterdigitstring	**6.9.1**
		Digit Letterdigitstring	
		Void	

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Librarycommunicator	=	defined in a particular implementation to conform to the style of the commoncommunicator	**6.8.3**
Lowerbound	=	Signedinteger	**6.3.3**
Macrobody	=	any sequence of characters in which string quotes are matched	**6.10.3** and **6.11**
Macrocall	=	Macroname Macroname (Macrostringlist)	**6.10.3**
Macrodefinition	=	DEFINE Macroname ⊰ Macrobody ⊱; DEFINE Macroname (Idlist) ⊰ Macrobody ⊱;	**6.10.3**
Macrodeletion	=	DELETE Macroname;	**6.10.3**
Macroname	=	Id	**6.10.3**
Macrostring	=	any sequence of characters in which string quotes are matched, commas are protected by round or square brackets and in which such brackets are properly matched and nested	**6.10.3**
Macrostringlist	=	Macrostring Macrostring, Macrostringlist	**6.10.3**
Multoperator	=	* /	**6.5.2**
Name	=	Id	**6.6.4**
Number	=	Real Integer	**6.9.2**
Numberdec	=	Numbertype Idlist Presetlist	**6.3.2**
Numbertype	=	FLOATING FIXED Scale INTEGER	**6.3.1**
Octaldigit	=	0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7	**6.9.2**
Octallist	=	Octaldigit Octaldigit Octallist	**6.9.2**
Overlaydec	=	OVERLAY Base WITH Datadec	**6.3.8**
Parameterspec	=	Specifier Idlist Tablespec Procedurespec	**6.7.4.1** and **6.7.4.10**
Parameterspeclist	=	Parameterspec Parameterspec; Parameterspeclist	**6.7.3**
Partword	=	Id [Index] BITS [Totalbits, Bitposition] Typedprimary	**6.5.3.5**
Partwordreference	=	Id [Index] BITS [Totalbits, Bitposition] Wordreference	**6.6.2**
Partwordtype	=	Elementscale UNSIGNED Elementscale	**6.3.4.3.3**
Placespec	=	LABEL Idlist SWITCH Idlist	**6.8.2**
Presetlist	=	← Constantlist Void	**6.3.6.2**

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Primary	=	Untypedprimary	**6.5.3.1**
		Typedprimary	
Procedurecall	=	Id	**6.6.4**
		Id (Actuallist)	
Proceduredec	=	Answerspec PROCEDURE Procedureheading; Statement	**6.7.1**
		Answerspec RECURSIVE Procedureheading; Statement	
Procedureheading	=	Id	**6.7.3**
		Id (Parameterspeclist)	
Procedurespec	=	Answerspec PROCEDURE Procparamlist	**6.7.4.9**
Procparameter	=	Id	**6.7.4.9**
		Id (Typelist)	
Procparamlist	=	Procparameter	**6.7.4.9**
		Procparameter, Procparamlist	
Real	=	Digitlist . Digitlist	**6.9.2**
		Digitlist $_{10}$ Signedinteger	
		$_{10}$ Signedinteger	
		Digitlist . Digitlist $_{10}$ Signedinteger	
		HEX (Hexlist . Hexlist)	
		OCTAL (Octallist . Octallist)	
Scale	=	(Totalbits, Fractionbits)	**6.3.1**
Signedinteger	=	Integer	**6.9.2**
		Addoperator Integer	
Simpleexpression	=	Term	**6.5.2**
		Addoperator Term	
		Simpleexpression Addoperator Term	
Simplestatement	=	Assignmentstatement	**6.6.1**
		Gotostatement	
		Procedurecall	
		Answerstatement	
		Codestatement	
		Compoundstatement	
		Block	
		Dummystatement	
Sizelist	=	Dimension	**6.3.3**
		Dimension, Dimension	
Specifier	=	VALUE Numbertype	**6.7.4.1**
		LOCATION Numbertype	
		Numbertype ARRAY	
		LABEL	
		SWITCH	
Statement	=	Label : Statement	**6.6.1**
		Simplestatement	
		Conditionalstatement	
		Forstatement	

Table 1 — Alphabetical list of syntax rules

Syntax rule			Reference
Statementlist	=	Statement	**6.6.7**
		Statement; Statementlist	
String	=	⟊ sequence of characters with quotes matched ⟊	**6.9.4** and **6.11**
Subcondition	=	Subcondition AND Comparison	**6.5.6.2**
		Comparison	
Switch	=	Id	**6.4**
Switchdec	=	SWITCH Switch ← Labellist	**6.4**
Tabledec	=	TABLE Id [Width, Length] [Elementdeclist Elementpresetlist] Presetlist	**6.3.4.2**
Tablespec	=	TABLE Id [Width, Length] [Elementdeclist]	**6.7.4.6**
Term	=	Factor	**6.5.2**
		Term Multoperator Factor	
Totalbits	=	Integer	**6.3.1**
Type	=	Specifier	**6.7.4.9**
		TABLE	
		Answerspec PROCEDURE	
Typedprimary	=	Wordreference	**6.5.3.3**
		Partword	
		LOCATION (Wordreference)	
		Numbertype (Expression)	
		Procedurecall	
		Integer	
Typelist	=	Type	**6.7.4.9**
		Type, Typelist	
Unconditionalexpression	=	Simpleexpression	**6.5.1**
		String	
Untypedprimary	=	Real	**6.5.3.2**
		(Expression)	
Upperbound	=	Signedinteger	**6.3.3**
Variable	=	Wordreference	**6.6.2**
		Partwordreference	
Width	=	Integer	**6.3.4.2**
Wordposition	=	Signedinteger	**6.3.4.3.1**
Wordreference	=	Id	**6.5.3.4**
		Id [Index]	
		Id [Index, Index]	
		[Index]	

5.1.3 An implementation that includes features additional to those specified in clause **6** (i.e. "extensions") complies with the requirements of this standard provided that both the following conditions are fulfilled:

a) the extensions neither individually nor collectively alter in any way the meanings of the features covered by the requirements of clause **6**;

b) the extensions are clearly detailed in all descriptions of the implementation as being "extensions to CORAL 66 as specified by BS 5905".

NOTE A suite of programs exists that aids in the determination of whether a compiler complies with the requirements of this standard (see Appendix B). Compliance with the requirements of this standard is determined definitively by reference to clause **6** and not to the suite of programs.

5.2 Programs

5.2.1 A program written in CORAL 66 complies with the requirements of this standard provided that the program uses only those features of the language specified in clause **6**, and includes any desired sections of machine code in the manner allowed for by the requirements of clause **6**.

5.2.2 A program that includes any extensions to the language (see **5.1.2** and **5.1.3**), or includes machine code in a manner other than as allowed by the requirements of clause **6**, does not comply with the requirements of this standard.

6 Requirements

6.1 The CORAL 66 program

6.1.1 *Preliminary*. 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 shall be regarded as a sequence of symbols, each visibly representable by a unique character or combination of characters. The symbols of the language are specified in **6.11**, but the characters are not. For the purposes of specification of the language, words printed in this standard in upper case letters are treated as single symbols and are referred to as *language words*. Lower case letters are reserved for use in *identifiers*, which may also include digits in non-leading positions. Except where they are used in strings, layout characters shall be ignored by a CORAL 66 compiler.

6.1.2 *Objects*. A program shall be made up of symbols (e.g. BEGIN, =, 4) and arbitrary identifiers, which, by declaration, specification or setting, acquire the status of single symbols. Identifiers shall be names referring to *objects*, which shall be classified as follows:

a) data (numbers, arrays of numbers, tables);

b) places (labels and switches);

c) procedures (functions and processes).

