

The Programming Language DIOS:

A Case Study in Programming Language Design

DEAN WAKERLEY, BSc(Hons)(Open) MIAP MACM MIEEE

Dissertation submitted in partial fulfilment of the

Requirements for the degree of

Master of Science in Software Engineering

DEPARTMENT OF COMPUTING

UNIVERSITY OF BRADFORD

2003/4

Supervisor: Dr. D.R.W. Holton

ABSTRACT

Master of Science in Software Engineering 2003/4

The Programming Language DIOS: A Case Study in Programming
Language Design

DEAN WAKERLEY, BSc(Hons)(Open) MIAP MACM MIEEE

Supervisor: Dr. D.R.W. Holton

A project to design a high-order programming language with a Pascal, Modula-2, Ada flavour with realisation facilities of Ada but simplifying the language syntax and implementation.

The project involved examining the science and theoretical aspects of programming languages, their implementation and use. Individual language constructs were examined from many programming languages with emphasis on Pascal, Modula-2, Ada 83/95, C, C++, Java and Perl. From this analysis a programming language was designed by selecting and improving individual language constructs into a whole language.

The language is block structured like Modula-2 and Ada. The language provides facilities for modular design of small to very large programs. The data types provided are integer, character, enumeration, three real types and aggregates (array, record and union). There are no limits to the range of discrete types. There are extensive realisation facilities for data types. Expressions may include tuples. There are border controls for making the use of exceptions secure. There are generics.

The impetus and aims of the language are given before the complete language is detailed. Recommendations for future work are given. It is assumed that the reader is an expert in programming languages.

Copyright © Dean Wakerley, September 2004

Contents

1.	Introduction	1
2.	Aims of the Language	2
2.1	Name	3
2.2	White Elephants	3
3.	Syntax Notation	4
4.	Morphology	5
4.1	Characters	5
4.1.1	Escape Sequences	5
4.1.2	Illegal Characters	6
4.2	White Space	6
4.2.1	Word Separators	6
4.2.2	Line Separators	6
4.3	Comments	7
4.3.1	Multiline Comments	7
4.3.2	Single-Line Comments	7
4.4	Identifiers	8
4.5	Keywords	8
4.6	Symbols	9
4.6.1	Punctuation	9
4.6.2	Operators	9
4.7	Numbers	10
4.7.1	Denary Natural Literal	10
4.7.2	Decimal Real Literal	11
4.7.3	Based Natural Literal	11
4.7.4	Based Real Literal	12
4.8	Character Literals	12
4.9	String Literals	13
4.10	Analysis	13
5.	Compilation Units	15
5.1	The Main Program	15
5.2	Compilation Unit Header	15
5.2.1	Annexes	16
5.2.2	Revision Control	16
5.3	Analysis	17
6.	Modules	18
6.1	Simple Module	18
6.1.1	Exporting	18
6.2	Combined Module	18
6.3	Split Module	19
6.3.1	Interface	19
6.3.2	Implementation	19
6.4	Module Initialization & Finalisation	20
6.5	Coupling, Import and Visibility	20
6.6	Module Lifetime	21
6.7	Module Exceptions	21
6.8	Analysis	22

7.	Declarations	23
7.1	Blocks, Scopes, Nesting	23
7.2	Type Declarations	23
7.3	Variable Declarations	23
7.3.1	Initialization	24
7.3.2	Attributes	24
7.4	Constant Declarations	24
7.5	Procedure Declarations	24
7.6	Module Declarations	25
7.7	Renaming Declarations	25
7.8	Import & Export Declarations	25
7.9	Entry Visa Declarations	25
7.10	Analysis	25
8.	Type Declarations	27
8.1	Type Definitions	27
8.2	Sub-Types	27
8.3	Distinct Types	27
8.4	Attribute Definitions	28
8.5	Enumeration	28
8.5.1	Enumeration Sub-Type	29
8.5.2	Operations	29
8.5.2.1	Equality and Inequality	29
8.5.2.2	Order Relations	30
8.5.2.3	Membership	30
8.5.3	Attributes	30
8.5.3.1	First, Last and Range Attributes	30
8.5.3.2	Predecessor and Successor Function Attributes	30
8.5.3.3	Cyclic Types and Predecessor and Successor Function Attributes	31
8.5.3.4	Position and Value Function Attributes	31
8.5.3.5	Minimum and Maximum Function Attributes	31
8.6	Boolean	31
8.7	Characters	31
8.7.1	Predefined Character Type	32
8.7.2	Character Literals	32
8.7.3	Character Sub-Type	32
8.7.4	Operations & Attributes	33
8.8	Integers	34
8.8.1	Predefined Integer Type	34
8.8.2	Integer Literals	34
8.8.3	Integer Sub-Type	35
8.8.3.1	Partial Constraints	36
8.8.3.2	Modular or Cyclic Arithmetic	36
8.8.3.3	Realisations	36
8.8.4	Operations	38
8.8.5	Attributes	39
8.9	Real Types	40
8.9.1	Real Literals	40
8.9.2	Predefined Floating-Point Type	40
8.9.2.1	Operations	41

8.9.2.2	Attributes	42
8.9.3	Binary or Secundal Fixed-Point	43
8.9.4	Decimal Fixed-Point	43
8.10	Composites	44
8.10.1	Array	44
8.10.1.1	Multi-dimensional Array	45
8.10.1.2	Strings and Arrays	45
8.10.1.3	Operations	46
8.10.1.4	Attributes	46
8.10.2	Records	46
8.10.2.1	Bit Fields	47
8.10.2.2	Operations	47
8.10.2.3	Field Attributes	47
8.10.3	Variants	48
8.11	Pointers	48
8.12	Procedures (Closures)	49
8.13	Default Values	50
8.14	Type Compatibility	50
8.15	General Type Attributes	50
8.16	Analysis	51
9.	Expressions	52
9.1	Operands	52
9.1.1	Literal Value	52
9.1.2	Function Designator	52
9.1.3	Variable/Constant Designator	52
9.1.4	Attribute Designators	52
9.1.5	Array Element Designators	53
9.1.6	Record & Variant Designators	53
9.1.7	Pointer De-Referencing	53
9.1.8	Type Designator	54
9.1.9	Type Qualification	54
9.1.10	Type Conversion	54
9.2	Operators	54
9.3	Tuple Expressions	57
9.3.1	Named Tuple/Aggregate	57
9.4	Sub-Range Expressions	58
9.5	Analysis	58
10.	Statements	59
10.1	Statement Sequences	59
10.2	Nesting Statements	59
10.2.1	Named Statements	59
10.2.2	Block Statement	59
10.2.3	If Statement	60
10.2.4	Case Statement	61
10.2.5	Loop Statement	61
10.2.5.1	Unconditional Loop	61
10.2.5.2	Loop Conditions	62
10.2.5.3	Pre-test Loop	62
10.2.5.4	Post-test Loop	62

10.2.5.5	Dual-test Loop	63
10.2.5.6	Mid-test Loop	63
10.2.6	For Loop Statement	63
10.3	Flat Statements	64
10.3.1	Assignment Statement	64
10.3.2	Invoke Statement	65
10.3.3	Exit Statement	66
10.3.4	Next Statement	66
10.3.5	Return Statement	66
10.3.6	Goto Statement	66
10.3.7	Raise Statement	67
10.3.8	Null Statement	67
10.4	Variable Declaration Statement	67
10.5	Entry Visa Declaration Statement	67
10.6	Analysis	68
11.	Procedure Declarations	69
11.1	Procedure Interface	69
11.2	Procedure Body	69
11.3	Formal Parameters	70
11.3.1	Default Parameters	70
11.3.2	Parameter Attributes	70
11.3.3	Open Arrays	71
11.4	Functions	71
11.4.1	Tuple Return	72
11.4.2	Named Returns	72
11.4.3	Exit Visa	73
11.5	Overloading	73
11.6	Operators	73
11.7	Analysis	73
12.	Exceptions	75
12.1	Raising An Exception: Raise Statement	75
12.2	Exit Visa	76
12.3	Catching an Exception	76
12.4	Entry Visa	77
12.5	Propagation: Propagate Statement	78
12.6	Predefined Exceptions	78
12.7	Analysis	79
13.	Generics Declarations	80
13.1	Generic Types	80
13.2	Generic Procedures	80
13.3	Generic Modules	81
13.4	Generic Parameters	82
13.4.1	Value Parameters	82
13.4.2	Variable Parameters	83
13.4.3	Procedure Parameters	83
13.4.4	Type Parameters	84
13.4.5	Module Parameters	84
13.4.6	Generic Specialised Parameters	84
13.4.7	Parameter Restrictions	84

13.5	Analysis	85
14.	Analysis & Conclusions.....	86
14.1	Original Contribution	87
15.	Recommendations for Further Work	88
15.1	Formal Semantics	88
15.2	Runtime Support	88
15.3	Annexes	88
15.3.1	Real-Time Annex	88
15.3.2	Object-Oriented Annex — Class Type	88
15.4	Further Research	89
15.4.1	Closing Keywords	89
15.4.2	Nesting Identifier Rules	89
15.5	Future Research	90
15.5.1	Encapsulation/Information Hiding	90
	References	91

1 Introduction

During the last twenty years developing software for work or hobby, I found many bug-bears with the languages being used Pascal (Borland 1988-94, ISO10206:1991, ISO7185:1990), C (BS9899:1999, Schildt 2001), C++ (Stroustrup 2000, ISO14882:1998), Delphi (Borland 1995), Ada (BS7145:1990, ISO8652:1995, Barnes 1998), Perl (Wall *et al* 2000), Java (Schildt 2001). In some cases, features in one language omitted in another. Ada, although having flaws itself provides features I found lacking elsewhere, particularly in the area of realisation.

Programming is central to software engineering and computer science and it is fitting that research in to programming languages continues. However, the current state of affairs I find disconcerting. There is great emphasis on simplifying system development with web technologies for example and the popularity of languages derived from C that ignore developments of the last thirty years for example, the problem of the dangling else (Louden 1993: 197), which was resolved in Algol-68 with the *bracketing keyword* (*ibid.*). The lack of moral fibre when it come to ethical behaviour, I refer to the case against Microsoft for being a monopolist, is this not enough to be struck-off for professional misconduct and prevented from distributing software for five years? Would this not be so in the legal or medical professions?

As a reaction, this is a project to design a programming language. There is a first part to this report (Wakerley 2004) which explores the theoretical aspects and examines various language features as initial research for this project. This report describes the new language. Time was not available to implement and test the language and so this report only presents the design. I felt it appropriate to relax being too critical whilst designing the language, this allows the complete language to be developed. Later, in another project to implement the language there will be opportunity to be more critical with the benefit of feedback from the implementation.

I have attempted to develop a language in the direction I believe should be available to the professionals. That is, with a Modula-2 (ISO10514-1:1996) & Ada (Barnes 1998) flavour and with sufficient realisation facilities. I believe the language contains sufficient fine-grained features to provide greater expressive power than current languages and demonstrate my mastery of software engineering and the craft of programming.

2 Aims of the Language

The aim of the project is to create a general purpose programming language and that is suitable for the most demanding systems programming. There are many design criteria (Louden 1993: 47-58) for creating a good programming language. I explored many criteria in the previous report (Wakerley 2004: 6-8). In general it is the aim of the language to achieve the best blend of all the criteria but emphasis is on high efficiency, high abstraction, low insulation, high flexibility, high portability, high orthogonality, high scalability, high evolvability, high security & reliability, simplicity, easily implementable and greater realisation facilities. These are criteria for the overall design. On an individual language feature basis many of these will be irrelevant.

These criteria were explored in the previous report (Wakerley 2004: 6-8) but to summarise high efficiency means the program must be fast. High abstraction allows expression of algorithms without knowledge or reference to underlying machine architecture. Low insulation is similar to high efficiency and means the gap between the source code and the machine code should be minimal. Interpreters have high insulation, assemblers have low insulation. High portability allows programs to be compiled without modification for many platforms. Orthogonality is the mathematical property of things that looks similar behaving similar. Scalability and evolvability recognise that fact that programs grow like any living thing and provides facilities appropriate for different stages of life. Realisation facilities allow greater control at machine level whilst retaining a high degree of abstraction.

Measuring individual language features against the above criteria is particularly complex because some criteria are inappropriate for some language features. Thus it becomes very subjective.

I have also introduced facilities for extension and legacy compatibility. The former is provided by allowing annexes that switch-on new or specialised language facilities. The latter by allowing source code to specify which version of the language it conforms to and the compiler is obliged to support.

I have also tried to give the programmer greater emancipation and empowerment through expressive power with syntactic sugar. This is appropriate because a compiler is written once where syntactic sugar may be used extensively to improve readability and elegance of many programs.

2.1 Name

The name of the language I have chosen is “Dios”, which is Spanish for God, with this version of the language having the name “Janus” after the Roman god of doorways and beginnings.

2.2 White Elephants

One important criteria of the language is that it should NOT contain any form of garbage collection facilities. I believe this is most important that programmers learn correct resource allocation and de-allocation. Also garbage collectors do violate the criteria of low insulation.

Therefore there is no mention of dynamic memory allocation, it is expected these will be provided by a standard module. This may be explored in the project to implement the language.

3 Syntax Notation

The morphology (word structure) and syntax of the language is described formally using Extended Backus-Naur Form (EBNF). The EBNF used conforms to the current British Standard (ISO14977: 1996).

EBNF does not explicitly distinguish between the structure of words (tokens) and the structure of sentences (syntax). This leaves an area where the formalism is weak. This ambiguity is a problem because the implementation separates into the lexical analyser (scanner) and the syntax analyser (parser). Therefore explicit distinction is necessary. As a result I have made the distinction by preceding rules defining tokens with the § symbol and syntax rules with the \$ symbol.

It is expected that the reader will be familiar with EBNF conforming to the current standard (ISO14977: 1996) and that no explanation is necessary.

As a reminder square brackets [] indicate the contents are optional, braces {} indicate zero or more, braces with minus {}- indicate one or more of the contents. Single and double quote are used to enclose literal text.

The syntax of the language is written to aid the reader and not for the benefit of compiler design or generation with compiler-compiler tools.

4 Morphology

Morphology is the study of word structure or tokens. This chapter identifies all of the words (tokens) of the language. This is for the benefit of the implementation.

4.1 Characters

```

§ latin letter      = latin capital letter | latin small letter;
§ latin capital letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"
                        "Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
                        (* Unicode "LATIN CAPITAL LETTER A..Z" U+0041..U+005A *)
§ latin small letter = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"
                        "s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"
                        (* Unicode "LATIN SMALL LETTER a..z" U+0061..U+007A *)

```

Source text consists of characters that are part of a character encoding. Software is available to edit source text in most character encodings. Unfortunately encodings such as Unicode contain so many characters that the font or typeface currently cannot represent every character. This presents a problem if the character you wish to use is not supported. This is why the language provides escape sequences at the character level.

4.1.1 Escape Sequences

```

§ escaped character = escaped backslash | escaped single quote | escaped double quote |
                    numbered character | named character;
§ escaped backslash = backslash, backslash;
§ escaped single quote = backslash, single quote;
§ escaped double quote = backslash, double quote;
§ numbered character = backslash, integer literal, backslash;
§ named character = backslash, character name, backslash;
§ character name = {space|digit|latin letter|hyphen-minus}-;
                  (* unicode character names: Unicode Standard v4.0.0 pp.xxxiv *)
§ backslash = "\";
              (* Unicode "REVERSE SOLIDUS" U+005C *)

```

An escape sequence is used to represent a character that is difficult to insert into the program text or is visually ambiguous (with similarly looking glyphs). The escape sequence is interpreted and the identified character is substituted into the text.

An escape sequence is either numbered or named. A numbered escape sequence consists of a natural literal enclosed within backslashes.

```

\16#2028#\ The unicode line separator character
\65\      ASCII Latin Capital A
\SOLIDUS\ Unicode slash

```

The names of characters are those given by the character set. If the character set does not defined names then the name provided by the Unicode Standard shall be used for the equivalent character. Generally symbols do not appear in character names (except space and hyphen-minus).

The backslash character may be identified by empty backslashes.

```
\\ Backslash
```

There are two characters that do not have the closing backslash.

```
\' Single quote
\" Double quote
```

These are intended for inserting quotes into character or string literals.

4.1.2 Illegal Characters

Any characters inserted into the program text are illegal if they are illegal within the character encoding or are not defined as part of the language morphology.

4.2 White Space

\$ white space = word separator | line separator;

There are two types of white space, word separators and line separators. Word separators and line separators are full tokens in they own right. The parser may discard them.

4.2.1 Word Separators

\$ word separator = {space | tab}-;

\$ space = (* Unicode "SPACE" U+0020 *);

\$ tab = (* Unicode "CHARACTER TABULATION" U+0009 *);

Word separators are horizontal spacing characters. They are generally the space and tab characters. The character set, operating system and text editor conventions may define others.

4.2.2 Line Separators

\$ line separator = LS | NLF;

\$ LS = (* Unicode "LINE SEPARATOR": U+2028 *);

\$ NLF = LF | (CR,LF) | CR | NEL; (* Platform/Character encoding defined *)

\$ LF = (* Unicode "LINE FEED" U+000A *);

\$ VT = (* Unicode "VERTICAL TABULATION" U+000B *);

\$ FF = (* Unicode "FORM FEED" U+000C *);

\$ CR = (* Unicode "CARRIAGE RETURN" U+000D *);

\$ NEL = (* Unicode "NEXT LINE" U+0085 *);

\$ PS = (* Unicode "PARAGRAPH SEPARATOR" U+2029 *);

Line separators are vertical spacing characters. They are generally the unicode line separator or the newline function (NLF).

