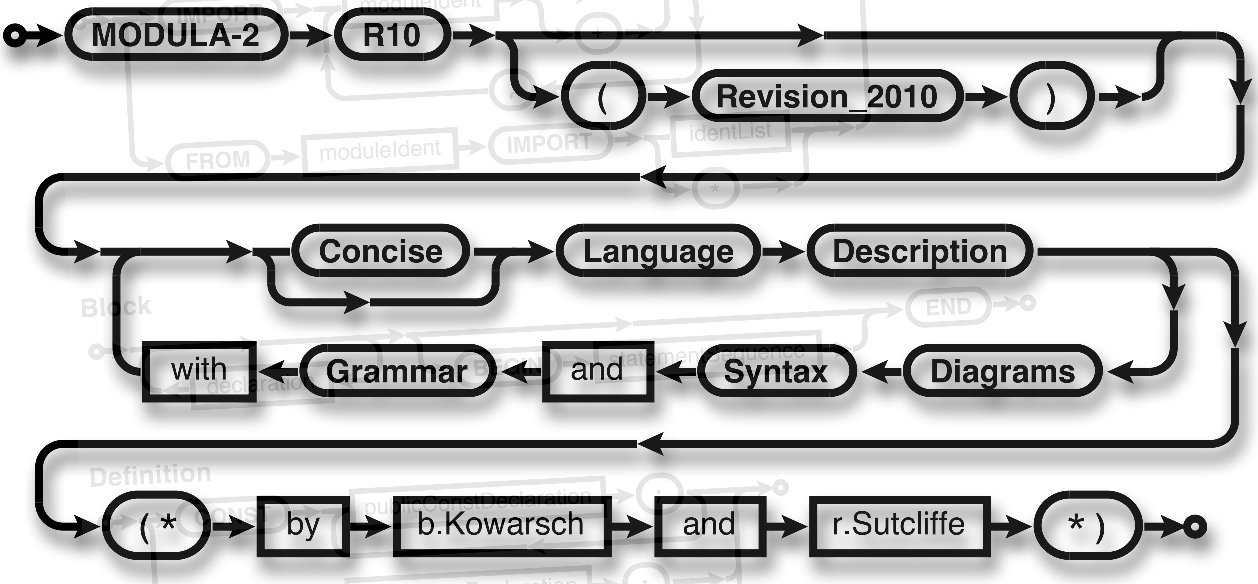
Modula-2 R10

Revision 2010

Last edited: January 31, 2014



Synopsis

This document contains the authoritative language report of Modula-2 R10, a modern revision of classic Modula-2, undertaken by B. Kowarsch and R. Sutcliffe in 2009 and 2010. Primary design goals were type safety, utmost readability and consistency, and suitability as a core language for domain specific supersets. Targeted areas of application are reliable systems implementation, engineering and mathematics. A particular strength of the design is a set of facilities to make library defined abstract data types practically indistinguishable from built-in types and thereby eliminate one of the primary causes of feature growth. R10 is a shorthand for "Revision 2010".

A first public working draft of this document was published in 2010. Pragmas were added in 2012. Work on the standard library then followed in 2013. A preliminary core language design freeze has been declared on January 15, 2014 and a general programming part is being written to prepare for publication in book form.

Abbreviations

ADT	Abstract Data Type	FFI	Foreign Function Interface
API	Application Programming Interface	IDE	Integrated Development Environment
ASCII	ISO464-US 7-bit character set	SXF	Scalar Exchange Format
BCD	Binary Coded Decimals	UCS	Unicode Character Set
BOM	Byte Order Mark	UTF8	UCS Transformation Format 8-bit
EBNF	Extended Backus-Naur Formalism	VLA	Variable Length Array

Syntax Notation

The notation used to describe syntax in this document is based on the EBNF notation used by the lexer and parser generator ANTLR, available from *http://www.antlr.org*:

- names that start with a capital letter represent terminal symbols
- names that start with a lowercase letter represent non-terminal symbols
- single and double quotes are used to delimit literals
- parentheses are used to group syntactic entities
- the vertical bar is used to separate alternatives
- a preceding tilde is used to denote logical not
- a trailing question mark is used to denote zero or one occurrence
- a trailing plus sign is used to denote one or more occurrences
- a trailing asterisk is used to denote zero or more occurrences
- a colon is used between a production rule's name and its body
- a semicolon is used to terminate a production rule

Work Items for Phase I

- editorial review and proofreading (ongoing)
- add general programming part, foreword and index prior to publishing

Work Items for Phase II

- add optional pragma `PRIORITY` for module priority
- definition and description of the `COROUTINE` pseudo-module
- definition and description of a new `ACTOR` library or pseudo-module for actor based concurrency
- explore the practicality of adding a pragma to disable language features in order to increase safety

Reference Compiler

Initial work on an open source reference compiler was undertaken in 2010 but had been paused in order to focus on finalising the language specification first. Some updates have been done in 2013. Work on the compiler will fully resume when the editorial review of the specification has been completed.

Compliance

The language specification of Modula-2 R10 is protected by copyright. The authors provide it as an open specification and grant any interested party the right to implement the specification provided that all of the following conditions are met:

- implementations acknowledge the copyright in the specification and give proper credit to the specification and its authors in associated documentation and marketing materials.
- implementations state the classification of compliance as defined in the glossary: fully compliant implementation, fully compliant superset or partially compliant subset.
- implementations furnish a completed compliance report sheet in associated documentation and marketing materials, using the form provided for this purpose in appendix C.
- implementations that are not fully compliant further document in detail where they do not comply with the specification in associated documentation and marketing materials.

Implementations that are fully compliant with Phase I of this specification will be reclassified as partially compliant or Phase I compliant when Phase II of this specification has been completed. Such an implementation may be classified fully compliant once again if it complies with Phase II.

The use of the specification for the implementation of non-compliant derivatives is discouraged. Implementors who wish to create such a derivative are requested to seek written permission from the authors. However, permission to use parts of the specification for scholarly research is granted.

The authors warrant that they did not and will not file patent applications for any innovations described in this specification and that to their best knowledge no part of this specification is encumbered by any third party patents.

Copying

Verbatim copying of this document in whole or in part for personal or scholarly use is permitted. Copying of the latest working draft for the purpose of placement in a search engine cache is permitted as long as the cached copy is updated no later than three months after a new draft is made available, the authors are not misrepresented and their privacy is respected. No other form of copying or reproduction is permitted without express written permission from both authors.

Creation of derivative works of this document, such as translations or modified versions of the document is not authorised and will require express written permission from both authors.

Citations of parts of this document in scholarly papers are permitted and encouraged.

Acknowledgements

The authors would like to thank Niklaus Wirth for the original design of Modula-2, inspiration and encouragement, Hans-Peter Mössenböck for his inspirational work on library based exception handling, further in alphabetical order, Tom Breeden, Dragiša Durić, Peter Eiserloh, Andreas Fischlin, Jürg Gutknecht, Gaius Mulley, Frode Odegard, Kees Pronk, Michael T. Richter, Diego Sardina and Marco van de Voort for their valuable feedback, further Günther Blaschek, Christian Maurer, Jeff Savit, Christoph Schlegel, Martin Schönhacker and Mark Woodman for their encouragement, and last but not least, the ANTLR and SQLite projects for sharing software tools.

This document has been created using the Pages word processor. The EBNF grammar has been prototyped and verified using ANTLRworks and the syntax diagrams have been created with a Modula-2 specific derivative of the SQLite project's syntax diagram drawing tool.

Table Of Contents

0 Glossary of Terms

0.1 Abstract Data Type

An *abstract data type*, or ADT, is a type whose internal structure and semantics are hidden from the user of the type and has its semantics defined by the library module that provides the ADT.

0.2 ADT Library Module

A library module that defines an abstract data type with the same name as its own module identifier is called an *ADT library module*. Such a module follows the *module-as-a-type* paradigm.

0.3 Auto-Casting, Auto-Casting Parameter

The property of a *formal parameter* to accept any constant or variable of any data-type and to *type cast* a passed-in actual parameter to the data type of the formal parameter is called *auto-casting*[1]. There are two kinds: cast to *open array* and cast to address. The parameter is called an *auto-casting formal parameter*.

0.4 Binding

Binding is the attachment of attributes to a syntactic entity. While most bindings are language defined and immutable, Modula-2 R10 provides three kinds of bindings that are user-definable.

0.4.1 Binding to Built-in Syntax

A library that implements an *abstract data type* may define a procedure and bind it to built-in syntax that would otherwise only be available in association with built-in types. The bound-to syntax may then be used with the *abstract data type* as if it was built-in.

0.4.2 Binding to an Operator

A library that implements an *abstract data type* may define a procedure and bind it to an operator. The bound-to operator may then be used with the *abstract data type* as if it was built-in.

0.4.3 Binding to a Predefined Procedure

A library that implements an *abstract data type* may define a procedure and bind it to the identifier of a predefined procedure. The bound-to procedure may then be passed parameters of the *abstract data type*.

0.5 Collection Type, Key-Value Pair

A type with a variable number of elements all of which are of the same type is called a *collection type*. A value or variable of a collection type is called a collection. The values of a collection are addressable by key and the elements are called *key-value pairs*.

0.6 Compliance

0.6.1 Full Compliance

An implementation that fully complies with the language specification in every aspect is a *fully compliant* implementation. A *fully compliant* implementation that adds syntax, operators, reserved words, predefined identifiers, pseudo-modules or language pragmas is a *fully compliant* superset. It may be domain specific.

0.6.2 Partial Compliance

An implementation that omits any syntax, operators, reserved words, predefined identifiers, pseudo-modules or mandatory pragmas, but complies with the specification in those parts that it implements is a *partially compliant* implementation or a *partially compliant* subset. Such a subset may be domain specific.

[1] Auto-casting semantics were first introduced in classic Modula-2 but without any associated terminology.

0.6.3 Non-Compliant Derivative

An implementation that provides any modified syntax, operators, predefined entities, pseudo-modules or language pragmas but is otherwise based on the specification is a *non-compliant derivative*.

0.7 Coordinated and Uncoordinated Superset

A compliant language superset whose additional reserved symbols, reserved words, predefined identifiers or language pragmas have been reserved in the language specification for exclusive use by the superset is a *co-ordinated superset*. A superset that is not coordinated is an *uncoordinated superset*.

0.8 Indeterminate Record, Indeterminate Field, Discriminant Field

An *indeterminate record* is a record with an *indeterminate field*. An *indeterminate field* is a record field whose size is determined only at runtime. A *discriminant field* is a record field that holds the size of an *indeterminate* field. The *discriminant field* has single-assignment semantics.

0.9 Module as a Manager, Module as a Type

Under the *module-as-a-manager* paradigm a module provides facilities to create, destroy, inspect and manipulate entities of a data type that is not provided by the module itself. Under the *module-as-a-type* paradigm a module provides both the type itself and the operations defined for the type.

0.10 Mutability and Immutability of Variables

A variable is always *mutable* when referenced from the scope in which it is defined. However, a variable may be *immutable* within the context of a different scope than that in which it was defined.

0.11 Named Type, Anonymous Type

A *named type* is a type that has a name associated with it and can be identified by its name which is its identifier. An *anonymous type* does not have a name associated with it and can only be identified by its structure.

0.12 Open Array, Open Array Parameter

An *open array* is an array whose size is not specified. An *open array parameter* is a *formal parameter* whose formal type is an *open array*. In a call to a procedure with an *open array parameter*, any array of the same dimension and base type may be passed-in for the *open array parameter*.

0.13 Parameter

A *parameter* is an entity to pass data into and possibly out of a procedure or function.

0.13.1 Formal Parameter, Actual Parameter

A *parameter* defined in the header of a procedure or function is called a *formal parameter*. A parameter passed in a call to a procedure or function is called an *actual parameter*. In a type safe language the types of formal and actual parameters are required to match.

0.14 Predefined, Predefined Identifier

A language defined constant, data type or procedure that is visible in any module scope without prior import is said to be *predefined* and is called a *predefined* entity. Its identifier is called a *predefined identifier*.

0.15 Pragma

A *pragma* is an in-source compiler directive that controls or influences the compilation process but does not alter the meaning of the program text in which it appears.

0.16 Procedure

A *procedure* is a named sequence of zero or more statements which may be invoked by calling the *procedure*. Zero or more *parameters* may be passed in and out of a *procedure*. There are two kinds.

0.16.1 Function Procedure

A *function procedure* is a *procedure* that returns a result in its own name[2].

0.16.2 Regular Procedure

A *regular*[3] *procedure* is a *procedure* that does not return a result in its own name.

0.16.3 Procedure Signature

The order, types and attributes of the *formal parameters* of a *procedure* as well as its return type are collectively called the *procedure's signature*. The signature of a procedure determines the compatibility of *actual* and *formal parameters* when the *procedure* is called. A *procedure signature* further determines whether the *procedure* is compatible with a given procedure type.

0.16.4 Function Signature

The signature of a *function procedure* is also referred to as a *function signature*.

0.17 Pseudo Entity

A *pseudo entity* is a built-in syntactic entity with special properties different from those of regular entities.

0.17.1 Pseudo-Module

A *pseudo-module* is a module that acts and looks like a library module but is built into the language because the facilities it provides would be difficult or impossible to implement outside of the compiler.

0.17.2 Pseudo-Procedure

A *pseudo-procedure* is a built-in intrinsic or macro that acts and looks like a *procedure* but is either inlined or its signature may be indeterminate or both. Due to these properties, it may not be passed as a procedure type parameter nor assigned to a procedure type variable.

0.17.3 Pseudo-Type

A *pseudo-type* is a built-in type whose use is restricted to one or more specific use cases, such as a type that may only be used as a formal type in a *formal parameter* list.

0.18 Soft, Hard and Promotable Compile Time Warnings

A compile time warning is a warning message emitted by a compiler during compile time. *Soft compile time warnings* may be silenced via compiler settings. *Hard compile time warnings* may not be silenced. *Promotable compile time warnings* may be promoted to compile time errors via compiler settings.

0.19 Type Equivalence

A regime that determines the equivalence of types is called *type equivalence*.

0.19.1 Name Equivalence

Under *name equivalence*, a type is considered equivalent to another if their type identifiers match.

---

[2] A procedure passing a result back to its caller other than by VAR parameter is said to *return a result in its own name*.

[3] In classic Modula-2 terminology, regular procedures were called proper procedures.

0.19.1.1 Loose Name Equivalence

Under *loose name equivalence* it is not possible to distinguish between intended and unintended alias types. An alias of a type is always considered equivalent to its aliased type.

0.19.1.2 Strict Name Equivalence

Under *strict name equivalence* it is either possible to distinguish between intended and unintended alias types or all alias types are not considered equivalent to their aliased type. If intended and unintended alias types are distinguished, then intended alias types are considered equivalent to their aliased type and unintended alias types are not. *Strict name equivalence* is the safest of all type regimes.

0.19.2 Structural Equivalence

Under *structural equivalence*, types are considered equivalent if their structures match.

0.20 Type Transfer

A *type transfer* is the transfer of a value from one type to another type. There are two kinds:

0.20.1 Type Cast

A *type cast* is a type transfer in which the bit representation of a value is not modified but simply reinterpreted as that of another type. The result of a *type cast* may or may not correspond to the original value or any approximation thereof. A *type cast* should therefore be regarded as unsafe.

0.20.2 Type Conversion

A *type conversion* is a type transfer by which the bit representation of a value is modified or replaced if necessary in order to obtain an equivalent value that corresponds to the original value or an approximation thereof in another type. The safety of a *type conversion* is guaranteed by its implementation.

0.21 Unsafe, Non-Portable

A feature is *unsafe* if the language cannot guarantee that a program using the feature will function properly *regardless* of the runtime environment and target architecture. A feature is *non-portable* if the language cannot guarantee that a program using the feature will function properly *depending* on the runtime environment and target architecture.

0.22 Unsafe Facility Enabler

Reserved words or pragmas that provide *unsafe facilities* are unavailable by default and must be enabled by unqualified import of a corresponding *unsafe facility enabler* from a pseudo-module before they can be used.

0.23 Variadic Procedure, Variadic Parameter

A *variadic procedure* is a *procedure* that can accept a variable number of *parameters*. A *variadic parameter* is a *formal parameter* for which a variable number of *actual parameters* may be passed-in.

0.24 Wirthian Macro

A *Wirthian macro* is a language defined lexical macro that acts and looks like a *procedure* where an invocation of the macro is replaced by a call to a library defined *procedure*. The list of *parameters* passed in the invocation does not necessarily match the list of *parameters* passed in the *procedure* call that replaces it. One or more *parameters* may be automatically substituted or inserted.[4]

---

[4] The semantics first appeared with `NEW` and `DISPOSE` in classic Modula-2 but without any associated terminology.

1 Lexical Entities

1.1 Character Sets

By default only the printable characters of the 7-bit ASCII character set, whitespace, tabulator and newline are legal within Modula-2 source text. Unicode characters may be permitted within quoted literals and comments, subject to recognition and verification of the encoding scheme used.

1.2 Special Symbols

Category	Symbol	Usage	Lexical Scope
Intra-Literal	\	Escape Prefix for LF and TAB within Quoted Literals	Within Literal
	‘	Digit Separator within Numeric Literals	
	.	Decimal Point within Real Number Literals	
	+ -	Exponent Sign within Real Number Literals	
Conversion-, String-, and Arithmetic Operators	::	Type Conversion	Program Text
	+	String Concatenation, Identity Function, Addition, Set Union	
	-	Sign Inversion, Subtraction and Set Difference	
	*	Multiplication and Set Intersection	
	/	Real Number Division and Symmetric Set Difference	
Relational Operators	=	Equality Test	Program Text
	#	Inequality Test	
	>	Greater-Than and Proper Superset Test	
	>=	Greater-Than-Or-Equal and Superset Test	
	<	Less-Than and Proper Subset Test	
	<=	Less-Than-Or-Equal and Subset Test	
	==	Identity Test	
Special Syntax	+	Re-Export Suffix and Enumeration Extension Prefix	Program Text
	*	Wildcard in Unqualified Import Directive	
	:=	Assignment Symbol	
	++	Increment Statement Suffix	
	--	Decrement Statement Suffix	
	..	Subrange Constructor and Slice Range Specifier	
	^	Pointer Dereferencing Suffix	
Punctuation	.	Name Separator and Module Terminator	Program Text
	,	Item Separator in an Item List	
	;	Separator in Declaration and Statement Sequences	
	:	Head-Body Separator in Declarations and Parameter Lists	
	|	Case Label Prefix and Blueprint Type-Alternative Separator	
Delimiters	‘ “	Single and Double Quoted String Literal Delimiters	Program Text
	( )	Expression Grouping and Parameter List Delimiters	
	{ }	Structured Value Delimiters	
	[ ]	Special Syntax, Array Index, Subrange and Slice Delimiters	
Pragmas	<* *>	Pragma Delimiters	Program Text
	@	Pragma Value Query Prefix	Within Pragma
Comments	(* *)	Block Comment Delimiters	Program Text
	//	Line Comment Prefix	First Column

1.3 Literals

There are three types of literals:

- numeric literals
- string literals
- structured literals

1.3.1 Numeric literals

There are four types of numeric literals

- decimal number literals
- base-2 number literals
- base-16 number literals
- character code literals

1.3.1.1 Decimal Number Literals

Decimal number literals represent decimal whole and real numbers. They are comprised of a mandatory integral part followed by an optional fractional part followed by an optional exponent. Integral and fractional part are separated by a decimal point. Fractional part and exponent are separated by the exponent prefix e followed by an optional sign. Integral part, fractional part and exponent are comprised of a non-empty sequence of decimal digits. Digits may be grouped using the single quote as a digit separator. A digit separator may only appear in between two digits.

EBNF:

```
DecimalNumber : DigitSeq ( "." DigitSeq ( "e" ( "+" | "-" )? DigitSeq )? )?
DigitSeq : Digit+ ( DigitSeparator Digit+ )* ;
Digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
DigitSeparator : "’" ;
```

Examples:

```
0, 42, 12300, 32767 (* whole numbers *)
0.0, 3.1415, 7.531e+12 (* real numbers *)
1’234’500’000, 0.987’654’321e+99 (* with digit separators *)
```

1.3.1.2 Base-2 Number Literals

Base-2 number literals represent whole numbers. They are comprised of base-2 number prefix 0b followed by a non-empty sequence of base-2 digits. Digits may be grouped using the single quote as a digit separator. A digit separator may only appear in between two digits.

EBNF:

```
Base2Number : "0b" Base2Digit+ ( DigitSeparator Base2Digit+ )* ;
Base2Digit : "0" | "1" ;
DigitSeparator : "’" ;
```

Examples:

```
0b0110 (* without digit separator *)
0b1111’0000’0101’0011 (* with digit separators *)
```

1.3.1.3 Base-16 Number Literals

Base-16 number literals represent whole numbers. They are comprised of base-16 number prefix 0x followed by a non-empty sequence of base-16 digits. Digits may be grouped using the single quote as a digit separator. A digit separator may only appear in between two digits.

EBNF:
```
Base16Number : "0x" Base16Digit+ ( DigitSeparator Base16Digit+ )* ;
Base16Digit : Digit | "A" | "B" | "C" | "D" | "E" | "F" ;
Digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
DigitSeparator : "'" ;
```

Examples:
```
0x80, 0xFF, 0xCAFED00D (* without digit separator *)
0x00'00'FF'FF, 0xDEAD'BEEF (* with digit separators *)
```

1.3.1.4 Character Code Literals

Character code literals represent Unicode code points. They are comprised of Unicode prefix 0u followed by a non-empty sequence of base-16 digits. Digits may be grouped using the single quote as a digit separator. A digit separator may only appear in between two digits.

EBNF:
```
CharCodeLiteral : "0u" Base16Digit+ ( DigitSeparator Base16Digit+ )* ;
Base16Digit : Digit | "A" | "B" | "C" | "D" | "E" | "F" ;
Digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
DigitSeparator : "'" ;
```

Examples:
```
0u7F (* DEL *)
0uA9 (* copyright *)
0u20'AC (* Euro sign *)
```

1.3.2 String Literals

String literals are sequences of quotable characters and optional escape sequences, enclosed in single quotes or double quotes. String literals may not contain any control code character.

EBNF:
```
String : "'" ( Character | '"' )* "'" | '"' ( Character | "'" )* '"' ;
Character : Digit | Letter | EscapedCharacter |
            " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" |
            "," | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" |
            "@" | "[" | "]" | "^" | "_" | "`" | "{" | "|" | "}" | "~" ;
Digit : "0" .. "9" ;
Letter : "A" .. "Z" | "a" .. "z" ;
EscapedCharacter : "\" ( "n" | "t" | "\" ) ;
```

Examples:
```
"it's nine o'clock"
'he said "Modula-2" and smiled'
"this is the end of the line\n"
```

1.3.3 Structured Literals

Structured literals are compound values consisting of zero or more terminal symbols, enclosed in braces. Structured literals may be nested.

EBNF:
```
structuredValue :
    "{" ( valueComponent ( "," valueComponent )* )? "}" ;
valueComponent :
    expression ( ( BY | ".." ) constExpression )? ;
```


Examples:
```
{ 1, 2, 3 }
{ “a”, “b”, “c” }
{ 42, 1.23, "foo", { 1, 2, 3 }, {“a”, “b”, “c”} }
{ 1 .. 5 }  (* equivalent to: { 1, 2, 3, 4, 5 } *)
{ 0 BY 5 }  (* equivalent to: { 0, 0, 0, 0, 0 } *)
```

1.4 Reserved Words

Reserved words are symbols that consist of a sequence of all-uppercase letters, are visible in any scope, have special meaning in the language and may not be redefined. There are 45 reserved words:

List of Reserved Words:

ALIAS	CONST	EXIT	INDETERMINATE	POINTER	TO
AND	DEFINITION	FOR	LOOP	PROCEDURE	TYPE
ARGLIST	DESCENDING	FROM	MOD	RECORD	UNTIL
ARRAY	DIV	GENLIB	MODULE	REFERENTIAL	VAR
BEGIN	DO	IF	NOT	REPEAT	WHILE
BLUEPRINT	ELSE	IMPLEMENTATION	OF	RETURN	
BY	ELSIF	IMPORT	OPAQUE	SET	
CASE	END	IN	OR	THEN	

1.5 Identifiers

Identifiers are names for syntactic entities in a program. They start with a letter, low-line or dollar sign, followed by any number and combination of letters, low-lines, dollar signs and digits.

EBNF:
```
Ident : ( "_" | "$" | Letter ) ( "_" | "$" | Letter | Digit )* ;
```

The use of the low-line and dollar sign within identifiers is permitted in support of environments and platforms where they are an integral part of the naming convention, for instance when writing components for or mapping to operating system APIs that use them. However, such an identifier must also contain at least one letter or digit. A non-conformant identifier shall cause a compile time error. The definition of an identifier in a foreign API style shall cause a soft compile time warning. The warning may be automatically silenced when `FFI` of module `UNSAFE` is imported into the scope of the compiling module.

Examples:
```
(* Modula-2 style *) Foo, setBar, getBaz, Str80, Matrix8x4, FOOBAR
(* Foreign API styles *) _foo, __bar, __baz__, foo_bar_123, $foo, sys$foo, SYS$BAR
```

There are two categories of identifiers:

- reserved identifiers
- user-definable identifiers

1.5.1 Reserved Identifiers

Reserved identifiers are language defined identifiers that may not be redefined. Reserved are:

- predefined identifiers
- module identifiers of pseudo-modules
- identifiers associated with core language semantics

```
ABS, ADDR, ADDRESS, ASSEMBLER, ATOMIC, BOOLEAN, BYTE, CARDINAL, CAST, CHAR, CHR,
COMPILER, CONCAT, CONVERSION, COPY, COROUTINE, COUNT, DUP, FALSE, FFI, INTEGER, LENGTH,
LONGCARD, LONGINT, LONGREAL, NEG, NEW, NIL, OCTET, ODD, ORD, PRED, PTR, READ, REAL,
REG, RELEASE, REMOVE, RETAIN, RETRIEVE, RUNTIME, STORE, SUBSET, SUCC, SXF, TBASE,
TLIMIT, TMAX, TMAXEXP, TMIN, TMINEXP, TPRECISION, TRUE, TSIGNED, TSIZE, UNICHAR,
UNSAFE, VAL, VARGC, VARGLIST, WORD, WRITE, WRITEF
```


1.5.2 User-definable Identifiers

Identifiers that do not coincide with reserved words or reserved identifiers may be defined or redefined in any scope of a program or library module.