6.1.3 *Program*. A program may be compiled in more than one unit and may be divided into *segments*. To make it possible to refer to chosen objects in different segments, the names and types of such objects shall be written outside the program segments in *communicators*. Objects fully defined within the program are rendered accessible to all segments by their mention in a communicator (see **6.2.3** and **6.8.2**). Objects whose full definition lies outside the program, e.g. library procedures, may be made accessible to all segments by mention in forms of communicator whose definition is implementation-dependent. A CORAL 66 program shall thus comprise, in some appropriate sequence:

a) name of program;

b) optional communicators;

c) one or more named segments.

Each program segment shall be in the form of a *block* (see **6.2.1**). This standard does not specify how the program or its segments shall be named nor how the segments shall be separated or terminated, but the typical form of a whole program compiled together is:

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

The program shall start running from the beginning of a segment, the choice of which depends upon a convention or mechanism outside the scope of this standard.

6.2 Scoping

NOTE A named object may be brought into existence for part of a program and may have no existence elsewhere (but see **6.3.7**). The part of the program in which it is declared to exist is known as its *scope*. One effect of scoping is to increase the freedom of choosing names for objects whose scopes do not overlap. Another effect is economy of computer storage space. The scope of an object is determined by the block structure of the program.

6.2.1 *Block structure*. A block shall be a statement consisting, internally, of one or more *declarations* followed by one or more *statements* punctuated by semicolons and all bracketed by a BEGIN and END.

The syntax shall be:

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 itself be a block merely by having the right form, blocks can be nested to an arbitrary depth. Except for global objects (see **6.2.3**), the scope of an object shall be 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*.

6.2.2 *Clashing of names*. Two objects that have the same name shall not have identical scopes. 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 that already exists. In this situation, the non-local object shall continue to exist through the inner block (e.g. a variable maintains its value), but it shall become temporarily inaccessible. The local meaning of the identifier shall always take precedence.

6.2.3 *Globals*. A program shall consist of one or more segments, each of which can be described as an *outermost block*, as there is no formal 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 shall be named in a communicator written outside the segments. For some types of object, such as COMMON data references, this shall take the form of a declaration, and shall be the only declaration required. Other types of object, specifically COMMON labels, COMMON switches and COMMON procedures, shall be fully defined within a segment. This means that COMMON labels shall be set, and COMMON switches and procedures shall be declared, in one of the outermost blocks of the program. Such objects are merely *specified* in the COMMON communicator (see **6.8.2**) and shall be treated as local in every outermost block of the program. Global objects *declared* outside the segments shall be treated as non-local. All globals shall be non-local in all the inner blocks of any segment. With these requirements for locality, questions of clashing shall be resolved in accordance with **6.2.2**.

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

6.2.5 *Restrictions connected with scoping*. No identifier other than a label shall be used before it has been declared or specified. Specification shall mean that the type of object to which an identifier refers has been given, but not necessarily the full definition of the object (see **6.8.2**). Typically, a procedure identifier is specified as referring to a certain type of procedure with certain types of parameters by the heading of the procedure declaration, but the procedure is not fully defined until the end of the declaration as a whole. As an example of this, assume that two procedures f and g are declared in succession after the beginning of a segment; the body of g may call on itself or on the procedure f, but the body of f shall not call on the procedure g unless g has been specified in a COMMON communicator.

6.3 References to data

6.3.1 *Numeric types*. There shall be three numeric types; floating-point, fixed-point and integer. Except in certain part-word table-elements (see **6.3.4.3.3**), all three types shall be signed. Numeric type shall be indicated by the language word FLOATING or INTEGER, or by the language word FIXED followed by scaling constants, which shall be given numerically, e.g.:

FIXED (13,5)

This example specifies five fractional bits and a minimum of 13 bits to represent the number as a whole, including the fractional bits and a sign. The number of fractional bits may be negative, zero or positive, and may cause the binary point to fall outside the significant field of the number.

NOTE For convenience and simplicity, this standard has been worded on the assumption that a number is confined within a single computer word. However, the representation and consequent range of values for any of the numeric types is not specified. Therefore, if in any implementation a system different from that assumed in this specification (e.g. two words for a floating-point number) is adopted, the requirements of this standard should be read accordingly.

The implementation shall define the system of numerical representation and the range of values. For FIXED type, the implementation shall define the status of any bits within the computer word that are not used by the declaration. The syntax for numeric type shall be:

Numbertype	=	FLOATING
		FIXED Scale
		INTEGER
Scale	=	(Totalbits, Fractionbits)
Totalbits	=	Integer
Fractionbits	=	Signedinteger

6.3.2 *Simple references*. The simplest objects of data are single numbers of floating, fixed-point or integer types. Simple references shall refer to such objects, e.g.:

INTEGER i, j, k;

FIXED (13,5) x, y

The declarations may include assignment of initial values; this is known as presetting, for which requirements are specified in **6.3.6**. The syntax for a number declaration shall be:

Numberdec	=	Numbertype Idlist Presetlist
Idlist	=	Id
		Id, Idlist

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

FIXED (13,5) ARRAY b[0:10];

FLOATING ARRAY c[1:3, 1:3]

The lower bound shall not 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 occur in a composite declaration, e.g.:

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

An array identifier shall refer to an array in its entirety, but its use in statements shall be confined to the communication of the array reference to a procedure. Elsewhere, an array identifier shall be indexed so that it refers to a single array element (but see Appendix B regarding run-time checks on subscript bounds). Indices shall have the form of arithmetic expressions, separated by commas, enclosed in square brackets after the array identifier.

Each index shall be evaluated to an integer as specified in **6.5.5**. The indices of a two-dimensional array shall be evaluated in the order met when reading the text from left to right. The syntax for an array declaration that includes a presetting facility (see **6.3.6.2**) shall be:

Arraydec	=	Numbertype ARRAY Arraylist Presetlist
Arraylist	=	Arrayitem
		Arrayitem, Arraylist
Arrayitem	=	Idlist [Sizelist]
Sizelist	=	Dimension
		Dimension, Dimension
Dimension	=	Lowerbound : Upperbound
Lowerbound	=	Signedinteger
Upperbound	=	Signedinteger

6.3.4 *Packed data*

6.3.4.1 *Preliminary*. There are two methods of referring to packed data; one in which an unnamed field is selected from any computer word that holds data (see **6.5.3.5**), and the other in which the data format is declared in advance. In the latter method, with which **6.3.4** is concerned, the format is replicated to form a *table*. A group of n words may be partitioned into bit-fields (where no fields cross a word boundary), and the same partitioning shall be applied to as many such groups (m, say) as are required. The total data-space for a table is thus nm words. Each group shall be known as a *table-entry*. The fields shall be named, so that a combination of field identifier and entry index selects data from all or part of one computer word, known as a *table-element*. The elements in an entry may occupy overlapping fields, and may leave unfilled spaces in the entry.

6.3.4.2 *Table declaration*. A table declaration shall serve two purposes. The first purpose shall be 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, e.g.

TABLE april [3, 30]

is the beginning of a declaration for the table “april” with 30 entries each three words wide, requiring an allocation of 90 words in all.

The second purpose of the declaration shall be to specify the structure of an entry by declaring the elements contained within it, as specified in **6.3.4.3**. Data-packing shall be implementation-dependent, and the examples assume a word length of 24 bits. The syntax for a table declaration shall be:

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

NOTE Requirements for the two presetting mechanisms are specified in **6.3.6.3**.