The characters that separate lines in text files differ based on character set, operating system and text editor conventions. The NLF denotes the new line character(s) pertaining to the implementation. Thus, on Unix/Linux this is the linefeed (LF) character.

4.3 Comments

§ comment = multiline comment | single line comment;

Comments permit the programmer to document the source code appropriately. The comments are borrowed from C++, which in turn are borrowed from BCPL.

“A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.” (Stroustrup 2000:139).

A comment is a fully-fledged token in it’s own right. It is semantically equivalent to the word separator token and therefore discarded by the parser.

Character escape sequences are not interpreted within comments, since their contents are disregarded.

4.3.1 Multiline Comments

§ multiline comment = slash, asterisk, multiline comment | (* any character *), asterisk, slash; Multiline comments are opened with `/*` and closed with `*/`. They may be nested. Any number of comments may exist on a single line.

```
/* comment 1 */ i: integer /* comment 2 */;
/*
   this comment
   is spread over
   several lines.
*/
/* outer comment
   /* nested comment
     /* inner nested comment */
     nested comment */
   outer comment
*/
```

It is an error for a multiline comment to be still open at the end-of-file.

4.3.2 Single-Line Comments

§ single line comment = slash, slash, (* any character up to but not including the line separator *); Single-line comments are opened by `//` and extend up to the character before the line separator or end-of-file, which ever is first. “BCPL’s `//` comment convention was (re)introduced in C++.” (Stroustrup 2000:10)

```
// this is a single-line comment
```

Note, multiline comment’s `/*` and `*/` are not recognised within single-line comments and the single-line comment’s `//` is not recognised within multiline comments.

```
// single-line comment /* this does NOT open a multiline
comment
/* // this is a multiline comment */
```

```

/*
// this is still a multiline comment
*/

```

4.4 Identifiers

§ qualified identifier = {identifier, period}, identifier | string literal;
 § identifier = letter, {letter | underscore | digit | other digit};
 § letter = (* any character with unicode character properties: Lu, Ll, Lt *);
 § underscore = " _"; (* Unicode "LOW LINE" U+005F *)
 § digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
 (* Unicode "DIGIT ZERO..NINE" U+0030..U+0039 *)
 § other digit = (* any character with unicode character property: Nd *);

Identifiers are user-defined words that represent procedures, variables, constants and other declarations.

4.5 Keywords

§ admit = "admit";
 § annex = "annex";
 § array = "array";
 § as = "as";
 § begin = "begin";
 § by = "by";
 § case = "case";
 § character = "character";
 § constant = "constant";
 § decimal = "decimal";
 § declare = "declare";
 § default = "default";
 § dios = "dios";
 § distinct = "distinct";
 § else = "else";
 § elsif = "elsif";
 § end = "end";
 § enumeration = "enumeration";
 § exit = "exit";
 § finalization = "finalization";
 § fixed = "fixed";
 § float = "float";
 § for = "for";
 § from = "from";
 § generic = "generic";
 § goto = "goto";
 § if = "if";
 § import = "import";
 § implementation = "implementation";
 § in = "in";
 § initialization = "initialization";
 § integer = "integer";
 § interface = "interface";
 § loop = "loop";
 § module = "module";

\$ next	= "next";
\$ nil	= "nil";
\$ null	= "null";
\$ of	= "of";
\$ on	= "on";
\$ out	= "out";
\$ pointer	= "pointer";
\$ procedure	= "procedure";
\$ propagate	= "propagate";
\$ raise	= "raise";
\$ renames	= "renames";
\$ record	= "record";
\$ return	= "return";
\$ specific	= "specific";
\$ then	= "then";
\$ to	= "to";
\$ type	= "type";
\$ union	= "union";
\$ until	= "until";
\$ while	= "while";
\$ with	= "with";

Keywords are predefined identifiers that are reserved for use by the language. It is an error to use a keyword as an identifier.

4.6 Symbols

Symbols are divided into punctuation and operators.

4.6.1 Punctuation

\$ double quote	= "";	(* Unicode "QUOTATION MARK" U+0022 *)
\$ single quote	= "'";	(* Unicode "APOSTROPHE" U+0027 *)
\$ open parenthesis	= "(";	(* Unicode "LEFT PARENTHESIS" U+0028 *)
\$ close parenthesis	= ")";	(* Unicode "RIGHT PARENTHESIS" U+0029 *)
\$ comma	= ",";	(* Unicode "COMMA" U+002C *)
\$ period	= ".";	(* Unicode "FULL STOP" U+002E *)
\$ colon	= ":";	(* Unicode "COLON" U+002A *)
\$ semicolon	= ";";	(* Unicode "SEMICOLON" U+002B *)
\$ open bracket	= "[";	(* Unicode "LEFT SQUARE BRACKET" U+005B *)
\$ close bracket	= "]";	(* Unicode "RIGHT SQUARE BRACKET" U+005D *)
\$ circumflex	= "^";	(* Unicode "CIRCUMFLEX ACCENT" U+005E *)
\$ vertical bar	= " ";	(* Unicode "VERTICAL LINE" U+007C *)
\$ assignment	= colon, equal;	
\$ double period	= period, period;	
\$ at	= "@";	(*Unicode "COMMERCIAL AT" U+0040 *)
\$ corresponds	= colon, greater-than;	

Punctuators are used as part of the sentence structure.

4.6.2 Operators

\$ plus	= plus sign;	
\$ plus sign	= "+";	(* Unicode "PLUS SIGN" U+002B *)
\$ minus	= minus sign hyphen-minus;	

§ hyphen-minus	= "-";	(* Unicode "HYPHEN-MINUS" U+002D *)
§ minus sign	= "−";	(* Unicode "MINUS SIGN" U+2212 *)
§ complement	= "∼";	(* Unicode "TILDE" U+007E *)
§ multiply	= multiply sign asterisk;	
§ multiply sign	= "×";	(* Unicode "MULTIPLICATION SIGN" U+00D7 *)
§ asterisk	= "*";	(* Unicode "ASTERISK" U+002A *)
§ divide	= divide sign divide slash slash;	
§ divide sign	= "÷";	(* Unicode "DIVISION SIGN" U+00F7 *)
§ divide slash	= "/";	(* Unicode "DIVISION SLASH" U+2215 *)
§ slash	= "/";	(* Unicode "SOLIDUS" U+002F *)
§ modulus	= "%";	(* Unicode "PERCENT SIGN" U+0025 *)
§ remainder	= "%%";	
§ not	= "¬";	(* Unicode "NOT SIGN" U+00AC *)
§ exponentiation	= up arrow double asterisk;	
§ up arrow	= "↑";	(* Unicode "UPWARDS ARROW" U+2191 *)
§ double asterisk	= asterisk, asterisk;	
§ root	= "√";	(* Unicode "SQUARE ROOT" U+221A *)
§ equal	= "=";	(* Unicode "EQUALS SIGN" U+003D *)
§ not equal	= inequality sign (not, equals);	
§ inequality sign	= "≠";	(* Unicode "NOT EQUAL TO" U+2260 *)
§ less-than	= "<";	(* Unicode "LESS-THAN SIGN" U+003C *)
§ greater-than	= ">";	(* Unicode "GREATER-THAN SIGN" U+003E *)
§ less-than or equal	= less-than or equal sign (less-than, equal);	
§ less-than or equal sign	= "≤";	(* Unicode "LESS-THAN OR EQUAL TO" U+003C *)
§ greater-than or equal	= greater-than or equal sign (greater-than, equal);	
§ greater-than or equal sign	= "≥";	(* Unicode "GREATER-THAN OR EQUAL TO" U+003E *)
§ and	= "∧";	(* Unicode "LOGICAL AND" U+2227 *)
§ or	= "∨";	(* Unicode "LOGICAL OR" U+2228 *)
§ xor	= "⊕";	(* Unicode "XOR" U+22BB *)
§ implies	= implies sign (equal, greater-than);	
§ implies sign	= "⇒";	(* Unicode "RIGHTWARDS DOUBLE ARROW" U+22BB *)
§ equivalent	= "⇔";	(* Unicode "LEFT RIGHT DOUBLE ARROW" U+21D4 *)
§ element of	= "∈";	(* Unicode "ELEMENT OF" U+2208 *)
§ not an element of	= "∉";	(* Unicode "NOT AN ELEMENT OF" U+2209 *)
§ subset	= "⊂";	(* Unicode "SUBSET OF" U+2282 *)
§ subset or equal	= "⊆";	(* Unicode "SUBSET OF OR EQUAL TO" U+2286 *)
§ not subset	= "⊄";	(* Unicode "NOT A SUBSET OF" U+2284 *)
§ superset	= "⊃";	(* Unicode "SUPERSET OF" U+2283 *)
§ superset or equal	= "⊇";	(* Unicode "SUPERSET OF OR EQUAL TO" U+2287 *)
§ concatenation	= "&";	(* Unicode "AMPERSAND" U+0026 *)

Operators are used to indicate actions, such as mathematical functions.

4.7 Numbers

§ numeric literal	= natural literal unsigned real literal;
§ natural literal	= denary natural literal based natural literal;
§ unsigned real literal	= decimal real literal based real literal;

Numbers follow the conventions of Ada with a few exceptions. Numerical tokens are unsigned, this is because the sign is an unary operator and therefore is a separate token.

4.7.1 Denary Natural Literal

§ denary natural literal	= numeral, [exponent];
--------------------------	------------------------

\$ numeral	= digit, {[underscore], digit};
\$ exponent	= latin small letter e, [numeric sign], numeral;
\$ latin small letter e	= "e"; (* Unicode "LATIN SMALL LETTER E" U+0065 *)
\$ numeric sign	= positive negative;
\$ positive	= plus;
\$ negative	= minus;

Because numerical tokens are unsigned, they are referred to as naturals. Denary naturals are whole numbers written in base 10. The simplest form is just a sequence of digits.

1, 23, 059

To aid readability single underscores may be used to group digits.

123_456_789

An underscore may not appear at the beginning or end of a sequence of digits. Two underscores may not appear together.

An exponent may be present, identified by a lowercase "e". The uppercase "E" is not allowed because it reduces readability since it is the same height as the digits. The following all represent the value 12300.

123e2, 1230e1, 12300e0

The exponent may have a sign and underscores for readability.

123e+2, 123e+123_456, 123e-1

Negative exponents are valid tokens and should be recognised by the lexical analyser (scanner). However, they will be reported by the syntax analyser (parser) as an error.

4.7.2 Decimal Real Literal

\$ decimal real literal	= numeral, radix point, numeral, [exponent];
\$ radix point	= period;

Decimal numerals are numbers written in base 10 with a decimal point and a fractional part. The simplest form is just a sequence of digits containing a decimal point.

1.5, 23.1, 059.6

Like denary naturals they may have underscores to aid readability.

123.456_789

They may also have an exponent, signed or unsigned and underscores for readability.

123e2, 1230e+1, 12300e-5, 1e+12_34_56, 123e0

4.7.3 Based Natural Literal

\$ based natural literal	= numeral, hash, radix numeral, hash, [exponent];
\$ hash	= "#"; (* Unicode "NUMBER SIGN" U+0023 *)
\$ radix numeral	= radix digit, {[underscore], radix digit};
\$ radix digit	= digit latin letter plus sign slash;

Like Ada, numbers may be expressed in a base other than 10. Unlike Ada the base may be any from binary to base 64. To represent higher bases new digits need to be found. Table 1, “Higher Base Digits,” on page 12, lists the new digits and their values.

Table 1: Higher Base Digits

Value/Digit	Value/Digit	Value/Digit	Value/Digit
0 0	16 G	32 W	48 m
1 1	17 H	33 X	49 n
2 2	18 I	34 Y	50 o
3 3	19 J	35 Z	51 p
4 4	20 K	36 a	52 q
5 5	21 L	37 b	53 r
6 6	22 M	38 c	54 s
7 7	23 N	39 d	55 t
8 8	24 O	40 e	56 u
9 9	25 P	41 f	57 v
10 A	26 Q	42 g	58 w
11 B	27 R	43 h	59 x
12 C	28 S	44 i	60 y
13 D	29 T	45 j	61 z
14 E	30 U	46 k	62 +
15 F	31 V	47 l	63 /

Like Ada a based numeric literal starts with a denary natural representing the base, followed by the based number enclosed in hash characters. Exponents may also be present.

```
2#1101#, 16#D#, 64#D# // all these are equal to 13
16#ABCD#e4, 16#ABCD0000# // both same
```

The exponent is read as “the base to the power of”. e.g. $2\#1101\#e4 = 1101_2 \times 2^4 = 208$

As with the convention a digit exceeding the base is illegal.

4.7.4 Based Real Literal

§ based real literal = numeral, hash, radix numeral, radix point, radix numeral, hash, [exponent];

Reals may also be based. A radix point, like decimals separates the whole number from the fractional part.

```
2#1101.1011#, 16#ABCD.0#e-2, 64#x/+D#
```

The last number is 15728525 in base 64. Note that the slash and plus characters are digits within the hash characters.

4.8 Character Literals

§ character literal = single quote, (* any single character *), single quote;

A character literal is used to express an abstract character. Single quotes enclose a single character. An escape sequence may be used to identify unrepresentable characters.

```
'A', 'x', '\DIVISION SIGN\','\255\','\\','\''
```

The last example shows the escaped single quote enclosed within single quotes.

A character literal must not extend beyond the end of a line.

4.9 String Literals

§ string literal = double quote, {(* any character *)}, double quote;

A string literal is a sequence of zero or more characters enclosed within double quotes but must not extended beyond the end of a line. Escape sequences may be used to represent unrepresentable characters.

The type of a string literal and strings are defined in “8.10.1.2 Strings and Arrays” on page 45.

```
"Bront\16#EB#\","\"Quoted\"\\NLF\""
```

The first example is the word “Brontë” (The diaeresis indicating that the e is sounded separately). The last example shows the escaped double quote enclosed within a string literal.

4.10 Analysis

The language restricts its tokens to the ASCII character set with the exception of operator symbols. In this case all may be represented with ASCII characters. The escape sequence is defined specifically for this purpose. This may reduce usability and readability but modern integrated development environments (IDE) can easily provide facilities for the relevant symbols and their input. The escape sequences provide a high degree of flexibility, compatibility and portability. They also provide orthogonality because they can be used anywhere in the source text.

Comments are simply inserted text that is read as is. If the developer chooses to use the ASCII character set for the source text then he is explicitly denying himself the use of Unicode characters. Escape sequences within comments may still be read as literal text. This is compatible with all design criteria. Comments allow nesting providing a high degree of flexibility.

Numbers, Strings and character literals are different and so look different, this provides a high degree of orthogonality.

Numbers provide high abstraction and flexibility by allowing integers and real to be expressed using up to radix 64.

Identifiers are allowed to contain Unicode letters this provides a high degree of flexibility.

5 Compilation Units

\$ compilation unit = [file header], program | module declaration | generic procedure | generic module;

\$ program = procedure declaration;

The smallest unit that may be compiled separately is the compilation unit. The language is arranged so the compilation unit is a single file or two in the case of the split module.

There are a number of compilation units including the main program, the four module declarations, the generic procedure and the generic modules. Like Ada the program is actually a procedure declaration in a file alone.

This is particularly good design because the build process is hierarchical based on imports from modules. If the compilation process starts with the main program text, its imports indicate the dependent modules that need compiling first. These modules in turn may have dependencies triggering their compilation first. This automatically gives the language control over the build process and suggests the compiler should contain the build tool.

The filename of a compilation unit should be the same as the identifier of the declaration within the file.

5.1 The Main Program

The main program is a lone procedure in its own file. The identifier may be any legal identifier. The example shows a program receiving of command line arguments as an array parameter and returning the okay value. This is assuming a POSIX compliant operating system.

```

procedure main(args: array(cardinal16)of string)
  return integer(16):
begin
  for i in args@range loop
    write(args(i));
    writeln;
  end loop;
  return 0;
end main;

```

If any exceptions are allowed to propagate out of the program they must be explicitly declared using the correct exit visa and correctly propagated, see “12 Exceptions” on page 75.

5.2 Compilation Unit Header

\$ file header = dios, {property}, semicolon, {annex header};

\$ property = identifier, equal, string;
\$ annex header = {annex, identifier, {property}, semicolon};

Every compilation unit may contain a header. The purpose of the header is similar to XML's header, it identifies the file syntax and allows properties to be declared. If the header is present the file will begin with the word "dios", the name of the language. Following this may be any number of property definitions separated by white space.

```
dios version="1.0" encoding="utf-8";
```

The properties are predefined by the language and may vary with differing versions of the language. The two properties currently defined are version and encoding.

The version property declares the version of the language that the file conforms. This allows tried and tested code written for a version of the language to be used with a new compiler without updating for a new version of the language.

The encoding property defines the character set/encoding that the file is using. The language will use the Unicode character set, this is currently a superset of all characters sets. The compiler will convert the character and string literals of source text into the Unicode character set for internal representation and for the resulting compiled files.

5.2.1 Annexes

Annexes may also be specified that provide extra language features. Such as real-time facilities, concurrent multi-tasking and other specialised facilities. These and others may be defined in the future.

```
dios version="1.0" encoding="utf-8";  
annex concurrency version="1.0";
```

The version property is defined for annexes.

5.2.2 Revision Control

It is good practice to make use of revision control tools. RCS is available under open source. It stores the various source texts in a form that allows any version (entered) to be recovered. This can be a godsend at times. See texts on RCS for further details (Tichy & Eggert 1997). There are many other popular tools and books available.

These tools have facilities to embed information in the text and are updated by the tools keeping track of version numbers etc.