1.6 Non-Semantic Symbols

Non-semantic symbols are symbols that do not impact the meaning of a program. They may occur anywhere in a program before or after semantic symbols but not within them. There are three types:

- pragmas
- comments
- lexical separators

1.6.1 Pragmas

Pragmas are directives to control or influence the compilation process but they do not change the meaning of the program. Pragmas consist of a pragma body enclosed in pragma delimiters `<*` and `*>`.

EBNF:
```
pragma : "<*" pragmaBody "*>" ;
```

A pragma body consists of a non-empty token sequence whose syntax is defined by the pragma grammar. Whitespace, horizontal tab and line breaks may occur between tokens within a pragma, but comments are not permitted. A comment delimiter encountered within a pragma shall cause a compile time error.

There are language defined and implementation defined pragmas. The pragma grammar defines a set of reserved words for use within language defined pragmas. They have special meaning within the pragma language and may not be used within implementation defined pragmas. There are 27 in-pragma reserved words:

List of In-Pragma Reserved Words:

ADDR	ELSIF	FATAL	FROM	INLINE	NORETURN	SINGLEASSIGN
ALIGN	ENDIF	FFI	GENERATED	LOWLATENCY	PADBITS	VOLATILE
DEPRECATED	ENCODING	FFIDENT	IF	MSG	PTW	WARN
ELSE	ERROR	FORWARD	INFO	NOINLINE	PURITY	

The symbols of implementation defined pragmas must be lowercase or mixed case words.

Examples:
```
<*ALIGN=TSIZE(LONGCARD)*>  (* language defined pragma *)
<*BoundsChecking=FALSE|WARN*)  (* implementation defined pragma *)
```

1.6.2 Comments

Comments are ignored by a compiler but are for annotation and documentation. There are two kinds:

- block comments
- line comments

1.6.2.1 Block Comments

Block comments are delimited by opening `(*` and closing `*)` comment delimiters.

EBNF:
```
BlockComment : "(*" ( ~( "*)" )* BlockComment? )* "*)" ;
```

Block comments are intended for annotating source code. They may span multiple lines and they may be nested but in order to ensure the portability of source code, a language defined nesting limit of ten including the outermost comment is imposed. A compile time error shall occur if this limit is exceeded.

Examples:
```
IF (* no match found *) this^.next = NIL THEN (* comment (* comment within *) *)
```


1.6.2.2 Line Comments

Line comments start with a // symbol at the beginning of a line and terminate at the end of the same line.

EBNF:

```
LineComment : "//" ~( EndOfLine )* EndOfLine ;
EndOfLine : LF | CR LF? ;
```

Line comments are provided in support of special documentation tags for automated documentation generators such as Doxygen. They are not intended for annotating source code.

Example:

```
(* Special documentation tags for Doxygen *)
//! @brief Modula-2 Standard Library
//! @authors B.Kowarsch & R.Sutcliffe
```

1.6.3 Lexical Separators

Lexical separators terminate a numeric literal, an identifier, a reserved word or a pragma symbol.

EBNF:

```
LexicalSeparator : " " | TAB | EndOfLine ;
EndOfLine : LF | CR LF? ;
```

1.7 Control Codes

The following control codes may appear within Modula-2 source text but not within string literals:

- `TAB` denoting horizontal tab code `0u9`
- `LF` denoting line feed code `0uA`
- `CR` denoting carriage return code `0uD`
- `UTF8 BOM` denoting code sequence `0uEF`, `0uBB`, `0uBF`, but permitted only at the very beginning of a file

Any other control codes within a source file shall cause a compile time error. An unrecognised BOM shall cause a fatal compile time error. Encoding support other than ASCII and UTF8 is implementation defined.

1.8 Symbols Reserved for Language Extensions and External Utilities

Although not part of the language itself, certain symbols are reserved for exclusive use by language extensions and external source code processing utilities. Others are taboo or reserved for possible future use.

1.8.1 Symbols Reserved for Use by Coordinated Language Supersets

A coordinated language superset is a compliant language superset for whose exclusive use certain symbols are reserved. The reserved symbols of coordinated language supersets are listed below:

Superset	Symbols Reserved for Exclusive Use by Superset	
Objective Modula-2	Special Symbols	@ `
	Reserved Words	BYCOPY BYREF CLASS CONTINUE CRITICAL INOUT METHOD ON OPTIONAL OUT PRIVATE PROTECTED PROTOCOL PUBLIC TRY
	Predefined Identifiers	NO OBJECT YES
	Pragmas	ACTION FRAMEWORK OUTLET QUALIFIED
Parallel Modula-2	Reserved Words	ALL PARALLEL SYNC
	Pragmas	LOCAL SPREAD CYCLE SBLOCK CBLOCK
Single-Pass Compilers	Pragmas	FORWARD

1.8.2 Symbols Reserved for Use by Uncoordinated Language Supersets

An uncoordinated language superset is a superset for which no particular symbols are reserved in the language specification. An uncoordinated superset may define any additional predefined identifiers, reserved words and language pragmas as long as they are prefixed with a single ampersand `&` or percent `%` character. Such a superset may use either the ampersand prefix or the percent prefix, but not both.

Examples:
```
&TRY &CATCH %DESCR %IMMED (* superset reserved words *)
<* &DISALLOW="Pointers" *> (* superset language pragma *)
```

1.8.3 Symbols Reserved for Other Language Facilities

Facility	Symbols Reserved for Exclusive Use by Facility
Qualified Name Mangling	~ !
Optional Inline Assembler	ASSEMBLER ASM REG
Phase II Deliverables	COROUTINE ACTOR PRIORITY

1.8.4 Symbols Reserved for Use by External Source Code Processors

To assist external source code processing prior to compilation, certain symbols are reserved for exclusive use by external source code processing utilities.

Utility	Symbols Reserved for Exclusive Use by Utility
Modula-2 Template Engine	## @@ <# #> /* */
Character Set Transliterators	/= (. .) (: :) /?/ /:/ (^) (/) (‘) (;) (?) (.) (+) (-) and ?? trigraphs of ISO C

1.8.5 Symbols that are Taboo or Reserved for Possible Future Use

Any special symbols not already reserved are either taboo or reserved for possible future use.

1.9 Lexical Parameters

1.9.1 Length of Literals

The minimum lengths of literals a conforming implementation shall support are:

- for string literals, 160 characters
- for character code literals, 8 characters
- for whole number literals, 24 characters
- for real number literals, 64 characters

The fractional part of a real number literal may be truncated. If it is truncated, a soft compile time warning shall be emitted.

If a string literal, a character code literal, a whole number literal or the significand or exponent of a real number literal is longer than an implementation is able to process, a compile time error shall occur.

1.9.2 Length of Identifiers and Pragma Symbols

The minimum lengths of identifiers and pragma symbols an implementation shall support are:

- for identifiers, 32 characters
- for pragma symbols, 32 characters

If an identifier or a pragma symbol name exceeds the maximum length supported by the implementation it may be truncated to the maximum supported length. If it is, a soft compile time warning shall occur.

1.9.3 Length of Comments

An implementation that generates source code of another language may choose to preserve comments by copying them into the output. In this case, the implementation may limit the length of comments copied into the output. The minimum lengths of comments to be fully preserved that such an implementation shall support are:

- for line comments, 250 characters
- for block comments, 2000 characters

If a comment to be preserved exceeds the maximum length supported by the implementation it may be truncated to the maximum supported length. If it is truncated, a soft compile time warning shall occur.

Furthermore, if a nested multi-line comment is truncated, an implementation shall insert all closing comment delimiters that have been lost as a result of truncation.

1.9.4 Line and Column Counters

An implementation may limit the capacity of its internal line and column counters. The minimum values a conforming implementation shall support are:

- for the line counter, 65000
- for the column counter, 250

In the event that a source file being processed exceeds the supported counter limits an implementation may either continue or abort compilation. A soft compile time warning shall occur if the implementation continues. A fatal compile time error shall be emitted if the implementation aborts.

1.9.5 Lexical Parameter Constants

Actual lexical parameters shall be provided as constants in standard library module `LexParams`.

2 Compilation Units

A compilation unit is a sequence of source code that can be independently compiled. There are four types of compilation units:

- a program module
- the definition[1] part of a library module
- the implementation part of a library module
- a blueprint definition

EBNF:
```
compilationUnit :
    IMPLEMENTATION? programModule | definitionOfModule | blueprint;
```

A Modula-2 program consists of exactly one program module and zero or more library modules and blueprint definitions.

2.1 Program Modules

A program module represents the topmost level of a Modula-2 program. It may import any number of identifiers from any number of library modules but does not itself export any identifiers.

EBNF:
```
programModule :
    MODULE moduleIdent ";"
    importList* block moduleIdent "." ;
moduleIdent : Ident ;
```

Example:
```
MODULE HelloWorld;
IMPORT IOSupport;
BEGIN
  WRITE(“Hello World\n”)
END HelloWorld.
```

2.2 Definition Part of Library Modules

The definition part of a library module represents the public interface of the library module. Any identifier defined in the definition part is automatically available for import by other modules.

EBNF:
```
definitionModule :
    DEFINITION MODULE moduleIdent ( "["blueprintToObey"]" )? ( FOR typeToExtend )? ";"
    importList* definition*
    END moduleIdent "." ;
```

Example:
```
DEFINITION MODULE Counting;
PROCEDURE LetterCount( str : ARRAY OF CHAR ) : CARDINAL;
PROCEDURE DigitCount( str : ARRAY OF CHAR ) : CARDINAL;
END Counting.
```

2.3 Implementation Part of Library Modules

The implementation part of a library module represents the implementation of the library module. Any identifier defined in the corresponding definition part is automatically available in the implementation part. Any identifier defined in the implementation part is not available outside the implementation part.

---

[1] The definition part of a library module may also be referred to as the interface or interface part of the library module.

EBNF:
```
implementationModule :
    IMPLEMENTATION programModule ;
```

2.2 Blueprint Definitions

A blueprint definition represents a common set of requirements that a library module may be declared to conform to. Declaration of conformance to a blueprint is mandatory for ADTs that bind constants or procedures to built-in syntax. The blueprint determines how the ADT shall be declared, what literals shall be compatible, and what bindings to operators and built-in procedures it shall provide.

EBNF:
```
blueprint :
    BLUEPRINT blueprintIdent ( "[" blueprintToRefine"]" )? ( FOR bpOfTypeToExtend )? ";"
    ( REFERENTIAL identList ";" )? moduleTypeRequirementOrImpediment ";"
    ( requiredConst “;” )* ( requiredProcedure “;” )*
    END blueprintIdent"." ;
blueprintIdent: Ident ;
blueprintToRefine, bpOfTypeToExtend : blueprintIdent;
moduleTypeRequirementOrImpediment :
    TYPE "=" ( permittedTypeDefinition ( "|" permittedTypeDefinition )*
    ( ":=" protoLiteral ( "|" protoLiteral )* )? ) | NIL ;
permittedTypeDefinition :
    RECORD | OPAQUE RECORD? ;
protoLiteral :
    simpleProtoliteral | structuredProtoliteral ;
simpleProtoliteral : Ident ;
structuredProtoliteral :
   “{“ ( VARIADIC OF simpleProtoliteral ( “,” simpleProtoliteral )* |
         structuredProtoliteral ( "," structuredProtoliteral )+ ) “}” ;
requiredProcedure : procedureHeader ;
requiredConst :
    CONST ( “[” constBindableProperty “]” )? Ident
    ( “:” ( range OF)? predefinedType | “=” constExpression ) ;
constBindableProperty :
    “:=” | DESCENDING | TSIGNED | TBASE | TPRECISION | TMINEXP | TMAXEXP ;
predefinedType : Ident ;
```

Example:
```
BLUEPRINT ProtoComplex [ProtoNumeric]; (* blueprint for complex numbers *)
TYPE = OPAQUE RECORD := { REAL, REAL };
PROCEDURE [+] add ( a, b : ProtoComplex ) : ProtoComplex; (* binding to + *)
PROCEDURE [-] sub ( a, b : ProtoComplex ) : ProtoComplex; (* binding to - *)
```


2.5 Module Initialisation

The body of the implementation part of a library module is the library's initialisation procedure. It is automatically executed by the Modula-2 runtime environment when a Modula-2 program is run.

The order in which modules are initialised is language defined and depends on the module dependency graph. During compilation a module dependency graph is built and the initialisation order is determined by depth-first traversal order of the dependency graph whereby initialisation takes place for each node from bottom to top on the way back up. However because of module interdependencies among library modules that may not be visible to a programmer making use of these, and because the order in which items are imported may affect the initialization sequence, a program that depends on a particular initialisation order for its meaning is wrong.

2.6 Module Termination

Module termination is not a core language feature but it is a facility provided by a standard library module. Module `Termination` provides an API for client modules that require termination to install their own termination handlers onto the library's termination handler stack.

Module `Termination` installs its own wind-down procedure in the runtime environment during module initialisation. The wind-down procedure then calls the installed termination handlers in reverse order when the program is about to be terminated.

3 Import of Identifiers

Identifiers defined in the interface of a library module may be imported by other modules using an import directive. There are two types of import:

- qualified import
- unqualified import

EBNF:
```
import :
    IMPORT moduleIdent ( "+" | "-" )? ( "," moduleIdent ( "+" | "-" )? )* |
    FROM moduleIdent IMPORT ( identList | "*" ) ";"
```

3.1 Qualified Import

When an identifier is imported by qualified import, it must be qualified with the exporting module's module name when it is referenced in the importing module. This avoids name conflicts when importing identically named identifiers from different modules.

Example:
```
IMPORT FileIO; (* qualified import of module FileIO *)
VAR status : FileIO.Status; (* qualified identifier of Status *)
```

3.1.1 Import Aggregation

A module imported by qualified import may be automatically re-exported to any importing client module. Modules to be re-exported in this way are marked with a plus sign after their identifiers.

A module that imports other modules for the sole purpose of re-export is called an import aggregator. This facility is useful for importing an entire library collection with a single import statement.

Example:
```
DEFINITION MODULE FooBarBaz;
IMPORT Foo+, Bar+, Baz+; (* import Foo, Bar and Baz into importing module *)
END FooBarBaz.
MODULE ImportAggregate;
IMPORT FooBarBaz; (* equivalent to: IMPORT Foo, Bar, Baz; *)
```

3.1.2 Importing Modules as Types

If the interface of a module defines a type that has the same name as the module then the type is referenced unqualified. This facility is useful in the construction of abstract data types as library modules.

Example:
```
DEFINITION MODULE Colour;
TYPE Colour = ( red, green, blue );
(* public interface *)
END Colour.
IMPORT Colour; (* import module Colour *)
VAR colour : Colour; (* type referenced as Colour instead of Colour.Colour *)
```


3.2 Unqualified Import

When an identifier is imported by unqualified import, it is made available in the importing module as is. This facility is intended predominantly for import from pseudo-modules and in cases where its use will reduce clutter and improve readability. If two identically named identifiers from different modules are imported unqualified, a name conflict occurs and a compile time error shall be emitted.

Examples:
```
FROM COMPILER IMPORT MIN, MAX, HASH; (* unqualified import *)
CONST Start = MIN( foo, bar, baz ); (* MIN instead of COMPILER.MIN *)
CONST BufSize = MAX( x, y, z ); (* MAX instead of COMPILER.MAX *)
CONST FooIdent = HASH("Foo"); (* HASH instead of COMPILER.HASH *)
```

3.2.1 Wildcard Import

An unqualified import directive may import all available identifiers of a pseudo module by using an asterisk as a wildcard. This facility is available for import from pseudo modules only, because it significantly increases the likelihood of name conflicts. Any other use shall cause a compile time error.

Example:
```
(* import all identifiers from pseudo-module RUNTIME *)
FROM RUNTIME IMPORT *;
(* then use unqualified ... *)
RaiseRTFault(RTFault.InvalidAccessor);
(* instead of qualified, thereby avoiding clutter ... *)
RUNTIME.RaiseRTFault(RUNTIME.RTFault.InvalidAccessor);
```

3.3 Repeat Import

3.3.1 Qualified Import of an Already Imported Module

Qualified import of a module that has already been imported by qualified import into the same module scope is permissible. Only the first import is significant, any subsequent imports are redundant. A redundant import has no effect but shall cause a soft compile time warning.

3.3.2 Unqualified Import of an Already Imported Identifier

Unqualified import of an identifier that has already been imported unqualified from the same origin into the same module scope is permissible. Only the first import is significant, any subsequent imports are redundant. A redundant import has no effect but shall cause a soft compile time warning.

3.3.3 Qualified and Unqualified Import of an Identifier

Unqualified import of an identifier that is also imported qualified into the same module scope is permissible. The imported entity may then be referenced both qualified and unqualified. No compile time warning shall occur.

3.3.4 Unqualified Import from an Already Imported ADT Library Module

Unqualified import of the type identifier of an ADT whose library module is also imported qualified in to the same module scope results in a name conflict and shall cause a compile-time error. However, unqualified import of any other identifier from an already imported ADT library module is permissible.

4 Data Types

Modula-2 is a strongly typed language. Constants and variables are always associated with a data type. A data type is an abstract property of a constant or variable that determines the storage size and structure, the compatibility with other constants or variables and the operations that are permitted on entities of that type. There are ten predefined data types:

Predefined Types:
```
BOOLEAN, CHAR, UNICHAR, (* non-numeric *)
OCTET, CARDINAL, LONGCARD, INTEGER, LONGINT, REAL, LONGREAL (* numeric *)
```

Other types may be defined using built-in type constructor syntax:

Enumeration types, set types, array types, record types, pointer types, procedure types and abstract data types.

Character strings are represented by character arrays or abstract data types.

4.1 Type Compatibility

A relationship between types that determines whether one type may be substituted for the other is called type compatibility. The default type regime in Modula-2 R10 is strict name equivalence. Two types are compatible only if they are one and the same, unless their compatibility has intentionally been declared through `ALIAS` type or subrange type declaration. Thus, there are three kinds of type compatibility:

- type compatibility by name
- type compatibility by alias
- type compatibility by subrange

4.1.1 Type Compatibility By Name

Two types are compatible if their names match, that is, if they are one and the same type.

4.1.2 Type Compatibility By Alias

Two types are compatible if one is defined directly or indirectly as an `ALIAS` type of the other, or if both types are defined directly or indirectly as `ALIAS` types of a third type. An `ALIAS` type is a type that has been defined using the `ALIAS OF` type constructor. An `ALIAS` type is compatible with its base type.

`ALIAS` type compatibility is commutative: If `A` is an `ALIAS` type of type `T`, then all values of type `A` are compatible with `T` and all values of type `T` are compatible with `A`. `ALIAS` type compatibility is also transitive: If `A` is an `ALIAS` type of `T1` and `T1` is an `ALIAS` type of `T2`, then type `A` is also compatible with `T2`. Moreover, if two types `T1` and `T2` are `ALIAS` types of `T3`, then `T1` and `T2` are compatible.

4.1.3 Type Compatibility by Subrange

A type is compatible with another if it is defined directly or indirectly as a subrange type of the other. A subrange type is a derived type that has been defined using the `[n .. m] OF` subrange type constructor.

Subrange type compatibility is not commutative: A subrange type is compatible with its base type, but the reverse is not true. That is, if `S` is a subrange type of type `T`, or a subrange type of a subrange type of `T`, then values of type `S` are compatible upwards with `T` but values of type `T` are not compatible downwards with `S`. Also, if `S1` and `S2` are both subrange types of type `T`, then `S1` and `S2` are not compatible.

4.2 Value Compatibility

A relationship between value representing entities that determines their compatibility for a given use case is called value compatibility. There are three such use cases. Thus, there are three kinds of value compatibility.

4.2.1 Assignment Compatibility

A relationship between a value and a variable that determines whether the value may be assigned to the variable is called assignment compatibility. A value is said to be assignment compatible.

4.2.2 Expression Compatibility

A relationship between a value and a given operation that determines whether the value may be used as a given operand of the operation is called expression compatibility.

A relationship between entities that determines whether the entities may be operands of a given operation is called expression compatibility. In Modula-2 R10, expression compatibility is defined by the operation and thus it is always expressed in respect of a given operation.

4.2.3 Parameter Passing Compatibility

A relationship between an entity and a formal parameter that determines whether the value may be passed as an argument to the parameter is called parameter passing compatibility.

4.1.3 Compatibility of Literals

Whole number literals are assignment compatible with:

- unsafe types `UNSAFE.BYTE` , `UNSAFE.WORD` and `UNSAFE.ADDRESS`
- predefined types `OCTET`, `CARDINAL`, `LONGCARD`, `INTEGER` and `LONGINT`
- any subrange type of `OCTET`, `CARDINAL`, `LONGCARD`, `INTEGER` and `LONGINT`
- any ADT that conforms to a prototype that permits the use of whole number literals provided the ADT also defines a procedure to bind to the assignment operator

Real number literals are assignment compatible with:

- predefined types `REAL` and `LONGREAL`
- any ADT that conforms to a prototype that permits the use of real number literals provided the ADT defines a procedure to bind to the assignment operator

Character code and string literals are assignment compatible with:

- predefined types `CHAR` and `UNICHAR`
- any ADT that conforms to a prototype that permits the use of character literals provided the ADT defines a procedure to bind to the assignment operator

Structured literals are assignment compatible with:

- types that are structurally equivalent to the structured literal
- any ADT that conforms to a prototype that permits the use of structured literals provided the literal is structurally equivalent to the proto-literal defined by the prototype and further provided the ADT defines a procedure to bind to the assignment operator

4.1.4 Assignment Compatibility

The value of an expression `e` may be assigned to a mutable variable `v` if any of the following is true:

- both `e` and v are of the same type
- `e` is compatible with `v` under `ALIAS` type compatibility rules
- `e` is compatible with `v` under subrange type compatibility rules
- `e` is the identifier of a procedure `p`, `v` is of a procedure type `T` and the signatures of `p` and `T` match
- `e` is a literal that is compatible with `v` under literal compatibility rules

Regardless of type compatibility, assignment may not be made to an immutable variable.

4.1.5 Parameter Passing Compatibility

4.1.5.1 Named Type Parameters

The value of an expression `e` may be passed to a named-type `VAR` parameter `p` if:

- `e` is a mutable variable designator and `e` is assignment compatible with `p`

An expression `e` may be passed to a named-type `CONST` or value parameter `p` if:

- `e` is assignment compatible with the type of `p`

4.1.5.2 Open Array Parameters

The value of an expression `e` may be passed to an open array `VAR` parameter `p` if:

- `e` is a mutable variable designator,
  the type of `e` is an array type,
  and the base type of `e` is assignment compatible with the base type of `p`

The value of an expression `e` may be passed to an open array `CONST` or value parameter `p` if:

- the type of `e` is an array type,
  and the base type of `e` is assignment compatible with the base type of `p`

4.1.5.3 Auto-Casting Open Array Parameters

The value of an expression `e` may be passed to an open array `VAR` parameter `p` if:

- `e` is a mutable variable designator
  and the formal type of `p` is `CAST ARRAY OF OCTET`,
  or `CAST ARRAY OF UNSAFE.BYTE`, or `CAST ARRAY OF UNSAFE.WORD`

The value of an expression `e` may be passed to an open array `CONST` or value parameter `p` if:

- the formal type of `p` is `CAST ARRAY OF OCTET`,
  or `CAST ARRAY OF UNSAFE.BYTE`, or `CAST ARRAY OF UNSAFE.WORD`

4.1.5.4 Variadic Parameters

A comma separated list of values may be passed to a variadic parameter if:

- the formal variadic parameter is of pseudo-type `UNSAFE.VARGLIST`
- the formal variadic parameter is the last parameter of the procedure
  and the list is structurally equivalent to the formal variadic parameter

A structured literal may be passed to a variadic parameter if:

- the formal variadic parameter is of pseudo-type `UNSAFE.VARGLIST`
- the literal is structurally equivalent to the formal variadic parameter

4.2 Type Conversions

A value of type `T1` may be converted to an equivalent value of an incompatible type `T2` using the type conversion operator if `T1` is convertible to `T2`.

4.2.1 Convertibility of Non-Numeric Ordinal Types

A value `v1` of a non-numeric ordinal type `T1` is convertible to an equivalent value of another non-numeric ordinal type `T2` if `T2` has a legal value `v2` for which the relation `ORD(v1) = ORD(v2)` is true.

A value `v` of a non-numeric ordinal type `T1` is convertible to an equivalent value of a predefined whole number type `T2` if `ORD(v)` is a legal value of `T2`.

4.2.2 Convertibility of Numeric Predefined Types

A value `v` of a numeric predefined type `T1` is convertible to an equivalent value of another numeric predefined type `T2` if `v` is also a legal value of `T2`.

A value `v1` of a predefined whole number type is convertible to an equivalent value of a non-numeric ordinal type `T2` if `T2` has a legal value `v2` for which the relation `v1 = ORD(v2)` is true.

4.2.3 Convertibility of Set Types

A value `s` of a set type `T1` is convertible to an equivalent value of another set type `T2` if every element in `T1` may also be a legal element of `T2`.

4.2.4 Convertibility of Array Types

A value of an array type `T1` is convertible to a value of another array type `T2` if `T1` and `T2` have the same number of components and the base type of `T1` is assignment compatible with the base type of `T2`.

4.2.5 Convertibility of Record Types

A value of a record type `T1` is convertible to a value of record type `T2` if `T1` is an extension of `T2`. Fields present in `T1` that are not present in `T2` are lost during conversion.

4.2.6 Convertibility of Pointer Types

A value of a pointer type `T1` is convertible to a value of another pointer type `T2` if the base type of `T1` is assignment compatible with the base type of `T2`.

4.2.7 Convertibility of Procedure Types

A value of a procedure type `T1` is convertible to a value of procedure type `T2` if types `T1` and `T2` are structurally equivalent.

4.2.8 Convertibility of Opaque Types

A value `v` of an opaque type `T1` is convertible to an equivalent value of another type `T2` if:

- `T1` is an ADT that provides a conversion procedure for conversions from type `T1` to type `T2`, the procedure is bound to the conversion operator, and `v` is a legal value of `T2`
- `T1` and `T2` are scalar types, `T1` is convertible to scalar exchange format, `T2` is convertible from scalar exchange format, and `v` is a legal value of `T2`

4.2.9 Convertibility of Scalar Types

A type `T` is convertible to scalar exchange format if:

- `T` is a numeric predefined type
- `T` is an ADT that provides a conversion primitive to scalar exchange format

A type `T` is convertible from scalar exchange format if:

- `T` is a numeric predefined type
- `T` is an ADT that provides a conversion primitive from scalar exchange format

4.2.10 Non-Convertibility of UNSAFE Types

Types provided by pseudo-module `UNSAFE` are not convertible. No conversion operator bindings may be defined that convert to or from `UNSAFE` types. To transfer the value of an `UNSAFE` type to another type, or to transfer a value to an `UNSAFE` type, the `CAST` operation must be used.