6.3.4.3 *Table-element declaration*

6.3.4.3.1 *General*. A table-element declaration shall associate an element name with a numeric type and with a particular field of each and every entry in the table. The field shall be the whole or part of a computer word, and the form of declaration shall differ accordingly. The syntax for an element declaration, more fully specified in **6.3.4.3.3**, shall be:

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

Bit-position shall be numbered from zero upwards, and the least significant bit of a word shall be designated bit-position zero. Normally, table-elements should be located so that they fall within the declared width of the table, but a CORAL 66 compiler shall not check the limits. To improve program legibility, a compiler may permit the language word BIT as an alternative to the comma in the foregoing syntax. The meaning of Bitposition is specified in **6.3.4.3.3**.

6.3.4.3.2 *Whole-word table-elements*. As specified in **6.3.4.3.1**, the form of declaration for whole-word table-elements shall be:

Id Numbertype Wordposition

For example,

tickets INTEGER 0

declares a pseudo-array of elements named “tickets”. (True array elements shall be located consecutively in store (see **6.3.5**).) Each element shall refer to a (signed) integer occupying word-position zero in an entry. Similarly, the example

weight FIXED (16, – 4) 1

locates “weight” in word-position 1 with a significance of 16 bits, stopping four bits short of the binary point. Floating-point elements shall be similarly permitted.

6.3.4.3.3 *Part-word table-elements*. Elements that occupy fields narrower than a computer word (and only such elements) shall be 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 shall be given in brackets (total bits and fraction bits), followed by the word-and bit-position of the field within the entry. Word-position shall be the word within which the field is located, and bit-position shall be the bit at the least significant end of the field.
The language word UNSIGNED increases the capacity of the field for positive numbers at the expense of eliminating negative numbers. For example, (4,2) allows numbers from – 2.00 to 1.75, whilst UNSIGNED (4,2) allows them from 0.00 to 3.75. If the scale contains only a single integer, e.g.

sunshine UNSIGNED (4) 2,4

the number in brackets shall represent the total number of bits for a *part-word integer*. Though (4,0) and (4) have essentially the same significance, (4,0) indicates fixed-point type and (4) indicates an integer type. The syntax of Partwordtype, for substitution in the syntax of **6.3.4.3.1**, shall be:

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

The requirements for Totalbits and Fractionbits are specified in **6.3.1**. The number of fraction bits may be negative, zero or positive, and the binary point may lie outside the declared field.

6.3.4.4 *Complete table declaration*. The complete table declaration built up as an illustrative example in **6.3.4.2** and **6.3.4.3** would be:

```
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]
```

All the numbers used to describe and locate fields shall be constants.

6.3.4.5 *Reference to tables and table-elements*. A table-element shall be selected by indexing its field identifier. To continue from the example given in **6.3.4.4**, the rain for 6 April would be written rain[5]. (An entry shall always have the conventional lower bound of zero.) In use, the names of table-elements shall always be indexed, although 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 shall select a computer word from the table data regarded as a conventional array of single computer words, with lower index bound zero (but see Appendix B regarding run-time checks on subscript bounds). Thus the implied bounds of the array "april" are 0 : 89. A word so selected shall be treated as a signed integer, from which it follows that april[6] in the example quoted in **6.3.4.4** would be equivalent to tickets[2]. A table name shall normally be 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 **6.3.8**).

6.3.5 *Storage allocation*. Computer storage space for data shall be 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 shall perform allocation serially. For example, the declarations

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

makes the locations of a, b and c become n, n + 1 and n + 2 respectively, and those of p and q become m and m + 1 respectively, where n and m are undefined and unrelated.

In two-dimensional arrays, the second index shall be stepped first; the declaration

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

locates the elements

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

in consecutive ascending locations.

6.3.6 ***Presetting***

6.3.6.1 *General*. 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 shall not be dynamic, and preset values that are altered by program shall not be restored unless the program or segment is reloaded. An object shall not be eligible for presetting if it is declared anywhere within:

a) the body of a recursive procedure; *or*

b) an inner block of the program; *or*

c) an inner block of a procedure body.

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

```
SEGMENT segment name;
BEGIN PROCEDURE f;
   BEGIN PROCEDURE g;
      BEGIN INTEGER i
```

6.3.6.2 *Presetting of simple references and arrays*. The preset constants shall be listed at the end of the declaration after an assignment symbol, and shall be allocated in the order specified in **6.3.5**, e.g.:

```
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 shall be ignored by the compiler. The number of constants in the presetlist shall not exceed, but may be less than, the number of words declared, and presetting shall cease when the presetlist is exhausted. The preset assignment symbol may optionally be the only part of the presetlist that is present (see **6.3.7**). The syntax shall be:

```
Presetlist      = ← Constantlist
                  Void
Constantlist    = Group
                  Group, Constantlist
Group           = Constant
                  (Constantlist)
                  Void
```

NOTE The main purpose of the final void may be seen by reference to **6.3.6.3**. For the expansion of Constant, see **6.9.2**.

6.3.6.3 *Presetting of tables*. Two alternative mechanisms shall be available for presetting a table. Either the internal structure of a table shall be completely disregarded and the table treated as an ordinary one-dimensional array of whole computer words, and preset as such, or all the table-elements shall be preset after their declaration list, as shown at Elementpresetlist in the syntax specified in **6.3.4.2**, e.g.:

```
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 language word PRESET shall be used instead of the assignment symbol specified in **6.3.6.2**. Each entry of the table shall be preset in succession as a group of elements, taken in the order of their declaration. Voids in the list shall imply absence of assignment; this may be necessary to avoid duplication when fields overlap, as do “ratio” and “arc” in the foregoing example. As specified in **6.3.6.2**, brackets used for grouping constants in the list of presets shall be ignored by the compiler. The syntax shall be:

```
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)
```

6.3.7 *Preservation of values*. Objects of data that have not been preset shall not be required to have existence outside the scope of their declarations. The values to which local identifiers refer shall in general be assumed to be undefined when a block is first entered and whenever it is subsequently re-entered.

NOTE This is consistent with 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.

When a data declaration contains a presetlist as permitted by the requirement of **6.3.6**, the values of all the objects named in that declaration shall remain undisturbed between successive entries to the block or procedure body, like "own" variables in ALGOL 60. Appearance of a preset assignment symbol at the end of the declaration shall suffice, even though the list of preset constants is void.

6.3.8 *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 reused for other purposes. 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.

NOTE Indiscriminate use of overlaying should be avoided, as it can lead to confusion and obscurity.

To form an overlay declaration, an ordinary data declaration shall be preceded by a phrase of the form

OVERLAY Base WITH

,where Base is a data reference that has previously been covered by a declaration in the same COMMON communicator or in the same segment. The base shall be a simple reference, a one-dimensional array reference or a table reference treated as a one-dimensional array of whole words. If the array or table identifier is not indexed, it shall refer to the location of its zero th element (which may be conceptual). Storage allocated by the overlay declaration shall start from the base, shall proceed serially (as specified in **6.3.5**) and shall not be overlaid by succeeding declarations unless these are themselves overlay declarations. There shall be no requirement to reorder storage that is already allocated. The syntax of an overlay declaration shall be:

Overlaydec	=	OVERLAY Base WITH Datadec
Base	=	Id
		Id [Signedinteger]

6.4 Place references: switches. Place references shall refer to positions of program statements, and the simplest position marker shall be the *label* (see **6.2.4**). A *switch* shall be a preset and unalterable array of labels with lower index bound one. These labels shall be within scope at the switch declaration. Any use of the indexed switch name shall refer 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 shall be:

Switchdec	=	SWITCH Switch ← Labellist
Labellist	=	Label
		Label, Labellist
Switch	=	Id
Label	=	Id

6.5 Expressions

6.5.1 *General*. The term "expression" shall be reserved for arithmetic expressions. CORAL 66 shall have no designational expressions of ALGOL 60 type. As there are no Boolean variables and no bracketed Boolean expressions (see **6.5.6.2**), the expressions after IF shall be termed *conditions*. The syntax for expression shall be:

Expression	=	Unconditionalexpression
		Conditionalexpression
Unconditionalexpression	=	Simpleexpression
		String

NOTE Requirements for strings are specified in **6.9.4**.