The RCS annex allows this information to be understood by the compiler and attributes to be available to use in the program. The version property is optional since there is currently only one version available.

```
dios version="Janus" encoding="utf-8";  
annex RCS version="1.0";  
rcs  
    $Id: ...$  
    $Log: ...$  
end rcs;  
procedure main;  
begin  
    ..  
end main;
```

The annex introduces the **rcs** statement as shown in the example. The RCS keywords are enclosed in dollar symbols (as defined by RCS), these should be placed in a **rcs** statement. The statement may be placed only inside the compilation unit header. It performs no action other than understanding rcs information.

The rcs information within **rcs** statement(s) is available as attributes of the compilation unit identifier. In the above example, this is `main` and the attributes defined would be accessed with `main@rcs.Id` and `main@rcs.Log` (the case must match). These attributes are of type string.

The RCS keywords should be placed on a line by themselves. The *log* keyword is more complicated because it extends beyond one line and the compiler needs to understand how this expands, it usually ends with a blank line.

5.3 Analysis

Compilation units may be expressed using any character set/encoding (subject to the implementation) which will be converted by the compiler to Unicode, this provides a high degree of flexibility.

The version specification feature provides formal legacy support, portability over time since old code does not need to be updated.

High security and reliability is obtained by preventing exception implicitly propagating out of the program. Explicit entry and exit visa are required making a program inherently secure and reliable.

The annex facility provides a means of extension for the future without risking compatibility when an annex is not required.

The RCS annex provides a unique facility for integration with external tools.

6 Modules

§ module declaration = simple module | combined module | module interface | module implementation;

Modules are based on Modula-2's modules (Wirth 1982) and Turbo Pascal's units (Borland 1988-94). The semantics are the same as Modula-2 but the syntax provides greater flexibility and scalability for the evolutionary nature of software development.

6.1 Simple Module

§ simple module = module, qualified identifier, colon, module declarations, end, qualified identifier, semicolon;

The simplest module is Modula-2's local module. It basically consists of a collection of declarations bracketed with the module declaration. Declarations may be exported and imported with the use of the import and export declarations.

The language is designed with evolutionary development process in mind. The simple module is designed for modules with few declarations. As a module grows it can be upgraded to the combined module.

```
module simple:
  export s;
  procedure s:
  begin
    ...
  end s;
end simple;
```

6.1.1 Exporting

§ export declaration = export, identifier list, semicolon;

The export declaration is only legal within the simple module and consists of the **export** keyword followed by a comma separated list of identifiers that are to be visible outside the module.

6.2 Combined Module

§ combined module = module, qualified identifier, colon, interface section, implementation section, end, qualified identifier, semicolon;

§ interface section = interface, [declaration section];

§ implementation section = implementation, module declarations;

§ module declarations = [declaration section], [initialization procedure], [finalization procedure];

The combined module is the next step for a module that is evolving in size and maturity.

It is similar to UCSD/Turbo Pascal's unit.

The module is split into two sections the interface and the implementation section. The interface section replaces the export list of the simple module and declarations here are exported.

Declarations in the implementation section are private to the module. Procedure declarations must be placed in the implementation section and only the forward declaration (procedure heading) may be placed in the interface section.

```
module combined:
interface
  procedure s;
implementation
  procedure s:
  begin
    ...
  end s;
end combined;
```

6.3 Split Module

The split module is the standard Modula-2 module where the definition and implementation are separated into their own files. In this language the definition module is called interface.

6.3.1 Interface

\$ module interface = module, interface, qualified identifier, colon, [declaration section], end, qualified identifier, semicolon;

The interface of a split module is identical to the interface of a combined module with the exception of the declaration heading and it is placed into a separate compilation unit (file).

```
module interface simple:
  procedure s;
end simple;
```

6.3.2 Implementation

\$ module implementation= module, implementation, qualified identifier, colon, module declarations, end, qualified identifier, semicolon;

The implementation section of a split module is identical to the implementation of a combined module with the exception of the declaration heading and the fact that it is placed into a separate compilation unit (file).

```
module implementation simple:
  procedure s:
  begin
    ...
  end s;
end simple;
```

6.4 Module Initialization & Finalisation

\$ initialization procedure= initialization, [exceptional return], [attribute definition], colon,
procedure block, end, initialization, semicolon;

\$ finalization procedure= finalization, [exceptional return], [attribute definition], colon,
procedure block, end, finalization, semicolon;

The initialisation of module variables and other declarations are performed automatically in order of declaration. However, a module may have other initialisation or finalisation needs such as resource acquisition and release.

Unlike Modula-2 there is no **begin** and **end** block for modules to be initialised. Instead a special procedure may be declared for initialisation or finalisation of the module. The initialisation procedure looks like this:

```
initialization:
... // declarations
begin
... // statements
end initialization;
```

The finalisation procedure looks like this:

```
finalization:
... // declarations
begin
... // statements
end finalization;
```

The initialisation and finalisation procedure may not declare parameters or return values. Exceptional returns may be declared.

6.5 Coupling, Import and Visibility

\$ import declaration = [from, qualified identifier], import, import item, {comma, import item}, semicolon;

\$ import item = qualified identifier, [as, identifier];

Good design principles with modules are to minimise coupling and maximise cohesion. As such, the coupling facilities of modules have been improved (over Modula-2) by allowing the import to appear in any declaration area. Modula-2 only permits importing at the top of a module making the identifier visible to the whole module. This is good practice if the imported identifier is used in many places. However, I have learned from experience that minimising the visibility to the smallest required is very good practice. This improves maintainability.

```
declare
import STextIO.write, STextIO.writeln;
begin
STextIO.write("Hello World!");
```

```
    STextIO.writeln;  
end;
```

This example shows a simple import. Identifiers must be fully qualified both in the import and in the invocation. When there are a number of imported identifiers from the same module it is possible to use the *from* clause to specify the module. In this case all identifiers are automatically qualified.

```
declare  
  from STextIO import write, writeln;  
begin  
  STextIO.write("Hello World!");  
  STextIO.writeln;  
end;
```

Note that in the above example only the import has implied qualification. The invocation must still be fully qualified. This may be inconvenient so renaming to an unqualified identifier is possible.

```
declare  
  from STextIO import write as write, writeln as newline;  
begin  
  write("Hello World!");  
  newline;  
end;
```

`STextIO.write` is renamed to `write` and `STextIO.writeln` to `newline`. The import and any renaming has the visibility and lifetime of the scope where declared.

6.6 Module Lifetime

Modules that are compilation units have global lifetime. They would be initialised at the start of the program and finalised at the termination of the program. The initialisation and finalisation procedures would be called if present at these times.

Modules that are declared in a declaration area would have the lifetime associated with that scope. Initialisation would occur when the scope comes into being and finalisation would occur when the scope ceases to exist.

6.7 Module Exceptions

For global lifetime modules, exceptions raised by the initialisation or finalisation must be caught by the program block. This is consistent with the invocation statement that raises exceptions. Relevant entry visa must be declared, See “12.4 Entry Visa” on page 77.

Blocks where modules are declared must catch any exceptional returns from the initialisation or finalisation procedures and have the correct entry visas.

6.8 Analysis

Modules are an abstraction for dividing programs into smaller units. The three types of module provide high degree of flexibility, scalability and evolvability because a small module may start with a simple module and grows through the combined module into a large split module with minimal work for the developer.

The module initialisation and finalisation are *contained* within the module and so contribute to high abstraction and encapsulation. High security and reliability are also provided because the initialisation and finalisation must explicitly declare exceptional returns.

Having three types of module does reduce simplicity overall but the beginner only needs to focus on the simplest and learn incrementally.

There is high encapsulation provided by the module abstraction. Coupling is minimised by the import mechanism working at the scope level like any other declaration.

Orthogonality is high because each type of module may be a compilation unit or generic and nesting is permitted.

7 Declarations

§ declaration section = {declaration}-;
 § declaration = type declaration | variable declaration | constant declaration | procedure declaration | module declaration | rename declaration | import declaration | export declaration | entry visa;

Declarations introduce an identifier to its property definition such as a constant, procedure, module, type or variable.

7.1 Blocks, Scopes, Nesting

The language is block-structured and follows the same scope rules of Pascal, Modula-2 and Ada.

Basically a declaration is visible only within the block it is declared and any nested block. Identifiers that have the same identifier of a declaration in an enclosing block will hide the outer identifier. Further details can be found in any books on Pascal, Modula-2 or Ada. Refer to (Barnes 1998: 74) and (Wirth 1982:143).

7.2 Type Declarations

A type declaration declares an identifier and gives that identifier a type definition. A type declaration has the form:

```
type t: ... ;
```

What appears between the colon and the semicolon is the type definition. There are a number of type definitions within the language and are covered by the chapter “8 Type Declarations” on page 27.

7.3 Variable Declarations

§ variable declaration = identifier, {comma, identifier}, colon, type definition, [default clause], semicolon;

A variable declaration declares one or more identifiers and gives them a fixed data type. The data type determines the set of values a variable may assume and the operation that may be performed.

```
a: integer(32);
b: boolean;
c,d,e: character(8);
s: array [1..10] of (character(8));
date: record
  day: [1..31];
  month: month;
  year: integer(16);
end date;
```

The examples show various variable declarations.

7.3.1 Initialization

§ default clause = assignment, expression, semicolon;

It is possible to initialise a variable with a default clause, which looks like an assignment.

```
a: integer(32) := 123;
b: boolean := true;
c,d,e: character(8) := ('X', 'Y', 'Z');
f,g,h: character(8) := (default :> 'X', h :> 'A');
```

For a list of variables with the same type a tuple expression must be used to initialise them.

7.3.2 Attributes

Variables have a number of attributes available for examination or overriding. These are `address` and `lifetime`.

The `address` attribute is the memory location of the first byte occupied by the variable. It has the type of a pointer to the variable type (`pointer to t@type` where *t* is the variable type).

The `lifetime` attribute refers to the lifetime of the variable. It may have *scoped* or *global* lifetime. The default is *scoped*. Variables with global lifetime must be initialised and are persistent for the lifetime of the program (equivalent the C's static variables).

7.4 Constant Declarations

§ constant declaration = identifier, colon, constant, [type definition], assignment, expression, semicolon;

A constant declaration is similar to a variable declaration but the constant may not be re-assigned a new value.

```
a: constant int8 := 5;
b: constant boolean := true;
nul: constant ascii_char := '\0\';
pi: constant short_float := 3.142;
day1: constant day := Monday;
Waterloo: constant date := (18, June, 1815);
```

The type is optional for discrete types and strings.

```
a: constant := 5;
nul: constant := '\0\';
pi: constant := 3.142;
day1: constant := Monday;
s: constant := "123";
```

7.5 Procedure Declarations

Procedure declarations are covered in chapter “11 Procedure Declarations” on page 69.

7.6 Module Declarations

Module declarations are covered in chapter “6 Modules” on page 18.

7.7 Renaming Declarations

\$ rename declaration = (rename variable|rename constant|rename procedure|rename module), semicolon;

\$ rename variable = identifier, colon, type definition, renames, qualified identifier;

\$ rename constant = identifier, [colon, type definition], renames, qualified identifier;

\$ rename procedure = procedure interface, renames, qualified identifier;

\$ rename module = module, identifier, renames, qualified identifier;

It is always good practice to use meaningful names for identifiers but with large programs consisting of many modules, fully qualified identifiers can become inconveniently long. Renaming is provided to create a new identifier within the local scope for a previously defined identifier elsewhere. This can be a variable, constant, procedure or module. The original identifier remains unchanged.

The new identifier adopts the definition of the original. For variables the type is required and for procedures the full interface is required. This must match the original definition. However, for procedure parameters the default clause is redefined. This is because the default values are expanded at the invocation. Therefore to retain the same default a parameter must declare the same default clause, omission will remove any default.

This feature is borrowed from Ada *as is*, see (Barnes 1998: 253-257) and the Ada standard for details.

7.8 Import & Export Declarations

Importing and exporting from modules is covered in “6 Modules” on page 18.

7.9 Entry Visa Declarations

The entry visa declaration may be in a declaration section or a statement sequence and is covered in “12.4 Entry Visa” on page 77.

7.10 Analysis

The block structure provides superior encapsulation than C and its descendants because it allows nesting of declarations where identifiers have restricted visibility thereby allowing reuse in different scopes. Many times I have been thwarted by the fact C prevents nesting procedures. This satisfies a higher level of abstraction than C and its descendants.

Renaming declarations (including import renaming) are an excellent feature improving readability by allowing long fully qualified identifiers to be shortened within the local

scope. They provide also provide a means to redefine default parameters for procedures. This provides a high degree of flexibility.

8 Type Declarations

§ type declaration = type, type designator, colon, type definition, semicolon;
 § type designator = identifier;
 § type definition = [distinct], (subrange expression | scalar definition | nonscalar definition), [default clause];
 § scalar definition = scalar base definition, [attribute definition], [subrange expression];
 § scalar base definition = type designator | enumeration definition;
 § nonscalar definition = nonscalar base definition, [attribute definition];
 § nonscalar base definition = type designator | array definition | record definition | variant definition | class definition | pointer definition | closure definition;
 § attribute definition = [with], tuple expression;

The type system is intended to be strongly typed with rigorous type checking performed during compilation.

Discrete types have distinct values. Integers, characters and enumeration types are discrete types. Discrete and real types are scalar types.

8.1 Type Definitions

A type declaration declares an identifier and gives that identifier a definition. A type declaration has the form:

```
type t: ... ;
```

What appears between the colon and the semicolon is the type definition. There are a number of type definitions within the language the rest of this chapter deals with them.

8.2 Sub-Types

A type declaration where the type being declared is based on an existing type is a sub-type declaration. Most type declarations are actually subtype declarations. A sub-type declaration without any additional constraints as the form:

```
type t: boolean;
```

Here *t* is declared to be a subtype of `boolean` with the whole range of values. As a sub-type *t* is type and assignment compatible with `boolean`.

8.3 Distinct Types

Types are equivalent if they resolve to the same definition and not based on the identifier in a type declaration. The exception to this rule when a type is declared with the **distinct** keyword, in this case a completely distinct type is created.

```
type t: distinct boolean;
```

In this declaration `t` is declared to be a distinct sub-type. This means that it is not compatible with the `boolean` type. However for the enumeration types the values within the type are duplicated and are the values of the new type. Thus, for `t` the identifiers `false` and `true` are overloaded and are values of both types.

8.4 Attribute Definitions

Types have a number of attributes, some relate to the abstract level whilst others relate to the physical level or realisation of the type. Typical attributes are `first`, `last` and `range` these are like constants that contain information about the type. Because these are common attributes they may be expressed using a sub-range expression for scalar types. This appears after the general attribute clause.

The general attribute definition consists of an optional **with** keyword followed by a tuple. This may be called the **with** clause. The tuple may have positional or named values. The various types have predefined positional arrangements for attribute definitions. For the other type attributes named values must be used within the tuple.

The **with** clause is used to override the default value for any attribute. If attributes are specified they must result in a consistent type definition. Conflicts will be reported as an error.

The positional or named values must correspond with the attribute identifiers and the values must be assignment compatible.

For example:

```
type byte: integer(first :> 0, last :>255);
```

defines an integer type with a range of `[0..255]`.

8.5 Enumeration

\$ enumeration definition = enumeration, open parenthesis, enumeration literal, {vertical bar, enumeration literal}, close parenthesis;

\$ enumeration literal = identifier;

An enumeration type is a data type where its values are identifiers explicitly specified one by one in the type definition. These identifiers are simultaneously declared to be constants and are the only values of this type; no other values belong to this type.

Enumeration types are ordinal, this means the values are ordered. This order corresponds to the order in which the values are declared. Given this example:

```
type suit: enumeration (club|diamond|heart|spade);
type day: enumeration
```

```
(Sunday|Monday|Tuesday|Wednesday|Thursday|
Friday|Saturday);
```

The values `club` to `spade` would have the positions 0 to 3, and the values `Sunday` to `Saturday` the positions 0 to 6.

8.5.1 Enumeration Sub-Type

Having defined an original enumeration type it is possible to define a sub-type by specifying a sub-range expression. For example, to define a type representing the working days of the week from the previously defined `day` type, then any of these declarations could be used:

```
type workingday: [Monday .. Friday];
type workingday: day [Monday .. Friday];
type workingday: enumeration (Sunday|Monday|Tuesday|Wednesday|
Thursday|Friday|Saturday) [Monday .. Friday];
```

The `workingday` type is an enumeration sub-type with the values `Monday` through `Friday`. The first example may be used where there will be no ambiguities. The second and third examples explicitly specifies the base type, the second using the identifier and the third the complete definition. The second example is the preferred format and the third example is not a good idea.

A sub-type is compatible with the base type and other sub-types derived from the same base-type provided the sub-ranges intersect. If there is no intersection, no values will be assignable. Where there is an intersection, if an attempt to assign a value to a variable that is outside the range an `upper_bound_violation` or `lower_bound_violation` exception will be raised.

8.5.2 Operations

Enumeration types have a number of relational operators defined. These are equality, inequality, order relations and membership.

8.5.2.1 Equality and Inequality

Equality (`=`) and inequality (`≠`) compare two values for equality and returns a Boolean value. Obviously equality returns true if the two values are equal and inequality returns true on inequality.

```
d: day := Monday;
...
if d=Tuesday then
...
```

Equality test is also intrinsically embedded in the case statement.

8.5.2.2 Order Relations

The order relations' compare two values and return a Boolean based on the position of each value within type's order.

There are four order relation operators less-than ($<$), less-than or equal ($\leq$), greater-than ($>$), greater-than or equal ($\geq$). These have the expected mathematical semantic.

8.5.2.3 Membership

The membership operator ($\in$) tests if a value belongs to a particular type. There is a not a member operator ($\notin$) that test for not membership.

```
Tuesday  $\in$  day // returns true
```

8.5.3 Attributes

Enumeration types have a number of attributes that may be defined or accessed by the program.