4.3 Semantics of Types

Every data type has an associated set of language defined semantics. These semantics define the interpretation of values, the compatibility of literals and a set of operations. Many data types share a common set of semantics with other data types. A common set of shared semantics is called a prototype. Every data type is thus defined in terms of its prototype.

4.3.1 The Semantics of Non-Numeric Ordinal Types

Non-numeric ordinal types are data types with non-numeric ordered values, including a start value that is interpreted as the type's zero-th value. The ordinal value of any n-th value is n for all n. The following operations are defined for non-numeric ordinal types:

- assignment of literals and expressions (`:=`)
- type conversion (`::`)
- smallest value (`TMIN`)
- largest value (`TMAX`)
- ordinal value (`ORD`)
- predecessor value (`PRED`)
- successor value (`SUCC`)
- iteration (`FOR value IN type`)
- equal (=)
- not-equal (#)
- less (<)
- less-or-equal (<=)
- greater (>)
- greater-or-equal (>=)

Predefined data types `BOOLEAN`, `CHAR` and `UNICHAR`, and all enumeration types are non-numeric ordinal types. Literals for type `BOOLEAN` are `TRUE` and `FALSE`, literals for types `CHAR` and `UNICHAR` are character code literals and string literals of length one.

4.3.2 The Semantics of the Boolean Type

The boolean type is a non-numeric ordinal type with two values, interpreted as boolean truth values, represented by the predefined constants `TRUE` and `FALSE`, where `TRUE` > `FALSE`. Further to the operations defined for non-numeric ordinal types, three additional operations are defined for the boolean type:

- logical-not (`NOT`)
- logical-and (`AND`)
- logical-or (`OR`)

Predefined data type `BOOLEAN` is the one and only boolean type. No facility exists to define other data types as boolean types.

4.3.3 The Semantics of Set Types

Set types are data types that represent mathematical sets with a finite number of elements. The following operations are defined for set types:

- assignment of literals and expressions (`:=`)
- type conversions (`::`)
- element capacity (`TLIMIT`)
- number of actual elements (`COUNT`)
- membership test (`IN`)
- include element (`set[element] := TRUE`)
- exclude element (`set[element] := FALSE`)
- iteration (`FOR element IN set`)
- set union (+)
- set difference (`-`)
- set intersection (`*`)
- symmetric set difference (`/`)
- equal (=)
- not-equal (#)
- proper subset (<)
- proper superset (>)
- subset (<=)
- superset (>=)

All types defined using the `SET OF` type constructor are set types.

4.3.4 The Semantics of Whole Number Types

Whole number types are data types that represent subsets of the mathematical set of integers ℤ, with a finite number of values. The following operations are defined for whole number types:

- assignment of literals and expressions (`:=`)
- type conversions (`::`)
- scalar conversion (`SXF`, `VAL`)
- smallest value (`TMIN`)
- largest value (`TMAX`)
- absolute value (`ABS`)
- sign reversal (`NEG`), for signed types only
- odd/even test (`ODD`)
- addition (+)
- postfix increment (++)
- difference (-)
- postfix decrement (`--`)
- multiplication (`*`)
- integer division (`DIV`)
- modulo (`MOD`)
- iteration (`FOR value IN type`)
- equal (=)
- not-equal (#)
- less-than (<)
- less-or-equal (<=)
- greater-than (>)
- greater-or-equal (>=)

Predefined data types `OCTET`, `CARDINAL`, `LONGCARD`, `INTEGER` and `LONGINT` are whole number types.

4.3.5 The Semantics of Real Number Types

Real number types are data types that represent subsets of the mathematical set of real numbers ℝ, with a finite number of values which may be approximations of the real numbers they represent.

The following operations are defined for real number types:

- assignment of literals and expressions (`:=`)
- type conversions (`::`)
- scalar conversion (`SXF`, `VAL`)
- smallest value (`TMIN`)
- largest value (`TMAX`)
- absolute value (`ABS`)
- sign reversal (`NEG`)
- addition (+)
- postfix increment (++)
- difference (-)
- postfix decrement (`--`)
- multiplication (`*`)
- division (`/`)
- equal (=)
- not-equal (#)
- less-than (<)
- less-or-equal (<=)
- greater-than (>)
- greater-or-equal (>=)

Predefined data types `REAL` and `LONGREAL` are real number types.

4.3.6 The Semantics of Array Types

Array types are compound data types whose components are all of the same type. The following operations are defined for array types:

- assignment of structured literals and expressions (`:=`)
- store component (`array[index] := value`)
- retrieve component (`value := array[index]`)
- obtain component capacity limit (`TLIMIT`)
- obtain number of components (`COUNT`)
- component iteration (`FOR index IN array`)
- equal (=)
- not-equal (#)

All data types defined using the `ARRAY OF` type constructor are array types.

4.3.7 The Semantics of Character String Types

Character string types are arrays or ordered collections whose components are character types. The following operations are defined for character string types:

- assignment of string literals, structured literals and expressions (`:=`)
- store component (`string[index] := value`)
- retrieve component (`value := string[index]`)
- obtain character capacity limit (`TLIMIT`)
- obtain string length (`LENGTH`)
- component iteration (`FOR char IN string`)
- concatenation (+)
- equal (=)
- not-equal (#)

All character string data types defined using the `ARRAY OF CHAR` and `ARRAY OF UNICHAR` type constructor are character string types.

4.3.8 The Semantics of Collection Types

Collection types are data types that represent containers for an arbitrary number of key-value pairs. The following operations are defined for collection types:

- allocation (`NEW`)
- retention (`RETAIN`)
- release and deallocation (`RELEASE`)
- assignment of entities of the same type (`:=`)
- store value by key (`collection[key] := value`)
- retrieve value for key (`value := collection[key]`)
- remove value for key (`collection[key] := NIL`)
- obtain key/value pair capacity limit (`TLIMIT`)
- obtain number of key/value pairs (`COUNT`)
- key is present test (`IN`)
- iteration by key (`FOR key IN collection`)
- equal (=)
- not-equal (#)

All data types defined using the `ASSOCIATIVE ARRAY OF` type constructor are collection types.

4.3.9 The Semantics of Record Types

Record types are compound data types whose components are of arbitrary types. The following operations are defined for record types:

- assignment of structured literals and expressions (`:=`)
- store component (`record.component := value`)
- retrieve component (`value := record.component`)
- equal (=)
- not-equal (#)

All data types defined using the `RECORD` type constructor are record types.

4.3.10 The Semantics of Pointer Types

Pointer types are data types that represent references to a storage location. The following operations are defined for pointer types:

- assignment of `NIL` and expressions (`:=`)
- allocation (`NEW`)
- deallocation (`RELEASE`)

- dereference (`^`)
- equal (=)
- not-equal (#)

The invalid pointer value `NIL` may not be dereferenced. An attempt to do so shall cause the runtime system to raise a runtime fault of type `DerefNil`.

All data types defined using the `POINTER TO` type constructor are pointer types.

4.3.11 The Semantics of Procedure Types

Procedure types are special pointer types that reference the storage location of a procedure and store the formal parameters of a procedure prototype. The following operations are defined for procedure types:

- assignment of `NIL` and expressions (`:=`)
- procedure call
- equal (=)
- not-equal (#)

All procedure types defined using the `PROCEDURE` type constructor are procedure types and all procedures and functions are values of a procedure type.

4.3.12 The Semantics of Opaque Types

Opaque types are data types whose structure and semantics are only available in the implementation part of the library module that defines the opaque type. Outside the implementation part the following operations are defined:

For opaque pointers:

- allocation (`NEW`)
- deallocation (`RELEASE`)
- assignment (`:=`)
- equal (=)
- not-equal (#)

For opaque records:

- assignment (`:=`) of entities of the same type only
- equal (=)
- not-equal (#)

All data types defined using a sole `OPAQUE` type constructor are opaque pointer types. All data types defined using the `OPAQUE RECORD` type constructor are opaque record types.

4.3.13 The Semantics of UNSAFE Types

Types in module `UNSAFE` are data types that represent low-level storage units or storage locations. These types do not overflow nor underflow but wrap-around instead. The following operations are defined for `UNSAFE` types:

- assignment (`:=`)
- odd/even test (`ODD`)[1]
- addition (+)
- postfix increment (++)
- difference (`-`)
- postfix decrement (`--`)
- equal (=)
- not-equal (#)

Types `BYTE`, `WORD`, `MACHINEBYTE`, `MACHINEWORD` and `ADDRESS` in module `UNSAFE` are `UNSAFE` types.

[1] For some target architectures odd/even tests on addresses may always return the same result.

5 Definitions and Declarations

A definition is a directive that defines an identifier in the public interface of a library module. A declaration is a directive that declares an identifier in a program module or in the implementation part of a library module. There are four types of definitions and declarations:

- constant definitions and declarations
- variable definitions and declarations
- type definitions and declarations
- procedure definitions and declarations

5.1 Constant Definitions and Declarations

A constant is an immutable value determined at compile time. A constant may be defined or declared as an alias of another constant, but not as an alias of a module, a variable, a type or a procedure.

EBNF:
```
constDeclaration :
    CONST ( Ident "=" constExpression ";" )* ;
```

Examples:
```
CONST zero = 0; maxInt = TMAX(INTEGER); bufSize = MAX(fooLen, barLen, bazLen);
```

5.2 Variable Definitions and Declarations

A variable is an entity to store a value. A variable always has a type and it can only hold values of its type. The type of a variable is immutable. The value of a variable is undetermined when it is allocated. However, variables of pointer types are automatically initialised with the invalid pointer value `NIL`.

EBNF:
```
varDeclaration :
    VAR identList “:” ( range OF )? typeIdent ";" ;
```

Examples:
```
VAR ch : CHAR; x, y, z : REAL; cardinals1to100 : [1 .. 100] OF CARDINAL;
```

5.2.1 Global Variables

A variable defined or declared in the top level of a module has a global life span. It exists throughout the entire runtime of the program. However, a global variable does not have global scope. It is only visible within the module where it is defined or declared, and if it is exported, within modules that import it.

A variable defined in the top level of a library module’s definition part is always exported immutable. It may be assigned to within the library module's implementation part but it may not be assigned to within modules that import it. A pointer variable defined in the top level of a definition part shall cause a promotable hard compile time warning unless its type is opaque or a `POINTER TO CONST` type.

Example:
```
DEFINITION MODULE GlobalPtrVars;
TYPE ImmFooPtr : POINTER TO CONST Foo;  TYPE FooPtr = POINTER TO Foo;
VAR immfoo : ImmFooPtr; (* OK *)  VAR foo : FooPtr; (* compile time warning *)
```

5.2.2 Local Variables

A variable declared within a procedure has local life span and local scope. It only exists during the lifetime of the procedure and it is only visible within the procedure where it is declared, and within procedures local to the procedure where it is declared.

5.3 Type Definitions and Declarations

Types are defined and declared using a type definition or declaration.

EBNF:
```
typeDefinition :
    TYPE ( Ident = ( type | opaqueType ) ";" )* ;
typeDeclaration :
    TYPE ( Ident = type ";" )* ;
type :
    (( ALIAS | SET | range ) OF )? namedType | enumerationType |
    arrayType | recordType | setType | pointerType | procedureType ;
range :
    "[" constExpression ".." constExpression "]" ;
namedType : qualident ;
```

Examples:
```
TYPE Volume = INTEGER;
TYPE HashTable = OPAQUE;
```

5.3.1 Strict Name Equivalence

By default, types of different names are always incompatible even if they are derived from the same base type.

Example:
```
TYPE Celsius = REAL; Fahrenheit = REAL;
VAR celsius : Celsius; fahrenheit : Fahrenheit;
celsius := fahrenheit; (* compile time error: incompatible types *)
```

In order to assign values across type boundaries, type conversion is required.

Example:
```
celsius := (fahrenheit :: Celsius - 32.0) * 100.0/180.0; (* type conversion *)
```

5.3.2 Alias Types

For a type to be compatible with another type it must be defined or declared as an `ALIAS` type using the `ALIAS OF` type constructor.

EBNF:
```
aliasType :
    ALIAS OF typeIdent ;
```

Example:
```
TYPE INT = ALIAS OF INTEGER;
VAR i : INT; j : INTEGER;
i := j; (* i and j are compatible *)
```

5.3.3 Set Types

A set type is a container type for a finite number of elements from a finite value space. The value space of a set contains all possible elements of the set.

Set types are defined using the `SET OF` type constructor.

EBNF:
```
setType :
    SET OF enumTypeIdent ;
enumTypeIdent : typeIdent ;
```


A value that is an element of the set is said to be a member of the set. To test or modify the membership of a value in a set, it may be addressed by selector. Membership is of type `BOOLEAN`, is is either `TRUE` or `FALSE`.

Example:
```
TYPE ColourSet = SET OF Colour;
VAR colours : ColourSet;
colours := { Colour.red, Colour.green }; colours[Colour.blue] := FALSE;
IF Colour.blue IN colours THEN WRITE("blue is on\n") END;
```

Set types defined with the `SET OF` type constructor can hold at most 128 elements.

5.3.4 Subrange Types

A subrange type is a type defined as a subset of a scalar or ordinal type. A subrange type is upwards compatible with its base type, but the base type is not downwards compatible with any of its subrange types. A subrange type inherits all of its properties from its base type, except for its `TMIN` and `TMAX` values.

Subrange types are defined using subrange type constructor syntax.

EBNF:
```
subrangeType :
    "[" lowerBound ".." upperBound "]" OF typeIdent ;
lowerBound :
    ">"? constExpression ;
upperBound :
    "<"? constExpression ;
```

Both lower and upper bound must always be compatible with the base type. A > symbol denotes an open lower bound. Values that are less than or equal to an open lower bound expression are not legal values of the subrange type. A < symbol denotes an open upper bound. Values that are greater than or equal to an open upper bound expression are not legal values of the subrange type.

Examples:
```
TYPE BaseColours = [red .. blue] OF Colours;
TYPE Natural = [1 .. TMAX(CARDINAL)] OF CARDINAL;
TYPE Radian = [0.0 .. tau] OF REAL;
TYPE PositiveReal = [>0.0 .. TMAX(REAL)] OF REAL;
TYPE NegativeReal = [TMIN(REAL) .. <0.0] OF REAL;
```

Only real number expressions may follow an open bound marker.

5.3.5 Opaque Types

A type may be defined as an opaque type. The identifier of an opaque type is available in the library where it is defined and in modules that import it. However, the implementation details of an opaque type are only available within the implementation part of the library where it is defined. This facility is useful for the construction of abstract data types. There are two types of opaque types:

- opaque pointer types
- opaque record types

5.3.5.1 Opaque Pointers

An opaque pointer type is a pointer to a type whose declaration is hidden in the corresponding implementation part. Entities of the abstract data type can only be allocated dynamically at runtime.

EBNF:
```
opaquePointerDefinition : TYPE Ident "=" OPAQUE ";" ;
```


Example:
```
DEFINITION MODULE Tree;
TYPE Tree = OPAQUE; (* opaque pointer *)
(* public interface *)
END Tree.

IMPLEMENTATION MODULE Tree;
TYPE Tree = POINTER TO TreeDescriptor;
TYPE TreeDescriptor = RECORD left, right : Tree; value : ValueType END;
(* implementation *)
END Tree.

IMPORT Tree;
VAR tree : Tree;
NEW(tree); (* dynamic allocation of a variable of abstract data type Tree *)
```

5.3.5.2 Opaque Records

An opaque record type is an opaque type that represents a record type instead of a pointer to a record type. Whereas an entity of an abstract data type that is based on an opaque pointer can only be allocated dynamically at runtime, entities of an abstract data type based on an opaque record are not limited to dynamic allocation. Variables of an opaque record type can also be allocated statically, both as global and as local variables.

In order for the compiler to be able to allocate a variable of an opaque record type statically, it must be able to determine its allocation size. However, the allocation size of a record can only be determined from the record type's declaration. For this reason, the declaration of an opaque record type is lexically located in the definition part. Nevertheless, it is semantically treated as if it was hidden in the corresponding implementation part in order to preserve encapsulation.

Therefore, only the identifier of an opaque record is visible to modules that import it. Its internal structure is not available to them. Any attempt to access the fields of an opaque record outside of the module in which it is implemented shall cause a compile time error.

EBNF:
```
opaqueRecordDefinition : TYPE Ident "=" OPAQUE recordType ";" ;
```

Example:
```
DEFINITION MODULE BigInteger;
TYPE BigInteger = OPAQUE RECORD highDigits, lowDigits : INTEGER END;
(* public interface *)
END BigInteger.

IMPLEMENTATION MODULE BigInteger;
...
i := bigInt.highDigits; (* fields visible in implementation part *)
...
END BigInteger.

IMPORT BigInteger;
VAR bigInt : BigInteger; i : INTEGER;
i := bigInt.highDigits; (* compile time error: hidden component *)
```

5.3.6 Anonymous Types

An anonymous type is a type that does not have a type identifier associated with it. In languages with name equivalence, the names of the types of variables must be examined to determine whether or not they are assignment or expression compatible. If the types do not have names, their compatibility cannot be determined. Anonymous types are therefore of very limited use in languages with name equivalence.

Nevertheless, Modula-2 R10 permits the use of anonymous types in three cases:

- in the form of formal open array parameters
- in the form of an anonymous array type in indeterminate record field declarations
- in the form of an anonymous subrange type in variable and record field declarations

Any other use of anonymous types shall result in a compile time error.

5.3.7 Enumeration Types

An enumeration type is an ordinal type whose legal values are defined by a list of identifiers. The identifiers are assigned ordinal values from left to right. The ordinal value assigned to the leftmost value is always zero.

EBNF:
```
enumerationType :
    "(" ( "+" namedType “,”)? identList ")" ;
```

When referencing an enumerated value, its identifier must be qualified with the name of its type, except within a subrange type constructor. This requirement fixes a flaw in classic Modula-2 where importing enumeration types could cause name conflicts.

Example:
```
TYPE Colour = ( red, green, blue, orange, magenta, cyan );
TYPE BaseColour = [red .. blue] OF Colour; (* unqualified identifiers *)
VAR colour : Colour;
colour := Colour.green; (* qualified identifier of value green *)
```

The very first item in the list of identifiers that define the legal values of an enumeration type may contain a reference to another enumeration type. When another enumeration type is referenced within an enumerated list all the identifiers listed in the referenced type become legal values of the new type.

The allocation size of an enumeration type is always 16-bit. Its maximum range is 65536 values.

Example:
```
TYPE Colour = ( red, green, blue );
TYPE MoreColour = ( +Colour, orange, magenta, cyan );
(* equivalent to: MoreColour = ( red, green, blue, orange, magenta, cyan ); *)
```

5.3.8 Indexed Array Types

An indexed array type is a compound type whose components are all of the same type and are addressable by cardinal index. The lowest index is always zero. The number of components is specified by the formal array index parameter which shall be of an unsigned whole number type and its value shall never be zero. Array types are defined using the `ARRAY` type constructor.

EBNF:
```
arrayType :
    ARRAY componentCount ( "," componentCount )* OF namedType ;
componentCount : cardinalConstExpression ;
```

Example:
```
TYPE IntArray = ARRAY 10 OF INTEGER;
VAR array : IntArray;
array := { 0 BY 10 }; (* initialise all values with zero *);
FOR item IN array DO item := 0 END; (* another way to initialise *)
WRITE(array);
```


5.3.9 Record Types

A record type is a compound type whose components are of arbitrary types. The components are called fields. Record types may be defined as extensions of other record types. Such a type is called a type extension, the type it is based on is called its base type. However, the base type of a type extension may not be an opaque record nor an indeterminate record. The names of the fields of the base type may not be used again as field names in the type extension.

Record types are defined using the `RECORD` type constructor.

EBNF:

```
recordType :
    RECORD ( fieldList ( ';' fieldList )* indeterminateField? |
    "(" baseType ")" fieldList ( ";" fieldList )* ) END ;
fieldList :
    identList ':' ( range OF )? typeIdent ;
baseType : typeIdent ;
indeterminateField :
    INDETERMINATE Ident ':' ARRAY discriminantField OF typeIdent ;
discriminantField : Ident ;
```

Example:

```
TYPE Point = RECORD x, y : REAL END;
TYPE ColourPoint = RECORD ( Point ) colour : Colour END;
VAR point : Point; cPoint : ColourPoint;
cPoint := { 0.0, 0.0, Colour.black }; point := cPoint :: Point;
```

5.3.10 Indeterminate Record Types

An indeterminate record type is a record type that contains exactly one indeterminate field and exactly one discriminant field. An indeterminate field is a field whose type is indeterminate. A type is indeterminate if its allocation size cannot be determined from its type declaration. A discriminant field is a field that determines the size of an indeterminate field.

5.3.10.1 Declaration of Indeterminate Record Types

The type declaration of an indeterminate record must declare:

- one discriminant field that is of a whole number type
- one indeterminate array field that references the discriminant field as its size

The discriminant field may be any field other than the last field and the indeterminate field is always the last field. An indeterminate record type may be the target type of a pointer type but it may not be a type extension, the type of a record field or the base type of an array or type extension.

Example:

```
TYPE VLADescriptor = RECORD
    size    : CARDINAL; (* discriminant field *)
    a, b, c : Foo; (* other, arbitrary fields *)
INDETERMINATE
    buffer  : ARRAY size OF OCTET (* indeterminate field *)
END; (* VLADescriptor *)
```

5.3.10.2 Allocating Indeterminate Records

Records of an indeterminate type may only be allocated dynamically at runtime using predefined procedure `NEW`. When the record is allocated, the discriminant value must be passed to `NEW` as an additional parameter. Any attempt to allocate a record of indeterminate type without passing the discriminant value shall result in a compile time error.

The compiler replaces any invocation of `NEW` for an indeterminate record type with a call to library procedure `ALLOCATE` passing the correct allocation size using the formula:

```
allocSize(T) = TSIZE(T) + discriminant * TSIZE(baseType(T.indeterminateField))
```

where `T` is the indeterminate record type, discriminant is the discriminant value passed to `NEW` and `baseType(T.indeterminateField)` is the base type of the array of the indeterminate field. The value returned by `TSIZE` for an indeterminate record type is its allocation size without the indeterminate field.

Example:
```
VAR vla : VLA;
NEW(vla, 100); (* allocate VLA record with 100 buffer elements *)
```
Compiled as:
```
ALLOCATE(vla, TSIZE(VLADescriptor) + 100 * TSIZE(OCTET)); vla^.size := 100;
```

5.3.10.3 Immutability of the Discriminant Field

The discriminant field of an indeterminate record type is automatically initialised when it is allocated. After initialisation the discriminant field becomes immutable. Its immutability is enforced as follows:

- a discriminant field may not be passed to any procedure as a `VAR` parameter
- a discriminant field may not appear on the left hand side of an assignment
- a discriminant field may not be the designator in a ++ or `--` statement

Examples:
```
INC(vla^.size); (* discriminant field may not be passed as VAR parameter *)
vla^.size := 42; (* discriminant field may not be assigned to *)
vla^.size++; (* discriminant field may not be used with ++ or -- *)
```

5.3.10.4 Run-time Bounds Checking

Access to the indeterminate array field of a record of indeterminate type is bounds checked at runtime in the same manner as access to a determinate array is checked. The compiler automatically inserts the code to check array indices against the discriminant field. Any attempt to access the array with a subscript that is out of bounds shall result in a run-time error.

5.3.10.5 Assignment Compatibility

The assignment compatibility of two records of indeterminate type cannot be verified at compile time. For this reason records of indeterminate type can only be copied field-wise, not record-wise.

5.3.8.6 Parameter Passing

Since the compatibility of records of indeterminate types cannot be determined at compile time, they may not be formal types. A record of indeterminate type may therefore only be passed to an auto-casting formal open array parameter `CAST ARRAY OF OCTET`, `CAST ARRAY OF UNSAFE.BYTE` or `CAST ARRAY OF UNSAFE.WORD`, or to a formal pointer type parameter whose target type is the indeterminate type.

The indeterminate field of an indeterminate record may be passed to an open array parameter whose base type is assignment compatible with the base type of the indeterminate field.

5.3.10.7 Deallocating Indeterminate Records

Records of indeterminate type may only be deallocated using predefined procedure `RELEASE`.

5.3.11 Pointer Types

A pointer type is a container for a typed reference to an entity at a storage location. The type of the entity pointed to is called the base type. Pointer types are defined using the `POINTER TO` type constructor.

EBNF:
```
pointerType : POINTER TO CONST? typeIdent ;
```

Example:
```
TYPE IntPtr = POINTER TO INTEGER;
TYPE ImmIntPtr = POINTER TO CONST INTEGER;
VAR intPtr: IntPtr; immPtr : ImmIntPtr; int : INTEGER;
intPtr := PTR(int, IntPtr); immPtr := PTR(int, ImmIntPtr); int := 0;
intPtr^ := 1; (* OK, modifying a mutable entity *)
immPtr^ := 1; (* compile time error: attempt to modify an immutable entity *)
```

5.3.12 Procedure Types

A procedure type is a special container for typed references to procedures of given procedure headers. Procedure types are defined using the `PROCEDURE` type constructor.

EBNF:
```
procedureType :
    PROCEDURE ( "(" formalTypeList ")" )? ( ":" returnedType )? ;
formalTypeList :
    formalType ( "," formalType )* ;
formalType :
    attributedFormalType | variadicFormalType ;
attributedFormalType :
    ( CONST | VAR )? simpleFormalType ;
simpleFormalType :
    CAST? ( ARRAY OF )? namedType ;
variadicFormalType :
    VARIADIC OF
    ( attributedFormalType |
      "(" attributedFormalType ( "," attributedFormalType )* ")"  )
returnedType : namedType ;
```

Example:
```
TYPE WriteStrProc = PROCEDURE ( CONST ARRAY OF CHAR );
TYPE FSM = PROCEDURE ( CONST ARRAY OF CHAR, FSM );
VAR WriteStr : WriteStrProc;
WriteStr := Terminal.WriteString; WriteStr("hi!");
```

5.4 Procedure Definitions and Declarations

A procedure is a sequence of statements with its own local scope, identified by a name. In Modula-2, procedures may have zero or more associated parameters and they may or may not return a result. Procedures that return a result are called function procedures, those that do not return a result are called regular procedures. A procedure consists of two parts:

- procedure header
- procedure body

Typically, procedure definitions are placed in a library module's definition part and corresponding procedure declarations are placed in the library's implementation part.

5.4.1 The Procedure Header

The procedure header represents the interface of a procedure. A procedure header always defines the identifier of the procedure. It may further define a binding to an operator or predefined procedure, the procedure's formal parameter list and its return type.