6.5.2 *Simple expressions*. Arithmetic shall be performed with the monadic and dyadic adding operators + and −, and with the dyadic multiplying operators * (multiply) and / (divide). The plus and minus operators shall join *terms*. The multiplication and division operators shall join *factors* to form terms. There shall be no exponentiation operator. The syntax for simple expression shall be:

Simpleexpression	=	Term
		Addoperator Term
		Simpleexpression Addoperator Term
Term	=	Factor
		Term Multoperator Factor
Addoperator	=	+
		−
Multoperator	=	*
		/

6.5.3 ***Primaries***

6.5.3.1 *General*. Primaries shall be the basic operands in expressions, e.g. in the analysis of the expression

x + y * (a + b) – 4

,there are 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 shall be similarly analysed until no further reduction is possible and no expression brackets remain. When an expression contains no word-logical operators (see **6.5.4**), a factor shall be a primary, whether or not of a defined type. Thus, the syntax shall be:

Factor	=	Primary
		Booleanword
Primary	=	Untypedprimary
		Typedprimary

6.5.3.2 *Untyped primaries*. Untyped primaries shall be those operands that cannot be classed as integer, floating-point or fixed-point (of known scale) without reference to their context, e.g. the number 3.1416 can be represented, with varying degrees of accuracy, in many different ways within a computer word. The same applies to an expression, whose type is determined by context (see **6.5.5**). The syntax shall be:

Untypedprimary	=	Real
		(Expression)

A “real” (see **6.9.2**) shall be an unsigned numerical constant containing a decimal, octal, or hexadecimal point, or a tens exponent, or a decimal point and a tens exponent.

6.5.3.3 *Typed primaries*. Typed primaries shall be classified as follows:

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

6.5.3.4 *Word references*. A simple reference, or a reference to an array element or whole-word table-element, shall have 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*, shall take the form

[Index]

, where the index is any expression evaluated as an integer to give the actual location of a computer word. An anonymous reference shall possess all the properties of an identified reference, except that it shall lack 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 shall refer to the contents of the address defined by Index. Such contents shall be taken to be of numeric type INTEGER, irrespective of any declaration associating that word with some other type.
See also **6.5.3.6**. The syntax for word reference shall be:

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

6.5.3.5 *Part-words*. Any single item of packed data may act as a typed primary. Such an item shall be *either:*

a) a reference to a part-word table-element; *or*

b) a specified field of any typed primary.

In a), the type shall be defined in the table declaration. In b), the desired field shall be selected by a prefix of the form

BITS [Totalbits, Bitposition]

in front of the typed primary to be operated upon. The result of this operation shall be a positive integer value of width Totalbits and in units of the bit at "Bitposition". Totalbits shall not be set equal to the full word length. The value will in general be implementation-dependent, even though the operand shall be typed, as no requirements are laid down for the internal representation of data. In all cases, however, the numeric type resulting from the application of BITS shall be INTEGER. The syntax for a part-word, which should be distinguished from that of a "part-word reference" (see **6.6.2**), shall be:

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

6.5.3.6 *Locations*. The computer location of any word reference shall be obtainable by the location operator, which shall be written in the form

LOCATION (Wordreference)

, and shall have a value of type INTEGER.

NOTE 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.

6.5.3.7 *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 specified in **6.5.5**.

6.5.3.8 *Functions*. The call of a typed procedure (see **6.7**) may be treated as a function and used as a primary in any expression. (For the syntax of a procedure call, see **6.6.4**.)

6.5.3.9 *Integers*. An integer used in any expression (see **6.9.2**) shall be assumed to have the numeric type INTEGER before any necessary type-changes are enforced by context.

6.5.4 *Word-logic*. Three dyadic logical operators are defined and shall be used between typed primaries. The effect of these operators is implementation-dependent to the extent that the word-representation of data is not specified by this standard. The i th bit of the result shall be a given logical function of the i th bits of the two operands, and the result as a whole shall be a typed primary of numeric type INTEGER. To avoid confusion with Boolean operators in "conditions" (see **6.5.6.2**), a different terminology is used. The operators shall be:

DIFFER

	0	1
0	0	1
1	1	0

UNION

	0	1
0	0	1
1	1	1

MASK

	0	1
0	0	0
1	0	1

DIFFER is recognizable as "not equivalent", UNION as "inclusive or" and MASK as "and". The operators are specified above in order of increasing tightness of binding. As bracketed expressions are untyped, the use of brackets to overcome binding priorities necessitates explicit integer scaling, e.g.:

a MASK INTEGER (b UNION c)

The syntax, continued from **6.5.3.1**, shall be:

Booleanword = Booleanword2
Booleanword4 DIFFER Booleanword5

Booleanword2 = Booleanword3
Booleanword5 UNION Booleanword6

Booleanword3 = Booleanword6 MASK Typedprimary

Booleanword4 = Booleanword
Typedprimary

Booleanword5 = Booleanword2
Typedprimary

Booleanword6 = Booleanword3
Typedprimary

6.5.5 *Evaluation of expressions*. Expressions are used in assignment statements, as value parameters of procedures and as integer indexes, all of which contexts shall determine the numeric type finally required. CORAL 66 expressions shall be automatically evaluated to this type, but in the process of calculation, data may be subjected by the compiler to various intermediate transformations. This standard does not specify an algorithm for the evaluation of expressions, nor does an algorithm to determine a particular method of rounding form part of the language specification (for explanation of the term "rounding" see BS 3527-2). In particular, this standard does not specify the results of arithmetic operations that overflow or underflow. However, any rounding algorithm adopted shall behave in a consistent manner, and all syntactically outermost terms in an expression shall be evaluated to the required numeric type before the adding operators are applied. In the simplest cases, this requirement ensures predictable results, although rounding-off errors are not minimal and overflow may occur. If an expression is enclosed in round brackets, its terms are not "outermost", the requirement no longer applies, and the algorithm for the particular compiler shall determine the sequence of events. The programmer may impose any desired system of evaluation by the use of Numbertype (Expression), which is a typed primary (see **6.5.3.3**), and any occurrence of which behaves like a variable, e.g. ref, declared as

Numbertype ref;

and assigned a value by

ref ← Expression

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

x ← i – j

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

x ← INTEGER (i – j)

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

6.5.6 *Conditional expressions*

6.5.6.1 *General.* The syntax for a conditional expression shall be:

Conditionalexpression	=	IF Condition THEN Expression ELSE Expression

The expressions following THEN and ELSE are known as the *consequent expression* and the *alternative expression* respectively. The value of a conditional expression shall be the value of the consequent expression if the condition is true (see **6.5.6.2**); it shall be the value of the alternative expression if the condition is false.

The numeric type used to evaluate the condition shall have no effect on the evaluation of the consequent or alternative expressions. Consequent and alternative expressions shall not be prevented from being regarded as syntactically outermost by their appearance in a conditional expression.

6.5.6.2 *Conditions.* A condition shall consist of one or more arithmetic comparisons. Comparisons shall be connected by the Boolean operators OR and AND, of which AND shall be the more tightly binding. The permissible arithmetic comparators shall be “less than”, “less than or equal to”, “equal to”, “greater than or equal to”, “greater than”, and “not equal to”. The syntax shall be:

Condition	=	Condition OR Subcondition Subcondition
Subcondition	=	Subcondition AND Comparison Comparison
Comparison	=	Simpleexpression Comparator Simpleexpression
Comparator	=	< $\leqslant$ = $\geqslant$ > $\neq$

The Boolean operators shall have their usual meanings, the OR being inclusive. Conditions and subconditions shall be evaluated from left to right only as far as is necessary to determine truth or falsity. Comparisons shall be evaluated in the order in which they appear when a condition is read from left to right.