There is no predefined positional tuple to define attributes. Any attribute definition must therefore use named values throughout.

Using attributes instead of constant values or literals directly in the appropriate places can lead to fewer errors or work when modifying program.

8.5.3.1 First, Last and Range Attributes

Three attributes refer to the range of values: `first`, `last` and `range`. For example:

```
type color: enumeration (red|green|blue);
```

The attribute `color@first` has the value `red`, `color@last` has the value `blue` and `color@range` has the value `[red..blue]`. If the `color` type is later modified to:

```
type color: enumeration (red|yellow|green|cyan|blue|magenta);
```

The attribute `color@last` changes to the value `magenta` and `color@range` changes to the value `[red..magenta]`. Therefore, using attributes is sometimes better.

8.5.3.2 Predecessor and Successor Function Attributes

The predecessor and successor function attributes return the predecessor (`previous`) and successor (`next`) values respectively. Both `day@pred(Wednesday)` and `day@succ(Monday)` return the value `Tuesday`.

Attempting to obtain the predecessor of the first value (`day@pred(Sunday)`) or the successor of the last value (`day@succ(Saturday)`) will result in a bounds violation exception.

8.5.3.3 Cyclic Types and Predecessor and Successor Function Attributes

The days of the week are cyclic by nature. To capture this property it is possible to make the type cyclic:

```
type day: enumeration
(Sunday|Monday|Tuesday|Wednesday|Thursday|
Friday|Saturday) (mode :> cyclic);
```

Attempting to obtain the predecessor of the first value (`day@pred(Sunday)`) will return Saturday and the successor of the last value (`day@succ(Saturday)`) will return Sunday. No exception will be raised.

8.5.3.4 Position and Value Function Attributes

The position and value function attributes convert between the enumeration values and integers. The function attribute `day@pos(Wednesday)` would return the value 3 and `day@val(2)` would return the value Tuesday.

8.5.3.5 Minimum and Maximum Function Attributes

The minimum and maximum function attributes compare two values and return the lowest or highest in the types' order respectively. The function attribute `day@min(Friday, Wednesday)` would return the value Wednesday and `day@max(Friday, Wednesday)` would return the value Friday.

8.6 Boolean

Boolean type is a globally scoped predefined enumeration type. It is predefined because some statements take a parameter that is required to evaluate to a value representing the truth-values true or false. The declaration of the `boolean` type can be assumed to be:

```
type boolean: enumeration (false|true);
```

As an enumeration type boolean has all the properties for enumeration types.

8.7 Characters

Unlike the enumeration type the character type is predefined. It has the reserved word `character` defined and a word structure predefined for its values (character literals).

Character types are ordinal, this means the values are ordered. This order corresponds to the position value underlying the character type. *The character type is implemented physically with integers, this is hidden from the abstract level.*

8.7.1 Predefined Character Type

The predefined type **character** represents the complete infinite and contiguous set of characters and is distinct from all other types. The reserved word **character** is a reserved word and may not be redefined, overloaded or otherwise hijacked.

The set of characters is the infinite or unconstrained set, mapping the integer values $n = \{-\infty, \dots, -3, -2, -1, 0, 1, 2, 3, \dots, \infty\}$ to characters with the functional attribute **character**@val(n).

As an unconstrained type it cannot be used directly to declare variables. This code fragment would result in an unconstrained type error.

```
declare
  c: character; // illegal
begin
  ...
end;
```

To use a character type it is necessary to use a sub-type.

8.7.2 Character Literals

As a predefined type the word structure (tokens) are predefined for its values (character literals). Character literals are the only values compatible with the character type but are compatible with all character sub-types even *distincts*.

The structure of character literals was defined in “4.8 Character Literals” on page 12, which is basically a single character enclosed in single quotes. The character may be an escape sequence.

```
'A' 'Z' '0' '\COLON\' '\FULL STOP'
```

It is possible to express a character literal for any integer value using a numeric escape sequence.

```
'\0\'      // a character with integer value 0
'\255\'    // a character with integer value 255
'\65535\'  // a character with integer value 65535
'\16#10FFFF#\' // a character with integer value 16#10FFFF#
```

8.7.3 Character Sub-Type

All usable character types are sub-type definitions of the **character** type. There are a number of forms of character sub-type definition.

The simplest is the implied subrange definition. This example declares a character type with the range of character being the latin capital letters. The base type will be assumed to be the predefined **character** type.

```
type char: ['A' .. 'Z'];
```

It is better to specify the base type, this can be the predefined **character** type or a previously defined character type.

```
type char16: character ['\0\' .. character@val((216)-1)];
type char8: char16 ['\0\' .. '\255\'];
```

The first example defines a character sub-type based on the predefined **character** with 65536 characters. The second example defines a character sub-type based on the first example with a range that could represent the Latin-1 character encoding.

The character types have a predefined positional tuple for specifying attributes. This consists of a single cardinal expression. The cardinal expression specifies the size (binary digits) of the character type. So, a definition of the form:

```
character(n) OR character with(n)
```

is short-hand for:

```
character [character@val(0) .. character@val((2n)-1)]
```

Thus the declaration:

```
type char32: character(32);
```

declares a 32 bit character type with the range ['\0\' .. '\16#FFFFFFFF#\'].

8.7.4 Operations & Attributes

Operations and attributes for character types are the same as those defined for enumeration types. These are the operators: equality, inequality, order relations and membership.

All character literals have membership of the predefined **character** type and so the membership operator would always return true:

```
'x' ∈ character // returns true for all character literals
```

The not a member operator would always return false. Sub-types would test against the sub-range.

The three attributes `first`, `last` and `range`. are defined for all constrained sub-types. They are not defined for the predefined **character** type.

The predecessor, successor, position, value, minimum and maximum function attributes are all defined for character sub-types, the predefined **character** type and all character literals. They are all intrinsic functions. The value function attribute can be used to define character sub-types of any range.

The `mode` attribute may be overridden with a **with** clause to make a type cyclic. The function attributes like the enumeration types will raise a bound violation exception for values outside of the sub-types range.

8.8 Integers

Like the character type the integer type is predefined. It has the reserved word **integer** defined and a word structure predefined for its values (integer literals).

Integer types are ordinal, this means the values are ordered. This order corresponds to the position value that just happens to be the same as the integer value.

8.8.1 Predefined Integer Type

The predefined type **integer** represents the complete infinite and contiguous set of whole numbers and is distinct from all other types. The reserved word **integer** is reserved and may not be redefined, overloaded or otherwise hijacked.

The set of integers is the infinite or unconstrained set $-\infty, \dots, -3, -2, -1, 0, 1, 2, 3, \dots, \infty$. As an unconstrained type it cannot be used directly to declare variables. This code fragment would result in an unconstrained type error.

```
declare
  i: integer; // illegal
begin
  ...
end;
```

To use an integer type it is necessary to use a sub-type.

8.8.2 Integer Literals

§ integer literal = [numeric sign], natural literal;

As a predefined type the integer values (integer literals) have a predefined word structure (tokens). Integer literals are the only values compatible with the integer type but are compatible with all integer sub-types even *distincts*.

The structure of unsigned integer (natural) literals was defined in “4.7 Numbers” on page 10. A sign may also precede them.

```
+123, -123_456_789, +10, -53, 16#3FFF#, 2#1101_0101#
```

It is an error for an integer literal to have a negative exponent, even if the resulting value is an integer.

```
12300e-2 = 123
```

For readability, why obfuscate using the former when the latter is more readable?

8.8.3 Integer Sub-Type

All usable integer types are sub-type definitions of the **integer** type. The **integer** type has a similar form to the character type definition. There are a number of forms for sub-type definition.

The simplest is the implied sub-range definition. This example declares an integer type with the range of integer being -128 to 127. The base type will be assumed to be the predefined **integer** type.

```
type int8: [-128 .. 127];
```

It is better to specify the base type, this can be the predefined **integer** type or a previously defined integer type.

```
type int16: integer [-32768 .. 32767];
type uint8: int16 [0 .. 255];
```

The first example defines an integer sub-type based on the predefined **integer** with the range -32768 to 32767. The second example defines an integer sub-type based on the first example with a range that could represent a byte.

The integer types have a predefined positional tuple for specifying attributes. This consists of a single cardinal expression. The cardinal expression specifies the size or number of binary digits. So, a definition of the form:

```
integer(n)
```

is short-hand for:

```
integer [-(2n-1) .. (2n-1)-1]
```

Thus the declaration:

```
type int16: integer(16);
```

declares a 16 bit integer sub-type with the range -2^{15} to $2^{15}-1$ or [-32768 .. 32767].

The size may be **any** non-negative value including zero that would result in a null range. This allows large integers to be used.

```
type large_int: integer(4096);
```

This declares a type with 4096 bits or 512 bytes long. Large integers like this may not be implemented intrinsically but using subprograms that work on integers of variable length.

These two forms provide portability across all platforms because the range or size is specified, the compiler generates correct code for that type. This is unlike C where the size of **int** may vary across platforms, which may break algorithms and from past experience hard to find bugs.

Unsigned types may be specified by using a range with `first ≥ 0`. Unsigned types will be stored as unsigned values.

```
type cardinal32: integer(32, first :> 0);
```

Here the `last` attribute will be calculated as 2^{32} .

8.8.3.1 Partial Constraints

Partial constraints may be specified using a **with** clause.

```
type cardinal: integer (first :> 0);  
type natural: integer (first :> 1);
```

Here the upper range attribute (`last`) is omitted requiring it to be specified later. As with sub ranges later range constraints must be equal or smaller than the base type. In the `cardinal` type example it will not be possible to specify `first` or lower range less-than zero.

8.8.3.2 Modular or Cyclic Arithmetic

Cyclic or modular arithmetic may be used, by specifying a modular type.

```
type byte: integer (modulus :> 255);
```

Modular integers are unsigned and use cyclic arithmetic. With a range zero to the specified positive integer. No bound violation exceptions will be raised because assignments of values outside the range will wraparound.

Specifying the modulus attribute is shorthand for the attribute definition (`first :> 0, last :> n, mode :> cyclic`).

It is possible to specify a modular type with a range using an attribute definition. The range need not be unsigned.

```
type buf_index: integer(mode :> cyclic) [0..255];  
type buf_index: integer(mode :> cyclic) [-128..127];
```

A common security problem with software results from buffer overflow errors. Cyclic integer types are excellent for use as buffer indices because there is no possibility of buffer overflow.

```
type idx: integer(mode :> cyclic) [0..1023];  
type buffer: array idx of byte;
```

8.8.3.3 Realisations

The size of integers defined with a range adopt the smallest size required and adopt the format: unsigned or two's complement etc. based on the range. There are a number of at-

tributes that may be specified to realise physical attributes. The compiler will generate code to perform any conversions transparently, however performance may be effected.

It is possible to make integers use less space by using a translation.

```
type a: integer(translation :> 100) [100..300];
```

Would fit in a unsigned byte with the physical range 0..200 using.

```
type b: integer(translation :> +228) [100..300];
```

Would fit in a signed byte with the physical range -128..72 using two complement.

```
type c: integer(translation :> +173) [100..300];
```

Would fit in a signed byte with the physical range -73..127 using two's complement.

It is possible to specify a different format to store integers than the processor uses.

```
type d: integer(32, format :> ones_complement);
```

Some hardware store integer with the least significant byte (little endian) and other with the most significant byte first (big endian). Some computers may use other weird formats. The compiler will automatically generate the correct code for the hardware. Protocols may specify their own byte ordering for storing binary data. For example, the TCP/IP protocol defines the network standard byte order to be big endian. Therefore, it is possible to specify a realisation attribute within the integer type definition.

```
type int32be: integer(32, byte_ordering :> big_endian);
```

The compiler will generate code that will physically store the integer in the specified order whilst being transparent at the abstract level.

The integer types will be implemented using the hardware provided operations. This can lead to unportable behaviour. For example integer division will have a rounding behaviour. There are generally five rounding behaviours. It depends on the hardware designers which is used.

To prevent non-portable behaviour numerical types have a rounding attribute. There is a predefined enumeration type:

```
type rounding:
enumeration(toward_nearest|toward_zero|toward_sinf|
toward_pinf|toward_ninf);
```

The `rounding` enumeration defines the five possible rounding behaviours. Round towards nearest whole number, towards zero, towards infinity with same sign (away from zero), towards positive infinity and towards negative infinity.

The rounding attribute is defined to have type `rounding`. The default is based on the platform. Overriding it with a attribute definition for a numerical type will make all operations default to this behaviour.

```
type int16: integer (16, rounding := toward_nearest);
```

This defines a sixteen-bit integer type where all operations involving rounding will round to the nearest whole number. This guarantees portable behaviour for all platforms. It may however result in slower performance if the hardware implements rounding differently and the operations must be performed in software.

8.8.4 Operations

Integer types have the same operators as character types along with many more. The equality, inequality, order relations and membership operators all work as one would expect.

All integer literals have membership of the predefined `integer` type and so the membership operator would always return true:

```
56 ∈ integer // returns true for all integer literals
```

The not a member operator would always return false. Sub-types would test against the sub-range.

The integer types also have arithmetic operators. These are: addition, subtraction, multiplication, division, modulus, remainder, exponentiation, root, absolute value, bitwise logical, shift and rotate operators. Each have the usual mathematical definition.

The modulus and remainder have the Ada definition; that is the remainder returns the correct value when dealing with negative values.

The bitwise logical operators are complement, conjunction, disjunction, implication, exclusive disjunction and equivalence. The table gives the result of each operation applied

Table 1: Result Table for Bitwise Operators

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \vee q$	$p \Leftrightarrow q$
0	0	1	0	0	1	0	1
0	1	1	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	1	1	1	0	1

to two bits. Corresponding bits in each of the operands would have a corresponding result

from the table. The shift and rotate operators move the bits left or right. The rotate operators would move the ejected bit back in the other end.

8.8.5 Attributes

The three attributes `first`, `last` and `range` are defined for all constrained sub-types. They are not defined for the predefined `integer` type.

The predecessor, successor, position, value, minimum and maximum function attributes are all defined for integer sub-types, the predefined `integer` type and all integer literals. They are all intrinsic functions.

The position and value function attributes are somewhat redundant as they return the same value unless of course it is out-of-range, in which an exception will be raised.

The magnitude function attribute returns the magnitude (or distance from zero) of a number. That is, the number without sign. This function is also known as modulus or absolute value.

```
integer@magnitude(-50)    // returns 50
integer@magnitude(0)      // returns 0
integer@magnitude(103)    // returns 103
```

The sign function attribute returns -1, 0 or 1 depending on whether the parameter is less-than zero, zero or greater-than zero.

```
integer@sign(-50)         // returns -1
integer@sign(0)           // returns 0
integer@sign(103)         // returns 1
```

There is a divide function attribute. This basically performs a division operation similarly to the divide operator with the addition of a third argument. The third argument allows the specification of the rounding properties for the division. It has the interface declaration:

```
procedure type@div(a,b: type, round: rounding :=
  toward_nearest)
  return type
  raise divide_by_zero, lower_bound_violation,
    upper_bound_violation;
```

The rounding argument defaults to nearest if not specified.

```
integer@div(25.2, 3, toward_nearest)
integer@div(25.2, 3, toward_zero)
integer@div(25.2, 3, toward_sinf)
integer@div(25.2, 3, toward_pinf)
integer@div(25.2, 3, toward_ninf)
```

8.9 Real Types

Real numbers implemented on a computer can only be approximations because of the finite nature so the real type is an abstract concept only.

Because of the different applications and properties required of real calculations, there are three real types: floating-point, binary fixed-point and decimal fixed-point. These are represented by the predefined types: **float**, **fixed** and **decimal**. All three belong to the abstract real type but have different physical representations and the strong typing prevents type compatibility but type conversion is possible.

The predefined types **float**, **fixed** and **decimal** are also reserved words and so cannot be overloaded. Like the **integer** type the real types are all unconstrained types and cannot be used in variable declarations without additional constraints.

8.9.1 Real Literals

§ real literal = [numeric sign], unsigned real literal;

The real literals do belong to the abstract real type and are compatible with the three real types. Real literals are a syntactic construct consisting of an optional sign token and the unsigned literal token. The unsigned real literal tokens were defined in “4.7 Numbers” on page 10.

```
-0.5, +10.123, 3.141_592_653_589, -16#3FFF.ABCD#, 2#1101.101#
```

8.9.2 Predefined Floating-Point Type

Floating-point numbers follow the IEEE 754-1985 Standard for binary floating-point arithmetic (BS 7237:1990). They may be implemented using hardware or software.

The IEEE standard defines single and double precision types. It also allows for extended precision types to be defined. The single and double precision float-point types are 32 and 64 bits in size respectively. However, it is possible to specify floating-point types of any size.

```
type float32: float (size:>32);  
type float64: float (size:>64);  
type float100: float (size:>100);
```

Where the size matches the size of a type defined in the IEEE standard that format will be used if not the compiler is free to use its own format provided the full size is used.

Unlike integers, floats have a more complex internal structure and specifying only the overall size may result in non-portable behaviour. The internal structure consists of three components: sign, exponent and fraction. The sign only requires one bit and there is no benefit to defining a larger size. The exponent and fraction parts do effect the precision

and range of numbers representable. It is therefore possible to specify the sizes for the exponent and fraction part. This allows greater control, portability and predictable behaviour. The positional tuple expects two values: the exponent size and the fraction size (number of bits). Both must be specified or derivable otherwise the type will be unconstrained.

```
type float32: float (8,23);  
type float64: float (11,52);
```

These two examples have the same sizes as the single and double precision type defined in the IEEE standard and would be behaviour exactly the same. The size of the sign will default to one.

```
type float100: float (19,80);
```

This type has an exponent of 19 bits and a fraction of 80 bits. Its behaviour will be consistent with the IEEE standard but have the extra precision the size allows.