A procedure header may only define a binding to an operator or predefined procedure if it belongs to a global definition in an ADT library module that specifies a prototype in its module header and if the binding is required by the prototype's definition.

The signature of a procedure that binds to an operator or predefined procedure must conform to the language defined signature for the respective operator.[1]

EBNF:

```
procedureHeader :
    PROCEDURE
    ( "[" ( "::" | bindableEntity ) "]" )?
    ident ( "(" formalParamList ")" )? ( ":" returnedType )? ;
procedureIdent : Ident ;
```

Examples:

```
PROCEDURE IsNegative ( x : INTEGER ) : BOOLEAN;
PROCEDURE [+] add ( a, b : BCD ) : BCD; (* procedure binding to + operator *)
```

5.4.2 The Procedure Body

A procedure body consists of zero or more local variable declarations, zero or more local procedure declarations and the procedure's execution block that represents the sequence of actions that perform the procedure's intended task. A procedure declaration always repeats the procedure definition and is followed by the procedure body.

EBNF:

```
procedureBody : block ident ;
procedureDeclaration : procedureHeader ";" procedureBody ;
```

Example:

```
PROCEDURE IsNegative ( x : INTEGER ) : BOOLEAN; (* header *)
BEGIN (* body *)
    IF x < 0 THEN RETURN TRUE ELSE RETURN FALSE END
END IsNegative;
```

5.4.3 Formal Parameters

The parameters in the parameter list of a procedure header are called the procedure's formal parameters. There are simple formal parameters and variadic formal parameters.

EBNF:

```
formalParamList : formalParams ( ";" formalParams )* ;
formalParams : simpleFormalParams | variadicFormalParams ;
```

5.4.3.1 Simple Formal Parameters

A formal parameter always specifies a type, and it may or may not specify an attribute. A formal parameter's attribute determines the parameter passing convention of the formal parameter.

EBNF:

```
simpleFormalParams :
    ( CONST | VAR )? identList ":" simpleFormalType ;
```

There are three parameter passing conventions:

- pass by value
- pass by reference, mutable
- pass by reference, immutable

[1] Language defined signatures for bindings have yet to be documented.

5.4.3.2 Pass By Value

The default parameter passing convention is pass-by-value. It is used when no attribute is specified for a parameter or parameter list. A parameter passed by value is called a value parameter. When a variable or value is passed to a value parameter, a copy of the value is passed to the called procedure and the scope of the copy is the procedure's block.

Example:
```
PROCEDURE IsNegative ( x : INTEGER ) : BOOLEAN;
(* no attribute => pass-by-value *)
```

5.4.3.3 Pass By Reference – Mutable

The pass-by-mutable-reference convention is used when the `VAR` attribute is specified for a parameter or parameter list. A parameter passed this way is called a `VAR` parameter. When a variable is passed to a `VAR` parameter, a mutable reference to the variable is passed to the called procedure which may then modify the value of the variable. Immutable entities may therefore not be passed by mutable-reference.

Example:
```
PROCEDURE Increment ( VAR x : INTEGER );
(* VAR => pass-by-reference, mutable *)
BEGIN
    x++; (* modifies original *)
END Increment;
number := 1; Increment(number); (* value of number is now 2 *)
CONST zero = 0; Increment(zero); (* compile time error: immutable entity *)
```

5.4.3.4 Pass By Reference – Immutable

The pass-by-immutable-reference convention is used when the `CONST` attribute is specified for a parameter or parameter list. A parameter passed by immutable-reference is called a `CONST` parameter. When the pass-by-immutable-reference convention is used, an immutable reference to the parameter is passed to the called procedure and the procedure may not modify a passed-in value. That is, within the scope of the procedure the parameter is treated as if it was a constant. Both mutable and immutable entities may be passed as `CONST` parameters.

Example:
```
PROCEDURE WriteString ( CONST s : ARRAY OF CHAR );
(* CONST => pass-by-reference, immutable *)
```

5.4.3.5 Variadic Formal Parameters

A variadic procedure is a procedure that can accept a variable number of parameters. A variadic parameter is a formal parameter to which a variable number of actual parameters may be passed.

Modula-2 R10 provides variadic formal parameters both for safe and unsafe use cases:

- unsafe variadic formal parameters for interfacing to unsafe foreign variadic procedures
- type safe variadic formal parameters for implementing type safe variadic procedures in Modula-2

Facilities to define procedure headers with unsafe variadic formal parameters for interfacing to unsafe foreign variadic procedures are provided by pseudo-module `UNSAFE` and are described in detail in section 11.4.1 (“Mapping to Unsafe Variadic Procedures in Foreign APIs”).

In support of procedures with type safe variadic formal parameters, a formal parameter list may contain one or more variadic parameters denoted by reserved word `ARGLIST`.

EBNF:
```
variadicFormalParams :
    ARGLIST numberOfArgumentsToPass? OF
    ( simpleFormalType | "{" simpleFormalParams ( ";" simpleFormalParams )* "}" )
    ( "|" variadicTerminator )? ;
numberOfArgumentsToPass, variadicTerminator : constExpression ;
```

There are two variadic parameter passing conventions:

- variadic counter
- variadic list terminator

5.4.3.6 Variadic Counter

When the variadic-counter convention is used, the compiler determines the number of actual parameters passed in the procedure call and inserts the resulting value as an additional parameter immediately before the variadic argument list. The counter is of type `CARDINAL`.

Example:
```
PROCEDURE Variadic ( v : ARGLIST OF INTEGER );
```
Invoked as:
```
Variadic(0, 1, 2, 3, 4); (* passing five arguments *)
```
Compiled as:
```
Variadic(5, 0, 1, 2, 3, 4); (* argument count inserted before argument list *)
```

Within the body of a variadic procedure, the variadic argument list may be iterated using a `FOR IN` loop over the variadic parameter. Predefined function `COUNT` returns the value of the variadic counter.

Example:
```
PROCEDURE Average ( v : ARGLIST OF REAL ) : REAL;
VAR sum : REAL;
BEGIN sum := 0.0;
  (* iterate over variadic argument list v *)
  FOR item IN v DO
    sum := sum + item
  END;
  (* calculate average from sum and argument count *)
  RETURN sum / COUNT(v) :: REAL;
END Average;
```

The variadic-counter convention may also be used when mapping to or replacing a variadic C function that expects a variadic counter of unsigned type immediately before its variadic argument list.

5.4.3.7 Variadic List Terminator

When the variadic-list-terminator convention is used, the compiler appends a terminator value specified in the formal variadic parameter to the end of the list of actual parameters passed. The terminator value must be of the same type as the base type of the variadic list it terminates.

Example:
```
CONST terminator = -1; (* variadic list terminator *)
PROCEDURE Variadic ( v : ARGLIST OF INTEGER | terminator );
```
Invoked as:
```
Variadic(0, 1, 2, 3, 4); (* passing five arguments *)
```
Compiled as:
```
Variadic(0, 1, 2, 3, 4, -1); (* list terminator appended to argument list *)
```

If the formal variadic parameter consists of multiple components, then the terminator must be of the same type as the first component of the formal variadic parameter.

Example:
```
CONST terminator = “”; (* Empty string to terminate variadic list *)
PROCEDURE Variadic ( v : ARGLIST  OF { key : Str; val : REAL } | terminator );
```

Within the body of a variadic procedure, the variadic argument list may be iterated using a `FOR IN` loop over the variadic parameter. The variadic parameter may not be passed to predefined function `COUNT`.

Example:
```
CONST terminator = -1; (* variadic list terminator *)
PROCEDURE Variadic ( v : ARGLIST OF INTEGER | terminator );
BEGIN
   FOR item IN v DO
     WRITE(f, item)
   END
END Variadic;
```

The variadic-list-terminator convention may also be used when mapping to or replacing a variadic C function that expects a terminator value to indicate the end of its variadic argument list.

5.4.3.8 Variadic List With Multiple Components

A variadic formal parameter may contain multiple components. This is useful to define procedures that can accept a variable number of value pairs or other tuples.

Example:
```
PROCEDURE Insert ( tree : Tree; arglist : ARGLIST OF { key : Key; value : Value } );
```
Invoked as:
```
Insert(tree, "foo", 123, "bar", 456, "baz", 789);
```
Compiled as:
```
Insert(tree, 3, "foo", 123, "bar", 456, "baz", 789);
```

Alternatively, a variadic parameter list may be passed as a structured value as long as the structured value is structurally equivalent to the formal variadic parameter to which it is passed.

Example:
```
Insert(tree, {"foo", 123, "bar", 456, "baz", 789});
```

5.4.3.9 Variadic List Followed By Further Parameters

If a variadic formal parameter is followed by further formal parameters, then the actual variadic parameter list can only be passed as a structured value in order to allow the compiler to determine with certainty where the variadic list ends. Failing to pass the variadic parameter list as a structured value will result in a compile-time error.

Example:
```
PROCEDURE Insert ( t : Tree; a : ARGLIST OF { k : Key; v : Value }; VAR s : Status );
```
Invoked as:
```
Insert(tree, {"foo", 123, "bar", 456, "baz", 789}, status);
```
Compiled as:
```
Insert(tree, 3, "foo", 123, "bar", 456, "baz", 789, status);
```

5.4.3.10 Requiring A Fixed Number Of Arguments

A formal `ARGLIST` parameter may specify a requirement for a fixed number of arguments to be passed to it.

Example:
```
PROCEDURE initWithList ( VAR a : Array; list : ARGLIST 5 OF REAL );
initWithList(array, 1.0, 2.0, 3.0, 4.0, 5.0); (* OK *)
initWithList(array, 1.0, 2.0, 3.0); (* compile time error: incorrect argument count *)
```


5.4.3.11 Open Array Parameters

A formal parameter may be declared as an open array parameter. An open array parameter has an anonymous array type without any index specified. Any array whose component type matches that of the open array may then be passed as an actual parameter.

Example:
```
TYPE String10 = ARRAY 10 OF CHAR;
VAR str : String10;
PROCEDURE Write ( s : ARRAY OF CHAR );
str := "hello"; Write(str); (* any CHAR array may be passed *)
```

The component count of an array passed as an open array parameter is automatically passed as a hidden parameter before the open array parameter. The count parameter is of type `LONGCARD`.

Example:
```
PROCEDURE Write ( s : ARRAY OF CHAR );
```
Invoked as:
```
Write("the quick brown fox"); (* 19 characters plus null-terminator *)
```
Compiled as:
```
Write(20, "the quick brown fox"); (* component count 20 inserted before s *)
```

Within the body of the procedure, the passed-in array may be iterated using a `FOR IN` loop over the open array parameter. Predefined function `COUNT` returns the component count.

Example:
```
PROCEDURE LetterCount ( s : ARRAY OF CHAR );
VAR letters : LONGCARD;
BEGIN letters := 0;
  FOR ch IN s DO
    IF ASCII.IsLetter(ch) THEN letters++ END;
  END;
  WRITE("character count : "); WRITE(COUNT(s)); WriteLn;
  WRITE("letter count    : "); WRITE(letters); WriteLn
END LetterCount;
```

5.4.3.12 Auto-Casting Open Array Parameters

A formal parameter may be declared as an auto-casting open array parameter with component type `OCTET`, `UNSAFE.BYTE` or `UNSAFE.WORD`. Any value of any type may then be passed as an actual parameter and it is cast automatically to the array type of the formal parameter. This facility is useful for system-level programming tasks but type safety is no longer guaranteed. Therefore `CAST` must be explicitly imported from pseudo-module `UNSAFE` to declare an auto-casting formal parameter.

Example:
```
FROM UNSAFE IMPORT CAST;
VAR str : String10; x : LONGBITSET;
PROCEDURE Copy ( CONST source : CAST ARRAY OF OCTET;
                 VAR target   : CAST ARRAY OF OCTET );
str := "hello"; Copy(str, x); (* casting copy *)
```


5.4.4 Procedure Type Compatibility

The types of the formal parameters and the return type of a procedure are collectively called the procedure's signature. A procedure's signature determines its type. Procedures and procedure variables are compatible if they are of the same type, that is, their respective signatures must match.

Example:
```
VAR p : PROCEDURE ( VAR ARRAY OF CHAR );
PROCEDURE StripTabs ( VAR s : ARRAY OF CHAR );
PROCEDURE WriteString ( CONST s : ARRAY OF CHAR );
p := StripTabs; (* OK *)
p := WriteString; (* compile time error: incompatible types *)
```

5.4.5 Operator Bound Procedures

A procedure may be defined to bind to an operator or a predefined procedure in respect of an abstract data type defined to conform to a blueprint. Except for bindings to the conversion operator which are always permitted, only bindings required by the blueprint the ADT conforms to may be defined.

Example:
```
(* Module BCD is required to conform to blueprint ProtoReal,
   which requires a binding to the + operator to be defined *)
DEFINITION MODULE BCD [ProtoReal];
TYPE BCD = OPAQUE RECORD value : ARRAY 8 OF OCTET END;
(* binding procedure add to the + operator for operands of type BCD *)
PROCEDURE [+] add ( a, b : BCD ) : BCD;
(* binding procedure toREAL to the :: operator for conversions to type REAL *)
PROCEDURE [::] toREAL ( b : BCD ) : REAL;
...
END BCD.
```

6 Statements

A statement is an action that can be executed to cause a transformation of the computational state of a program. Statements are used for their effects only, they do not return any values and may not be used within expressions. There are ten types of statements:

- assignments
- post-increment and post-decrement statements
- procedure calls
- if statements
- case statements
- while statements
- repeat statements
- loop statements
- for statements
- exit statements
- return statements

6.1 Assignments

An assignment statement is used to assign a value to a mutable variable. The right hand side of the assignment must be assignment compatible with its left hand side.

EBNF:

```
assignment : designator ":=" expression ;
designator : qualident ( ( "[" exprListOrSlice "]" | "^" ) ( "." ident )* )+ ;
```

Examples:

```
VAR ch : CHAR; i : INTEGER; r : REAL; z : COMPLEX; a : Array10;
ch := "a"; i := 12345; r := 3.1415926; z := { 1.2, 3.4 }; a[5] := 0;
```

6.2 Postfix-Increment and Postfix-Decrement Statements

A postfix-increment adds one to, a postfix-decrement subtracts one from a whole number variable.

EBNF:

```
incrementOrDecrement : designator ( "++" | "--" ) ;
```

Examples:

```
lineCounter++; index--;
```

6.3 Procedure Calls

A procedure call statement is used to invoke a procedure. It may include a list of parameters to be passed to the called procedure. Parameters passed are called actual parameters, those defined in the procedure's header are called formal parameters. In every procedure call, the types of actual and formal parameters must match. Procedure calls may be recursive, that is, a procedure may call itself. Recursive calls shall be optimised by eliminating tail call recursion.

EBNF:

```
procedureCall : designator ( "(" expressionList? ")" )? ;
```

Examples:

```
Insert( tree, “Fred Flintstone”, 42 ); ClearBuffers;
```

6.4 IF Statements

An `IF` statement is a conditional flow-control statement. It evaluates a condition in form of a boolean expression. If the condition is true then flow control passes to its `THEN` block. If the condition is false and an `ELSIF` branch follows, then flow control passes to the `ELSIF` branch to evaluate yet another condition in the

ELSIF branch. Again, if the condition is true then flow control passes to the THEN block of the ELSIF branch. If there are no ELSIF branches or if the conditions of all ELSIF branches are false, and if an ELSE branch follows, then flow control passes to the ELSE's block. IF-statements must always be terminated with an END. At most one block in the statement is executed.

EBNF:
```
ifStatement :
    IF booleanExpression THEN statementSequence
    ( ELSIF booleanExpression THEN statementSequence )*
    ( ELSE statementSequence )?
    END ;
```

Example:
```
IF i > 0 THEN WRITE("Positive")
ELSIF i = 0 THEN WRITE("Zero")
ELSE WRITE("Negative")
END;
```

6.5 CASE Statements

A CASE statement is a flow-control statement that passes control to one of a number of labeled statements or statement sequences depending on the value of an ordinal expression.

EBNF:
```
caseStatement :
    CASE expression OF case ( "|" case )+
    ( ELSE statementSequence )?
    END ;
case : caseLabelList ":" statementSequence ;
caseLabelList : caseLabels ( "," caseLabels )* ;
caseLabels : constExpression ( ".." constExpression )? ;
```

Example:
```
CASE colour OF
|   Colour.red   : WRITE("Red")
|   Colour.green : WRITE("Green")
|   Colour.blue  : WRITE("Blue")
ELSE
    UNSAFE.HALT(1) (* fatal error *)
END;
```

A case label shall be listed at most once. If a case is encountered at run time that is not listed and there is no ELSE clause, no case label statements shall be executed and no error shall result.

6.6 WHILE Statements

A WHILE statement is used to repeat a statement or sequence of statements depending on a condition. The condition is evaluated each time before the DO block is executed. The DO block is repeated as long as the condition evaluates to TRUE.

EBNF:
```
whileStatement : WHILE booleanExpression DO statementSequence END ;
```

Example:
```
WHILE NOT EOF(file) DO READ(file, ch) END;
```

6.7 REPEAT Statements

A REPEAT statement is used to repeat a statement or sequence of statements depending on a condition. The condition is evaluated each time after the REPEAT block has executed. If the condition is TRUE the REPEAT block is repeated, otherwise not.

EBNF:
```
repeatStatement : REPEAT statementSequence UNTIL booleanExpression;
```

Example:
```
REPEAT Read(file, ch) UNTIL ch = terminator END;
```

6.8 LOOP Statements

The `LOOP` statement is used to repeat a statement sequence indefinitely unless terminated by an `EXIT` statement.

EBNF:
```
loopStatement : LOOP statementSequence END ;
```

Example:
```
LOOP
  READ(file, ch);
  IF ch IN TerminatorSet THEN
    EXIT
  END (* IF *)
END; (* LOOP *)
```

6.9 FOR IN Statements

The `FOR IN` statement is used to repeatedly execute a statement or statement sequence while iterating over all values of an ordered value set. Before each iteration, a control variable declared in the loop header is assigned a new value from the value set until no more values are available. By default the iteration order is ascending. If the `DESCENDING` attribute is specified in the loop header, it is descending.

EBNF:
```
forInStatement :
    FOR DESCENDING? controlVariable
    IN ( range OF namedType | designator ) )
    DO statementSequence END ;
controlVariable : Ident;
```

6.9.1 The Control Variable

The control variable of a `FOR IN` loop is declared in the loop header and its type is the component type of the value set. If the type cannot be determined from the value set, it must be specified in the loop header.The scope of the control variable is the loop. It no longer exists after the loop has exited. It is treated as a mutable variable within the loop header and as an immutable variable within the loop body:

- it may not be the left hand side of an assignment
- it may not be the designator in an increment or decrement statement
- it may not be passed as an argument to any `VAR` parameter of a procedure
- it may not be assigned to any pointer other than a `POINTER TO CONST` pointer

6.9.2 The Value Set

The value set of a `FOR IN` loop must be ordered. It may be:

- the designator of an ordinal type
- an anonymous subrange of an ordinal type
- the designator of a variable of an `ARRAY` or `SET` type
- the designator of a variable of an ADT that provides a binding to `FOR`

6.9.2.1 Iterating Over An Ordinal Type

If the value set is the designator of an ordinal type or an anonymous subrange thereof, the type of the control variable is the ordinal type. The loop iterates over all values of the ordinal type or the subrange given in the loop header.

Example:
```
TYPE Colours = (red, green, blue);
FOR colour IN Colours DO WRITE(NameOfColour(colour)) END;
```

6.9.2.2 Iterating Over A Whole Number Type

If the value set is the designator of a whole number type or an anonymous subrange thereof, the type of the control variable is the whole number type. The loop iterates over all values of the whole number type or the subrange given in the loop header.

Examples:
```
FOR number IN CARDINAL DO BottlesOfBeer(number) END;
FOR i IN [0..9] OF CARDINAL DO array[2*i+1] := odd END; (* indices 1, 3, 5, ... *)
```

6.9.2.3 Iterating Over An Array

If the value set is the designator of an array, the type of the control variable is the component type of the array and the loop iterates over all components of the array.

Example:
```
TYPE Array = ARRAY 100 OF REAL; VAR array : Array;
LabExperiment(array); FOR number IN array DO WRITE(number) END;
```

6.9.2.4 Iterating Over A Set

If the value set is the designator of a set, the type of the control variable is the element type of the set and the loop iterates over all elements of the set.

Example:
```
TYPE ColourSet = SET OF Colours; VAR colourSet : ColourSet;
TakeMeasurement(colourSet); FOR colour IN ColourSet DO counter++ END;
```

6.9.2.5 Iterating Over A Collection

If the value set is the designator of a collection, the type of the control variable is the component type of the collection's ADT and the loop iterates over all components of the collection.

Example:
```
IMPORT Dictionary; VAR dictionary : Dictionary;
NEW(dictionary); READ(file, dictionary);
(* iterate over all components of a collection *)
FOR item IN dictionary DO WRITE(item) END;
(* iterate over all keys of a collection *)
FOR key IN Dictionary.KeyType DO WRITE(dictionary[key]) END;
```

6.9.2.6 Iterating Over All Components Of A Structured Value

Iteration over all components of a structured value is a specific use case of iteration over an array or a set. The structured value is assigned to an array or set variable which is then used as value set in the iteration.

Example:
```
IMPORT CHARSET; VAR charset : CHARSET;
charset := {“A”..”Z”, “a”..”z”, “0”..”9”};
FOR char IN charset DO WRITE(char) END;
```


6.10 EXIT Statements

An `EXIT` statement in the body of a `WHILE`, `REPEAT`, `LOOP` or `FOR IN` statement terminates execution of the statement's body and transfers control to the first statement after the body. `EXIT` statements may occur within the body of a `LOOP`, `WHILE`, `REPEAT` or `FOR IN` statement but not anywhere else.

EBNF:

```
exitStatement : EXIT ;
```

Example:

```
LOOP ch := nextChar(stdIn);
  CASE ch OF
  | ASCII.ESC : EXIT
  | (* other case labels *)
```

6.11 RETURN Statements

The `RETURN` statement is used within a procedure body to return control to the calling procedure and in the main body of the program to return control to the operating environment that activated the program. Whether or not a value is returned depends on the type of procedure. When returning from a regular procedure, no value may be returned but when returning from a function procedure, a value of the procedure's return type must be returned, otherwise a compile time error shall occur.

EBNF:

```
returnStatement : RETURN expression? ;
```

Example:

```
PROCEDURE Successor ( x : CARDINAL ) : CARDINAL ;
BEGIN
    RETURN x+1;
END Successor;
```

6.12 Statement Sequences

Statements in a sequence of statements are separated by semicolons.

EBNF:

```
statementSequence : statement ( ";" statement )* ;
```

Example:

```
x := x * 5; counter++; WRITE(file, x)
```

7 Expressions

An expression is a computational formula that evaluates to a value. An expression consists of operands and operators. Operands and operators are described in sub-sections 7.3 and 7.4 respectively.

7.1 Runtime and Compile Time Expressions

Expressions are classified according to their time of evaluation. An expression that may only be evaluated at runtime is called a runtime expression. An expression that is evaluated at compile time is called a compile time, or constant expression. Constant expression only have operands whose values are known at compile time and only invoke built-in functions or macros that may be evaluated at compile time.

7.2 Evaluation Order

The evaluation order of expressions is determined by operator precedence. There are five levels of operator precedence. However, sub-expressions enclosed in parentheses, designators and function calls always take precedence over the the default evaluation order that is defined by operator precedence. This effectively constitutes an implicit sixth level of evaluation.

Expression evaluation levels and their associated sub-expression EBNF definitions are given below from lowest to highest precedence. Levels one to five coincide with operator precedence levels.

7.2.1 Evaluation Level 1

EBNF:
```
expression :
    simpleExpression ( relOperator simpleExpression )? ;
```

7.2.2 Evaluation Level 2

EBNF:
```
simpleExpression :
    ( "+" | "-" )? term ( addOperator term )* ;
```

7.2.3 Evaluation Level 3

EBNF:
```
term :
    factorOrNegation ( mulOperator factorOrNegation )* ;
```

7.2.4 Evaluation Level 4

EBNF:
```
factorOrNegation :
    NOT? factor ;
```

7.2.5 Evaluation Level 5

EBNF:
```
factor :
    simpleFactor ( "::" typeIdent )? ;
```

7.2.6 Evaluation Level 6

EBNF:
```
simpleFactor :
    NumericLiteral | StringLiteral | structuredValue |
     designatorOrFunctionCall | "(" expression ")" ;
```


7.3 Operands

An operand may be a literal, a designator or a sub-expression. Whether any given operand may legally occur at a given position within an expression is determined by expression compatibility. Expression compatibility of operands is dependent on the operator of an expression or sub-expression as each operator defines what operand types it can accept.

7.3.1 Designators

Designators consist of an identifier that refers to a constant, a variable or a function call, followed by an optional tail that consists of one or more selectors. A designator's identifier may be qualified.

EBNF:
```
designator :
    qualident designatorTail? actualParameters? ;
```

7.3.2 Selectors

A selector may denote a component of an array, an element of a set, a value of a collection, a slice of a string, the pointer dereferencing suffix, a field of a record, or the actual parameter list of a function call.

EBNF:
```
designatorTail :
    ( ( "[" exprListOrSlice "]" | "^" ) ( "." ident )* )+ ;
exprListOrSlice :
    expression ( ( "," expression )+ | ".." expression )? ;
actualParameters :
    "(" expressionList? ")" ;
expressionList :
    expression ( "," expression )* ;
```

Examples:
```
array[index], cube[i, j, k], stringSlice[5..9], pointer^, record.field, write("a")
```

7.3.2.1 The Collection Value Selector

TO DO

7.3.2.2 The Character String Slice Selector

TO DO

7.3.2.3 The Pointer Dereference Selector

TO DO

7.3.2.4 The Record Field Selector

TO DO

7.3.2.5 The Function Call Selector

TO DO

7.4 Operators

Operators are special symbols or reserved words that represent an operation. An operator may be unary or binary. Unary operators are prefix, binary operators are infix. An operator may be either left-associative or non-associative and it has a precedence level between one and five, where five represents the highest operator precedence level. Arity, associativity and precedence determine the order of evaluation in expressions that consist of multiple sub-expressions and may contain different operators.