6.6 Statements

6.6.1 *General.* The syntax shall be:

Statement	=	Label : Statement Simplestatement Conditionalstatement Forstatement
Simplestatement	=	Assignmentstatement Gotostatement Procedurecall Answerstatement Codestatement Compoundstatement Block Dummystatement

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

6.6.2 *Assignments.* The left-hand side of an assignment statement shall always be a data reference, and the right-hand side shall be an expression for procuring a numerical value. The location of the left-hand side shall be evaluated prior to the right-hand side expression; the result of assignment shall be 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 **6.3.7**). The expression on the right-hand side shall be evaluated to the numeric type of the reference, with automatic scaling and rounding as necessary. Functions occurring in an assignment statement shall be evaluated in the order in which they are met when reading the text from left to right (subject to compliance also with the requirements of **6.5.5**). The left-hand side may be a word reference as specified in **6.5.3.4** or 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 shall remain unaltered. As examples of assignment,

```
INTEGER i;
i ← 4
```

has the effect of placing the integer 4 in the location 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 that 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 shall be:

Assignmentstatement	=	Variable ← Expression
Variable	=	Wordreference
		Partwordreference
Partwordreference	=	Id[Index]
		BITS [Totalbits, Bitposition] Wordreference

There shall be no form of multiple assignment statement.

6.6.3 *Goto statements.* The goto statement shall cause the next statement for execution to be the one having a given label. The label may be written explicitly after GOTO, or referenced by means of a switch whose index shall lie within the range 1 to n, where n is the number of labels specified in the switch declaration (see also **6.2.4** and **6.4**). The syntax shall be:

Gotostatement	=	GOTO Destination
Destination	=	Label
		Switch [Index]

6.6.4 *Procedure calls.* A procedure identifier, followed in parentheses by a list of actual parameters (if any), shall constitute 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. The call of the procedure shall cause:

a) the formal parameters in the procedure declaration to be replaced by the actuals in a manner that shall depend on the formal parameter specifications (see **6.7.4**); the replacement shall be effected in the order in which the parameters are read when reading from left to right; *and*

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

The syntax for a procedure call shall be:

Procedurecall	=	Id Id (Actuallist)
Actuallist	=	Actual Actual, Actuallist
Actual	=	Expression Wordreference Destination Name
Name	=	Id

NOTE The purpose of the four types of actual parameter is described in **6.7.4**.

6.6.5 *Answer statements.* An answer statement shall be used only within a typed procedure body, and shall be the means by which a value is given to the procedure. It shall cause the expression in the answer statement to be evaluated to the numeric type of the procedure, followed by immediate exit from the procedure body. The syntax shall be:

Answerstatement	=	ANSWER Expression

6.6.6 *Code statements.* Any sequence of code instructions enclosed by CODE BEGIN and END may be used as a CORAL 66 statement. Code statements should provide for the inclusion of nested CORAL text. The form of the code is not specified; it may be the assembly code for a particular computer, or it may be at a higher level enabling available compiler features to be exploited. The code should, above all, enable the CORAL programmer to exploit all available hardware facilities of the computer. For communication between code and other statements, it shall be possible to use any identifier of the program within the code statement, provided such identifiers are in scope.

NOTE 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, but this standard specifies no corresponding requirement regarding a CORAL 66 compiler. This type of extension to other forms of statement does not comply with the requirements of this standard.

The syntax for a code statement shall be:

Codestatement	=	CODE BEGIN Codesequence END

,where Codesequence is as defined in each particular implementation.

6.6.7 *Compound statements.* A compound statement shall be a sequence of statements grouped to form a single statement, for use where the syntactic structure of the language demands. A compound statement shall be transparent to scopes and a goto statement may therefore refer to a label that is set inside a compound statement. The syntax shall be:

Compoundstatement	=	BEGIN Statementlist END
Statementlist	=	Statement Statement; Statementlist

6.6.8 *Blocks.* See **6.2**.

6.6.9 *Dummy statements.* A dummy statement shall be a void whose execution has no effect, e.g., a dummy statement follows the colon in

; label : END

This syntax shall be:

Dummystatement	=	Void

6.6.10 *Conditional statements*. The syntax of the conditional statement shall be:

Conditionalstatement	=	IF Condition THEN Consequence
		IF Condition THEN Consequence ELSE Alternative
Consequence	=	Simplestatement
		Label : Consequence
Alternative	=	Statement

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

6.6.11 ***For-statements***

6.6.11.1 *General*. The for-statement shall comprise a means of repeatedly executing a given statement, the "controlled statement", for different values of a chosen variable, the "control variable", which may (or may not) occur within the controlled statement. The implementation shall define the effect of jumps into the controlled statement, and the consequent state of the control variable. One 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 third 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 shall be used. The syntax shall be:

Forstatement	=	FOR Wordreference ← Forlist DO Statement
Forlist	=	Forelement
		Forelement, Forlist
Forelement	=	Expression
		Expression WHILE Condition
		Expression STEP Expression UNTIL Expression

The control variable shall be a word reference, i.e. either an anonymous reference or a declared word reference. The location of the control variable shall be evaluated once only, prior to the evaluation of the for-list.

6.6.11.2 *For-elements with STEP*. Let the element be denoted by:

e1 STEP e2 UNTIL e3

The expressions shall be evaluated once only. First, their values shall be evaluated in the order in which they are met when reading from left to right. Let these values be denoted by v1, v2 and v3 respectively. Then, in sequence:

a) v1 shall be assigned to the control variable;

b) v1 shall be compared with v3; if (v1 – v3) * v2 > 0, the for-element shall be exhausted; *otherwise*

c) the controlled statement shall be executed;

d) the value v1 shall be set from the control variable, then incremented by v2 and the cycle shall be repeated from a).

6.6.11.3 *For-elements with WHILE*. Let the element be denoted by:

e1 WHILE Condition

Then, in sequence:

a) e1 shall be evaluated and assigned to the control variable;

b) the condition shall be tested; if false, the for-element shall be exhausted; *otherwise*

c) the controlled statement shall be executed and the cycle repeated from a).

Unlike the expressions considered in **6.6.11.2**, the expression e1 and those occurring in the condition shall be evaluated repeatedly.

6.7 Procedures

6.7.1 *General*. A procedure shall be a body of program, written out once only, named with an identifier, and available for execution anywhere within the scope of the identifier. There shall be three possible methods of communication between a procedure and its program environment, as follows.

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 shall be replaced by *actual* parameters, in one-to-one correspondence.

b) The body may use non-local identifiers whose scopes embrace the body. Such identifiers shall be accessible outside the procedure.

c) The body may include an answer statement which shall compute a single value for the procedure, making its call suitable for use as a function in an expression. A procedure that possesses a value shall be known as a *typed procedure*.

The syntax for a procedure declaration shall be:

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

The second of the foregoing syntax alternatives shall be the form of declaration used for recursive procedures. If a procedure is defined in a manner that directly or indirectly calls on itself at run-time, that procedure is said to be "recursive" and shall be explicitly declared as such. The statement following the procedure heading shall be the procedure body, which shall contain an answer statement (see **6.6.5**) unless the answer specification is void (see **6.7.2**), and which shall be treated as a block whether or not it includes any local declarations (see **6.7.5**).