It is possible to a floating-point type with a sub-range but the base type must already have the precision defined.

```
type fp: float [-10.0 .. 10.0]; // error
```

This would result in an error because the **float** type is unconstrained and the size of the exponent and fraction part cannot be calculated base on a sub-range. This would be better:

```
type fp: float32 [-10.0 .. 10.0];
```

The previously defined `float32` type already has its size attributes defined and the sub-range constrains the range.

```
type fp: float (8,23) [-10.0 .. 10.0];
```

It is possible to use an attribute definition to specify a complete type.

```
type fp: float (exponent_size := 8, fraction_size := 23,  
  range := [-10.0 .. 10.0]);
```

As a numerical type the rounding attribute is available for overriding. Its default is defined by the IEEE standard to be round to nearest.

```
type fp: float (8,23, rounding:= toward_sinf);
```

This declaration defines a floating-point type of single precision with rounding away from zero.

8.9.2.1 Operations

Float types have the similar operators as integer types. The order relations and membership operators all work as one would expect.

The equality and inequality operators are defined for float types. However, there are replacement function attributes available that should be used instead. This is due to the problems of accuracy, rounding properties and the derivation of values.

All real literals have membership of the predefined **float** type and so the membership operator would always return true:

```
56.0 ∈ real // returns true for all real literals
```

The not-a-member operator would always return false. Sub-types would test against the sub-range.

The arithmetic operators are addition, subtraction, multiplication, division, modulus, remainder, exponentiation and root. Each has the same mathematical definition as integers.

The modulus and remainder have integer divide behaviour, obviously.

There are no bitwise logical operators.

8.9.2.2 Attributes

The three attributes *first*, *last* and *range* are defined for all constrained float sub-types. They are not defined for the predefined **float** type.

There are no predecessor, successor, position or value function attributes.

The minimum, maximum, magnitude and sign function attributes are defined for float sub-types and the predefined **float** type. They have similar behaviour as the integer types.

There is a *tolerance* attribute available for overriding. Its purpose is to provide a tolerance for the equality and inequality operators. However, it can only be defined in the **with** clause in the type definition. It cannot be changed during runtime. The equality function attribute is provided for fine grain control.

The equality function attribute should be used to test for equality. It takes three arguments. The first two are the values to be compared. The third is the tolerance or accuracy of the test.

```
float@eq(56.32, 56.335, 0.0001)
float@neq(a, b, tolerance := 0.0001)
```

The equality function attribute return the result of $|a-b| \leq t$ where *a* and *b* are the values and *t* is the tolerance. The inequality function attribute will return the opposite boolean value.

The divide function attribute also exists for the float type. It as the interface declaration:

```

procedure type@div(a,b: type, round: rounding :=
toward_nearest)
  return type
  raise divide_by_zero, overflow, underflow, inexact,
    invalid_operation, lower_bound_violation,
    upper_bound_violation;

```

The raised exceptions are the IEEE standard defined exceptions along with the upper and lower bound exception for when the range is exceeded.

```

float@div(25.2, 3, toward_zero)

```

8.9.3 Binary or Secundal Fixed-Point

The word for binary numbers with fractional part is secundal (from Latin secundus, on the form of decimal derived from Latin decim, a tenth) other candidates are binal and bicimal.

Basically secundal or binary fixed-point numbers are actually stored as scaled integers. The scaling relates to the multiplier that moves the radix point. The number of (secundal) digits and the number of secundal places are values expected by the predefined positional attribute definition tuple. Both must be specified or derivable otherwise the type will be unconstrained.

```

type fixed12: fixed (12,4);
a: fixed12 := 11101101.0101; // decimal 237.3125

```

Would be able to represent the secundal 11101101.0101 or decimal 237.3125.

Specifying zero for the number of places is equivalent to integers. Specifying negative number of places will result in a discrete type with legal values being separated by more than one.

```

type granular16: fixed (12,-4);

```

This type as values that are multiples of 16. Number in between will be rounded based on the rounding attribute.

The operations and attributes are the same as for floating-point.

8.9.4 Decimal Fixed-Point

The decimal fixed-point type is provided for applications such as financial software. Like secundal fixed-point types the number of (decimal) digits and decimal places must be specified. This is the predefined positional attribute definition tuple. Both values must be specified or derivable otherwise the type will be unconstrained.

```

type dec12: decimal (12,4);

```

Decimal fixed-point type will most likely be implemented using binary-coded-decimal format but this may not be the case. The type will have the specified digits and places for decimal.

The operations and attributes are the same as for floating-point.

8.10 Composites

There are four composite types: the array, the record, the variant (union) and the class. Composite types permit the manipulation of tuples or aggregates.

8.10.1 Array

- \$ array definition = array, [open parenthesis], array dimension list, [close parenthesis], of, array element type;
- \$ array dimension list = array dimension, {comma, array dimension};
- \$ array dimension = subrange expression | unbound subrange | discrete definition;
- \$ discrete definition = discrete base definition, [attribute definition], [unbound subrange | subrange expression];
- \$ discrete base definition = type designator | enumeration definition;
- \$ unbound subrange = open bracket, close bracket;
- \$ array element type = [open parenthesis], type definition, [close parenthesis];

The array is a composite of homogenous elements. They a similar appearance to Pascal or Ada arrays.

```
type intarr: array [1..10] of int16;
```

This example is an array type of ten `int16`s. The index is implied from the type of subrange expression, which in this case is `integer`.

It is possible to use a type definition for the index.

```
type intarr: array character(32) ['A'..'Z'] of int16;
```

Here a definite index is declared but like Ada it is possible to declare unconstrained indices.

```
type a: array character(32)[] of int16;
type b: array integer of int16;
```

Here the first example declares an index to be of type `character(32)` but the actual range to be defined later. The second example `integer` is already unconstrained and so the range is also to be defined later.

It is also possible to specify unknown index types using an empty sub-range expression.

```
type arr: array [],[] of int16;
...
b: arr(day, [0..23]);
```

The above example shows the type declaration with two dimensions but unknown index types. The variable declaration shows the completed index types, the first the enumeration `day` and the second an integer range.

The indices and element type may be enclosed in parentheses for readability. If the element type uses an attribute definition the element type must be enclosed in parentheses otherwise the compiler will report an ambiguity.

```
type a: array(integer)of integer(16);
type b: array(integer)of integer(element_size:>32);
```

Here there are two ambiguities. The correct syntax is:

```
type a: array(integer)of (integer(16));
type b: array(integer)of (integer) (element_size:>32);
```

It is possible to use the range attribute to specify an index.

```
type a: array(t.range)of (integer(16));
type b: array(integer t@range)of (integer(16));
```

8.10.1.1 Multi-dimensional Array

An array may be of any static number of dimensions. One could define the following two-dimensional array type.

```
type a: array([1..10])of array([3..7])of int;
```

However, one could define the same array with this shorthand.

```
type b: array([1..10], [3..7])of int;
```

Where the dimensions are a comma separated list of index declarations. Variables declared with types `a` and `b` are assignment compatible.

```
x: a;
y: b;
...
y := x;
```

8.10.1.2 Strings and Arrays

A string is actually an array of characters. A string literal as the type:

```
array(integer(first:>0))of (character)
```

The character type will be inferred from the source character encoding. Basically the smallest size that would represent all of the characters in the string. Storage size (`element_size` attribute) would be rounded to the usual alignment requires set by the compiler or CPU architecture.

The index integer type will adopt a size required to represent the require range (number of characters).

String literal tokens which were described in “4.9 String Literals” on page 13 are actually a short-hand for a positional aggregate or tuple.

```
"Red" = ('R', 'e', 'd')
```

8.10.1.3 Operations

The operations on array are limited. Equality, inequality and membership is defined.

For assignment, equality and inequality the operations are the composite of the individual operations performed on each element.

The membership operation tests a tuple expression for compatibility with the array type. It too is a conjunction of the membership operator being apply to corresponding values in the tuple and with the elements in the array. The tuple may have named values.

8.10.1.4 Attributes

The four attributes `first`, `last`, `range` and `length` are defined for arrays. They refer to the range of the indices of the array. They actually take a natural parameter value (greater-than zero). This value refers to the dimension of the array. The default is one if absent.

The `length` attribute is the number of elements in a index.

The `element_size` attribute specifies the size of each element including any padding if different to the element types size.

8.10.2 Records

```
$ record definition    = record, {record field, semicolon}, end, identifier;
$ record field        = identifier, colon, type definition, [in, integer expression, subrange
                        expression];
```

A record is an aggregate structure consisting of a fixed number of components; it could be considered a fixed collection of variables. The components are called fields and like variables have an identifier and a type definition.

Because the fields are components of the record their scope is that of the record itself and so are accessed with field designators.

```
type date: record
    day: integer [1..31];
    month: month;
    year: int;
end date;

type point: record
    x: int32_t;
    y: int32_t;
end point;
```

8.10.2.1 Bit Fields

The physical format of a record may be defined for portability when accessing external resources such as file formats. This is done using the **in** clause which if absent for any field, the compiler juggle the format as it sees fit. Usually the compiler will pad the field for alignment with memory boundaries.

The **in** clause takes a byte offset into the record and an integer sub-range expression indicating the bits being occupied. They are additive, that is 0 [8..15] is the same as 1 [0..7].

```
type date: record
  day: integer [1..31] in 0 [0..4];
  month: month in 0 [5..8];
  year: int in 1 [1..15];
end date;
```

Fields may not occupy the same bits.

Having created a record type with a physical format for external resource access it is also appropriate to create a padded version optimised for use in memory. This can be achieved by creating a subtype and then using a pragma to force the compiler to juggle the format.

```
type optdate: date;
pragma optimize (optdate);
```

8.10.2.2 Operations

The operations on records are limited. Equality, inequality and membership are defined.

For assignment, equality and inequality the operations are the composite of the individual operations performed on each field.

Thus, for two variables of the previous `date` record, equality would be the conjunction:

$$a.day = b.day \wedge a.month = b.month \wedge a.year = b.year$$

The membership operation tests a tuple expression for compatibility with the record type. It too is a conjunction of the membership operator being apply to corresponding values in the tuple and with the fields in the record. The tuple may have named values.

8.10.2.3 Field Attributes

The fields of a record type have additional attributes relating to their position within the record. These are `plot_first`, `plot_last`, `plot_range` and `plot_offset`. The word *plot* is used in the sense a *plot* of land. The attribute `plot_offset` gives the offset in storage elements (byte) of the first occupied. The `plot_first`, `plot_last` and `plot_range` attributes are the bit range the field is occupying.

```
date.day@plot_first
```

8.10.3 Variants

\$ variant definition = union, {vertical bar, record field, semicolon}, end, identifier;

The variant or union is similar to C's union. All fields within a variant are aligned to the same address. This allows for binary data to be interpreted differently. The only operations defined are assignment, equality and inequality. They simply operate on blocks of memory with no interpretation of the typing of fields.

```
type date: union
| day: integer [1..31];
| month: month;
| year: int;
end date;

type point: union
| x: int32_t;
| y: int32_t;
end point;

type test: union
| x: integer [0..10];
| y: integer (32);
end point;
```

The variant is included because it exists in C. C and C++ at the current time, are the most popular languages in use, whilst Ada and more restrictive languages are fading away.

C's union and Pascal's anonymous variant are considered to be insecure in some quarters. To ensure the variant is a secure abstraction the fields of a variant are tested on access for membership of the fields' type. An exception will be raised based on the transgression of the data. For an integer this could be the lower or upper bounds violation.

8.11 Pointers

\$ pointer type = pointer, to, [open parenthesis], type definition, [close parenthesis];

The pointer type is an abstraction of an address in assembly language. It is effectively an indirect type.

```
type pbyte: pointer to byte;
```

This declaration defines a type which values are addresses that point to variables of type byte.

There is only one literal defined for pointer type **nil**, it is a reserved word and represents a pointer that points to no variable at all.

The address of variables may be obtained using the address attribute. For type compatibility pointers must de-reference to compatible types.

```

declare
  a: int16 := 56;
  p: pointer to int16 := nil;
begin
  p := a@address;
end;

```

Attempting to de-reference **nil** value will result in an error or exception (**nil_dereferenced**).

8.12 Procedures (Closures)

\$ **procedure type** = **procedure**, [formal type list], [formal return], [exceptional return], [attribute definition];

\$ **formal type list** = open parenthesis, formal type, {comma, formal type}, close parenthesis;

\$ **formal type** = [in], [out], type definition;

\$ **formal return** = **return**, type definition | tuple type definition;

\$ **tuple type definition** = open parenthesis, type definition, {comma, type definition}, close parenthesis;

\$ **exceptional return** = **raise**, qualified identifier list;

\$ **qualified identifier list** = qualified identifier, {comma, qualified identifier};

A closure or *procedure type* is a type that has procedures as its values. This allows variables of procedure types to be assigned procedures and the procedure held in the variable to be invoked. This provides a form of polymorphic behaviour or dynamic dispatching.

```

declare
  type func: procedure (int) return int;
  procedure a(x: int) return int:
  begin
  end a;
  procedure b(x: int) return int:
  begin
  end b;
  f: func;
  x: int;
begin
  f := a;
  x := f(5); // invokes a
  f := b;
  x := f(5); // invokes b
end;

```

The **null** statement is actually a predefined procedure that does nothing. It may be assigned to any procedure type variable or passed as an argument to a procedure, provided there is no return value.

Note: procedures that are nested are not type compatible with a procedure type due to the nested environment being part of a procedure's type.

8.13 Default Values

Type declarations may also declare default values which variables will be initialised with unless otherwise explicitly initialised.

Here the `bool` type is a sub-type of `boolean` with the default of `false`.

```
type bool: boolean := false;
```

8.14 Type Compatibility

Integer types are type compatible with other integer types; character types are type compatible with other character types and real types are type compatible with other real types provided that their range intersects. Where ranges intersect but also overlap bound checking will be performed on assignment and out-of-bounds values will result in exceptions being raised.

Enumeration types are compatible if they have the same definition or are a sub-type of the same definition.

Distinct types are only compatible with themselves or non-distinct types derived from themselves.

The type definition is used to check for type compatibility and not the identifier, except with distinct types.

Aggregate types are compatible if they have the same degree and corresponding elements are type compatible.

8.15 General Type Attributes

All type have a number of general attributes. These are **type**, `basetype` and `roottype`. The first contains the type definition of the type. The second is the type definition of the immediate parent type. Finally the `roottype` attribute is the type definition of the ultimate ancestor type.

For example given these declarations.

```
type int16: integer(range => -32768..32767);
type int8:  int16  (range => -128..127);
x: int8;
```

Then these expressions would be true.

```
integer.type=integer
int16.type=int16
int16.base=integer
int16.root=integer
int8.type=int8
int8.base=int16
```

```
int8.root=integer  
x.type=int8  
x.base=int16  
x.root=integer
```

8.16 Analysis

These data types provide a high degree of abstraction and portability by representing abstract concepts such as number, characters, enumeration and aggregates. There is a high degree of orthogonality, there is a clear distinction between numbers, characters, strings and aggregates.

A high degree of flexibility and scalability is provided by the ability to define types with any finite range of integers and characters for example.

A high degree of security and reliability is provided. For example, using an array with a modular type for the index as a buffer will prevent buffer overflow bugs and exploitations.

There is a great amount of realisation facilities that give greater control of various type attributes thereby reducing the insulation between the language and the underlying machine. For example the physical format of a record can be controlled so compatibility with file formats or memory mapped input/output ports can be maintained.

9 Expressions

§ expression = singleton expression | tuple expression | sub-range expression;

Expressions were introduced into programming languages to allow calculations to be expressed in an algebraic fashion. This improves readability and comprehension.

Expressions generally consist of operators and operands. An operator is a function that is represented using a symbol and has one or two operands. Unlike functions, operators may use prefix, infix or suffix notation. An operand is the name given to a parameter of an operator.

The language improves orthogonality of expressions by allowing tuple expressions.

9.1 Operands

§ operand = literal value | function designator | variable designator | constant designator | attribute designator | element designator | field designator | pointer dereference | type designator | qualification | conversion;

Operands are the parameters of operators. An operand maybe another expression, a literal value, a designator or a function result.

9.1.1 Literal Value

§ literal value = enumeration literal | character literal | string literal | integer literal | real literal | nil | null;

A literal value is the textual representation of a constant.

9.1.2 Function Designator

§ function designator = invoke statement;

Procedures that return values are know as functions and can be used in an expression. The returned value is used in the expression. The function designator is similar to a procedure invocation with the relevant parameters.

9.1.3 Variable/Constant Designator

§ variable designator = identifier;

§ constant designator = identifier;

Variables and constants are associated with an identifier when declared. These identifiers are used in expressions to refer to the value associated with the variable or constant.

$a * b + c$

Here the identifiers a , b and c are used in the expression are replaced with their values just like using algebra.

9.1.4 Attribute Designators

§ attribute designator = qualified identifier, at, identifier;

Attributes exist in Ada and are included in the language because they provide such important values and functions other languages omitted. This gives the language richness over the other languages.

Attributes are accessed using the attribute (“at”) symbol.

```
d@first
d@last
d@range
d@type
d@address
```

9.1.5 Array Element Designators

§ element designator = qualified identifier, open parenthesis, expression, {comma, expression}, close parenthesis;

When accessing array elements it is necessary to identify which elements are being referred. Like Ada array elements are accessed using parentheses enclosing element index.

The element index is an integer value.

```
d(1)
e(5)(6)
e(5,6)
```

Multiple dimensions may be referred to using a comma-separated list as in the last example.