An overview of operators with their operations, arity, associativity and precedence is given below:

Operator	Represented Operations	Arity	Associativity	Precedence
`::`	Type Conversion	binary	left	5 (highest)
`NOT`	Logical Negation	unary	none	4
`*`	Multiplication, Set Intersection	binary	left	3
`/`	Division, Symmetric Set Difference	binary	left	3
`DIV`	Euclidean Integer Division	binary	left	3
`MOD`	Modulus of Euclidean Integer Division	binary	left	3
`AND`	Logical Conjunction	binary	left	3
`+`	Arithmetic Identity	unary	none	2
	Addition, Set Union, String Concatenation	binary	left	2
`-`	Sign Inversion	unary	none	2
	Subtraction, Set Difference	binary	left	2
`OR`	Logical Disjunction	binary	left	2
`=`	Equality Test	binary	none	1
`#`	Inequality Test	binary	none	1
`>`	Greater-Than Test, Proper Superset Test	binary	none	1
`>=`	Greater-Than-Or-Equal Test, Superset Test	binary	none	1
`<`	Less-Than Test, Proper Subset Test	binary	none	1
`<=`	Less-Than-Or-Equal Test, Subset Test	binary	none	1
`==`	Identity Test	binary	none	1
`IN`	Membership Test	binary	none	1

7.4.1 The Type Conversion Operator

Symbol `::` denotes the type conversion operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression :: typeIdent ;
```

The operator always represents the type conversion operation. Its left operand must be of convertible type. Its right operand indicates the target type and must be a type identifier. Its result type is the target type. Any use of the operator with operands that do not meet these conditions shall cause a compile time error. The type conversion operator is bindable.

7.4.2 The Asterisk Operator

Symbol `*` denotes a multi-purpose operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression “*” expression ;
```


The operator may represent different operations, depending on the type of its operands. If the operand type is a numeric type, it represents multiplication. If it is a set type, it represents set intersection. Its operands must be type compatible. Its result type is the operand type. Any use of the operator with incompatible operand types shall cause a compile time error. The asterisk operator is bindable.

7.4.3 The Slash Operator

Symbol / denotes a multi-purpose operator.. It is left-associative and requires two operands.

EBNF:
```
term :
    expression “/” expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is a real or complex number type, it represents division. If it is a set type, it represents symmetric set difference. Its operands must be type compatible. Its result type is the operand type. Any use of the operator with incompatible operand types shall cause a compile time error. The slash operator is bindable.

7.4.4 The DIV Operator

Reserved word `DIV` denotes the `DIV` operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression DIV expression ;
```

The operator always represents Euclidean integer division. Its operands must be of a whole number type and they must be type compatible. Its result type is the operand type. Any use of the operator with an operand type that is not a whole number type or with incompatible operand types shall cause a compile time error. The `DIV` operator is bindable.

7.4.5 The MOD Operator

Reserved word `MOD` denotes the `MOD` operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression MOD expression ;
```

The operator always represents the modulus of Euclidean integer division. Its operands must be of a whole number type and they must be type compatible. Its result type is the operand type. Any use of the operator with an operand type that is not a whole number type or with incompatible operand types shall cause a compile time error. The `MOD` operator is bindable.

7.4.6 The Plus Operator

Symbol + denotes a multi-purpose operator. There are two variants:

- unary plus
- binary plus

7.4.6.1 The Unary Plus Operator

The unary plus operator is non-associative and requires one operand.

EBNF:
```
term :
    “+” expression ;
```


The operator always represents the arithmetic identity operation. Its operand must be a numeric type. Its result type is the operand type. Any use of the operator with an operand type that is not numeric shall cause a compile time error. The unary plus operator is not bindable.

7.4.6.2 The Binary Plus Operator

The binary plus operator is left-associative and requires two operands.

EBNF:
```
term :
    expression “+” expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is a numeric type, it represents addition. If it is a set type, it represents set union. If it is a character string type it represents concatenation. Its operands must be type compatible. Its result type is the operand type. Any use of the operator with incompatible operand types shall cause a compile time error. The plus operator is bindable.

7.4.7 The Minus Operator

Symbol `-` denotes a multi-purpose operator. There are two variants:

- unary minus
- binary minus

7.4.7.1 The Unary Minus Operator

The unary minus operator is non-associative and requires one operand.

EBNF:
```
term :
    “-” expression ;
```

The operator represents alternative syntax for invocations of predefined procedure `NEG` which represents the sign inversion operation. Its operand must be a signed numeric type. Its result type is the operand type. Any use of the operator with an operand type that is not numeric shall cause a compile time error. The unary minus operator is not bindable.

7.4.7.2 The Binary Minus Operator

The binary minus operator is left-associative and requires two operands.

EBNF:
```
term :
    expression “-” expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is a numeric type, it represents subtraction. If it is a set type, it represents set difference. Its operands must be type compatible. Its result type is the operand type. Any use of the operator with incompatible operand types shall cause a compile time error. The minus operator is bindable.

7.4.8 The NOT Operator

Reserved word `NOT` denotes the logical `NOT` operator. It is non-associative and requires one operand.

EBNF:
```
term :
    NOT expression ;
```


The operator always represents the logical negation operation. Its single operand may be any expression of type `BOOLEAN` and its result type is `BOOLEAN`. Any use of the operator with an operand whose type is not `BOOLEAN` shall cause a compile time error. The `NOT` operator is not bindable.

7.4.9 The AND Operator

Reserved word `AND` denotes the logical `AND` operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression AND expression ;
```

The operator always represents the logical conjunction operation. Its operands must be of type `BOOLEAN` and its result type is `BOOLEAN`. Any use of the operator with incompatible operand types shall cause a compile time error. The `AND` operator is not bindable.

7.4.10 The OR Operator

Reserved word `OR` denotes the logical `OR` operator. It is left-associative and requires two operands.

EBNF:
```
term :
    expression OR expression ;
```

The operator always represents the logical disjunction operation. Its operands must be of type `BOOLEAN` and its result type is `BOOLEAN`. Any use of the operator with incompatible operand types shall cause a compile time error. The `OR` operator is not bindable.

7.4.11 The Equality Operator

Symbol = denotes the equality operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression "=" expression ;
```

The operator always represents the equality test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use of the operator with incompatible operand types shall cause a compile time error. The equality operator is bindable.

7.4.12 The Inequality Operator

Symbol # denotes the inequality operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression "#" expression ;
```

The operator always represents the inequality test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use of the operator with incompatible operand types shall cause a compile time error. The inequality operator is not bindable.

7.4.13 The > Operator

Symbol > denotes a dual-purpose relational operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression ">" expression ;
```


The operator may represent different operations, depending on the type of its operands. If the operand type is numeric, it represents the greater-than test operation. If it is a collection type, it represents the proper-superset test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use with incompatible operand types shall cause a compile time error. The > operator is bindable.

7.4.14 The >= Operator

Symbol >= denotes a dual-purpose relational operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression “>=” expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is numeric, it represents the greater-or-equal test operation. If it is a collection type, it represents the superset test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use with incompatible operand types shall cause a compile time error. The >= operator is not bindable.

7.4.15 The < Operator

Symbol < denotes a dual-purpose relational operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression “<“ expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is numeric, it represents the less-than test operation. If it is a collection type, it represents the proper-subset test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use with incompatible operand types shall cause a compile time error. The < operator is bindable.

7.4.16 The <= Operator

Symbol <= denotes a dual-purpose relational operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression “<=” expression ;
```

The operator may represent different operations, depending on the type of its operands. If the operand type is numeric, it represents the less-or-equal test operation. If it is a collection type, it represents the subset test operation. Its operands must be type compatible. Its result type is `BOOLEAN`. Any use with incompatible operand types shall cause a compile time error. The <= operator is not bindable.

7.4.17 The Identity Operator

Symbol == denotes the identity operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression “==” expression ;
```

The operator always represents the identity test operation. Its operands must be type compatible pointer types. Its result type is `BOOLEAN`. Any use of the operator with operands that do not meet these conditions shall cause a compile time error. The identity operator is not bindable.

7.4.18 The IN Operator

Reserved word `IN` denotes the `IN` operator. It is non-associative and requires two operands.

EBNF:
```
term :
    expression IN expression ;
```

The operator always represents the membership test operation. Its right operand must be of a collection type and its left operand must be of the component type of said collection type. Its result type is `BOOLEAN`. Any use of the operator with operands that do not meet these conditions shall cause a compile time error. The `IN` operator is bindable.

7.5 Operations

Operations are functions represented by an operator. They have one or two parameters, called operands and they return a result. The operands of an operation are the operands of the operator that represents it.

7.5.1 Conversion Operations

7.5.1.1 Type Conversion

The type conversion operation is represented by the double colon operator. It requires two operands. The left operand is called the input, the right operand is called the target type of the operation.

EBNF:
```
typeConversion :
    input "::" targetType ;
input : expression ;  targetType : typeIdent ;
```

The operation converts the value of its input into an equivalent or approximate value of its target type. The input may be an expression of any convertible type and the target type must be a type identifier. The result type is the target type. The type of the input must be convertible to the target type. Any attempt to convert an input to a target type to which it is not convertible shall cause a compile time error.

Examples:
```
VAR n : CARDINAL; i : INTEGER; r : REAL;
n :: INTEGER; i :: REAL; r :: CARDINAL;
```

7.5.2 Arithmetic Operations

Arithmetic operations perform arithmetic algebra with numeric operands and return a numeric result. For any operation with more than one operand, the operands must be type compatible. The operand type may be any numeric predefined type or any ADT that conforms to a numeric blueprint. The result type is always the operand type. An operation with mismatched operands shall cause a compile time error.

7.5.2.1 Arithmetic Identity

The arithmetic identity operation is represented by the + operator when it is not preceded by any operand but followed by a single operand. It returns the operand itself.

Examples:
```
+1 (* 1 *);  +10’000’000 (* 10’000’000 *);  +42.0 (* 42.0 *);  +intValue
```

7.5.2.2 Sign Inversion

Sign inversion is represented by the – operator when it is not preceded by any operand but followed by a single operand. The operation returns the sign reversed value of its operand.

Examples:
```
-1 (* -1 *);  -10’000’000 (* -10’000’000 *);  -42.0 (* -42.0 *);  -intValue
```


7.5.2.3 Addition

Addition is represented by the `+` operator when it occurs between two numeric operands. The operation adds the values of its operands and returns the sum.

Examples:
```
1 + 2 (* 3 *);  (-2) + 5 (* 3 *);  1.0 + 0.5 (* 1.5 *)
```

7.5.2.4 Subtraction

Subtraction is represented by the `-` operator when it occurs between two numeric operands. The operation subtracts the value of its right operand from that of its left operand and it returns the difference.

Examples:
```
3 - 2 (* 1 *);  (-2) - (-3) (* 1 *);  67.5 - 0.25 (* 67.25 *)
```

7.5.2.5 Multiplication

Multiplication is represented by the `*` operator when it occurs between two numeric operands. The operation multiplies the values of its operands and it returns the product.

Examples:
```
3 * 4 (* 12 *);  (-2) * (3) (* -6 *);  1.5 * 3.0 (* 4.5 *);  5 * 2.0 (* error *)
```

7.5.2.6 Real Number Division

Real number division is represented by the `/` operator when it occurs between two real number operands. The operation divides the value of its left operand by the value of its right operand and it returns the quotient. The divisor may not be zero. A constant divisor that is zero shall cause a compile time error. A zero divisor encountered at runtime shall raise runtime fault `DivByZero`.

Examples:
```
4.5 / 3.0 (* 1.5 *);
6 / 3,  5 / 2.0,  1.5 / 3,  r / 0.0  (* compile time errors *)
```

7.5.2.7 Integer Division

Integer division is represented by the `DIV` operator. It requires two operands. The operation divides the value of its left operand by the value of its right operand using the Euclidean definition of `DIV` and returns its quotient. The Euclidean definition of `DIV` (Bout, 1992) is given by the equation:

$$\mathrm{a}\ \mathtt{DIV}\ \mathrm{n} = \mathit{sgn}(\mathrm{n}) \times \mathit{floor}(\mathrm{a} \div \mathit{abs}(\mathrm{n}))$$

The divisor may not be zero. A constant divisor that is zero shall cause a compile time error. A zero divisor encountered at runtime shall raise runtime fault `DivByZero`.

Examples:
```
7 DIV 2 (* 3 *);  7 DIV -2 (* -3 *);  -7 DIV 3 (* -4 *);  -7 DIV -3 (* 4 *)
```

7.5.2.8 Modulus Operation

The modulus operation is represented by the `MOD` operator. It requires two operands. The operation divides the value of its left operand by the value of its right operand using the Euclidean definition of `MOD` and returns its remainder. The Euclidean definition of `MOD` (Bout, 1992) is given by the equation:

$$\mathrm{a}\ \mathtt{MOD}\ \mathrm{n} = \mathrm{a} - \mathit{sgn}(\mathrm{n}) \times \mathit{floor}(\mathrm{a} \div \mathit{abs}(\mathrm{n})) \times \mathrm{n}$$

The divisor may not be zero. A constant divisor that is zero shall cause a compile time error. A zero divisor encountered at runtime shall raise runtime fault `DivByZero`.

Examples:
```
7 MOD 2 (* 1 *);  7 MOD -2 (* 1 *);  -7 MOD 3 (* 2 *);  -7 MOD -3 (* 2 *)
```

7.5.3 Set Operations

Set operations perform set algebra with set operands and return a set as result.

7.5.3.1 Set Union

Set union is represented by the `+` operator when it occurs between two set operands. The operation returns the union of its operands. Operands must be type compatible. The result type is the operand type.

Examples:
```
VAR s1, s2 : BITSET;  s1 := { 1, 3, 5, 7 };  s2 := { 1, 2, 3 };
s1 * s2 (* returns { 1, 2, 3, 5, 7 } *);  3 * s1 (* compile time error *)
```

7.5.3.2 Set Difference

Set difference is represented by the `-` operator when it occurs between two set operands. The operation returns the set difference of its operands. Operands must be type compatible. The result type is the operand type.

Examples:
```
VAR s1, s2 : BITSET;  s1 := { 1, 3, 5, 7 };  s2 := { 1, 2, 3, 4 };
s1 * s2 (* returns { 2, 4 } *);  3 * s1 (* compile time error *)
```

7.5.3.3 Set Intersection

Set intersection is represented by the `*` operator when it occurs between two set operands. The operation returns the set intersection of its operands. Operands must be type compatible. The result type is the operand type.

Examples:
```
VAR s1, s2 : BITSET;  s1 := { 1, 3, 5, 7 };  s2 := { 1, 2, 3 };
s1 * s2 (* returns { 1, 3 } *);  3 * s1 (* compile time error *)
```

7.5.3.4 Symmetric Set Difference

Symmetric set difference is represented by the `/` operator when it occurs between two set operands. The operation returns the symmetric set difference of its operands. Operands must be type compatible. The result type is the operand type.

Examples:
```
VAR s1, s2 : BITSET;  s1 := { 1, 3, 5, 7 };  s2 := { 1, 2, 3 };
s1 / s2 (* returns { 2, 5, 7 } *);  s1 / 2.5 (* compile time error *)
```

7.5.4 Logical Operations

Logical operations perform boolean algebra with one or two `BOOLEAN` operands and return a `BOOLEAN` result. Modula-2 uses boolean short-circuit evaluation. Whenever the result of a binary logical operation is determined by the left operand alone, its right operand shall not be evaluated.

7.5.4.1 Logical Negation

Logical negation is represented by the `NOT` operator. The operation returns the logical negation of its operand.

Examples:
```
NOT TRUE (* FALSE *); NOT FALSE (* TRUE *); NOT (element IN set)
```


7.5.4.2 Logical Conjunction

Logical conjunction is represented by the `AND` operator. The operation returns the logical conjunction of its operands.

Examples:
```
TRUE AND FALSE (* FALSE *);  isOrdered AND isMutable
```

7.5.4.3 Logical Disjunction

Logical disjunction is represented by the `OR` operator. The operation returns the logical disjunction of its operands.

Examples:
```
TRUE OR FALSE (* TRUE *);  isOrdered OR isMutable
```

7.5.5 Character String Operations

Character string operations perform character string transformations with character string operands as input and return a character string as result.

7.5.5.1 Concatenation

Character string concatenation is represented by the + operator when it occurs between character string operands. The operation requires two operands. It appends the contents of its right operand to those of its left operand and returns the resulting string. The result type is the operand type.

Examples:
```
”red” + “fox” (* “redfox” *);  str + ASCII.LF;  str + “\n”
```

7.5.6 Relational Operations

Relational operations test the relationship between two operands and return a `BOOLEAN` result.

7.5.6.1 Equality Test

The equality test is represented by the = operator. The operation requires two operands. It compares the values of its operands. If the values are equal, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR i : INTEGER; n : CARDINAL; r : REAL;  i := -1;  n := 1;  r := 1.0;
ABS(i) = 1(* TRUE *);  n = 0 (* FALSE *);  r = 1.0 (* TRUE *)
i = n,  i = r,  n = r  (* compile time errors, incompatible types *)
```

7.5.6.2 Inequality Test

The inequality test is represented by the # operator. The operation requires two operands. It compares the values of its operands. If the values are unequal, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR i : INTEGER; n : CARDINAL; r : REAL;  i := -1;  n := 1;  r := 1.0;
ABS(i) # 1(* FALSE *);  n # 0 (* TRUE *);  r # 1.0 (* FALSE *)
i # n,  i # r,  n # r  (* compile time errors, incompatible types *)
```

7.5.6.3 Greater-Than Test

The greater-than test is represented by the > operator when it occurs between two numeric operands. The operation compares the values of its operands. If the value of its left operand is greater than that of its right operand, the result is `TRUE`, otherwise `FALSE`. If the operand type is unordered, a compile time error shall occur.

Examples:
```
0 > -1 (* TRUE *);  0.5 > 1.0 (* FALSE *);  set1 > set2;  fooTree > barTree
```

7.5.6.4 Greater-Than-Or-Equal Test

The greater-than-or-equal test is represented by the >= operator when it occurs between two numeric operands. The operation compares the values of its operands. If the value of its left operand is greater than or equal to that of its right operand, the result is `TRUE`, otherwise `FALSE`. If the operand type is unordered, a compile time error shall occur.

Examples:
```
0 >= -1 (* TRUE *);  1.0 >= 1.0 (* TRUE *);  set1 >= set2;  fooTree >= barTree
```

7.5.6.5 Less-Than Test

The less-than test is represented by the < operator when it occurs between two numeric operands. The operation compares the values of its operands. If the value of its left operand is less than that of its right operand, the result is `TRUE`, otherwise `FALSE`. If the operand type is unordered, a compile time error shall occur.

Examples:
```
0 < -1 (* FALSE *);  0.5 < 1.0 (* TRUE *);  set1 < set2;  fooTree < barTree
```

7.5.6.6 Less-Than-Or-Equal Test

The less-than-or-equal test is represented by the <= operator when it occurs between two numeric operands. The operation compares the values of its operands. If the value of its left operand is less than or equal to that of its right operand, the result is `TRUE`, otherwise `FALSE`. If the operand type is unordered, a compile time error shall occur.

Examples:
```
0 <= -1 (* FALSE *);  1.0 <= 1.0 (* TRUE *);  set1 <= set2;  fooTree <= barTree
```

7.5.6.7 Proper Superset Test

The proper-superset test is represented by the > operator when it occurs between two set operands. The operation tests the proper superset relationship of its operands. If the value of its left operand is a proper superset of its right operand, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR set1, set2: BITSET;  set1 := { 1, 2, 3 };  set2 := { 1, 2 };
set1 > set1 (* FALSE *);  set1 > set2 (* TRUE *)
```

7.5.6.8 Superset Test

The-superset test is represented by the >= operator when it occurs between two set operands. The operation tests the superset relationship of its operands. If the value of its left operand is a superset of its right operand, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR set1, set2: BITSET;  set1 := { 1, 2, 3 };  set2 := { 1, 2 };
set1 >= set1 (* TRUE *);  set1 >= set2 (* TRUE *)
```

7.5.6.9 Proper Subset Test

The proper-subset test is represented by the < operator when it occurs between two set operands. The operation tests the proper subset relationship of its operands. If the value of its left operand is a proper subset of its right operand, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR set1, set2: BITSET;  set1 := { 1, 2 };  set2 := { 1, 2, 3 };
set1 < set1 (* FALSE *);  set1 < set2 (* TRUE *)
```

7.5.6.10 Subset Test

The subset test is represented by the <= operator when it occurs between two set operands. The operation tests the subset relationship of its operands. If the value of its left operand is a subset of its right operand, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
VAR set1, set2: BITSET;  set1 := { 1, 2 };  set2 := { 1, 2, 3 };
set1 <= set1 (* TRUE *);  set1 <= set2 (* TRUE *)
```

7.5.6.11 Identity Test

The identity test is represented by the == operator. The operation tests the identity relationship of its operands. If both operands are references to the very same entity, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
TYPE IntPtr = POINTER TO INTEGER;  VAR foo, bar : INTEGER; baz : IntPtr;
baz := PTR(foo, IntPtr) (* baz points to foo *)
foo == bar (* returns FALSE, different variables *)
foo == baz^ (* returns TRUE, baz^ is an alias for foo *)
bar == baz (* compile time error: operands of incompatible types *)
```

7.5.6.12 Membership Test

The membership test is represented by the `IN` operator. The operation tests if there is a membership relation between its operands. If the value of its left operand is a member of a value set represented by its right operand, the result is `TRUE`, otherwise `FALSE`.

Examples:
```
FileMode.write IN mode;  “A” IN charSet;  “foo” IN tree
```

7.6 Structured Values

Structured values are compound values that consist of comma separated component values, enclosed in braces. A component value may be any literal or identifier denoting a value or structured value.

EBNF:
```
structuredValue :
    "{" ( valueComponent ( "," valueComponent )* )? "}" ;
valueComponent :
    expression ( ( BY | ".." ) constExpression )? ;
```

An expression in a structured value that is followed by the repetition clause `BY` or by the range constructor `..` must be a constant expression.

Examples:
```
{ 0 BY 100 }, { “a” .. “z” }, { 1 .. 31 }
{ "abc", 123, 456.78, { 1, 2, 3 } }, { 1970, Month.Jan, 1, 0, 0, 0.0, TZ.UTC }
```


8 Predefined Identifiers

Predefined identifiers are identifiers available in every module scope of a program without import. They are reserved identifiers and therefore may not be redefined. There are four groups:

- predefined constants
- predefined types
- predefined procedures
- predefined functions

8.1 Predefined Constants

There are three predefined constants:

`NIL`	invalid pointer value
`TRUE`	shorthand for `BOOLEAN.TRUE`
`FALSE`	shorthand for `BOOLEAN.FALSE`

8.2 Predefined Types

There are ten predefined types:

`BOOLEAN`	boolean type
`CHAR`	7-bit character type, subset of UTF-8
`UNICHAR`	4-octet character type, full UTF-32 set
`OCTET`	8-bit unsigned integer type of range `[0..255]`, smallest unit
`CARDINAL`	unsigned integer type of range $[0..2^b-1]$, where $b = 8n$, for $n \in \mathbb{N} \wedge n > 1$
`LONGCARD`	unsigned integer type of range $[0..2^b-1]$, where $b = 8m$, for $m \in \mathbb{N} \wedge m \geq n$
`INTEGER`	signed integer type of range $[-(2^{b-1})..2^{b-1}-1]$, where $b = 8n$, for $n \in \mathbb{N} \wedge n > 1$
`LONGINT`	signed integer type of range $[-(2^{b-1})..2^{b-1}-1]$, where $b = 8m$, for $m \in \mathbb{N} \wedge m \geq n$
`REAL`	real number type with implementation defined precision and range
`LONGREAL`	real number type with a precision and range equal to or higher than that of `REAL`

No predefined type may be an `ALIAS` type of another, even if the types share the same implementation.

8.2.1 Ranges of Numeric Predefined Types

An implementation shall use the same value of n for types `CARDINAL` and `INTEGER`, and it shall use the same value of m for types `LONGCARD` and `LONGINT`, where m and n are natural numbers, and m shall be larger than or equal to n. The value of n shall be equal to the value returned by predefined function `TSIZE` for types `CARDINAL` and `INTEGER`. The value of m shall be equal to the value returned by `TSIZE` for types `LONGCARD` and `LONGINT`.

In order to ensure upward conversions from whole number types to their corresponding real number types without the risk of type overflow, the following conditions shall be satisfied:

- `TMIN(REAL) ≤ TMIN(INTEGER):: REAL ∧ TMIN(LONGREAL) ≤ TMIN(LONGINT):: LONGREAL`
- `TMAX(REAL) ≥ TMAX(CARDINAL):: REAL ∧ TMAX(LONGREAL) ≥ TMAX(LONGCARD):: LONGREAL`

8.2.2 IO Support For Predefined Types

Although predefined types are built-in, their IO operations are not. The IO operations corresponding to `READ`, `WRITE` and `WRITEF` for predefined types are provided in the standard library and need to be imported to become available.

To this end, the standard library provides a runtime support library module for each predefined type whose module identifier matches the type identifier of the corresponding type. Library modules may therefore reuse predefined type identifiers as their module identifiers. Such reuse does not redefine the corresponding type.

8.3 Predefined Procedures

All predefined procedures are Wirthian macros. They act and look like library defined procedures but they may not be assigned to procedure variables, they may not be passed as parameters to any procedure and their invocations are replaced by the compiler with a predefined statement or statement sequence or a call to a corresponding library procedure. There are ten predefined procedures:

```
NEW RETAIN RELEASE COPY STORE REMOVE CONCAT READ WRITE WRITEF
```

8.3.1 Procedure NEW

Procedure `NEW` dynamically allocates storage for a variable of a pointer type. Its pseudo-definition is:

```
PROCEDURE NEW ( VAR p : <AnyPointerType>; (*OPTIONAL*) n : <UnsignedType> );
```

A call to procedure `NEW` is replaced by the compiler with a call to library procedure `ALLOCATE` which must be imported before `NEW` can be used. The standard library provides an `ALLOCATE` procedure in module `Storage`. `NEW` automatically invokes `RETAIN` if the type of `p` is a reference counted type.

Library procedure `ALLOCATE` always requires a second parameter to specify the allocation size of the type that the pointer variable points to. The compiler automatically determines the allocation size for the pointer variable passed to `NEW` and passes the appropriate size value as a second parameter to library procedure `ALLOCATE` when it replaces the procedure call.

Examples:
```
TYPE FooPtr = POINTER TO Foo;
VAR fooptr : FooPtr;
NEW(fooptr); (* replaced by ALLOCATE(fooptr, TSIZE(Foo)); *)
```

When `NEW` is used to allocate storage for a variable of indeterminate type a second parameter is required to pass the discriminant value for the type.