6.7.2 *Answer specification*. The value of a typed procedure shall be given by one or more answer statements (see **6.6.5**) in its body, and its numeric type shall be specified at the front of the procedure declaration. An untyped procedure shall have no answer statement, shall possess no value, and shall have no answer specification in front of the word PROCEDURE or RECURSIVE. The syntax shall be:

Answerspec	=	Numbertype
		Void

6.7.3 *Procedure heading*. The procedure heading shall give the procedure its name. It shall also describe and list any identifiers used as formal parameters in the body. On a call of the procedure, the compiler shall set up a correspondence between the actual parameters in the call and the formal parameters specified in the procedure heading. The syntax of the heading shall be:

Procedureheading	=	Id
		Id (Parameterspeclist)
Parameterspeclist	=	Parameterspec
		Parameterspec; Parameterspeclist

6.7.4 *Parameter specification*

6.7.4.1 *General*. Any object in scope at the position of a procedure call may be passed to that procedure by means of a parameter, whether it is an object of data, a place in the program, or a procedure to be executed. For data, there shall be two distinct levels of communication; *numerical values* (for input to the procedure) and *data references* (for input or output). Table 2 specifies all the types of object that may be passed, the syntactic form of specification, and the corresponding form of the actual parameter that shall be supplied in the procedure call. The syntax shall be:

Parameterspec	=	Specifier Idlist
		Tablespec
		Procedurespec
Specifier	=	VALUE Numbertype
		LOCATION Numbertype
		Numbertype ARRAY
		LABEL
		SWITCH

Table 2 — Parameters of procedures

Object	Formal specification	Actual parameter
Numerical value	VALUE Numbertype Id[a]	Expression
Location of data word	LOCATION Numbertype Id[a]	Wordreference
Name of array	Numbertype ARRAY Id[a]	Id
Name of table	Tablespec[b]	Id
Place in program	LABEL Id[a]	Destination
Name of switch	SWITCH Id[a]	Id
Name of procedure	Procedurespec[c]	Id

[a] Composite specification of similar parameters has Idlist in place of Id.
[b] See **6.7.4.6**.
[c] See **6.7.4.9**.

6.7.4.2 *Value parameters*. The formal parameter shall be treated as though declared in the procedure body; upon entry to the procedure, the actual expression shall be evaluated to the type specified (including scaling if the numeric type is FIXED), and the value shall be forthwith assigned to the formal parameter.

The formal parameter may be used subsequently for working space in the body; if the actual parameter is a variable, its value shall be unaffected by assignments to the formal parameter.

6.7.4.3 *Data reference parameters*. Location, array and table parameters are all examples of data references. Upon entry to the procedure, these formals shall be 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 shall therefore be operations on the actual parameters. For example, the values of the actual parameters may be altered by assignments within the procedure.

6.7.4.4 *Word location parameters*. The actual parameter shall be a word reference, i.e. a simple data reference, an array element, an indexed table identifier, a whole-word table-element or an anonymous reference. Index expressions shall be evaluated upon entry to the procedure as part of the process of obtaining the location of the actual parameter. The numeric type of the actual parameter shall agree exactly with the formal specification. Part-word references, such as table-elements, shall not be used 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(5 * i + 2, n [i])

6.7.4.5 *Array parameters*. As in an array declaration, the specified numeric type shall apply to all the elements of the array named. The numeric type of the actual array name shall agree with this formal specification. By indexing within the body, the procedure may refer to any element of the actual array.

6.7.4.6 *Table parameters*. The specification of a table parameter shall be identical in form to a table declaration except that presetting shall not be permitted. The syntax shall be:

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

The element declaration list shall include such fields as are used in the procedure body. Unused fields may be omitted.

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

6.7.4.8 *Place parameters: switch parameters*. The actual parameter shall be a switch identifier. By indexing within the procedure body, the procedure shall be able to refer to any of the individual labels that form the elements of the switch.

6.7.4.9 *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 shall be formally specified in the heading of the procedure within which it is called. For 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 (underlined for clarity) might be

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

A typical call of q would be q(lab, g). At the inner level of parameter specification, no formal identifiers shall be required, no composite specifications shall be permitted (as for i and j in g) and the specifications shall be 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 underlined in the last example, no further parameter specifications shall be required.

The syntax for a procedure specification shall be:

Procedurespec	=	Answerspec PROCEDURE Procparamlist
Procparamlist	=	Procparameter
		Procparameter, Procparamlist
Procparameter	=	Id
		Id (Typelist)
Typelist	=	Type
		Type, Typelist
Type	=	Specifier
		TABLE
		Answerspec PROCEDURE

6.7.4.10 *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 **6.7.4.1**) may be used:

Parameterspec	=	VALUE Formalpairlist
		LOCATION Formalpairlist
		Specifier Idlist
		Tablespec
		Procedurespec
Formalpairlist	=	Formalpair
		Formalpair, Formalpairlist
Formalpair	=	Id : Id

At the call of the procedure, each formal pair shall correspond to a single actual parameter. The first identifier shall be 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 shall arrange 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) shall take account of this known scale.

6.7.5 *The procedure body*. For purposes of scoping, a procedure declaration shall be regarded as a block at the place where it appears on the program sheet. Everything except the body may be disregarded, and the formal parameters may be treated as though declared within the body, labels included. Identifiers that are non-local to the procedure body shall be those in scope at the place of the procedure declaration, subject to the requirements specified in **6.2.5**. Actual parameters shall 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.

6.8 Communicators

6.8.1 *General*. The segments of a program may communicate with each other through COMMON (see **6.8.2**), and with objects external to the program by means of the communicators such as LIBRARY, EXTERNAL or ABSOLUTE, as defined in particular implementations.

6.8.2 *COMMON communicators*. Global objects declared within a program (see **6.2.3**) shall be communicated to all segments through a COMMON communicator. This shall consist of a list of COMMON items separated by semicolons all within round brackets following the word COMMON. Such items shall be of three kinds, corresponding to the division of objects into data, places and procedures. A COMMON data item shall be a declaration of the identifiers listed within it, exactly as specified in **6.3**, storage being allocated as specified in **6.3.5**, presets and overlays as specified in **6.3.6** and **6.3.8**. Communication of places and procedures shall take the form of *specification*, as in the equivalent parameters of a procedure declaration (see **6.7.4.7** and **6.7.4.9**). For each identifier specified in a COMMON communicator, there shall correspond an appropriate declaration (or for labels, a setting) in one and only one outermost block of the program. The syntax shall be:

```
Commoncommunicator  =  COMMON (Commonitemlist)
Commonitemlist      =  Commonitem
                       Commonitem; Commonitemlist
Commonitem          =  Datadec
                       Overlaydec
                       Placespec
                       Procedurespec
                       Void
Placespec           =  LABEL Idlist
                       SWITCH Idlist
```

6.8.3 *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 shall begin with the word LIBRARY and shall be styled to conform to the rest of the language.

The syntax shall be:

Librarycommunicator = defined in a particular implementation to conform to the style of the commoncommunicator

NOTE The relative importance attached to COMMON and LIBRARY as means of inter-segment communication borders on questions of implementation that fall outside the scope of this standard.

6.8.4 *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 shall be permitted through the use of an EXTERNAL communicator similar in form to a COMMON communicator.

The syntax shall be:

Externalcommunicator = defined in a particular implementation to conform to the style of the commoncommunicator

6.8.5 *ABSOLUTE communicators*. CORAL 66 programs may refer to objects having absolute addresses in the computer by the use of ABSOLUTE communicators that 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 should take the syntactic form Id/Integer.

The syntax shall be:

Absolutecommunicator = defined in a particular implementation to conform to the style of the commoncommunicator

6.9 Names and constants

6.9.1 *Identifiers*. Identifiers shall be used for naming objects of data, labels and switches, procedures, macros and their formal parameters. An identifier shall consist of an arbitrary sequence of lower case letters and digits, starting with a letter. It shall carry no information in its form, e.g. single-letter identifiers shall not be reserved for special purposes. It may be of any length, though compilers may disregard all but the first 12 printing characters. As layout characters are ignored, spaces may be used in identifiers without acting as terminators.