Slices are a subset of an array. They may be identified using a sub-range expression where an element index would be used.

```
d([10..25])
e([10..25], [35..60])
```

The last example show a multi-dimensional slice.

9.1.6 Record & Variant Designators

§ field designator = qualified identifier, period, identifier;

Records are variables like any other variable and accessed using normal variable designators. The fields of records are accessed using normal qualified identifier. That is by using a full-stop character between the variable identifier and the field identifier.

```
d.day
d.month
d.year
```

Variants and their components are accessed in the same way that records and their fields are.

9.1.7 Pointer De-Referencing

§ pointer dereference = qualified identifier, circumflex;

Pointer variables store addresses to values of another type. To access the values being pointed-to it is necessary to de-reference the pointer (follow the address). This is done with the de-reference operator (represented by the circumflex character) after the variable identifier.

```
d^
e^.b // b is a record field
```

9.1.8 Type Designator

\$ type designator = qualified identifier;

Types in well-designed source code will be declared using meaningful identifiers.

```
integer
character
int16
date
day
```

9.1.9 Type Qualification

\$ qualification = type designator, period, open parenthesis, expression, close parenthesis;

Type qualification is necessary in situations where ambiguity can occur. Typical situation is where multiple enumeration type are defined and use the same identifier as a literal.

The qualification syntax is the type designator followed by a period (indicating a component of), followed by a value in parentheses that is to be qualified.

```
boolean.(false)
character.('x')
float.(6.4)
integer.(x)
```

9.1.10 Type Conversion

\$ conversion = type designator, open parenthesis, expression, close parenthesis;

Type conversion converts a value of one type into a value of another type. The difference between qualification and conversion is that checks are made to ensure the value is within legal range.

```
integer(x)
```

9.2 Operators

\$ singleton expression = operand | brackets | prefix operation | infix operation;

\$ brackets = open parenthesis, singleton expression, close parenthesis;

\$ prefix operation = prefix operator, singleton expression;

\$ infix operation = singleton expression, infix operator, singleton expression;

\$ prefix operator = plus | minus | complement | not;

\$ infix operator = plus | minus | multiply | divide | modulus | remainder | exponentiation | root | equal | not equal | less-than | less-than or equal | greater-than | greater-than or equal | element of | not an element of | subset | subset or

equal | not subset | superset | superset or equal | and | or | implies | xor |
equivalent | concatenation;

Operators come in two forms infix and prefix. The prefix operators include the sign identity of a number: plus or minus. Prefix operators have their symbol preceding the operand. Infix operators include arithmetic operators such addition, subtraction etc. and have their symbol between two operands. This is consistent with algebra.

Operators have different behaviour depending on the type of the operands but have these basic categories: arithmetic, bitwise logical, boolean logical and relational. The table “Table 1: Precedence of Operators” on page 55 listed the operators with their precedence.

Table 1: Precedence of Operators

Priority	Operator	Escape Seq.	Operation	Operand(s)	Returns
0	(...)		Parenthesis	Any	Any
1	+ − ~	\plus\ \minus\ \comp\	Positive Identity Negative Identity Bitwise Complement	Numeric Numeric Boolean	Same Same Boolean
2	× or * ÷ or / % %%	\multiply\ \divide\ \mod\ \rem\	Multiplication Division Modulus Remainder	Numeric Numeric Numeric Numeric	Same Same Same Integer
3	+ −	\plus\ \minus\	Addition Subtraction	Numeric Numeric	Same Same
4	↑ or ** √	\power\ \root\	Exponentiation Root	Numeric Numeric	Same Same
5	= ≠ or ≠= < ≤ or <= > ≥ or >= ∈ ∉ ⊂ ⊆ ⊄ ⊃ ⊇	\eq\ \neq\ \lt\ \lteq\ \gt\ \gteq\ \in\ \not in\ \subset\ \subteq\ \notsubset\ \superset\ \supseteq\	Equality Inequality less than less than or equal greater than greater than or equal member of not member of subset subset or equal not subset superset superset or equal	Any Any Any Any Any Any Value, Type Value, Type Types Types Types Types	Boolean Boolean Boolean Boolean Boolean Boolean Boolean Boolean Boolean Boolean Boolean Boolean
6	¬	\not\	Logical not	Boolean	Boolean
7	∧	\and\	Conjunction	Int. or Bool.	Same

Table 1: Precedence of Operators

Priority	Operator	Escape Seq.	Operation	Operand(s)	Returns
8	$\vee$	<code>\or\</code>	Disjunction	Int. or Bool.	Same
9	$\Rightarrow$ or $=>$	<code>\imp\</code>	Implication	Int. or Bool.	Same
10	$\underline{\vee}$ $\Leftrightarrow$	<code>\xor\</code> <code>\same\</code> or <code>\nxor\</code>	Exclusive Disjunction Equivalence	Int. or Bool. Int. or Bool.	Same Same
11	$\&$	<code>\concat\</code>	Concatenation	Arrays	Array
12	<code>f()</code> , identifier		function call, designators	Any	Any

The arithmetic operators perform the usual mathematical action. These are addition, subtraction, multiplication, division, modulus, remainder, exponentiation and root. The modulus and remainder operators perform integer division, discard the quotient and return the remainder in the fashion defined for Ada (Barnes 1998: 83).

The logical operators are not, conjunction, disjunction, implication, exclusive disjunction and equivalence. The table “Table 2: Truth Table for Operators” on page 56 gives the results of each operator. These are the usual logical action.

Table 2: Truth Table for Operators

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \underline{\vee} q$	$p \Leftrightarrow q$
false	false	true	false	false	true	false	true
false	true	true	false	true	true	true	false
true	false	false	false	true	false	true	false
true	true	false	true	true	true	false	true

The bitwise logical operators are the same as boolean operators. The table “Table 1: Result Table for Bitwise Operators” on page 38 shows the result of the bitwise operators.

The relational operators are equality, inequality, less-than, less-than or equal, greater-than, greater-than or equal. The equality and inequality test the two operands for the same value and return true or false accordingly. The order relations test the two operands for the usual ordinality property within the positional number line.

The membership operator’s test a value is within a type or sub-range.

There are subset and superset operators. The two operands are type designator or type definitions and the operators return true or false.

The concatenation operator is used to concatenate two one-dimensional arrays. This is defined similarly to Ada.

9.3 Tuple Expressions

§ tuple expression = open parenthesis, tuple value, {comma, tuple value}, close parenthesis;
 § tuple value = expression|named expression;
 § named expression = named values name, {vertical bar, named values name}, corresponds, expression;
 § named values name = qualified identifier|subrange expression;
 An **aggregate** or “**tuple** is a sequence of objects separated by commas and enclosed in parentheses.” (Pratt & Zelkowitz 1996: 604).

The tuple is used where multiway assignment or correspondence is required. The most common use of tuples is passing arguments to procedures and functions.

The standard tuple is generally called a positional tuple or aggregate because the position of a value within the tuple has significance within the context. Such as in a procedure invocation, the first argument corresponds with the first parameter, the second argument with the second parameter and so on.

This example shows a positional tuple being assigned to a two-dimensional array variable.

```
a: array(1..3,3..7) of int;
a := ((1,2,3,4,5), (6,7,8,9,10), (11,12,13,14,15));
```

However, there exists a named tuple or aggregate. This can be seen in Ada and has advantages of readability, comprehension and qualification where procedure overloading recreates ambiguities.

9.3.1 Named Tuple/Aggregate

The named aggregate is so called because each (not necessarily all) value within the tuple are identified. This is done by proceeding a value with an identifier followed by the corresponds symbol. Those familiar with Ada should note the difference of symbol.

```
write(item :> "Hello World!");
```

This example shows a procedure invocation with the argument being corresponded with a formal parameter called `item`.

It is possible to mix both positional and named values within the same tuple. The rule is that there must be a one-to-one correspondence between the tuple and its destination.

For more details on named aggregate see (Barnes 1998) or other Ada text.

9.4 Sub-Range Expressions

§ subrange expression = open bracket, subrange literal, double period, subrange literal, close bracket;

§ subrange literal = integer literal | character literal | enumeration literal | real literal;

A sub-range expression specifies a range of values. These values must evaluate to scalar types.

<code>[-128 .. 127]</code>	a range of integers
<code>['a' .. 'z']</code>	a range of characters
<code>[Monday .. Friday]</code>	a range of enumeration values
<code>[-10.0 .. 10.0]</code>	a range of real values

As can be seen from these examples, a sub-range expression is enclosed in square brackets and consist of two values separated by the *to* symbol. The two values must be of the same type.

9.5 Analysis

The expressions provide a means to express calculations in the familiar mathematical form this satisfies the high abstraction criteria. Orthogonality has been improved by generalising the use of tuple.

There are a high number of levels in the table of precedence of operators. This is comparable with C. The levels of precedence are arrange to minimise bracket use. Precedence is usual arithmetic (BODMAS), power operators are below arithmetic so brackets can be omitted, relational, logical and concatenation. The logical operator precedence (and symbols) was adopted from Z (Woodcock & Davies 1996), (Spivey 1992).

10 Statements

\$ statement = flat statement | nesting statement | variable declaration | entry visa;
Statements are divided into two categories: flat or nesting statements.

10.1 Statement Sequences

\$ statement sequence = {statement, semicolon}-;

Sequence is one of the three characteristics of an imperative language (Louden 1993: 11).

A statement sequence consists of one or more statements that are executed in order. If no action is required, the **null** statement may be used.

10.2 Nesting Statements

\$ nesting statement = block statement | if statement | case statement | loop statement | for statement;

Nesting statements are statements that contain nested statement sequences. These include the block, decision and loop statements.

10.2.1 Named Statements

All nesting statement may be named, this aids identification of a specific level of nesting for accessing hidden variables or exit and next statements.

With the exception of the block statement all nesting statement are closed with the **end** keyword followed by the statement keyword. This is mandatory for unnamed statements but for named statements it becomes optional. However, the same name must also appear after the closing keywords irrespective of the presence of the second closing keyword.

```
loop    ...    end loop;           // unnamed statement
12: loop ... end 12;              // named without closing keyword
13: loop ... end loop 13;         // named with closing keyword
```

This example shows the three possible naming/keyword combinations using the loop statement.

10.2.2 Block Statement

\$ block statement = block | named block;

\$ block = [declare, declaration section], block body, end;

\$ block body = begin, statement sequence, [exception handler];

\$ named block = identifier, colon, block body, end, identifier;

\$ exception handler = on, {exception clause}-;

\$ exception clause = vertical bar, qualified identifier, corresponds, statement sequence;

The block statement introduces a nested statement sequence and an optional declaration section. This allows local declarations and statements having a local scope and visibility.

The simplest block statement consists of the **begin** and **end** keywords enclosing a statement sequence.

```
begin
  x;
end;
```

To include a declaration section the **declare** keyword precedes the **begin** keyword. There must be at least one declaration if the **declare** clause is used.

```
declare
  type int16: integer(16);
  a: int16 := 0;
  b: constant := 23;
begin
  a := b;
end;
```

Here the declarations spring into life when the the block is entered, are visible during the block and die when the block is exited.

The following example shows the naming of the block statement.

```
X1:
declare
  ... // declarations
begin
  ... // statements
end X1;
```

The exception handler catches exceptions within the block. Details can be found in “12.3 Catching an Exception” on page 76.

10.2.3 If Statement

§ if statement	= unnamed if named if;
§ unnamed if	= if body, if;
§ named if	= identifier, colon, if body, [if], identifier;
§ if body	= if, boolean expression, then clause, {elsif clause}, [else clause], end;
§ then clause	= then, statement sequence;
§ elsif clause	= elsif, boolean expression, then clause;
§ else clause	= else, statement sequence;

The **if** statement is the first decision statement. It follows the syntax of Ada. The expressions following the **if** or **elsif** keyword are evaluated in order. They must evaluate to a boolean value. The first that is true the associated statement sequence following the **then** keyword is executed. If no expression evaluates to true the statement sequence following the **else** keyword is executed, if present.

There may be any number of **elsif** clauses.

```
if x<0 then
  ... // statements
```

```

elseif x>y then
    ... // statements
else
    ... // statements
end if;

```

10.2.4 Case Statement

§ **case statement** = unnamed case | named case;
 § **unnamed case** = case body, case;
 § **named case** = identifier, colon, case body, [case], identifier;
 § **case body** = case, expression, {item clause}-, end;
 § **item clause** = vertical bar, item expression list | default, corresponds, statement sequence;
 § **item expression list** = expression, {comma, expression};

The **case** statement is the second decision statement. The expression following the **case** keyword is evaluated. It must evaluate to an ordinal expression (enumeration, integer or character type). The guarded statement sequence with the same value as the expression will be executed.

The item clause expression must be static and type equivalent to the case expression. There must be one or more item clause.

```

case x
| 1 :> ... // statements
| 3 :> ... // statements
| 5, [10..52] :> ... // statements
| default :> ... // statements
end case;

```

The default item clause may appear anywhere within the list but may appear only once. The default clause is executed for values not explicitly listed.

10.2.5 Loop Statement

§ **loop statement** = unnamed loop | named loop;
 § **unnamed loop** = [loop condition], loop body, loop, [loop condition];
 § **named loop** = identifier, colon, [loop condition], loop body, [loop condition], identifier, [loop condition];
 § **loop condition** = (while | until), boolean expression;
 § **loop body** = loop, statement sequence, end;

The loop statements specify the repeated execution of a statement sequence. A loop statement may be unconditional or may specify optional conditions before and/or after the statement sequence.

10.2.5.1 Unconditional Loop

The simplest form of loop statement is the unconditional loop.

```

loop
    ...
end loop;

```

The unconditional loop is necessary to express the continuous repetition of statements where termination is not required. There are few situations where unconditional loop is required.

10.2.5.2 Loop Conditions

The loop statement has two condition clauses: *while* or *until*. Each are followed by a boolean expression. The *while* clause will repeat the loop if the boolean expression is true while the *until* clause will repeat the loop if the boolean expression is false.

To allow pre-test and post-test conditions, the condition clauses may be placed before or after the loop body. This allows the programmer to express the semantic equivalent of Pascal's *while* statement, *repeat until* statement and other variations.

The condition clauses along with the exit statement are necessary to express the termination of a loop.

10.2.5.3 Pre-test Loop

The pre-test loop is a condition clause followed by a loop body. The boolean expression is tested before the statement sequence is executed, so they may never be executed.

```
// continue looping while x<0 is true
while x<0 loop
    ...
end loop;

// until x<0 is true continue looping
until x<0 loop
    ...
end loop;

outer:
while x>0 loop
    inner:
        until y>x loop
            ...
        end loop inner; // optional keyword present
    end outer; // optional keyword omitted
```

10.2.5.4 Post-test Loop

The post-test loop is a loop body followed by a condition clause. Because the boolean expression is tested after the loop body the statement sequence is guaranteed to execute once.

```
// continue looping while y>50 is true
loop
    ...
end loop while y>50;

// until x<0 is true continue looping
loop
```

```

...
end loop until x<0;

```

10.2.5.5 Dual-test Loop

Although rare, there are occasions when both a pre-test and post-test are required.

```

while y>50 loop
...
end loop until x<0;

```

10.2.5.6 Mid-test Loop

The **exit** statement facilitates the testing and exiting from anywhere within the statement sequence. It may also exit from any level of nesting. Any number of exit statements may be used within the loop body.

```

11:
loop
  loop
    ...
    exit 11 if b;
    ...
  end loop
end 11;

```

10.2.6 For Loop Statement

§ for statement = for body | named for body;
 § named loop body = identifier, colon, for body, identifier;
 § for body = for, identifier, for range, loop body;
 § for range = (in, discrete range) | (assignment, discrete expression, to, discrete expression), [by, discrete expression];
 § discrete range = subrange expression | discrete definition;
 § discrete definition = scalar definition;

The **for** loop statement syntax is adapted from Modula-2 and Ada. There is an identifier which is the control variable. Actually the control variable emulates a constant and therefore may not be assigned to within the loop scope.

The simplest for statement is the same as Modula-2.

```

for i := 1 to 10 loop
...
end loop;

```

In this example the increment is assumed to be one. But for an integer range any integer may be specified for the increment.

```

for i := 100 to 1 by -10 loop
...
end loop;

```

Here the range is descending in multiples of ten. The values iterated over would be (100, 90, 80, 70, 60, 50, 40, 30, 20, 10). When the **by** clause is omitted the default is 1 or -1

depending on the direction of the range. For enumeration and character types the **by** clause is not allowed.

```
type color: enumeration (red|green|blue);
for i := blue to red loop
  ...
end loop;
```

In this example the enumeration is iterated over using the predecessor function attribute.

More complex ranges may be specified with the **in** clause. The range is specified with a sub-range expression, a type designator or a discrete type definition. The range must be of discrete type (enumeration, character or integer). The loop body will be iterated once for each value within the sub-range.

```
for i in [1..10] loop ... end loop;
for i in integer(4) loop // range is [-8..7]
  ... end loop;
for i in integer[1..10] loop ... end loop;
for i in int16@range by 100 loop ... end loop;
```

These examples show various ranges including the **by** clause being used to specify an increment. Note that the may only be ascending when using the **in** clause.

10.3 Flat Statements

§ flat statement = assignment statement | invoke statement | exit statement | next statement | return statement | goto statement | raise statement | propagate statement | null statement;

A flat statement is a statement that does not contain a nested statement sequence. They may not be named since they have no nested statements and therefore no need.