Examples:
```
TYPE VLA = RECORD items : CARDINAL; array : ARRAY items OF INTEGER END;
TYPE VLAPtr = POINTER TO VLA;
VAR v : VLAPtr;
NEW(v, 100); (* replaced by ALLOCATE(v, TSIZE(VLA) + 100*TSIZE(INTEGER)); *)
```

8.3.2 Procedure RETAIN

Procedure `RETAIN` retains a variable of a reference counted type. Its pseudo-definition is:

```
PROCEDURE RETAIN ( VAR p : <refCountedPointerType> );
```

An invocation of `RETAIN` is replaced by the compiler with a call to the procedure that has been bound to `RETAIN` for the type of the argument. An invocation of `RETAIN` with an argument whose type is not reference counted shall raise runtime fault `NotRefCounted`.

Examples:
```
VAR str : STRING;
NEW(str); ... RETAIN(str); (* replaced by STRING.retain(str); *)
```

8.3.3 Procedure RELEASE

Procedure `RELEASE` releases a variable of a pointer type. Its pseudo-definition is:

```
PROCEDURE RELEASE ( VAR p : <AnyPointerType> );
```

Releasing a variable whose type is **not** reference counted causes the variable to be deallocated immediately. Releasing a variable whose type **is** reference counted causes a prior invocation of `RETAIN` for the variable to be canceled. The variable is ultimately deallocated when all its prior invocations of `RETAIN` have been canceled.

An invocation of `RELEASE` with an argument whose type is **not** reference counted is replaced by the compiler with a call to library procedure `DEALLOCATE` which must be imported before `RELEASE` can be used in this way. The standard library provides a procedure `DEALLOCATE` in module Storage.

Example:
```
FROM Storage IMPORT DEALLOCATE;
RELEASE(nonRefCountedVar); (* replaced by DEALLOCATE(nonRefCountedVar); *)
```

An invocation of `RELEASE` with an argument whose type **is** reference counted is replaced by the compiler with a call to the library procedure that has been bound to `RELEASE` for the type of the argument. The implementor of the type is responsible for implementing these semantics within the procedure that is bound to `RELEASE`. The procedure must first cancel a single `RETAIN` and then check if any further are outstanding. If no more are then it must call library procedure `DEALLOCATE` to deallocate the argument.

Example:
```
IMPORT RefCountedType; VAR refCountedVar : RefCountedType; NEW(refCountedVar);
RELEASE(refCountedVar); (* replaced by RefCountedType.release(refCountedVar); *)
```

8.3.4 Procedure STORE

Procedure `STORE` stores a value in a component of a collection. Its pseudo-definition is:

```
PROCEDURE STORE ( c : <CollectionType>; s : <SelectorType>; v : <ValueType> );
```

TO DO: detailed description of operands and semantics.

8.3.5 Procedure INSERT

Procedure `INSERT` inserts data into a target variable, starting at a given index. It has two signatures:

```
PROCEDURE INSERT ( VAR t : <TargetType>; i : <IndexType>; data : <TargetType> );
```

```
PROCEDURE INSERT ( VAR t : <TargetType>; i : <IndexType>; data : ARGLIST OF <ValueType> );
```

`TargetType` may be `ARRAY OF CHAR`, `ARRAY OF UNICHAR` or any indexed collection ADT that provides bindings to `INSERT`. The insert data must be of the target type or a variadic list of values of its value type.

TO DO: detailed description of operands and semantics.

8.3.6 Procedure REMOVE

Procedure `REMOVE` removes a component from a collection. Its pseudo-definition is:

```
PROCEDURE REMOVE ( c : <CollectionType>; s : <SelectorType> );
```

TO DO: detailed description of operands and semantics.

8.3.7 Procedure CONCAT

Procedure `CONCAT` concatenates two or more character arrays or string variables given in its variadic source argument list and passed the concatenated result back in its target parameter. The arguments may be quoted literals or variables of any character array type or string ADT. The component type of all arguments must be the same, either `CHAR` or `UNICHAR`, but not mixed. Its pseudo-definition is:

```
PROCEDURE CONCAT ( VAR target : <StringType>; source : ARGLIST OF <StringType> );
```

8.3.8 Procedure READ

Procedure `READ` reads a value from a file or stream and assign it to a variable. Its pseudo-definition is:


```
PROCEDURE READ ( f : File; VAR v : <AnyType> );
```

A call to procedure `READ` is replaced by the compiler with a call to a library procedure `Read` which must be defined in a library module that has the same name as the type of the variable for which a value is being read.

The standard library provides a `Read` procedure for each predefined type in a corresponding module. The IO modules for all predefined types may be imported at once by importing aggregator module `IOSupport`.

In order to be able to call `READ` on library defined types, the library module that defines the type must have the same name as the type and it must provide its own `Read` procedure.

Examples:

```
IMPORT IOSupport;
VAR n : CARDINAL;
READ(n); (* replaced by CARDINAL.Read(stdIn, n); *)

IMPORT BCD;
VAR balance : BCD;
READ(balance); (* replaced by BCD.Read(stdIn, balance); *)
```

8.3.9 Procedure WRITE

Procedure `WRITE` is used to write a value to a file or stream. Its pseudo-definition is:

```
PROCEDURE WRITE ( f : File; v : <AnyType> );
```

A call to procedure `WRITE` is replaced by the compiler with a call to a library procedure `Write` which must be defined in a library module that has the same name as the type of the value being written.

The standard library provides a `Write` procedure for each predefined type in a corresponding module. The IO modules for all predefined types may be imported at once by importing aggregator module `IOSupport`.

In order to be able to call `WRITE` on library defined types, the library module that defines the type must have the same name as the type and it must provide its own `Write` procedure.

Examples:

```
IMPORT IOSupport;
VAR n : CARDINAL;
WRITE(n); (* replaced by CARDINAL.Write(stdOut, n); *)

IMPORT BCD;
VAR balance : BCD;
WRITE(balance); (* replaced by BCD.Write(stdOut, balance); *)
```

8.3.10 Procedure WRITEF

Procedure `WRITEF` is used to write one or more values to a file or stream using a given format depending on a format string. Its pseudo-definition is:

```
PROCEDURE WRITEF ( f : File; fmt : ARRAY OF CHAR; v : ARGLIST OF <AnyType> );
```

A call to procedure `WRITEF` is replaced by the compiler with a call to a library procedure `WriteF` which must be defined in a library module that has the same name as the type of the value or values being written.

The standard library provides a `WriteF` procedure for each predefined type in a corresponding module. The IO modules for all predefined types may be imported at once by importing aggregator module `IOSupport`.

In order to be able to call `WRITEF` on library defined types, the library module that defines the type must have the same name as the type and it must provide its own `WriteF` procedure.

Procedure `WRITEF` is variadic. It accepts one or more values to be written. However, all values must be of the same type. The format string strictly determines the formatting of values only. This is in contrast to the `printf` function of C where the format string also determines the types of values. In Modula-2 R10 all values must be of the same type to ensure type safety.

Examples:
```
IMPORT IOSupport;
VAR n1, n2, n3 : CARDINAL;
WRITEF("", n1, n2, n3); (* => CARDINAL.WriteF(stdOut, "", n1, n2, n3); *)

IMPORT BCD;
VAR balance : BCD;
WRITEF("", balance); (* => BCD.WriteF(stdOut, "", balance); *)
```

8.4 Predefined Functions

Predefined functions appear like library defined functions but they may not be assigned to procedure variables, may not be passed as parameters. Calls to them are typically replaced by the compiler with an expression rather than a call to a corresponding function. There are 17 predefined functions:

```
ABS NEG ODD PRED SUCC ORD CHR DUP COUNT LENGTH PTR RETRIEVE SUBSET
TMIN TMAX TSIZE TLIMIT
```

8.4.1 Function ABS

Function `ABS` returns the absolute value of its operand. Its operand may be of any numeric type. Its return type is the operand type. Its pseudo-definition is:

```
PROCEDURE ABS ( x : <NumericType> ) : <OperandType> ;
```

8.4.2 Function NEG

Function `NEG` returns the sign reversed value of its operand. Its operand maybe of any signed numeric type. Its return type is the operand type. Its pseudo-definition is:

```
PROCEDURE NEG ( x : <SignedNumericType> ) : <OperandType> ;
```

8.4.3 Function ODD

Function `ODD` returns `TRUE` if its operand is an odd number or `FALSE` if it is not. Its operand may be of any whole number type. Its return type is the boolean type. Its pseudo-definition is:

```
PROCEDURE ODD ( x : <WholeNumberType> ) : BOOLEAN ;
```

8.4.4 Function PRED

Function `PRED` returns the n-th predecessor of its first operand where n is the second operand. The type of its first operand may be any non-numeric ordinal type and that of its second operand may be `OCTET`, `CARDINAL` or `LONGCARD`. Its return type is the type of the first operand. Its pseudo-definition is:

```
PROCEDURE PRED ( x : <NonNumericOrdinalType>; n : <PredefCardType> ) : <typeOf(x)> ;
```


8.4.5 Function SUCC

Function `SUCC` returns the n-th successor of its first operand where n is the second operand. The type of its first operand may be any non-numeric ordinal type and that of its second operand may be `OCTET`, `CARDINAL` or `LONGCARD`. Its return type is the type of the first operand. Its pseudo-definition is:

```
PROCEDURE SUCC ( x : <NonNumericOrdinalType>; n : <PredefCardType> ) : <typeOf(x)> ;
```

8.4.6 Function ORD

Function `ORD` returns the ordinal value of its operand. The type of its operand may be any non-numeric ordinal type. If the operand is of type `UNICHAR` then its return type is `LONGCARD`, otherwise its return type is `CARDINAL`. Its pseudo-definition is:

```
PROCEDURE ORD ( x : <NonNumericOrdinalType> ) : <CardinalOrLongCardinalType> ;
```

8.4.7 Function CHR

Function `CHR` returns the character for the code point given by its operand. Its operand may be of any predefined whole number type. If the value of its operand is less than 128 then its return type is `CHAR`, otherwise its return type is `UNICHAR`. Its pseudo-definition is:

```
PROCEDURE CHR ( x : <UnsignedType> ) : <CharOrUnicharType> ;
```

8.4.8 Function DUP

Function `DUP` returns a newly allocated duplicate of its operand. The operand must be a dynamically allocated variable which may be of any type. Its return type is the operand type. Its pseudo-definition is:

```
PROCEDURE DUP ( CONST p : <AnyType> ) : <OperandType> ;
```

8.4.9 Function SUBSET

Function `SUBSET` returns `TRUE` if its first operand is a subset of its second operand, or `FALSE` if it is not. Its operand may be of any set type or any collection ADT that provides a binding to `SUBSET`. Its return type is the boolean type. Its pseudo-definition is:

```
PROCEDURE SUBSET ( a, b : <SetOrCollectionType> ) : BOOLEAN ;
```

8.4.10 Function COUNT

Function `COUNT` returns the number of items stored in its operand. Its operand may be a variable of any set type or collection type, or the identifier of a formal open parameter, formal variadic parameter or formal variadic parameter list. Its return type is `LONGCARD`. Its pseudo-definition is:

```
PROCEDURE COUNT ( c : <SetOrCollectionOrFormalOpenOrVariadicParam> ) : LONGCARD ;
```

8.4.11 Function LENGTH

Function `LENGTH` returns the number of characters stored in its operand. Its operand may be of any character string type. Its return type is `LONGCARD`. Its pseudo-definition is:

```
PROCEDURE LENGTH ( CONST s : <CharacterArrayOrStringADT> ) : LONGCARD ;
```

8.4.12 Function PTR

Function `PTR` returns a pointer to its first operand. Its return type is given by its second operand. Its first operand may be a variable of any type. Its second operand must be a pointer type whose target type is the type of the first operand. If the first operand is immutable within the scope where `PTR` is called, then the second operand must be a `POINTER TO CONST` type. Its pseudo-definition is:

```
PROCEDURE PTR ( variable : <AnyType>; T : <TypeIdentifier> ) : <T> ;
```

8.4.13 Function RETRIEVE

Function `RETRIEVE` retrieves and returns a component value from a collection. Its first operand denotes the collection from which to retrieve the value. Its second operand is a selector that identifies the component whose value to retrieve. Its pseudo-definition is:

```
PROCEDURE RETRIEVE ( c : <CollectionType>; s : <ComponentType> ) : <ReturnType>;
```

The collection given by the first operand may be an array or a set, or a collection ADT that binds a retrieval function to predefined function `RETRIEVE`. The selector given by the second operand and the return type depend on the first operand:

If the first operand is an array or array ADT, the selector denotes the index of the component to retrieve. If it is a set or set ADT, the selector denotes the element to retrieve. If it is an associative array or collection ADT, the selector denotes the key to identify the component to retrieve. If the first operand is a set or set ADT, the return type is `BOOLEAN`, otherwise it is the component type of the collection.

8.4.14 Function TMIN

Function `TMIN` returns the smallest legal value of its operand. Its operand is an identifier denoting any ordered type. Its return type is the operand. Its pseudo-definition is:

```
PROCEDURE TMIN ( T : <TypeIdentifier> ) : <T> ;
```

8.4.15 Function TMAX

Function `TMAX` returns the largest legal value of its operand. Its operand is an identifier denoting any ordered type. Its return type is the operand. Its pseudo-definition is:

```
PROCEDURE TMAX ( T : <TypeIdentifier> ) : <T> ;
```

8.4.16 Function TSIZE

Function `TSIZE` returns the required allocation size of a type. The value returned represents the number of octets required to allocate a variable of the type denoted by its operand. Its operand is an identifier denoting a type. Its return type is `LONGCARD`. Its pseudo-definition is:

```
PROCEDURE TSIZE ( T : <TypeIdentifier> ) : LONGCARD ;
```

8.4.17 Function TLIMIT

Function `TLIMIT` returns the capacity limit of a set, array, string or collection type. The identifier of the type whose capacity limit is being requested is passed as its operand. Its return type is `LONGCARD`. The return value represents the maximum number of components that a variable of the type can hold. A return value of zero indicates that the type does not have a fixed capacity limit. Its pseudo-definition is:

```
PROCEDURE TLIMIT ( T : <TypeIdentifier> ) : LONGCARD ;
```


9 Scalar Conversion

Scalar conversion is the process of converting a value of a scalar numeric type into the equivalent value of another scalar numeric type or a close approximation thereof.

In an any-to-any type conversion system the number of conversion paths grows exponentially with the number of convertible types. This makes it impractical to provide a conversion procedure for each conversion path. Instead, a two-stage conversion via an intermediate representation for scalar values may be used to reduce the number of required conversion procedures to two for each convertible type.

9.1 Scalar Exchange Format

Scalar Exchange Format (SXF) is a language defined intermediate representation for scalar values to facilitate scalar conversion. Any two scalar numeric types `T1` and `T2` are convertible to each other if both `T1` and `T2` provide conversions to and from scalar exchange format:

- The intermediate conversion path from `T1` to `T2` is: `T1` → `SXF` → `T2`.
- The intermediate conversion path from `T2` to `T1` is: `T2` → `SXF` → `T1`.

The current version of the SXF protocol is 1.00 and its syntax is given below:

EBNF:

```
serialisedScalarFormat :
    version length sigDigitCount expDigitCount digitRadix
    sigSign sigDigits ( expSign expDigits )? terminator ;
version:
    digitB64 digitB64; (* protocol version, valid range 100 to 4095 *)
length :
    digitB64 digitB64; (* allocated length, valid range 16 to 4095 *)
sigDigitCount :
    digitB64 digitB64; (* digit count of significand, valid range 1 to 4000 *)
expDigitCount :
    digitB64; (* digit count of optional exponent, valid range 0 to 63 *)
digitRadix :
    ":" | "@" ; (* digit radix, ":" for base 10, "@" for base 16 *)
sigSign :
    "+" | "-" ; (* sign of the significand *)
sigDigits :
    digitB10+ | digitB16+ ; (* digits of the significand *)
expSign :
    "+" | "-" ; (* sign of the exponent, if digit count > 0 *)
expDigits :
    digitB10+ | digitB16+ ; (* digits of the exponent, if digit count > 0 *)
digitB10 :
    "0" .. "9" ; (* representing values between 0 and 9 *)
digitB16 :
    "0" .. "?" ; (* representing values between 0 and 15 *)
digitB64 :
    "0" .. "o" ; (* representing values between 0 and 63 *)
terminator :
    ASCII(0) ;
```

The protocol supports variable length base-10 and base-16 representation of integer and real number values with up to 4000 digits in the significand and for real numbers up to 63 digits in the exponent. For maximum efficiency, digits are encoded without gaps in the encoding table and the radix of the source value is preserved when converting to scalar exchange format, thereby avoiding the need for radix conversion where source and target type use the same radix.

9.2 Pseudo-Module CONVERSION

Module `CONVERSION` provides an interface to low-level conversion facilities used internally by the compiler to convert scalar numeric types to and from scalar exchange format. Facilities are listed below:

9.2.1 Constant SXFVersion

Constant `SXFVersion` indicates the compound version of the SXF protocol used by an implementation. The two least significant digits represent the minor version, remaining digits the major version.

`SXFVersion` whole number value in the range of `100 .. 4095`

9.2.2 Macro SXFSizeForType

An invocation of macro `SXFSizeOfType` is replaced by the allocation size in octets required to represent arbitrary values of the given numeric type in SXF. Its replacement value is a compile time value.

Pseudo Definition:
```
PROCEDURE SXFSizeForType ( <ScalarType> ) : CARDINAL;
```

9.2.3 Macro TSIGNED

An invocation of macro `TSIGNED` is replaced by the signed property of the scalar type given by its argument.

Pseudo Definition:
```
PROCEDURE TSIGNED ( <NumericScalarType> ) : BOOLEAN;
```

9.2.4 Macro TBASE

An invocation of macro `TBASE` is replaced by the radix of the scalar type given by its argument.

Pseudo Definition:
```
PROCEDURE TBASE ( <NumericScalarType> ) : CARDINAL;
```

9.2.5 Macro TPRECISION

An invocation of macro `TPRECISION` is replaced by the significand capacity of the scalar type given by its argument. The resulting compile time value indicates the number of digits for the type's default radix.

Pseudo Definition:
```
PROCEDURE TPRECISION ( <NumericScalarType> ) : CARDINAL;
```

9.2.6 Macro TMINEXP

An invocation of macro `TMINEXP` is replaced by the value of the smallest exponent the scalar type given by its argument can encode.

Pseudo Definition:
```
PROCEDURE TMINEXP ( <NumericScalarType> ) : INTEGER;
```

9.2.7 Macro TMAXEXP

An invocation of macro `TMAXEXP` is replaced by the value of the largest exponent the scalar type given by its argument can encode.

Pseudo Definition:
```
PROCEDURE TMAXEXP ( <NumericScalarType> ) : CARDINAL;
```

9.2.8 Primitive SXF

The `SXF` primitive converts a value of a scalar numeric type to scalar exchange format and passes the result back in an octet array.

Pseudo Definition:
```
PROCEDURE SXF ( value : <ScalarType>; VAR sxfValue : ARRAY OF OCTET );
```

If the capacity of the octet array passed-in is too small to hold the scalar exchange format representation of the value to be converted, a runtime fault of type `TypeOverflow` shall be raised.

9.2.9 Primitive VAL

The `VAL` primitive converts a value in scalar exchange format to an equivalent or closely approximate value of a given scalar numeric type.

Pseudo Definition:
```
PROCEDURE VAL ( sxfValue : ARRAY OF OCTET; VAR value : <ScalarType> );
```

If the value represented in scalar exchange format is smaller than `TMIN` or larger than `TMAX` of the target type, a runtime fault of type `TypeOverflow` shall be raised.

10 Interfaces to Compiler and Runtime System

Modula-2 R10 provides facilities to interface and communicate with an implementation's compile-time and run-time systems. These facilities are available from two corresponding pseudo-modules:

- pseudo-module `COMPILER`
- pseudo-module `RUNTIME`

10.1 Pseudo-Module COMPILER

Pseudo-module `COMPILER` provides information about the compiler itself, compile time facilities and interfaces to intrinsics internal to the compiler. All facilities are compile-time expressions.

10.1.1 Identity Of The Compiler

Module `COMPILER` provides a set of constants pertaining to the identity of the compiler:

`Name`	string containing the short name of the compiler
`FullName`	string containing the full name of the compiler
`EditionName`	string containing the name of the compiler edition
`MajorVersion`	whole number denoting the major version of the compiler
`MinorVersion`	whole number denoting the minor version of the compiler
`SubMinorVersion`	whole number denoting the sub-minor version of the compiler
`ReleaseYear`	whole number denoting the year of release of the compiler version
`ReleaseMonth`	whole number denoting the month of release of the compiler version
`ReleaseDay`	whole number denoting the day of release of the compiler version

10.1.2 Testing The Availability Of Optional Capabilities

Module `COMPILER` provides a set of boolean macros to test the availability of optional capabilities. An availability macro expands to `TRUE` if the corresponding facility is supported, otherwise to `FALSE`.

Boolean Macro	Indicating Availability Of	Capability Available From
`SupportsUTF8EncodedSource`	support for UTF8 encoded source	language core
`SupportsEncodingVerification`	support for encoding verification	language core
`SupportsAlignmentControl`	pragma `ALIGN`	language core
`SupportsBitPadding`	pragma `PADBITS`	language core
`SupportsAttrNoReturn`	pragma `NORETURN`	language core
`SupportsAttrPurity`	pragma `PURITY`	language core
`SupportsAttrSingleAssign`	pragma `SINGLEASSIGN`	language core
`SupportsAttrLowLatency`	pragma `LOWLATENCY`	language core
`SupportsAttrVolatile`	pragma `VOLATILE`	language core
`SupportsAddressMapping`	pragma `ADDR`	pseudo-module `UNSAFE`
`SupportsCFFI`	foreign function interface to C	pseudo-module `UNSAFE`
`SupportsFortranFFI`	foreign function interface to Fortran	pseudo-module `UNSAFE`
`SupportsInlineCode`	raw machine code inline facility	pseudo-module `ASSEMBLER`
`SupportsInlineAssembly`	symbolic assembly inline facility	pseudo-module `ASSEMBLER`
`SupportsRegisterAccess`	intrinsics `SETREG` and `GETREG`	pseudo-module `ASSEMBLER`
`SupportsRegisterMapping`	formal parameter attribute `REG`	pseudo-module `ASSEMBLER`

Values may be target dependent. An implementation may not add any availability macros to pseudo-module `COMPILER`. To allow testing of implementation defined facilities, an implementation shall provide boolean availability constants in library module `ImplDef` instead. Identifiers shall be prefixed with `Supports`.

10.1.3 Information About The Compiling Source

Module `COMPILER` provides a set of lexical macros to insert information about the compiling source into the compiled library or program in order to support user defined warnings and error messages:

`MODNAME`	expands to a string constant with the name of the module being compiled
`PROCNAME`	expands to a string constant with the name of the procedure being compiled
`LINENUM`	expands to a whole number constant representing the line number being compiled

10.1.4 Implementation Models of REAL and LONGREAL

Module `COMPILER` provides an enumeration type and two constants to provide information about the implementation defined implementation model of predefined types `REAL` and `LONGREAL`:

```
TYPE IEEE754Support = ( None, Binary16, Binary32, Binary64, Binary128 );
```

`ImplModelOfTypeReal`	implementation defined constant of type `IEEE754Support`
`ImplModelOfTypeLongReal`	implementation defined constant of type `IEEE754Support`

10.1.5 Build Settings Interface

Module `COMPILER` provides a means to obtain *extra-source* build settings at compile time.

10.1.5.1 Macro DEFAULT

Macro `DEFAULT` permits retrieval of values from the compiler's settings and user preferences database at compile time. Keys are ASCII strings, values are ordinal. If a value is stored in the database for the key given by its first operand, the macro expands to the stored value, otherwise it expands to the fallback value given by its second operand. Facilities to store values in the database are implementation defined.

Pseudo Definition:

```
PROCEDURE DEFAULT
  ( key : ARRAY OF CHAR; fallback : <PredefOrdinalType> ) : <PredefOrdinalType>;
```

Example:

```
CONST maxFoobar = DEFAULT("MaxFoobar", 42); (* retrieve value, if undefined use 42 *)
```

10.1.6 Type Checking Interface

Module `COMPILER` provides a set of lexical macros to determine compatibility and convertibility of constants, variables and types, and prototype conformance of types.

10.1.6.1 Macro IsCompatibleWithType

An invocation of macro `IsCompatibleWithType` is replaced by boolean value `TRUE` if the type of the identifier passed as its first argument is compatible with the type passed as its second argument. Otherwise it is replaced by boolean value `FALSE`. Its first argument is an identifier of a constant, a variable or a type. Its second argument is an identifier of a type. An invocation of this macro results in a compile time expression.

Pseudo Definition:

```
PROCEDURE IsCompatibleWithType ( <qualident>; <AnyType> ) : BOOLEAN;
```

10.1.6.2 Macro IsConvertibleToType

An invocation of macro `IsConvertibleToType` is replaced by boolean value `TRUE` if the type of the identifier passed as its first argument is convertible to the type passed as its second argument. Otherwise it is replaced by boolean value `FALSE`. Its first argument is an identifier of a constant, a variable or a type. Its second argument is a type identifier. An invocation of this macro results in a compile time expression.

Pseudo Definition:

```
PROCEDURE IsConvertibleToType ( <qualident>; <AnyType>) : BOOLEAN;
```


10.1.6.3 Macro IsExtensionOfType

An invocation of macro `IsExtensionOfType` is replaced by boolean value `TRUE` if the type passed as its first argument is a type extension of the type passed as its second argument. If not, it is replaced by `FALSE`. Both arguments are type identifiers. An invocation of this macro results in a compile time expression.

Pseudo Definition:
```
PROCEDURE IsExtensionOfType ( <qualident>; <AnyType> ) : BOOLEAN;
```

10.1.6.4 Macro IsMutableType

An invocation of macro `IsMutableType` is replaced by boolean value `TRUE` if the identifier passed as its argument is a mutable type, otherwise `FALSE`. A mutable type is a type whose variables are mutable. By default all types are mutable. An ADT may declare itself immutable by binding a constant of value `FALSE` to the assignment symbol. An invocation of this macro results in a compile time expression.

Pseudo Definition:
```
PROCEDURE IsMutableType ( <qualident> ) : BOOLEAN;
```

10.1.6.5 Macro IsOrderedType

An invocation of macro `IsOrderedType` is replaced by boolean value `TRUE` if the identifier passed as its argument is is an ordered type, otherwise `FALSE`. An ordered type is a type with discrete values of a well defined order. By default only built-in ordinal types, enumeration types, `ARRAY` and `SET` types are ordered types. A numeric ADT is ordered if it is scalar. A collection ADT is ordered if it is defined ordered. An invocation of macro `IsOrderedType` results in a compile time expression.

Pseudo Definition:
```
PROCEDURE IsOrderedType ( <qualident> ) : BOOLEAN;
```

10.1.6.6 Macro IsRefCountedType

An invocation of macro `IsRefCountedType` is replaced by boolean value `TRUE` if the identifier passed as its argument is an ADT that provides bindings to both predefined procedures `RETAIN` and `RELEASE`. Otherwise it is replaced by `FALSE`. An invocation of this macro results in a compile time expression.