The syntax shall be:

Id	=	Letter Letterdigitstring
Letterdigitstring	=	Letter Letterdigitstring Digit Letterdigitstring Void
Letter	=	a *or* b *or* c *or* d *or* e *or* f *or* g *or* h *or* i *or* j *or* k *or* l *or* m *or* n *or* o *or* p *or* q *or* r *or* s *or* t *or* u *or* v *or* w *or* x *or* y *or* z
Digit	=	0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7 *or* 8 *or* 9

6.9.2 *Numbers*. Numerical constants specified elsewhere in this specification are of the following types:

a) *constants* for presetting, optionally signed;

b) *integers* and *reals* as primaries in expressions (a sign attached to a primary shall belong syntactically to the expression and not to the number);

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

The syntax shall be:

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

The further expansions shall be:

Digitlist	=	Digit
		Digit Digitlist
Hexlist	=	Hexdigit
		Hexdigit Hexlist
Octallist	=	Octaldigit
		Octaldigit Octallist
Hexdigit	=	Digit
		A *or* B *or* C *or* D *or* E *or* F
Octaldigit	=	0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7

6.9.3 *Literal constants*. A printing character shall be 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, e.g.

LITERAL(a)

has an integer value uniquely representative of “a”. The form is included within the syntax of integer (see **6.9.2**). The printing characters shall be implementation-dependent, but the set shall be assumed to include one 26-letter alphabet and a set of 10 digits (see **6.11**). Layout characters shall not occur as arguments of literal.

6.9.4 *Strings*. A string shall be 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 shall be properly paired within the string (but see **6.11**). A string shall be classed as an unconditional expression (see **6.5**), and its value shall be its location, but it shall not be used as a LOCATION parameter. This standard does not specify further the nature of a string, but an implementation shall specify the stored format of strings and shall state whether multiple copies of equivalent strings are generated.

Procedures capable of selecting individual characters from a string should be so designed that characters are represented by the same integer values as are defined for literal constants.

The syntax shall be

String = ⊰ sequence of characters with quotes matched ⊱

6.10 Processing text in a program

6.10.1 *Comment*

6.10.1.1 *General*. A program may be annotated by the insertion of textual matter and this comment shall be ignored by the compiler.

6.10.1.2 *Comment sentences*. A comment sentence may be written within a segment wherever a declaration or statement may appear, or within a communicator wherever a communicator item may appear. It shall consist of the word COMMENT followed by text and terminated by a semicolon. For obvious reasons, the text shall not contain a semicolon. The entire comment sentence shall be ignored by the compiler.

6.10.1.3 *Bracketed comment*. Bracketed comment shall comprise any textual matter enclosed within round brackets immediately after a semicolon of the program. The text may contain brackets provided that they are matched. Bracketed comment (including the brackets) shall be ignored by the compiler.

6.10.1.4 *END comment*. Annotation may be inserted after the word END provided that it takes the form of an identifier only. The "identifier" shall be ignored by the compiler.

6.10.2 *Macro facility*

6.10.2.1 *General*. A CORAL 66 compiler shall embody a macro processor, which may be regarded as a self-contained routine which processes the text of a CORAL program before passing it to the compiler proper. Its function shall be to enable the programmer to define and use convenient macro names, in the form of identifiers, to stand in place of cumbersome or obscure portions of text, typically code statements. Once a macro name has been defined, the processor shall expand it in accordance with the definition wherever it is subsequently used, until the definition is altered or cancelled (see **6.10.2.5**). However, the macro processor shall treat comments, constant character strings and the representations of numbers as indivisible entities, and shall not expand any objects with the form of identifiers within these entities. No character that could form part of an identifier shall be written adjacent to the use of a macro name or formal parameter of a macro, as this would inhibit the recognition of such names. A macro definition may be written into the source program wherever a declaration or a statement may appear, and shall be removed from it by the action of the macro processor.

6.10.2.2 *String replacement*. In the simplest use, a macro name shall stand 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, shall be any sequence of characters in which string quotes are matched (but see **6.11**).

NOTE Care needs to be taken to include brackets, such as BEGIN and END, as part of the macro body whenever there is the possibility that the context of the expansion may demand them.

6.10.2.3 *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 **6.10.2.2**.

A formal parameter, such as n above, shall be written as an identifier. An actual parameter (e.g. 6) shall be 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 requirement enables commas to be used for separating actual parameters. The number of actual parameters shall be the same as the number of formals, which shall also be separated by commas.

6.10.2.4 *Nesting of macros*. A macro definition may embody definitions or uses of other macros. When a macro is defined, the body shall be kept but not expanded. When the macro is used, it shall be as though the body were substituted into the program text, and it shall be during this substitution that any other macros encountered are processed. The use of a macro with parameters shall be regarded as introducing virtual macro definitions for the formal parameters before the macro body is substituted. Thus, to continue the example from **6.10.2.3**, the occurrence of shift(6) would be 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 shall not be defined as a macro name. A formal parameter shall 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, shall be the same as any formal parameter of the calling macro.

6.10.2.5 *Deletion and redefinition of macros*. The scope of a macro definition shall be from the point of definition until either the end of the program text is reached or the macro name is redefined or deleted. The scope of a macro shall be independent of the block structure of the program. To delete a macro, the construct

DELETE Macroname;

shall be used wherever the requirements of this standard allow a declaration or statement to appear. The construct shall be removed by the action of the macro processor. Alternatively, a macro name may be redefined. Macro definitions that have the same name shall be stacked and so processed that the most recent shall be the one that applies when the name is used. If a redefined macro is deleted, the most recent definition shall be deleted, and the previous one shall be reinstated.

NOTE "Recent" and "previous" refer to the sequence as processed by the macro processor.

6.10.3 *Syntax of comment and macros*. The syntax shall be:

Commentsentence	=	COMMENT any sequence of characters not including a semicolon;
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 Macrostring, Macrostringlist
Macrostring	=	any sequence of characters in which string quotes are matched, commas are protected by round or square brackets and in which such brackets are properly matched and nested

6.11 List of language symbols. The language symbols shall consist of the language words as specified in Table 3 and other symbols as specified in Table 4.

Table 3 — Language words

Language word	Reference	Language word	Reference	Language word	Reference
ABSOLUTE	**6.8.5**	END	**6.2.1** and **6.6.7**	OR	**6.5.6.2**
AND	**6.5.6.2**	EXTERNAL	**6.8.4**	OVERLAY	**6.3.8**
ANSWER	**6.6.5**	FINISH	**6.1.3**	PRESET	**6.3.6.3**
ARRAY	**6.3.3**	FIXED	**6.3.1**	PROCEDURE	**6.7.1**
BEGIN	**6.2.1** and **6.6.7**	FLOATING	**6.3.1**	RECURSIVE	**6.7.1**
BIT	**6.3.4.3.1**	FOR	**6.6.11**	SEGMENT	**6.1.3**
BITS	**6.5.3.5**	GOTO	**6.6.3**	STEP	**6.6.11.2**
CODE	**6.6.6**	HEX	**6.9.2**	SWITCH	**6.4**, **6.7.4.1** and **6.8.2**
COMMENT	**6.10.1.2**	IF	**6.5.6.1** and **6.6.10**	TABLE	**6.3.4.2**
COMMON	**6.8.2**	INTEGER	**6.3.1**	THEN	**6.5.6.1** and **6.6.10**
CORAL	**6.1.3**	LABEL	**6.7.4.1** and **6.8.2**	UNION	**6.5.4**
DEFINE	**6.10.2.2**	LIBRARY	**6.8.3**	UNSIGNED	**6.3.4.3.3**
DELETE	**6.10.2.5**	LITERAL	**6.9.3**	UNTIL	**6.6.11.2**
DIFFER	**6.5.4**	LOCATION	**6.5.3.3** and **6.7.4.1**	VALUE	**6.7.4**
DO	**6.6.11**	MASK	**6.5.4**	WHILE	**6.6.11.3**
ELSE	**6.5.6.1** and **6.6.10**	OCTAL	**6.9.2**	WITH	**6.3.8**