10.3.1 Assignment Statement

§ assignment statement = expression, assignment, expression;

Simple put the assignment statement evaluates an expression on the right of the assignment symbol and is copied into a variable on the left.

```
a := b;
```

This is complicated by tuple assignment or multiway assignment. This example shows a positional date tuple being assigned to a date variable.

```
type month: enumeration
  (January|February|March|April|May|June|
   July|August|September|October|November|December);
type date: record
  day: integer [1..31];
  month: month;
  year: int;
end date;
```

```
Abdication: date := (6, April, 1814) // Abdication of Napoleon
```

Named tuples may also be used. This example a named date tuple being assigned to a date variable.

```
Waterloo: date := (month:>June, day:>18, year:>1815) //
Defeated
```

Named tuples being assigned to arrays use the element index as the name.

```
a: array(1..5) of int := (1 :> 15, 5 :> 57, 4 :> 45, 2 :> 5, 3
:> 6);
```

Also multiple assignment may be reduced when the same value is to be used for many elements.

```
d: array(day) of int;
d := (sat|sun :> false, [mon..fri] :> true);
```

This example shows the array being assigned false to the days sat & sun but true for the range.

Functions may also return tuples.

```
procedure f(a,b,c: int)
  return (d,e,f: int):
begin
  return (a,b,c);
end f;
...
a,b,c: int;
(a, b, c) := f(d, e, f);
```

“This notation eliminates the asymmetry (non-orthogonality) of functions in most programming languages, which allow an arbitrary number of arguments, but only a single return.” (Scott 2000: 262)

Multiway assignment should also honour swapping without misbehaving.

```
(a, b) := (b, a);
```

is semantically equivalent to:

```
t := a; a := b; b := t;
```

where *t* is a temporary variable.

10.3.2 Invoke Statement

\$ invoke statement = qualified identifier, [actual parameters];

\$ actual parameters = open parenthesis, expression, close parenthesis;

The invoke statement is where a procedure is called upon to be executed. First the arguments are evaluated and then passed to the procedure on invocation.

```
write("Hello World!");
```

The arguments are actually tuple expressions that are explained in the chapter “9 Expressions” on page 52.

As with Ada, tuple arguments can use the corresponds symbol to name arguments. These are called named parameters. This is very useful when using default argument or the arguments in a different order.

```
write(item :> "Hello World!");
```

10.3.3 Exit Statement

\$ exit statement = exit, [qualified identifier], [if, boolean expression];

The **exit** statement may be used to exit any nested statement. Effectively it is a goto statement jumping to the statement following the current or named statement. It may not be used to exit a procedure.

```
outer:
loop
  inner:
  loop
    ...
    exit outer if b;
    ...
  end inner;
end outer;
```

This example shows the exiting of a named statement and based on a condition.

10.3.4 Next Statement

\$ next statement = next, [qualified identifier], [if, boolean expression];

The **next** statement may be used within loop statements to initiate the next iteration. Basically it jumps to the end of the current or named loop. The loop control tests if another iteration should be performed.

10.3.5 Return Statement

\$ return statement = return, [expression];

The **return** statement exits a procedure. If the procedure is a function the **return** clause must contain a compatible expression.

```
return a+b*c;
```

10.3.6 Goto Statement

\$ goto statement = goto, identifier | label declaration;

\$ label declaration = identifier, colon, label;

The **goto** statement is provided for those rare occasions where its use necessary. The language has low insulation and there is no performance benefit from using the **goto** statement but clarity of the code may be improved.

```
goto done;  
...  
<<done>>
```

In this example the **goto** statement jumps to the label declaration. It is not possible to jump into a nested statement.

The exact rules for the **goto** statement and destination label positioning is the same as Ada (Barnes 1998).

10.3.7 Raise Statement

The **raise** statement initiates an exception. Exceptions are explained in “12 Exceptions” on page 75.

10.3.8 Null Statement

The null statement does nothing. It is provided for explicitly indicating no action.

```
begin  
  null;  
end;
```

10.4 Variable Declaration Statement

Besides declaring variables within the declaration sections, variable and only variable may be declare anywhere a statement may. The variable has the scope, visibility and lifetime of the enclosing scope. It cannot be accessed before the declaration. This is more secure because it reduces the risk of variables being used before they are initialised.

```
begin  
  x;  
  a: integer(32) := b;  
  y;  
  z;  
end;
```

10.5 Entry Visa Declaration Statement

The entry visa declaration may be in a declaration section or a statement sequence and is covered in “12.4 Entry Visa” on page 77.

10.6 Analysis

Collectively the statements provide a high degree of abstraction. The block statement provides encapsulation and flexibility by allowing declarations to be nested. The nesting statements provide a high degree of orthogonality by bracketing a statement sequence with a closing keyword(s). The assignment statement provides a high degree of orthogonality and flexibility by allowing both single and tuple value assignment.

The loops are more flexible than other languages. The general loop statement has been extended through generalising the conditional clause. The specialised loop (**for** statement) has been extended through allowing both the Modula-2 and the Ada styles syntax.

Nesting statements may be named but flat statements cannot, this is non-orthogonal. The name is associated with the level of nesting for use with the **next** and **exit** statements and so will never be needed for flat statements.

The variable declaration statement is a non-orthogonal form of declaration but improves security reducing the risk of using a variable before it is initially assigned.

11 Procedure Declarations

§ procedure declaration = procedure interface, [colon, procedure body], semicolon;

§ procedure interface = procedure, qualified identifier, [formal parameters], [function return], [exceptional return], [attribute definition];

§ procedure body = procedure block, end, qualified identifier;

§ procedure block = [declaration section], block body;

Sub-programs are a means of reusing code that would otherwise be repeated throughout a program and reducing a program text into something manageable. The abstraction for subprograms is the procedure.

A procedure declaration consists of an interface and a body. The procedure interface may be declared without the body within a module's interface section or elsewhere as a forward declaration enabling invocation within a procedure before full declaration.

The use of procedures is through the procedure invocation statement.

11.1 Procedure Interface

The procedure interface consists of four sections identification, formal parameters, formal return (functions) and exceptional return (exit visa).

All procedure declarations start with the identification section. This consists of the **procedure** keyword followed by the identifier. The other sections are optional, the absence of a section indicates nothing in that section.

```
procedure writeln;
```

For the purpose of overloading the entire interface is used for identification and resolution with the exception of parameter defaults.

11.2 Procedure Body

The procedure body is where the action is specified. It is basically a block statement but without the **declare** keyword and the procedure identifier follows the **end** keyword. As such it can contain declarations, a statement sequence and an exception handler.

```
procedure hello:
begin
    write("Hello World!");
    writeln;
end hello;
```

Any variables declared in a procedure body are unless otherwise specified are created upon entry into the relevant block in the same way as Pascal and Ada.

11.3 Formal Parameters

§ formal parameters = open parenthesis, [formal parameter list], close parenthesis;
 § formal parameter list = formal parameter, {semicolon, formal parameter};
 § formal parameter = identifier list, colon, [in], [out], type definition, [default clause];
 § identifier list = identifier, {comma, identifier};

Formal parameters are the specification of parameters that will be provided at invocation.

Formal parameter syntax is similar to Ada. Like Ada formal parameters may be specified as **in**, **out** or **in out**. These relate to the copy-in and copy-out behaviour. The default when not specified is **in**.

It is an error to assign a value to **in** parameters as they behave like constants. **out** parameters provide one form of returning values back to the invoking procedure.

```
procedure proc(a: int; b: in int; c: out; d: in out int):
begin
    c := d;
    d := b+a;
end proc;
```

11.3.1 Default Parameters

Default values may be specified for parameters. This allows parameters to be omitted at invocation.

```
procedure proc(a: int := 0):
begin
    ...
end proc;

...
proc; // parameter omitted default will be assumed
```

When an invocation omits a parameter the default will be assumed provided that there is no ambiguity with regards to procedure overloading.

11.3.2 Parameter Attributes

There are two physical parameter-passing mechanisms. The first is pass-by-value, here the parameter value is simply copied into the stack frame or activation record. The second is pass-by-reference, here the address of the variable is passed to the called procedure.

It is possible using an attribute definition to specify either mechanism for any parameter.

```
procedure proc(a: int with(pass_by := reference);
    b: in out int with(pass_by := value)):
begin
    b := a;
end proc;
```

These like other realisation facilities must be used with caution and are provided to give the expert greater control.

11.3.3 Open Arrays

Array parameters may be variable length. This allows general procedures to be defined to work on arrays of varying length.

```

procedure sum(a: array integer(32)[] of int16)
  return int64:
    t: int64 := 0;
begin
  for i in a@range loop
    t := t + a(i);
  end loop;
  return t;
end sum;

```

This procedure totals the integer values store in an array. The possible array index may be undefined (open) but the size of the ordinal type must be known.

11.4 Functions

\$ function return = return, type definition | formal tuple return;

\$ formal tuple return = open parenthesis, formal tuple parameter, {semicolon, formal tuple parameter}, close parenthesis;

\$ formal tuple parameter= [identifier list, colon], type definition, [default clause];

Using **out** parameters is an effective means to return values from procedures. It has the disadvantage that the procedure must be called first as a statement, before the returned values may be used as constituents of an expression.

A function is a procedure that returns a value or tuple in such a fashion that the function may be used within an expression. This overcomes the problem with using **out** parameters. To create functions use the formal return section to specify the return type.

```

procedure sign(x: int) return [-1..1]:
begin
  if x<0 then
    return -1;
  elsif x>0 then
    return 1;
  else
    return 0;
end sign;

```

A procedure that has a **return** clause is called a function, despite the use of the **procedure** keyword. It is called a function because it returns a value that may be used as a constituent within an expression.

11.4.1 Tuple Return

Besides returning a single value it is possible to return a tuple.

```

procedure f(a,b,c: int)
  return (int,int,int):
begin
  return (a,b,c);
end f;
...
(a, b, c) := f(d, e, f);

```

“This notation eliminates the asymmetry (non-orthogonality) of functions in most programming languages, which allow an arbitrary number of arguments, but only a single return.” (Scott 2000: 262)

11.4.2 Named Returns

By specifying a tuple as the function return, it is possible to name the constituents. This allows the return statement to be given a named tuple. This may aid readability.

```

procedure f(a,b,c: int)
  return (d:int):
begin
  return a;
end f;

procedure g(a,b,c: int)
  return (d,e,f: int):
begin
  if e.first > 5 then // named return allows access to
    attributes
    ...
  end if;
  return (e => a, f => b, d => c);
end g;
...
(a, b, c) := f(d, e, f);

```

Named returns are not assignable or readable. Thus, this is illegal:

```

procedure f(a,b,c: int)
  return (d: int):
begin
  if d=5 then // illegal: d is not readable
    ...
  end if;
  d:=a; // illegal: d is not assignable
  return a;
end f;

```

The **return** statement must contain an expression that is compatible with the formal return declaration.

11.4.3 Exit Visa

The exceptional return section relates to exception and is discussed in “12.2 Exit Visa” on page 76.

11.5 Overloading

Overloading procedure is available enabling multiple procedure to be defined with the same identifier. The invocation determines which is procedure is referred based on the full procedure interface. That is parameters (number, names and types), return type and exceptional returns.

```
procedure write(x: integer(32));
procedure write(x: cardinal(32));
procedure write(x: string);
...
begin
  write("A string");
  write(cardinal.(56));
  write(-100);
end s;
```

The second write invocation needs the qualification otherwise it would report an ambiguity error.

11.6 Operators

It is not possible to create new operators but the predefined operators may be overloaded by specifying a string literal for the identifier containing the operators symbol. This is the same as Ada’s operator overloading.

```
procedure "+"(a,b: point) return point:
begin
  return (a.x+b.x, a.y+b.y);
end "+";
```

This overloads the addition operator with a translation operation on a Cartesian co-ordinate type.

11.7 Analysis

Procedures are one of the most fundamental abstractions a programming language can provide. It is a form of reuse: define a procedure once and invoke it many times as required, which improves reliability.

Overloading and default parameters provide for greater readability and flexibility by allowing well-known identifiers such as read, write or printf or operators to be used with

different parameter types for similar actions. This provides a high degree of orthogonal; things that look similar behave similar.

Procedures have a high degree of orthogonality by allowing zero, one or a tuple value to be returned and having the same syntax for procedures and functions.

A high degree of security is provided by the exit visa only allowing declared exceptions to propagate out of the procedure. This improves reliability by the fact that exceptions cannot be hidden.

12 Exceptions

Software today is often found to be lacking robustness. This is reported frequently in the trade press (Computing and Computer Weekly). Robustness is the design criteria of security and reliability (Louden 1993: 234): a program must be able to recover from an error or fail in a controlled fashion.

Experience tells me that for any well-designed program around 75% of all lines of code will perform error detection and resolution. This is a rule-of-thumb that emphasises the importance of error handling.

Languages provide exception-handling abstractions for error handling. Unfortunately current abstractions and current practice tend to be lax. The problem is that current abstractions allow exceptions to be overlooked; C++ being the worst example. In this case the exception would propagate out of the program. The extreme example is a plane crashing due to an exception occurring that has been overlooked within the flight control software.

Given the pervasive nature of error handling and lax design of exception abstractions one of the design objectives was to minimise their use and to tighten up the abstraction. The basic idea is that exceptions are considered as people travelling across international borders and to cross borders one must possess entry and exit visas.

12.1 Raising An Exception: Raise Statement

§ `raise statement` = `raise`, qualified identifier;

Exceptions follow C++'s practice of being types; this allows additional values to be passed.

Therefore, to declare an exception one could use the declaration:

```
type divide_by_zero: record
end divide_by_zero;
```

This is an empty record and so does not pass any other information.

Exceptions are raised with the **raise** statement. The following the **raise** keyword is an identifier, which must be a variable or type designator. If the identifier refers to a variable the type is used as the exception identifier.

```
raise divide_by_zero;
```

The **raise** statement behaves similarly to the **return** statement in that it exits the procedure: what is called an exceptional return.

Adding other information one could:

```
type ioerror: enumeration (file_not_found, access_denied,
...);
```

```
...
ioerr: ioerror := file_not_found;
...
raise ioerr;
```

Then extract the information when handling the exception.

12.2 Exit Visa

For any exception to exit a procedure there must exist an exit visa. Thus, this procedure would cause a compiler error.

```
procedure proc_a:
begin
    raise divide_by_zero; // not allowed: no exit visa
end proc_a;
```

The exit visa comes in the form of an exceptional return declaration as part of the procedure's interface. The exceptional return declaration consists of the **raise** keyword followed by a comma-separated list of *all* the exceptions that exit the procedure.

```
procedure proc_a raise divide_by_zero:
begin
    raise divide_by_zero;
end proc_a;
```

12.3 Catching an Exception

When an exception occurs there needs to be some means to catch it and handle the error. Like Ada this is part of the block statement located after all the statements, this is called the exception or resolution section. It looks like the body of a case statement but with the **on** keyword.

```
begin
    ...
on
| divide_by_zero:> ... ;
| lower_bound_violation:> ... ;
| upper_bound_violation:> ... ;
end proc_a;
```

Like C++ it is possible to give an instance (variable) identifier for an exception. This allows the values (if any) to be accessed.

```
begin
    ...
on
| e: divide_by_zero:> ... ;
| lower_bound_violation:> ... ;
| upper_bound_violation:> ... ;
| e: ioerror:>
    if e = file_not_found then ... end if;
```

```

    ... ;
end proc_a;

```

Note: the same identifier may be used because like the case statement the variable is declared locally to only that clause.

12.4 Entry Visa

\$ entry visa = admit, qualified identifier list;

For exceptions to be allowed entry into a block there must be permission in the form of the entry visa. The entry visa is basically a declaration listing the exceptions allowed entry.

```

    admit ea, eb;

```

It may appear anywhere within a block including the declaration section and statement sequence. It stays valid until the next entry visa or the end of the block's statement sequence.

When a procedure is invoked that has an exceptional return declaration, the compiler will report an error unless the entry visa lists all of the exceptions raised. Also the same exceptions must be caught in the resolution section: propagation by default is not permitted.

```

procedure a raise ea, eb, ec;
procedure r:
begin
    admit ea, eb, ec;
    a;
on                                // handler
| ea  :> ... ;
| eb  :> ... ;
| ec  :> ... ;
end r;

```

Entry visas only apply to procedure invocations. It is an error when a procedure raises an exception without it being listed in the current entry visa or when a procedure invocation raises no exceptions when an entry visa is *not* empty.

This code fragment gives a good example of the entry visa.

```

procedure a raise ea;
procedure b raise eb;
procedure c;
procedure d raise ea, eb, ed;

procedure r:
    // default is admit no exceptions
begin
    c;
    admit ea, eb; // statements must raises at least one of these
    a;
    b;

```

```

    admit; // required: c raises no exceptions
    c;
    admit ea as eda, eb as edb, ed; // rename some for visibility
    d;
on                                // handler
| ea    :> ...;
| eb    :> ...;
| eda   :> ...;
| edb   :> ...;
| e:ed  :> ...;
end r;

```

The procedure `c` does not raise any exceptions and therefore the entry `visa` is empty. For statement `d` the exceptions raised are renamed locally so they are visibly separate the same exception raised by previous statements.

It is an error for any exceptional return to be omitted from the exception handler.

12.5 Propagation: Propagate Statement

§ `propagate statement` = `propagate`, qualified identifier;

As noted previously, propagation does not happen by default. Thus, this code would produce an error for not catching the exceptions raised by the invocation of `a`.

```

procedure a raise ea, eb, ec;
procedure r:
begin
    admit ea, eb, ec;
    a; // error: must catch exceptional returns
end r;

```

The **propagate** statement is provided to explicitly propagate an exception out of the current procedure.

```

procedure a raise ea;
procedure r raise ea:
begin
    admit ea;
    a;
on
| ea :> propagate;
end r;

```

Like the **raise** statement the **propagate** statement exits the current procedure with the same exception.

12.6 Predefined Exceptions

There are a number of predefined exceptions and the following is not an exhaustive list. Implementation may also result in others being defined.