Pseudo Definition:
```
PROCEDURE IsRefCountedType ( <qualident> ) : BOOLEAN;
```

10.1.6.7 Macro ConformsToBlueprint

An invocation of macro `ConformsToBlueprint` is replaced by boolean value `TRUE` if the identifier passed as its first argument conforms to the blueprint passed as its second argument. Otherwise it is replaced by `FALSE`. Its first argument is an unqualified type identifier of an abstract data type. Its second argument is a blueprint identifier. An invocation of this macro results in a compile time expression.

Pseudo Definition:
```
PROCEDURE ConformsToBlueprint ( <qualident>; <Blueprint> ) : BOOLEAN;
```

10.1.7 Compile-Time Arithmetic

Module `COMPILER` provides a number of lexical macros to assist in the compile time calculation of range and size requirements for scalars and data structures.

10.1.7.1 Macro MIN

An invocation of macro `MIN` is replaced by the smallest constant from its variadic argument list.

Pseudo Definition:
```
PROCEDURE MIN ( constant : ARGLIST OF <EnumOrScalarType> ) : <OperandType>;
```


Examples:
```
FROM COMPILER IMPORT MIN;
MIN( 1, 2, 3 ) => 1
MIN( 1.2, 3.4, 5.6 ) => 1.2
TYPE Fruit = ( Apple, Cherry, Mango, Orange, Strawberry );
MIN( Fruit.Orange, Fruit.Cherry, Fruit.Mango ) => Fruit.Cherry
MIN ( 1, 3.4, Fruit.Mango ) => compile time error: incompatible arguments
```

10.1.7.2 Macro MAX

An invocation of macro `MAX` is replaced by the largest constant from its variadic argument list.

Pseudo Definition:
```
PROCEDURE MAX ( constant : ARGLIST OF <EnumOrScalarType> ) : <OperandType>;
```

Examples:
```
FROM COMPILER IMPORT MAX;
MAX( 1, 2, 3 ) => 3
MAX( 1.2, 3.4, 5.6 ) => 5.6
TYPE Fruit = ( Apple, Cherry, Mango, Orange, Strawberry );
MAX( Fruit.Orange, Fruit.Cherry, Fruit.Mango ) => Fruit.Orange
MAX ( 1, 3.4, Fruit.Mango ) => compile time error: incompatible arguments
```

Macros `MIN` and `MAX` only accept argument lists of constants or constant expressions of an enumerated type or of a numeric predefined type.

A compile time error occurs if any one of the following conditions is met:

- the arguments are not of the same type
- any argument is not a constant or constant expression [1]
- any argument is not of an enumerated type or a numeric predefined type

10.1.7.3 Macro MaxOrd

An invocation of macro `MaxOrd` is replaced by the largest ordinal number that can be represented with the number of bits given by its constant whole number argument.

Pseudo Definition:
```
PROCEDURE MaxOrd( numOfBits : <PredefinedWholeNumberType> ) : <OperandType>;
```

Example:
```
FROM COMPILER IMPORT MaxOrd;
MaxOrd(15) => 32768
```

10.1.7.4 Macro ReqBits

An invocation of macro `ReqBits` is replaced by the minimum number of bits required to represent the ordinal number given by its constant whole number argument.

Pseudo Definition:
```
PROCEDURE ReqBits ( ord : <PredefinedWholeNumberType> ) : <OperandType>;
```

Example:
```
FROM COMPILER IMPORT ReqBits;
ReqBits(32000) => 15
```

10.1.7.5 Macro ReqOctets

An invocation of macro `ReqOctets` is replaced by the minimum number of octets required to represent the ordinal number given by its constant whole number argument.

---

[1] It should be noted that library defined conversions may not be used within constant expressions.

Pseudo Definition:

```
PROCEDURE ReqOctets ( ord : <PredefinedWholeNumberType> ) : <OperandType>;
```

Example:

```
FROM COMPILER IMPORT ReqOctets;
ReqOctets(32000) => 2
```

Macros `MaxOrd` , `ReqBits` and `ReqOctets` only accept constant whole number arguments within the range of type `LONGCARD`. A compile time error shall occur if any one of the following conditions is met:

- the argument value is not a whole number
- the argument is not a constant or constant expression
- the argument value exceeds the range of type `LONGCARD`
- the resulting replacement value exceeds the range of type `LONGCARD`

10.1.8 Miscellaneous

10.1.8.1 Macro HASH

An invocation of macro `HASH` is replaced by the hash value of the string literal or constant given by its argument. The hash value is calculated using the compiler's built-in hash function. The enclosing quotation marks of the argument do not have any influence on the calculated hash value.

```
PROCEDURE HASH ( CONST s : ARRAY OF CHAR ) : LONGCARD;
```

The underlying hash algorithm is implementation dependent. It may also differ between different releases of the same implementation. Calculated hash *values* are therefore non-portable between implementations and releases.

10.1.8.2 Macro TODO

Macro `TODO` represents an empty statement. It may be used to indicate unimplemented sections of code.

```
PROCEDURE TODO;
```

When compiling in debug mode, any occurrence of macro `TODO` shall cause a hard compile-time warning. When executing an image built in debug mode, any invocation of macro `TODO` shall raise a runtime warning. When compiling in production mode, any occurrence shall cause a compile-time error.

10.2 Pseudo-Module RUNTIME

Pseudo-module `RUNTIME` provides an interface to the runtime system.

10.2.1 Runtime Faults

Runtime system faults are defined as an enumerated type `RTFault`. Its definition is:

```
TYPE RTFault = ( DivByZero, DerefNil, NotRefCounted, InvalidAccessor,
                 TypeOverflow, IndexOutOfBounds, StringCapacityExceeded,
                 StackOverflow, OutOfMemory, LibAbort, UserAbort );
```

10.2.1.1 Raising Runtime Faults

Module `RUNTIME` provides procedure `RaiseRTFault` to raise a runtime fault of type `f`. Raising a runtime fault will cause program abort. The procedure never returns. Its definition is:

```
PROCEDURE RaiseRTFault ( f : RTFault );
```

10.2.2 Runtime Event Handling

Module `RUNTIME` provides facilities to install user-defined runtime system event handlers.

10.2.2.1 Runtime Event Notifications

Module `RUNTIME` provides procedure type `NotificationHandler` for user-defined notification handlers. A handler may not invoke procedure `RaiseRTFault` and it must accept two parameters:

- the address of the program counter (PC) at which the offending event occurred.
- the address of the stack pointer (SP) at which the offending event occurred.

The type definition is:

```
TYPE NotificationHandler = PROCEDURE (UNSAFE.ADDRESS, UNSAFE.ADDRESS );
```

10.2.2.1.1 Installing a Notification Handler

Procedure `InstallNotificationHandler` installs a user-defined notification handler `p` for runtime faults of type `f`. Its definition is:

```
PROCEDURE InstallNotificationHandler ( f : RTFault; p : NotificationHandler );
```

10.2.2.2 Runtime Event Handlers

Module `RUNTIME` provides facilities to install user-defined event handlers for post-initialisation, pre-termination and termination events.

10.2.2.2.1 Procedure InstallInitHasFinishedHandler

Procedure `InstallInitHasFinishedHandler` installs user-defined event handler `p` as the program's post-initialisation handler. The installed handler is called immediately after module initialisation has finished. Its definition is:

```
PROCEDURE InstallInitHasFinishedHandler ( p : PROCEDURE );
```

10.2.2.2.2 Procedure InstallWillTerminateHandler

Procedure `InstallWillTerminateHandler` installs user-defined event handler `p` as the program's pre-termination handler. The installed handler is called immediately before the program's termination handler. Its definition is:

```
PROCEDURE InstallWillTerminateHandler ( p : PROCEDURE );
```

10.2.2.2.2 Procedure InstallTerminationHandler

Procedure `InstallTerminationHandler` installs user-defined event handler `p` as the program's termination handler. The installed handler is called immediately before the program terminates. Its definition is:

```
PROCEDURE InstallTerminationHandler ( p : PROCEDURE );
```

10.2.3 Runtime System Facilities

Module `RUNTIME` may provide stack trace, post mortem and system reset facilities. Runtime system facilities are defined as an enumerated type `Facility`. Its definition is:

```
TYPE Facility = ( StackTrace, PostMortem, SystemReset );
```

10.2.3.1 Testing The Availability Of Runtime System Facilities

Function `IsAvail` returns the availability of the runtime system facility given by its operand. It returns `TRUE` if the facility is available, otherwise it returns `FALSE`. Its definition is:

```
PROCEDURE IsAvail ( f : Facility; ) : BOOLEAN;
```

10.2.3.2 StackTrace Facility

Module `RUNTIME` may provide facilities to enable, disable and initiate a stack trace.

10.2.3.2.1 Procedure InitiateStackTrace

Procedure `InitiateStackTrace` aborts the currently running program and writes a stack trace dump if the stack trace facility is available and enabled. Its definition is:

```
PROCEDURE InitiateStackTrace;
```

10.2.3.2.2 Procedure SetStackTrace

Procedure `SetStackTrace` sets the current stack trace mode. If `TRUE` is passed and the stack trace facility is available, the stack trace mode is enabled, otherwise it is disabled. Its definition is:

```
PROCEDURE SetStackTrace( enabled : BOOLEAN );
```

10.2.3.2.3 Function StackTraceEnabled

Function `StackTraceEnabled` returns the current stack trace mode. It returns `TRUE` if the stack trace facility is available and enabled, otherwise it returns `FALSE`. Its definition is:

```
PROCEDURE StackTraceEnabled : BOOLEAN;
```

10.2.3.3 PostMortem Facility

Module `RUNTIME` may provide facilities to enable, disable and initiate a post mortem dump.

10.2.3.3.1 Procedure InitiatePostMortem

Procedure `InitiatePostMortem` aborts the currently running program and writes a post mortem dump if the post mortem facility is available and enabled. Its definition is:

```
PROCEDURE InitiatePostMortem;
```

10.2.3.3.2 Procedure SetPostMortem

Procedure `SetPostMortem` sets the current post mortem mode. If `TRUE` is passed and the post mortem facility is available, the post mortem mode is enabled, otherwise it is disabled. Its definition is:

```
PROCEDURE SetPostMortem( enabled : BOOLEAN );
```


10.2.3.3.3 Function PostMortemEnabled

Function `PostMortemEnabled` returns the current post mortem mode. It returns `TRUE` if the post mortem facility is available and enabled, otherwise it returns `FALSE`. Its definition is:

```
PROCEDURE PostMortemEnabled : BOOLEAN;
```

10.2.3.4 SystemReset Facility

Module `RUNTIME` may provide facilities to enable, disable and initiate a system reset, targeting embedded and self hosting platforms.

10.2.3.4.1 Procedure InitiateSystemReset

Procedure `InitiateSystemReset` aborts the currently running system image and restarts it if the system reset facility is available and enabled. Its definition is:

```
PROCEDURE InitiateSystemReset;
```

10.2.3.4.2 Procedure SetSystemReset

Procedure `SetSystemReset` sets the current system reset mode. If `TRUE` is passed and the system reset facility is available, the reset mode is enabled, otherwise it is disabled. Its definition is:

```
PROCEDURE SetSystemReset( enabled : BOOLEAN );
```

10.2.3.4.3 Function SystemResetEnabled

Function `SystemResetEnabled` returns the current system reset mode. It returns `TRUE` if the system reset facility is available and enabled, otherwise it returns `FALSE`. Its definition is:

```
PROCEDURE SystemResetEnabled : BOOLEAN;
```


11 Low-Level Facilities

Modula-2 R10 provides a rich set of built-in low level programming facilities. However, low-level types and intrinsics have semantics that relax the strict typing rules of the language. Their use is therefore potentially unsafe and they are by default hidden in low-level pseudo-modules from where they must be imported before they can be used. There are five low-level pseudo-modules:

- pseudo-module `UNSAFE`
- pseudo-module `ATOMIC`
- pseudo-module `ACTOR` (Phase II)
- pseudo-module `COROUTINE` (Phase II)
- pseudo-module `ASSEMBLER` (optional)

11.1 Pseudo-Module UNSAFE

Pseudo-module `UNSAFE` provides implementation- and target-dependent facilities.

11.1.1 UNSAFE Constants

Constant	Definition	Type
`OctetsPerByte`	implementation defined non-zero size of a byte in octets	`OCTET`
`BytesPerWord`	implementation defined non-zero size of a word in bytes	`OCTET`
`BitsPerMachineByte`	target dependent non-zero size of a machine byte in bits	`OCTET`
`MachineBytesPerMachineWord`	target dependent non-zero size of a machine word in bytes	`OCTET`
`TargetName`	string with the name of the target architecture, length < 16	`ARRAY OF CHAR`
`TargetByteOrder`	value indicating the byte order of the target architecture	`ENDIANNESS`
`BigEndian`	enumerated value indicating 3-2-1-0 byte order	`ENDIANNESS`
`LittleEndian`	enumerated value indicating 0-1-2-3 byte order	`ENDIANNESS`
`BigLittleEndian`	enumerated value indicating 2-3-0-1 byte order	`ENDIANNESS`
`LittleBigEndian`	enumerated value indicating 2-1-0-3 byte order	`ENDIANNESS`
`TargetIsBiEndian`	`TRUE` if target supports bi-endianness, otherwise `FALSE`	`BOOLEAN`
`ByteBoundaryAddressing`	`TRUE` if every machine byte is addressable, otherwise `FALSE`	`BOOLEAN`
`MaxWordsPerOperand`	largest operand size of `UNSAFE` low-level intrinsics, value ≥ 4	`OCTET`

11.1.2 UNSAFE Types

Module `UNSAFE` provides the following types:

`BYTE`	implementation defined byte
`WORD`	implementation defined word
`MACHINEBYTE`	target dependent machine byte
`MACHINEWORD`	target dependent machine word
`ADDRESS`	target dependent machine address

Although these types are provided by module `UNSAFE`, their respective IO operations are not. The IO operations corresponding to `READ`, `WRITE` and `WRITEF` for `UNSAFE` types are provided by the standard library and need to be imported to become available.

11.1.2.1 Type ENDIANNESS

Type `ENDIANNESS` enumerates endianness values. Its pseudo-definition is:

```
TYPE ENDIANNESS = ( LittleEndian, BigEndian, LittleBigEndian, BigLittleEndian );
```


11.1.2.1 Type BYTE

`BYTE` is a system type whose size is implementation defined. Its pseudo-definition is:

```
TYPE BYTE = OPAQUE RECORD value : ARRAY OctetsPerByte OF OCTET END;
```

11.1.2.2 Type WORD

`WORD` is a system type whose size is implementation defined. Its pseudo-definition is:

```
TYPE WORD = OPAQUE RECORD value : ARRAY BytesPerWord OF UNSAFE.BYTE END;
```

11.1.2.3 Type MACHINEBYTE

`MACHINEBYTE` is a system type whose bit width is `BitsPerMachineByte` which is a target dependent value representing the actual bit width of a machine byte of the target architecture.

11.1.2.4 Type MACHINEWORD

`MACHINEWORD` represents the actual word size of a machine word of the target architecture.

```
TYPE MACHINEWORD = OPAQUE RECORD
    value : ARRAY MachineBytesPerMachineWord OF UNSAFE.MACHINEBYTE END;
```

11.1.2.5 Type ADDRESS

`ADDRESS` represents references to untyped memory locations.

```
<* IF ByteBoundaryAddressing *> TYPE ADDRESS = POINTER TO UNSAFE.MACHINEBYTE;
<* ELSE *> TYPE ADDRESS = POINTER TO UNSAFE.MACHINEWORD; <* ENDIF *>
```

11.1.3 Low-Level Intrinsics

Low-level intrinsics are pseudo-procedures that act and look like library defined procedures but they may not be assigned to procedure variables and may not be passed to any procedure as parameters. Invocations of intrinsics are translated by the compiler into a sequence of low-level instructions.

11.1.3.1 Intrinsic ADR

Intrinsic `ADR` returns the address of its operand, which must be a variable. Its pseudo-definition is:

```
PROCEDURE ADR ( var : <AnyType> ) : ADDRESS;
```

11.1.3.2 Intrinsic CAST

Intrinsic `CAST` returns the value of its second operand, cast to the target type denoted by its first operand. Its second operand may be a variable, a constant, or a literal. Its pseudo-definition is:

```
PROCEDURE CAST ( <AnyTargetType>; val : <AnyType> ) : <TargetType>;
```

`CAST` further acts as a facility enabler for auto-casting formal parameters.

11.1.3.3 Intrinsic INC

Intrinsic `INC` increments the value of its first operand by the value of its second operand. Any overflow is ignored. Its pseudo-definition is:

```
PROCEDURE INC ( VAR x : <AnyType>; n : OCTET );
```

11.1.3.4 Intrinsic DEC

Intrinsic `DEC` decrements the value of its first operand by the value of its second operand. Any underflow is ignored. Its pseudo-definition is:

```
PROCEDURE DEC ( VAR x : <AnyType>; n : OCTET );
```


11.1.3.5 Intrinsic ADDC

Intrinsic `ADDC` adds the value of its second operand to its first operand, adds 1 if `TRUE` is passed-in its third operand and passes the carry bit back in its third operand. Its pseudo-definition is:

```
PROCEDURE ADDC ( VAR x : <AnyType>; y : <TypeOf(x)>; carry : BOOLEAN );
```

11.1.3.6 Intrinsic SUBC

Intrinsic `SUBC` subtracts the value of its second operand from its first operand, adds 1 if `TRUE` is passed-in its third operand and passes the carry bit back in its third operand. Its pseudo-definition is:

```
PROCEDURE SUBC ( VAR x : <AnyType>; y : <TypeOf(x)>; carry : BOOLEAN );
```

11.1.3.7 Intrinsic SHL

Intrinsic `SHL` returns the value of its first operand shifted left by the number of bits given by its second operand. Any overflow is ignored. Its pseudo-definition is:

```
PROCEDURE SHL ( x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.8 Intrinsic SHR

Intrinsic `SHR` returns the value of its first operand logically shifted right by the number of bits given by its second operand. Any underflow is ignored. Its pseudo-definition is:

```
PROCEDURE SHR ( x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.9 Intrinsic ASHR

Intrinsic `ASHR` returns the value of its first operand arithmetically shifted right by the number of bits given by its second operand. Any underflow is ignored. Its pseudo-definition is:

```
PROCEDURE ASHR ( x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.10 Intrinsic ROTL

Intrinsic `ROTL` returns the value of its first operand rotated left by the number of bits given by its second operand. Any overflow is ignored. Its pseudo-definition is:

```
PROCEDURE ROTL ( x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.11 Intrinsic ROTR

Intrinsic `ROTR` returns the value of its first operand rotated right by the number of bits given by its second operand. Any underflow is ignored. Its pseudo-definition is:

```
PROCEDURE ROTR ( x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.12 Intrinsic ROTLC

Intrinsic `ROTLC` returns the value of its first operand rotated left by the number of bits given by its third operand, rotating through the same number of bits of its second operand. Its pseudo-definition is:

```
PROCEDURE
    ROTLC ( x : <AnyType>; VAR c : <TypeOf(x)>; n : OCTET ) : <TypeOf(x)>;
```

11.1.3.13 Intrinsic ROTRC

Intrinsic `ROTRC` returns the value of its first operand rotated right by the number of bits given by its third operand, rotating through the same number of bits of its second operand. Its pseudo-definition is:

```
PROCEDURE
    ROTRC ( x : <AnyType>; VAR c : <TypeOf(x)>; n : OCTET ) : <TypeOf(x)>;
```


11.1.3.14 Intrinsic BWNOT

Intrinsic `BWNOT` returns the bitwise logical NOT of its operand. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWNOT ( x : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.15 Intrinsic BWAND

Intrinsic `BWAND` returns the bitwise logical AND of its operands. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWAND ( x, y : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.16 Intrinsic BWOR

Intrinsic `BWOR` returns the bitwise logical OR of its operands. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWOR ( x, y : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.17 Intrinsic BWXOR

Intrinsic `BWXOR` returns the bitwise logical exclusive OR of its operands. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWXOR ( x, y : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.18 Intrinsic BWNAND

Intrinsic `BWNAND` returns the inverted bitwise logical AND of its operands. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWNAND ( x, y : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.19 Intrinsic BWNOR

Intrinsic `BWNOR` returns the inverted bitwise logical OR of its operands. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE BWNOR ( x, y : <AnyType> ) : <TypeOf(x)>;
```

11.1.3.20 Intrinsic SETBIT

Intrinsic `SETBIT` sets the n-th bit of its first operand to the value given by its third operand. The value of n is given by its second operand. Any overflow or underflow is ignored. Its pseudo-definition is:

```
PROCEDURE SETBIT ( VAR x : <AnyType>; n : OCTET; bitval : BOOLEAN );
```

11.1.3.21 Intrinsic TESTBIT

Intrinsic `TESTBIT` tests the n-th bit of its first and returns `TRUE` if it is set, otherwise `FALSE`. The value of n is given by its second operand. Its pseudo-definition is:

```
PROCEDURE TESTBIT ( x : <AnyType>; n : OCTET ) : BOOLEAN;
```

11.1.3.22 Intrinsic LSBIT

Intrinsic `LSBIT` returns the bit position of the least significant set bit of its operand. Its pseudo-definition is:

```
PROCEDURE LSBIT ( x : <AnyType> ) : CARDINAL;
```

11.1.3.23 Intrinsic MSBIT

Intrinsic `MSBIT` returns the bit position of the most significant set bit of its operand. Its pseudo-definition is:

```
PROCEDURE MSBIT ( x : <AnyType> ) : CARDINAL;
```

11.1.3.24 Intrinsic CSBITS

Intrinsic `CSBITS` counts and returns the number of set bits of its operand. Its pseudo-definition is:

```
PROCEDURE CSBITS ( x : <AnyType> ) : CARDINAL;
```

11.1.3.25 Intrinsic BAIL

Intrinsic `BAIL` returns program control to the penultimate caller of the procedure where `BAIL` was invoked and may pass its optional operand as the return value. If an operand is passed then its type must match the return type of the calling procedure. It is an error to invoke `BAIL` outside a procedure. Its pseudo-definition is:

```
PROCEDURE BAIL ( (* OPTIONAL *) x : <AnyType> );
```

11.1.3.26 Intrinsic HALT

Intrinsic `HALT` immediately aborts the running program and returns a status code to the operating environment. The meaning of status codes is target platform dependent. Its pseudo-definition is:

```
PROCEDURE HALT ( status : <OrdinalType> );
```

11.1.4 Unsafe Pragma Enablers

Unqualified import of a pragma enabler will enable the use of an associated unsafe pragma in the importing module. A pragma enabler occurring outside of an unqualified import list shall result in a compile time error.

11.1.4.1 Pragma Enabler ADDR

An implementation that supports pragma `ADDR` shall provide pragma enabler `ADDR`.

11.1.4.2 Pragma Enabler FFI

An implementation that supports pragma `FFI` shall provide pragma enabler `FFI`.

11.1.5 Interfacing to Unsafe Variadic Procedures in Foreign APIs

Implementations that support pragma `FFI` shall provide pseudo-type `VARGLIST` and macro `VARGC` to allow interfacing to foreign variadic procedures whose variadic parameter expects arguments of arbitrary type because type safe variadic procedure declarations in Modula-2 cannot be used to interface to such procedures.

11.1.5.1 Pseudo-Type VARGLIST

Pseudo-type `VARGLIST` may be used within the formal parameter list of a foreign procedure definition to declare an untyped variadic parameter `p` causing type checking to be disabled when passing arguments to `p`.

Example:

```
DEFINITION MODULE stdio <*FFI="C"*>; (* foreign library interface *)
FROM UNSAFE IMPORT FFI, VARGLIST;
PROCEDURE printf( fmt : ARRAY OF CHAR; varglist : VARGLIST );
```

`VARGLIST` may only appear as formal type of the last formal parameter of a foreign procedure definition in a definition part that is an interface to a foreign library. Any other use shall result in a compile-time error.

11.1.5.2 Macro VARGC

Macro `VARGC` is replaced at compile time by the argument count of an argument list passed to an untyped variadic parameter of formal type `VARGLIST` within a call to a foreign variadic procedure that has a foreign interface definition. Its replacement value is compatible with any predefined whole number type.

Given a foreign interface definition of a foreign variadic procedure `foreign` as shown below:

```
DEFINITION MODULE ForeignLib <*FFI="C"*>;
FROM UNSAFE IMPORT FFI, VARGLIST;
PROCEDURE foreign ( vargc : INTEGER; varglist : VARGLIST );
```

Macro `VARGC` may be placed within the argument list of a call to procedure `foreign` as shown below:

```
FROM UNSAFE IMPORT VARGC; FROM ForeignLib IMPORT foreign;
foreign( VARGC, 42, 1.23, "foo" ); (* equivalent to: foreign( 3, 42, 1.23, "foo"); *)
```

`VARGC` may only appear within the argument list of a call to a foreign variadic procedure with a foreign interface definition whose variadic parameter is of type `VARGLIST`. However, it may not be part of the argument list passed to the variadic parameter. Any non-compliant use shall result in a compile time error.

11.2 Pseudo-Module ATOMIC

Pseudo-module `ATOMIC` provides intrinsics for atomic operations.

11.2.1 Testing The Availability Of Atomic Intrinsics

The availability of atomic operations is dependent on the target architecture. Not all CPUs support all operations and operands. Module `ATOMIC` provides enumeration type `INTRINSIC` and function `AVAIL` to test the availability of atomic intrinsics. Type `INTRINSIC` enumerates mnemonics for all possible atomic intrinsics.

```
TYPE INTRINSIC = ( SWAP, CAS, INC, DEC, BWAND, BWNAND, BWOR, BWXOR );
```

Function `AVAIL` returns the availability of the atomic intrinsic given by its first operand for the bit width given by its second operand. It returns `TRUE` if the operation is available. Its definition is:

```
PROCEDURE AVAIL ( intrinsic : INTRINSIC; bitwidth : CARDINAL ) : BOOLEAN;
```

11.2.2 ATOMIC Intrinsics

`ATOMIC` intrinsics are pseudo-procedures that act and look like library defined procedures but they may not be assigned to procedure variables and may not be passed to any procedure as parameters. Invocations of intrinsics are translated by the compiler into their respective machine instructions.