Table 4 — Other symbols

Symbol	Description and reference
0 1 2 3 4 5 6 7 8 9	digits, **6.9.1**
abcdefghijklmnopqrstuvwxyz	letters, **6.9.1**
+ –	adding operators, **6.5.2**
* /	multiplying operators, **6.5.2**
< ⩽ = ⩾ > ≠	comparators, **6.5.6.2**
( )	expression brackets, etc.
[ ]	index brackets, etc.
⟊ ⟋	string quotes, **6.9.4** and **6.10.2**
, ;	separators for lists
:	separator for bounds, **6.3.3**
	terminator for label setting, **6.6.1**
←	assignment symbol, **6.3.6.2**, **6.4** and **6.6.2**
.	point, **6.9.2**
10	"times ten to the power of", **6.9.2**

NOTE The tabulated symbols are too numerous for representation by single characters on most printing peripheral equipment. It is recommended that, where necessary, the two alphabets should be distinguished by enclosure of language words between apostrophes or acute accents (position 2/7; see BS 4730:1974), colloquially known as primes, and that the following character representations should be adopted:

Specified symbol	Representation
⩽	< =
⩾	> =
≠	< >
←	: =
⟊	”
⟋	”
10	E or e

The use of the quote-character (”) is a desirable representation, but as this makes the opening and closing symbols indistinguishable, it is recommended that the ALGOL 68 system should be adopted[2)], in which a quote-symbol within a string is represented by a pair of quote-characters (“”)

If the character e or the character E is used in place of $_{10}$, the alternative $_{10}$ Signedinteger in the expansion of the syntax of Real should be omitted (see **6.9.2**).

6.12 Permissible options

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

Major features that may be omitted are:

a) RECURSIVE procedures;

b) TABLE facilities;

c) either FIXED or FLOATING point numbers;

d) OVERLAY of data;

e) BITS;

f) the set of dyadic logical operators DIFFER, UNION, and MASK;

g) up to and including three of the set of communicators COMMON, LIBRARY, EXTERNAL and ABSOLUTE, provided that a mechanism for segmented compilation is provided.

2) *Numerische Mathematik*, **14**, (1969) pp. 79 – 218, paragraph 5.1.4.

In addition, this standard (see note to **6.6.6**) optionally allows code statements to be said to possess a value and (see **6.7.4.10**) allows for the optional omission of the non-standard procedure parameter features.

6.12.2 Any, all, or none of the optional features listed in **6.12.1** may be omitted. Omissions shall be clearly indicated in any description of the implementation.

NOTE A full CORAL 66 compiler would handle all these features, but a compiler for an object machine lacking floating-point hardware would not normally be expected to handle the FLOATING type of number.

Appendix A Unspecified features

A.1 This British Standard specifies CORAL 66 as a kernel high level language that deliberately does not define operations that might favour implementation on one type of computer architecture to the detriment of others. Thus only those features of the language that are specified in this standard should be considered as formally defined. Since the absence of anything is generally difficult to identify, this appendix is included in this British Standard to aid the reader in the more rapid understanding of all known aspects of CORAL 66. This appendix should only be considered as a guide to the greater understanding of the undefined aspects of the defining text and it does not form part of the specification requirements. Any description of a CORAL 66 implementation should clearly define the implementation adopted for the unspecified features listed in **A.2**.

A.2 The following features, given with their appropriate references to clause **6** in brackets, are left unspecified by this standard:

a) the results of division (**6.5.5**);

b) the results of arithmetic operations that produce overflow or underflow (**6.5.5**);

c) the representation and consequent range of values for integer, fixed- or floating-point types (**6.3.1**, and see also Appendix B);

d) the effect of overlaying preset data on to data and the effect of overlaying preset data by non-preset data (**6.3.8**).

Appendix B 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 (with the exception of labels) are fully declared or specified before use simplifies the compiler by ensuring that all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analysers with no back-tracking to be employed. Features that require elaborate hardware in the object machine for efficient program execution, e.g. 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 or to check the dimensionality of arrays. No run-time checking of procedure entries is necessary. Certain features of the language have been left undefined; these include the effect of overlaying preset data on to or by non-preset data, and the resolution of conflicting preset values. Furthermore, this standard does not take full account of side effects and these should only be exploited with care. An underlying concept of this standard is a linearly addressable store of CORAL words (a CORAL word being defined by the length of an object of type INTEGER), and the discussion of the LOCATION operator and the anonymous reference facility is bounded by this concept. Suitable extensions, i.e. features additional to those specified in clause **6**, may have to be made to the definition to accommodate alternative store architectures (see also **5.1**). However, the fundamental relationship between LOCATION and anonymous reference should be maintained (see **6.3.5**, **6.5.3.4** and **6.5.3.6**). 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 this standard.

Examples, included in this standard for the purpose of illustration only, assume a twos complement representation of data, but such examples are not intended to add to, or to detract from, the requirements of clause **6**.

NOTE The Royal Signals and Radar Establishment, Malvern, (RSRE) has acted as a language authority for CORAL 66 since the publication of the "Official definition" in 1970. RSRE has developed a formal compiler assessment procedure that includes a set of test programs. This procedure has been applied to a large number of implementations and has proved to be sufficient in the past to produce a uniformity of implementation across a wide range of computer architectures. RSRE has stated that its assessment procedure will continue to be applied to new implementations in the foreseeable future. Thus users have available an independent assurance of the quality of a particular implementation and implementors have available a facility that helps to determine the compliance of their compilers with the requirements of this standard.

blank

Publications referred to

BS 3527, *Glossary of terms used in data processing.*
BS 4730, *The United Kingdom 7-bit data code (ISO-7-UK).*

BSI — British Standards Institution

BSI is the independent national body responsible for preparing British Standards. It presents the UK view on standards in Europe and at the international level. It is incorporated by Royal Charter.

Revisions

British Standards are updated by amendment or revision. Users of British Standards should make sure that they possess the latest amendments or editions.

It is the constant aim of BSI to improve the quality of our products and services. We would be grateful if anyone finding an inaccuracy or ambiguity while using this British Standard would inform the Secretary of the technical committee responsible, the identity of which can be found on the inside front cover. Tel: 020 8996 9000. Fax: 020 8996 7400.

BSI offers members an individual updating service called PLUS which ensures that subscribers automatically receive the latest editions of standards.

Buying standards

Orders for all BSI, international and foreign standards publications should be addressed to Customer Services. Tel: 020 8996 9001. Fax: 020 8996 7001.

In response to orders for international standards, it is BSI policy to supply the BSI implementation of those that have been published as British Standards, unless otherwise requested.

Information on standards

BSI provides a wide range of information on national, European and international standards through its Library and its Technical Help to Exporters Service. Various BSI electronic information services are also available which give details on all its products and services. Contact the Information Centre. Tel: 020 8996 7111. Fax: 020 8996 7048.

Subscribing members of BSI are kept up to date with standards developments and receive substantial discounts on the purchase price of standards. For details of these and other benefits contact Membership Administration. Tel: 020 8996 7002. Fax: 020 8996 7001.

Copyright

BSI
389 Chiswick High Road
London
W4 4AL