The `upper_bound_violation` and `lower_bound_violation` are raised when a value or array index exceeds the legal range.

The IEEE floating-point (BS 7237:1990) standard defines five exceptions. These are `divide_by_zero`, `overflow`, `underflow`, `inexact` and `invalid_operation`.

These are all defined with an empty record type.

12.7 Analysis

The aim for the exception mechanism was to improve security and reliability by preventing exceptions that propagate out of the program whilst being unknown to the developers. This has been successfully achieved by the entry and exit visas and the rule requiring propagation to be explicitly specified by the `propagate` statement.

13 Generics Declarations

\$ generic declaration = generic type|generic procedure|generic module;

Genericity is an important feature of any language. It provides a form of compile-time parameterisation of modules, procedures and types very similar to macros in some languages. This is static polymorphism unlike dynamic polymorphism which provides type extension and inheritance hierarchy. It is less flexible but is inherently more reliable because of the compile-time checking.

13.1 Generic Types

\$ generic type = generic, generic parameters, type definition;

Generic declarations are similar to ordinary declarations with the addition of the generic clause. The generic clause consists of the keyword **generic** followed by the parameters.

```
generic (item_t: type (integer|float|fixed|decimal))
type point: record
  x, y: item_t;
end point;
```

This example shows a generic cartesian co-ordinate type. The actual type of the x and y components is to be specified later but must be a numeric type. The parameter `item_t` has a type of type but the optional constraint restricts the type to one of those specified in the constraint (numeric).

To use the generic point type a specific type must be created. This is done with the specific clause.

```
type point_int: specific point(int);
```

Here the type `point_int` is created which uses `int` as the numeric type. It would also be possible to declare a variable straight from the generic type.

```
a: specific point(int);
```

13.2 Generic Procedures

\$ generic procedure = generic, generic parameters, procedure declaration;

Generic procedures can be declared in similar fashion by preceding the generic declaration before the procedure declaration.

```
generic (item_t: type)
procedure exchange(x,y: in out item_t):
  z: item_t := x;
begin
  x := y;
  y := z;
end display;
```

This example shows a procedure taking two parameters that have a generic type. When the procedure is specialised the type is given.

```
procedure swap_int: specific exchange(int);
procedure swap_int: specific exchange(date);
```

This example shows the generic procedure being specialised, creating two procedures applying different types.

Sometimes procedure interfaces are declared prior to the full declaration. For example, in the interface section of a module or to allow the procedure to be used prior to declaration.

```
// declared in an interface section
generic (item_t: type)
procedure exchange(x,y: in out item_t);
```

This example shows a procedure interface declaration for the previous example.

13.3 Generic Modules

\$ generic module = generic, generic parameters, module declaration;

Information structures or containers are a good application of generics where a generic stack or queue could be declared but used containing different types. Generic module declarations follow similar pattern.

```
generic (item_t: type)
module stack:
  export is_empty, top, pop, push;

procedure is_empty return boolean:
begin
end is_empty;

procedure top return item_t:
begin
end top;

procedure pop return item_t:
begin
end pop;

procedure push(item: item_t):
begin
end push;

initialization
begin
end initialization;

finalization
begin
end finalization;
```

```
end stack;
```

This example shows a possible generic stack module that may be specialised to contain any type.

```
module int_stack: specific stack(int);
module chr_stack: specific stack(char);
```

This example shows two modules being specialised. If these declarations are placed in a procedure declaration section then the modules would have totally local storage. Recursive procedures would create multiple instants just like variables they would be initialised and finalised as per scope rules. In this way modules behave very much like classes. The `lifetime` attribute may be used to create a static module.

Modules come in a few varieties and generics work the same for all. The generic clause would precede the module declaration which ever is used.

```
generic (item_t: type)
module interface stack:

  procedure is_empty return boolean;
  procedure top return item_t;
  procedure pop return item_t;
  procedure push(item: item_t);

end stack;
```

This example shows a module interface, the module implementation would have similar declarations except the full implementation would be given.

The specialisation would only have the one declaration and not one for separate interface and implementation modules.

13.4 Generic Parameters

```
$ generic parameters = [open parenthesis, generic parameter list, close parenthesis];
$ generic parameter list= generic parameter, {comma, generic parameter};
$ generic parameter = identifier list, colon, generic parameter type;
$ generic parameter type= generic value parameter|generic type parameter|generic module
                           parameter|generic specific parameter;
```

There are five kinds of parameters that may be used in the generic clause. These are values, variables, types, procedures and modules.

13.4.1 Value Parameters

```
$ generic value parameter= [in], [out], type definition, [default clause];
```

Values parameters are similar to procedure `in` parameters. These are constant values that are arguments at the specialisation of a generic. The `in` keyword is the default and so is optional.

```

generic (max: cardinal(32); item_t: type)
module queue:
  export ...;
  type q: array integer[0..max] of item_t;
  ...
end queue;

```

This example shows the use of a value parameter (`max`) which is used to define the size of an array within the module. The value must be of type `cardinal(32)`.

13.4.2 Variable Parameters

Variable parameters like Ada are declared like procedure **in out** parameters. Suppose the following procedure outputs a new line to the console and it also increments the variable keeping track of the line number.

```

y: cardinal(32) := 0;
...
procedure newline:
begin
  ...
  y := y + 1;
end queue;

```

The generic version of this would permit the variable to be a generic argument.

```

generic (line: in out cardinal(32))
procedure newline:
begin
  ...
  line := line + 1;
end queue;

```

Here the procedure increments the generic variable parameter. Now the specialisation would allow the actual variable to be incremented to given has a parameter.

```

x: cardinal(32) := 0;
y: cardinal(32) := 0;
...
procedure newline_x: specific newline(x);
procedure newline_y: specific newline(y);

```

In this example there are two newline procedures each with its separate variable that is incremented.

13.4.3 Procedure Parameters

Like Modula-2 the language provides closures or *procedure types*. This allows variables and parameters to be of procedure types. Therefore generic parameters of procedure type are no different than procedure parameters of procedure type. Procedure types may be value or variable parameters.

13.4.4 Type Parameters

\$ generic type parameter= type, generic type restriction;

Type parameters allow types to be specified as arguments to the specialisation of a generic. The previous examples have shown the use of type parameters.

```
generic (item_t: type)
procedure exchange(x,y: in out item_t):
    z: item_t := x;
begin
    x := y;
    y := z;
end display;

procedure swap_int: specific exchange(int);
procedure swap_int: specific exchange(date);
```

As shown in this example the type parameter as the keyword **type** for the parameters type.

13.4.5 Module Parameters

\$ generic module parameter= module, generic module restriction;

Modules may be generic parameters. The syntax is similar to the type parameters.

```
generic (m: module)
procedure p:
    import m. ...;
begin
    ...
end display;
```

A module is then given as an argument at specialisation.

13.4.6 Generic Specialised Parameters

\$ generic specific parameter= specific, [formal type list], [formal return], [exceptional return], [attribute definition];

It is possible to specify generics as parameters, this restricts the parameter to a specialisation of a generic. The generic being a type, procedure or module.

```
generic (s: specific stack)
generic (s: specific stack(int))
```

13.4.7 Parameter Restrictions

\$ generic type restriction= [open parenthesis, generic restricted type, {vertical bar, generic restricted type}, close parenthesis];

\$ generic restricted type= type definition|generic specific parameter;

\$ generic module restriction= [open parenthesis, generic restricted module, {vertical bar, generic restricted module}, close parenthesis];

\$ generic restricted module= qualified identifier|generic specific parameter;

Type and module parameters may be restricted using syntax similar to an enumeration type definition.

```
generic (item_t: type (integer|float|fixed|decimal))  
generic (s: module (stack_int|specific stack|specific  
stack(int)))
```

The first example restricts the type parameter to one of the specified numeric types. The second example restricts the module parameter to a variety of stack modules.

13.5 Analysis

The generics provide a high degree of abstraction, generality, flexibility and scalability. Generalisations of containers such stacks, queues and dequeues (Knuth 1997: 238-243) can be created without specifying the contained type. Specialisation provides the extra information to create a concrete container.

14 Analysis & Conclusions

Some of the aims of the language cannot be achieved by the language design but by the implementation. These are efficiency and insulation, to achieve these aims the compiler should produce tight fast machine code.

A high degree of abstraction, portability and especially flexibility has been achieved. In every area of the language flexibility has been improved in comparison with other languages.

Types provide greater realisation facilities than available in other languages. Ada being the only language that managed to provide these previously.

Orthogonality has been improved over many languages but without going too far. Algol-68 (Scott 2000: 256-257) went too far by not distinguishing between statements and expressions.

There is a high degree of scalability and evolvability provided by the module abstraction. Generics provide for a high degree of generality and scalability.

I have improved security and reliability over other languages by providing border controls for exceptions and a solution to the buffer overflow bugs (Bates 2004) that result in exploitation by malicious software and users.

I believe the language is less cryptic and has a more consistent syntax with an aesthetic nature that is an improvement over languages such as C and C++.

Ada when implemented was found to be very difficult. It was my aim to adopt some features from Ada and simplify their implementation. The proof of the pudding is in the tasting, as they say. This means that the implementation of this language will be the ultimate test of a successful design. In some cases, I have omitted some features specifically because they have not been tried and tested in other languages and I have no idea of their feasibility.

The language syntax is described formally with EBNF but the semantics are informally described. Ideally the semantics would be described using formal semantics but time constraints prevented this from happening.

I believe the language strikes a middle ground between the simplicity of Modula-2 and the complexity and power of Ada.

Altogether I believe the design of the language successfully achieves the aims of the language.

To implement the language only the compiler and linker need to be developed. A build tool is not required, as this functionality will be provided by the compiler. A debugger will be useful to the developer but not required. An editor or integrated development environment may be developed to handle the operator symbols but this is not required.

14.1 Original Contribution

I believe the design principle *insulation* was used for the first time in the part 1 report (Wakerley 2004) by myself. Insulation relates to the measurement between source code and the underlying machine code. Where assembly language has low insulation, compiled software would have low to high insulation depending on the code optimisation and interpreters would have high insulation because they execute many more machine code instructions for the same source code.

The entry visa relating to border controls for exceptions is an original design because there was no previously existing language feature that has similar use.

The complete design of the programming language described in this dissertation is a synthesis of mainly pre-existing language features that has never been done before and therefore constitutes an original contribution to knowledge (Phillips & Pugh 2000: 63).

15 Recommendations for Further Work

15.1 Formal Semantics

Describe the semantics formally using any of the techniques available such as denotational semantics (Schmidt 1986). Other methods are specified in (Schmidt 1986: 3-5) and (Nielson & Nielson 1992: 209-211, 233-234). This prevents mis-interpretations and allows implementation of the language to be proven correct.

15.2 Runtime Support

Unlike Pascal and like Modula-2, Ada and C the language does not provide built in input and output facilities for accessing the file system for example. A standard runtime support library provides these facilities. This library needs to be developed and would best be developed in connection with developing the compiler.

15.3 Annexes

The annex facility was designed for extensibility. Here are some ideas for annexes.

15.3.1 Real-Time Annex

An annex can provide concurrency and real-time abstractions such as threads (tasks for those familiar with Ada) and monitors, semaphores and mutexes.

More details are available in (Burns & Wellings 2001).

15.3.2 Object-Oriented Annex — Class Type

Another annex can provide object-oriented features such as a class type. The class type was deferred in light of the research suggested in “15.5.1 Encapsulation/Information Hiding” on page 90.

Java, C++ and C# are currently popular and whilst there are many factors in their success, some is related to their adoption of object-oriented features, such as the *class*. Object-oriented features do provide some very useful abstractions but there are also disadvantages. The inheritance and type extension mechanism also relaxes the strong typing system. Readability is reduced because inherited identifiers are used with no obvious declaration, one must explore the whole inheritance hierarchy. Bloat and efficiency is a major problem today, which goes unaddressed because of the power of current computers and upgrade expectations. This is not a valid reason for bloat and there is some distance to go for ethical behaviour within the industry.

Despite the disadvantages I believe for a language to be popular and successful it should have these features but with an emphasis on using them sparingly. This is contrary to current practice and beliefs.

It is the intention to provide a class type based on C++, Java and Delphi but not in this version. There are further research I would like to do into information hiding. It is anticipated that the class type will be in an annex.

This example code is a glimpse of the future class type.

```
generic (item: type)
type stack: class
  inherits a,b,c;
  function top return item public;
  function pop return item public;
  procedure push(x: item) public;
end stack;
```

15.4 Further Research

Designing the language raised some areas for further research. Some research related to feedback from general use of the language. This can only be performed after the language is implemented and been in use for some time when the irritations can be identified. This may best be done from a position of hindsight.

15.4.1 Closing Keywords

With named statements like:

```
inner:
loop
...
end loop inner;
```

Should the second closing keyword (**loop**) be compulsory, optional or omitted? Since the identifier uniquely identifies the statement (that is its purpose).

15.4.2 Nesting Identifier Rules

With named statements like:

```
a:loop
  a:loop
    ...
    exit a.a if ...;
    ...
  end loop a;
end loop a;
```

Should extra rules be made, preventing the same identifier being use within a set of nested statements.

The argument for, is the purpose of naming statements is to identify between different nestings and as such should not be allowed to be duplicated.

The argument against, is adding extra rules specifically for names of statements will break orthogonality of identifier visibility.

15.5 Future Research

This research is not specifically related to this language and may well be used elsewhere.

15.5.1 Encapsulation/Information Hiding

```
type rec:record
  a: int16;
  b: int16;
  c: int16;
  d: int16;
end r;
procedure x(ref r:rec):
begin ...
end x;
procedure y(ref r:rec):
begin ...
end y;
```

Design a syntax that would permit `c` and `d` to be inaccessible, read-only, write-only or read-write from `x` whilst something else from `y`.

References

- BS7145:1990, *The Programming language Ada*, BS 7145:1990 EN 28 652:1989 ISO 8652:1987, British Standards Institute (BSI)
- BS7237:1990, *Specification for Binary floating point arithmetic for microprocessor systems*, BS 7237:1990 IEC 559:1989, British Standards Institute (BSI)
- BS9899:1999, *Programming languages — C*, BS ISO/IEC 9899:1999, British Standards Institute (BSI)
- ISO7185:1990, *Programming languages — Pascal*, BS EN 27185:1992 ISO/IEC 7185:1990, British Standards Institute (BSI)
- ISO8652:1995, *Ada Reference Manual*, ISO/IEC 8652:1995(E) with Technical Corrigendum 1, Intermetrics, Inc., The MITRE Corporation, Inc., U.S. Government.
- BS EN 27185:1992 ISO/IEC 7185:1990, British Standards Institute (BSI)
- ISO10206:1991, *Programming languages — Extended Pascal*, BS ISO/IEC 10206:1991, British Standards Institute (BSI)
- ISO10514-1:1996, *Programming languages — Part1: Modula-2, Base Language*, BS ISO/IEC 10514-1:1996, British Standards Institute (BSI)
- ISO14882:1998, *Programming languages — C++*, ISO/IEC 14882:1998(E), International Standardization Organization (ISO), International Electrotechnical Commission (IEC)
- ISO14977:1996, *Syntactic Metalanguage — Extended BNF*, BS ISO/IEC 14977:1996, British Standards Institute (BSI)
- Barnes, J. (1998) *Programming in Ada 95*, 2nd Edition, Pearson Education Ltd.
- Bates, R., (2004) ‘Buffer Overrun Madness’, *ACM Queue*, 2(3) pp.12,14, May 2004
- Borland (1988-94) *Turbo Pascal 3.0-7.0 Reference Manuals*, Borland International
- Borland (1995) *(Delphi 2.0) Object Pascal Language Reference*, Borland International
- Burns, A., Wellings, A. (2001) *Real-Time Systems and Programming Languages*, 3rd Edition, Pearson Education Ltd.
- Knuth, D., E., (1997) *The Art of Computer Programming Vol 1.*, 3rd edition, Addison Wesley Longman
- Louden, K., C. (1993) *Programming Languages: principles and practice*, PWS Publishing Company
- Nielson, H., R., Nielson, F. (1992) *Semantics with Applications: A Formal Introduction*, John Wiley & Sons Ltd.

- Phillips, E., M., Pugh, D., S., (2000) *How to get a Ph.D.: a handbook for students and their supervisors*, 3rd edition, Open University, McGraw-Hill Education
- Pratt, T.W. & Zelkowitz M.V. (1996) *Programming Languages: Design and Implementation*, Prentice-Hall, Inc.
- Schildt, H., (2000) *C: The Complete Reference*, 4th edition, The McGraw-Hill Companies
- Schildt, H., (2001) *Java 2: The Complete Reference*, 4th edition, The McGraw-Hill Companies
- Schmidt, D., A. (1986) *Denotational Semantics*, Allyn and Bacon, Inc.
- Scott, M., L., (2000) *Programming Languages Pragmatics*, Morgan Kaufmann Publishers
- Spivey, J., M. (1992) *The Z Notation: a reference manual*, Prentice Hall International (UK) Ltd
- Stroustrup, B. (2000) *The C++ Programming Language*, Special edition, Addison Wesley Longman
- Tichy, W., F., Eggert, P. (1997) *RCS — A System for Version Control 5.7*, 1.1 Ed., available as Open Source via internet.
- Wakerley, D. (2004 May) *A Case Study in Programming Language Design-Part 1*, University of Bradford
- Wall, L., Christiansen, T., Orwant, J. (2000) *Programming Perl*, 3rd edition, O'Reilly & Associates
- Wirth, N. (1982) *Programming in Modula-2*, Springer-Verlag
- Woodcock, J., Davies, J. (1996) *Using Z: Specification, Refinement and Proof*, Prentice Hall Europe
-