11.2.2.1 Intrinsic SWAP

Atomic intrinsic `SWAP` atomically swaps the values of its operands. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE SWAP ( VAR x, y : <AnyType> );
```

11.2.2.2 Intrinsic CAS

Atomic intrinsic `CAS` atomically compares its first and second operands and if they match, swaps the values of its second and third operands, and returns the original value of the second operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE CAS ( VAR expectedValue, x, y : <AnyType> ) : <OperandType>;
```

11.2.2.3 Intrinsic BCAS

Atomic intrinsic `BCAS` atomically compares its first and second operands and if they match, swaps the values of its second and third operands. It returns the result of the compare operation. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE BCAS ( VAR expectedValue, x, y : <AnyType> ) : BOOLEAN;
```


11.2.2.4 Intrinsic INC

Atomic intrinsic `INC` atomically increments the values of its first operand by the value given by its second operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE INC ( VAR x : <AnyType>; y : <TypeOf(x)> );
```

11.2.2.5 Intrinsic DEC

Atomic intrinsic `DEC` atomically decrements the values of its first operand by the value given by its second operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE DEC ( VAR x : <AnyType>; y : <TypeOf(x)> );
```

11.2.2.6 Intrinsic BWAND

Atomic intrinsic `BWAND` atomically performs the bitwise logical `AND` of its operands and passes the result back in its first operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE BWAND ( VAR x : <AnyType>; y : <TypeOf(x)> );
```

11.2.2.7 Intrinsic BWNAND

Atomic intrinsic `BWNAND` atomically performs the bitwise logical `NAND` of its operands and passes the result back in its first operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE BWNAND ( VAR x : <AnyType>; y : <TypeOf(x)> );
```

11.2.2.8 Intrinsic BWOR

Atomic intrinsic `BWOR` atomically performs the bitwise logical `OR` of its operands and passes the result back in its first operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE BWOR ( VAR x : <AnyType>; y : <TypeOf(x)> );
```

11.2.2.9 Intrinsic BWXOR

Atomic intrinsic `BWXOR` atomically performs the bitwise logical exclusive `OR` of its operands and passes the result back in its first operand. The operands must be 8-, 16- 32- or 64-bit wide. Its pseudo-definition is:

```
PROCEDURE BWXOR ( VAR x : <AnyType>; y : <TypeOf(x)> );
```


11.3 Pseudo-Module COROUTINE

Pseudo-module `COROUTINE` will provide intrinsics for concurrency based on coroutines.

11.3.1 Objectives and Requirements

This pseudo-module will be defined in Phase II of the language revision. However, its objectives and high level requirements have already been defined during Phase I:

- Local procedures shall not be used as coroutines
- Coroutines shall be organised into coroutine pools
- Coroutines with the same pool shall be able to share non-global data
- Only coroutines within the same pool shall be able to yield to each other

The pseudo-module will need to provide:

- a means to create and destroy coroutine pools
- a means to create and destroy coroutine tasks within a pool
- a means to pass non-global data between coroutines of the same pool
- a means to pass execution control to another coroutine within the same pool
- miscellaneous means of introspection for coroutines and coroutine pools

11.4 Pseudo-Module ACTOR

Pseudo-module `ACTOR` will provide intrinsics for concurrency based on actors.

11.4.1 Objectives and Requirements

This pseudo-module will be defined in Phase II of the language revision. No objectives and no requirements have been defined during Phase I.

11.5 Optional Pseudo-Module ASSEMBLER

Pseudo-module `ASSEMBLER` is an optional module providing access to CPU registers, a raw machine code inline facility, and a symbolic assembly inline-facility with implementation defined assembly language syntax for one or more target architectures. The use of module `ASSEMBLER` is unsafe and non-portable.

All facilities provided are optional, subject to the following constraints:

- module `ASSEMBLER` may not be provided void of any facilities
- type `REGISTER` shall be provided if intrinsics `SETREG`, `GETREG` or attribute `REG` are provided
- intrinsics `SETREG` and `GETREG` shall be provided if intrinsic `CODE` is provided
- intrinsics `SETREG` and `GETREG` may only be provided as a pair, not one without the other

11.5.1 Testing The Availability Of Assembler Facilities

Module `COMPILER` provides boolean macros to test the availability of optional facilities supported by module `ASSEMBLER` for the current target architecture. For details see section 10.1.2.

11.5.2 Register Mnemonics Of The Target Architecture

Module `ASSEMBLER` may provide enumeration type `REGISTER` defining CPU register mnemonics for the current target architecture. Mnemonics for selected architectures are given in the documentation module for `ASSEMBLER` within the standard library. An example definition is shown below:

Pseudo-Definition Example:
```
(* Register mnemonics for target architecture Intel 8051 *)
TYPE REGISTER = ( a, b, r1, r2, r3, r4, r5, r6, r7, dp, sp, psw );
```

If module `ASSEMBLER` provides type `REGISTER`, it shall also provide a set of convenience constants for the unqualified use of register mnemonics:

Pseudo-Definition Example:
```
CONST (* convenience constants *)
  a = REGISTER.a;
  b = REGISTER.b;
  r1 = REGISTER.r1; (* etc. *)
```

11.5.3 Intrinsic SETREG

Module `ASSEMBLER` may provide intrinsic `SETREG` to load a value into a CPU register.

Pseudo-Definition:
```
PROCEDURE SETREG ( r : REGISTER; v : <ValueType> );
```

An invocation of intrinsic `SETREG` loads a value `v` of arbitrary type given by its second argument into register `r` given by its first argument. It is a compile time error if the bit width of `v` exceeds the bit width of `r`.

Example:
```
SETREG(r3, counter); (* load the value of counter into register r3 *)
```

11.5.4 Intrinsic GETREG

Module `ASSEMBLER` may provide intrinsic `GETREG` to load the contents of a CPU register into a variable.

Pseudo-Definition:
```
PROCEDURE GETREG ( r : REGISTER; VAR value : <ValueType> );
```

An invocation of intrinsic `GETREG` loads the contents of register `r` given by its first argument into variable `v` given by its second argument. It is a compile time error if the bit width of `v` exceeds the bit width of `r`.

Example:
```
GETREG(r7, listPtr); (* load the contents of register r7 into variable listPtr *)
```


11.5.5 Intrinsic CODE

Module `ASSEMBLER` may provide intrinsic `CODE` to insert raw machine code into a library or program.

Pseudo-Definition:
```
PROCEDURE CODE ( rawMachineCodeValues : ARGLIST OF [0..255] OF CARDINAL );
```

An invocation of intrinsic `CODE` inserts the raw machine code represented by its argument list at the location where the intrinsic appears in the source text. `CODE` is variadic. Arguments must be compile time expressions of type `[0..255] OF CARDINAL`. A compile time error shall occur if any argument value is out of range.

Example:
```
CODE(0xC3, 0x0F, 0x00);
```

11.5.6 Language Extension REG

Module `ASSEMBLER` may provide attribute `REG` to map a formal parameter `p` in a procedure definition or procedure type declaration to a machine register `r` causing parameter `p` to be passed in register `r`. An implementation that supports this language extension shall provide reserved word `REG` and facility enabler `REG`.

EBNF:
```
regAttribute :
    IN REG ( registerNumber | registerMnemonic ) ;
registerNumber : constExpression ;  registerMnemonic : Ident ;
```

The register mapping becomes part of the signature of a procedure or procedure type. For two signatures to be compatible, corresponding parameters must match and be mapped to the same registers.

Examples:
```
FROM ASSEMBLER IMPORT REG; (* enable language extension *)
PROCEDURE Foo ( n : CARDINAL IN REG 3 );  PROCEDURE Bar ( n : CARDINAL );
TYPE FooProc = PROCEDURE ( CARDINAL IN REG 3 );  VAR fooProc : FooProc;
fooProc := Foo; (* OK *)  fooProc := Bar; (* error: incompatible signatures *)
```

Multiple parameters may not be mapped to the same register. To map multiple parameters, each parameter requires a separate attribute. `REG` may not be used with open array parameters or variadic parameters. The bit width of a mapped parameter must not exceed the bit width of the machine register to which it is mapped.

11.5.7 Language Extension ASM

Module `ASSEMBLER` may provide a symbolic assembly language extension `ASM`.

EBNF:
```
assemblyBlock :
    ASM assemblySourceCode END ;
assemblySourceCode : <implementation defined syntax> ;
```

An `ASM` block is a statement. The syntax within is implementation defined and target dependent. An implementation that supports this language extension shall provide reserved word `ASM` and facility enabler `ASM`.

Example:
```
FROM ASSEMBLER IMPORT ASM; (* enable language extension *)
ASM
  mov eax, ebx;
  xor ebx, ebx
END;
```


12 Pragmas

Pragmas are directives to the compiler, used to control or influence the compilation process, but they do not change the meaning of a program. Language defined pragmas and their properties are listed below:

Pragma	Usage	Scope	Availability	Safety
`MSG`	Emit compile time console messages	Pragma	mandatory	safe
`IF`	Conditional compilation, if-branch	Pragma	mandatory	safe
`ELSIF`	Conditional compilation, elsif-branch	Pragma	mandatory	safe
`ELSE`	Conditional compilation, else-branch	Pragma	mandatory	safe
`ENDIF`	Conditional compilation, terminator	Pragma	mandatory	safe
`INLINE`	Suggest procedure inlining	Procedure	mandatory	safe
`NOINLINE`	Forbid procedure inlining	Procedure	mandatory	safe
`PTW`	Promise to write to a `VAR` parameter	Formal parameter	mandatory	safe
`GENERATED`	Date/time stamp of library generation	see 12.3.3.4	see 12.3.3.4	safe
`FORWARD`	Forward declarations	Pragma	see 12.3.3.5	safe
`ENCODING`	Specify source text character encoding	File	see 12.3.3.6	safe
`ALIGN`	Specify memory alignment	Module, Type, Field List	optional	safe
`PADBITS`	Insert padding bits into packed records	Field List	optional	safe
`NORETURN`	Promise never to return	Regular Procedure	optional	safe
`PURITY`	Specify procedure purity level	Procedure, Type	optional	safe
`SINGLEASSIGN`	Mark single-assignment variable	Global Var, Local Var	optional	safe
`LOWLATENCY`	Mark latency-critical variable	Local Var	optional	safe
`VOLATILE`	Mark volatile variable	Global Var	optional	safe
`DEPRECATED`	Mark deprecated entity	Definitions, Declarations	optional	safe
`ADDR`	Map procedure or variable to fixed address	Global Var, Procedure	optional	unsafe
`FFI`	Specify foreign function interface	Module	optional	unsafe
`FFIDENT`	Map identifier to foreign library identifier	Global Var, Procedure	optional	unsafe

12.1 Pragma Scope And Positioning

The position where a pragma may appear in the source text depends on its scope.

Scope	Applies to	Insertion Point
File	entire file	at beginning of a file or immediately after a BOM
Module	entire module	between module header and its trailing semicolon
Pragma	pragma itself	anywhere a block comment may appear
Type	array or procedure type declaration	between type declaration and its trailing semicolon
Field List	insertion point forward	anywhere a field list may appear within a record
Global Var	all variables in declaration	between variable declaration and its trailing semicolon
Procedure	procedure declaration	between procedure header and its trailing semicolon
Formal Parameter	formal parameter declaration	immediately after the formal type of the parameter
Local Var	all variables in declaration	between variable declaration and its trailing semicolon

12.2 Pragma Safety

A pragma is safe if it provides a safe facility. It is unsafe if it provides an unsafe facility. The use of any unsafe pragma must be enabled by unqualified import of its identically named enabler. Pragma enablers of supported unsafe pragmas shall be provided by module `UNSAFE`.

12.3 Language Defined Pragmas In Detail

12.3.1 Pragma MSG

Pragma `MSG` emits four different types of user defined console messages during compilation: informational messages, compilation warnings, compilation error messages and fatal compilation error messages. A message mode selector determines the type of message. Console messages consist of a quoted string literal, the value of a compile time constant or pragma, or a comma separated list of these components.

EBNF:
```
pragmaMSG :
    "<*" MSG "=" ( INFO | WARN | ERROR | FATAL ) ":"
         compileTimeMsgComponent ( "," compileTimeMsgComponent )* "*>" ;
compileTimeMsgComponent :
    String | constQualident | "@" ( ALIGN | ENCODING | implDefPragmaName ) ;
```

The value of a pragma that represents a compile time setting is denoted by the pragma symbol prefixed with a question mark. Language defined pragmas that represent compile time settings are `ALIGN` and `ENCODING`.

Examples:
```
<*MSG=INFO : "The current alignment is: ", @ALIGN*>  (* emits the alignment value *)
<*MSG=INFO : "The current encoding is: ", @ENCODING*>  (* emits the encoding's name *)
```

Only pragmas that represent a compile time setting may be queried in this way.

12.3.1.1 Message Mode INFO

Message mode selector `INFO` is used to emit user defined information during compilation. Emitting an informational message does not change the error or warning count of the current compilation run and it does not cause compilation to fail or abort. A compiler switch may be provided to silence informational messages.

Example:
```
<*MSG=INFO : "Library documentation is available at http://foolib.com"*>
```

12.3.1.2 Message Mode WARN

Message mode selector `WARN` is used to emit user defined warnings during compilation. Emitting a warning message increments the warning count of the current compilation run but it does not cause compilation to fail or abort. Warnings emitted via pragma `MSG` are always hard compile time warnings.

Example:
```
<*MSG=WARN : "foo exceeds maximum value. A default of 100 will be used."*>
```

12.3.1.3 Message Mode ERROR

Message mode selector `ERROR` is used to emit user defined error messages during compilation. Emitting an error message increments the error count of the current compilation run and will ultimately cause compilation to fail but it does not cause an immediate abort. Error messages may not be silenced.

Example:
```
<*MSG=ERROR : "Value of foo is outside of its legal range of [1..100]."*>
```

12.3.1.4 Message Mode FATAL

Message mode selector `FATAL` is used to emit user defined fatal error messages during compilation. Emitting a fatal error message increments the error count of the current compilation run and causes compilation to fail and abort immediately. Fatal error messages may not be silenced. Compilation abort may not be avoided.

Example:
```
<*MSG=FATAL : "Unsupported target architecture."*>
```


12.3.2 Pragmas For Conditional Compilation

Conditional compilation pragmas are used to denote conditional compilation sections. A conditional compilation section is an arbitrary portion of source text that is either compiled or ignored depending on whether or not a given condition in form of a boolean compile time expression within the pragma is met.

A conditional compilation section consists of an initial conditional compilation branch denoted by pragma `IF`, followed by zero or more alternative branches denoted by pragma `ELSIF`, followed by an optional default branch denoted by pragma `ELSE`, followed by closing pragma `ENDIF`.

EBNF:
```
conditionalCompilationSection :
    pragmaIF token* ( pragmaELSIF token* )* ( pragmaELSE token* )? pragmaENDIF ;
```

Example:
```
<*IF (TSIZE(INTEGER)=2)*> CONST model = Model.small;
<*ELSIF (TSIZE(INTEGER)=4)*> CONST model = Model.medium;
<*ELSIF (TSIZE(INTEGER)=8)*> CONST model = Model.large;
<*ELSE*> <*MSG=FATAL : "unsupported type model"*>
UNSAFE.HALT(Errors.UnsupportedTypeModel);
<*ENDIF*>
```

Conditional compilation sections may be nested up to a maximum nesting level of ten including the outermost conditional compilation section. A fatal compile time error occurs if this value is exceeded. Pragma `IF`, increments the current nesting level, `ELSIF` and `ELSE` leave it unchanged, and `ENDIF` decrements it.

12.3.2.1 Pragma IF

Pragma `IF` denotes the start of the initial branch of a conditional compilation section. The source text within the initial branch is only processed if the condition specified in the pragma is true, otherwise it is ignored.

EBNF:
```
pragmaIF : "<*" IF inPragmaExpression "*>" ;
```

12.3.2.2 Pragma ELSIF

Pragma `ELSIF` denotes the start of an alternative branch in a conditional compilation section. The source text within an alternative branch is only processed if the condition specified in the pragma is true and the conditions specified for the initial branch and all preceding alternative branches of the same nesting level are false, otherwise it is ignored.

EBNF:
```
pragmaELSIF : "<*" ELSIF inPragmaExpression "*>" ;
```

12.3.2.3 Pragma ELSE

Pragma `ELSE` denotes the start of a default branch within a conditional compilation section. The source text within the default branch is only processed if the conditions specified for the initial branch and all preceding alternative branches of the same nesting level are false, otherwise it is ignored.

EBNF:
```
pragmaELSE : "<*" ELSE "*>" ;
```

12.3.2.4 Pragma ENDIF

Pragma `ENDIF` denotes the end of a conditional compilation section.

EBNF:
```
pragmaENDIF : "<*" ENDIF "*>" ;
```


12.3.3 Pragmas To Control Code Generation

12.3.3.1 Pragma INLINE

Pragma `INLINE` represents a *suggestion* that inlining of a procedure is desirable. It must appear both in the definition and implementation. An informational message shall be emitted if the suggestion is not followed.

EBNF:
```
pragmaINLINE : "<*" INLINE "*>" ;
```

Example:
```
PROCEDURE Foo ( bar : Baz ) <*INLINE*>;
```

12.3.3.2 Pragma NOINLINE

Pragma `NOINLINE` represents a *mandate* that a procedure shall not be inlined. It must appear both in the definition and implementation. The use of pragmas `INLINE` and `NOINLINE` is mutually exclusive.

EBNF:
```
pragmaNOINLINE : "<*" NOINLINE "*>" ;
```

Example:
```
PROCEDURE Foo ( bar : Baz ) <*NOINLINE*>;
```

12.3.3.3 Pragma PTW

Pragma `PTW` marks a formal `VAR` parameter `p` in the header of a procedure `P` with a *promise to write*. The promise is kept if it can be proven that `p` is written to within the body of `P` in every possible runtime scenario, either by assignment or by passing `p` to a `PTW` marked `VAR` parameter in a procedure call other than via procedure variable. A promotable soft compile time warning shall occur if the promise is not kept.

EBNF:
```
pragmaPTW : "<*" PTW "*>" ;
```

Example:
```
PROCEDURE init ( VAR ch : CHAR <*PTW*> );
```

12.3.3.4 Pragma GENERATED

Pragma `GENERATED` encodes the name of the template a library was generated from and the date and time when it was last generated. The pragma is inserted into the source by the Modula-2 template engine. An implementation shall use the information recorded in the pragma to avoid unnecessary regeneration of libraries.

EBNF:
```
pragmaGENERATED :
    "<*" GENERATED FROM template "," datestamp , "," timestamp "*>" ;
datestamp :
    year "-" month "-" day ;
timestamp :
    hours ":" minutes ":" seconds "+" timezone ;
year, month, day, hours, minutes, seconds, timezone : wholeNumber ;
```

Example:
```
<*GENERATED FROM AssocArrays, 2013-12-31, 23:59:59+0100*>
```

12.3.3.5 Pragma FORWARD

Pragma `FORWARD` shall be the only means of forward declaration in a single-pass compiler. Multi-pass implementations shall silently ignore any occurrences of pragma `FORWARD` without analysis of its contents. Two kinds of forward declarations may be embedded in the pragma: Type and procedure declarations.

EBNF:
```
pragmaFORWARD :
    "<*" FORWARD ( TYPE identList | procedureHeader ) "*>" ;
```

Example:
```
<*FORWARD TYPE ListNode*>
TYPE ListNodePtr = POINTER TO ListNode;
TYPE ListNode = RECORD data : Foo; nextNode : ListNodePtr END;
```


12.3.3.6 Pragma ENCODING

Pragma `ENCODING` specifies the encoding of the source file in which it appears.

EBNF:
```
pragmaENCODING :
    "<*" ENCODING "=" ( '"ASCII"' | '"UTF8"' ) codePointSampleList? "*>" ;
```

The pragma controls whether in addition to the characters that are permitted by the grammar, any further printable characters are permitted within quoted literals and comments. Semantics are given below:

BOM	Encoding Pragma	Characters Permitted in Quoted Literals and Comments
No BOM in file	No encoding pragma in source	only characters that are permitted by the grammar
	with specifier "ASCII"	
	with specifier "UTF8"	additionally any printable character that is encodable in UTF8
UTF8 BOM	No encoding pragma in source	only characters that are permitted by the grammar
	with specifier "ASCII"	
	with specifier "UTF8"	additionally any printable character that is encodable in UTF8
Any other BOM	Support is implementation defined; Use of pragma is mandatory; BOM and specifier must match	

An implementation that supports ASCII only shall recognise encoding specifier `"ASCII"`. It shall ignore any UTF8 BOM but reject any non-ASCII characters in the source file. An implementation that supports UTF8 shall recognise encoding specifiers `"ASCII"` and `"UTF8"`. Support for other encodings is implementation defined. Only one encoding pragma per source file is permitted.

12.3.3.6.1 Encoding Verification

As an option, pragma `ENCODING` may provide encoding verification. If supported, a list of arbitrary samples with pairs of quoted characters and their respective code point values may follow the encoding specifier.

EBNF:
```
codePointSampleList :
    ":" codePointSample ( "," codePointSample )* ;
codePointSample :
    quotedCharacterLiteral "=" characterCodeLiteral ;
```

If a sample list is specified within the pragma body, a verification is carried out by matching the quoted literals in the sample list against their respective code points. Any mismatching pair in the sample list shall cause a fatal compilation error and compilation to abort immediately. The maximum number of code point samples is implementation defined. A maximum of at least 16 is recommended. Excess samples shall be ignored.

Example:
```
<*ENCODING="UTF8" : "é"=0uE9, "©"=0uA9, "€"=0u20AC*>
```

12.3.3.7 Optional Pragma ALIGN

Pragma `ALIGN` controls memory alignment. Alignment is specified in octets.

EBNF:
```
pragmaALIGN : "<*" ALIGN "=" inPragmaExpression "*>" ;
```

When the pragma is placed in a module header, it has module scope and determines the default alignment within the module. Permitted alignment values range from one to 32 octets.

Example:
```
DEFINITION MODULE Foolib <*ALIGN=TSIZE(CARDINAL)*>; (* module scope *)
```

When the pragma is placed at the end of an array type declaration, it has array scope and determines the alignment of array components. Permitted alignment values range from one to 32 octets.

Example:
```
TYPE Array = ARRAY 10 OF OCTET <*ALIGN=4*>; (* array scope *)
```

When the pragma is placed in the body of a record type declaration, it has field list scope and determines the alignment of record fields following the pragma. Permitted alignment values range from zero to 32 octets.

A value of zero specifies packing. When packing is specified, the allocation size of a field of an anonymous subrange of type `OCTET`, `CARDINAL` and `LONGCARD` is reduced to the smallest bit width required to encode its value range. Fields of any other type are aligned on octet boundaries when packing is specified.

Example:
```
TYPE Aligned = RECORD
<*ALIGN=2*>  foo, bar : INTEGER;  (* 16 bit aligned *)
<*ALIGN=4*>  baz, bam : INTEGER;  (* 32 bit aligned *)
<*ALIGN=0*>  bits, bobs : [0..15] OF OCTET  (* packed *)
END; (* Aligned *)
```

12.3.3.8 Optional Pragma PADBITS

Pragma `PADBITS` inserts a specified number of padding bits into a packed record type declaration. The maximum permitted value is 256 bits. The pragma is only permitted when alignment is set to zero.

EBNF:
```
pragmaPADBITS : "<*" PADBITS "=" inPragmaExpression "*>" ;
```

Example:
```
TYPE Packed = RECORD <*ALIGN=0*>
   oneBit    : [0..1] OF OCTET; (* 1 bit *)
<*PADBITS=2*>               (* unused 2 bits *)
   twoBits   : [0..3] OF OCTET; (* 2 bits *)
   threeBits : [0..7] OF OCTET; (* 3 bits *)
END; (* Packed *)
```

12.3.3.9 Optional Pragma NORETURN

Pragma `NORETURN` marks a regular procedure with a promise never to return in any runtime scenario. A soft compile time warning shall occur if the compiler cannot prove that the promise is kept.

EBNF:
```
pragmaNORETURN : "<*" NORETURN "*>" ;
```

Example:
```
PROCEDURE RebootSystem <*NORETURN*>;
```

12.3.3.10 Optional Pragma PURITY

Pragma `PURITY` marks a procedure with an intended purity level:

- level 0 : may read and modify global state, may call procedures of any level (Default)
- level 1 : may read but not modify global state, may only call level 1 and level 3 procedures
- level 2 : may not read but modify global state, may only call level 2 and level 3 procedures
- level 3 : pure procedure, may not read nor modify global state, may only call level 3 procedures

An implementation shall emit a promotable soft compile time warning for any purity level violation.

EBNF:
```
pragmaPURITY : "<*" PURITY "=" inPragmaExpression "*>" ;
```

Example:
```
PROCEDURE Foo ( bar : Bar) : Baz <*PURITY=3*>; (* pure and side-effect free *)
```

12.3.3.11 Optional Pragma SINGLEASSIGN

Pragma `SINGLEASSIGN` marks a variable as a single-assignment variable.

Such a variable should be assigned to only once in every possible runtime scenario. An implementation shall issue a promotable soft compile time warning for any single-assignment violation it may detect.

EBNF:
```
pragmaSASSN : "<*" SINGLEASSIGN "*>" ;
```

Example:
```
VAR foo : INTEGER <*SINGLEASSIGN*>;
```

12.3.3.12 Optional Pragma LOWLATENCY

Pragma `LOWLATENCY` marks a local variable as latency-critical.

Marking a variable latency-critical represents a *suggestion* that mapping the variable to a machine register is desirable. An informational message shall be emitted if the suggestion is not followed.

EBNF:
```
pragmaLOWLATENCY : "<*" LOWLATENCY "*>" ;
```

Example:
```
VAR foo : INTEGER <*LOWLATENCY*>;
```

12.3.3.13 Optional Pragma VOLATILE

Pragma `VOLATILE` marks a global variable as volatile.

By marking a variable volatile the author states that its value may change during the life time of a program even if no write access can be deduced from source code analysis. An implementation shall neither eliminate any variable so marked, nor shall it emit any unused variable warning for any variable so marked.

EBNF:
```
pragmaVOLATILE : "<*" VOLATILE "*>" ;
```

Example:
```
VAR foo : INTEGER <*VOLATILE*>;
```

12.3.3.14 Optional Pragma DEPRECATED

Pragma `DEPRECATED` marks a constant, variable, type or procedure as deprecated. A promotable soft compile time warning shall occur whenever an identifier of a deprecated entity is encountered.

EBNF:
```
pragmaDEPRECATED : "<*" DEPRECATED "*>" ;
```

Example:
```
PROCEDURE foo ( bar : Baz ) <*DEPRECATED*>;
```

12.3.3.15 Optional Pragma ADDR

Pragma `ADDR` maps a procedure or a global variable to a fixed memory address.

EBNF:
```
pragmaADDR : "<*" ADDR "=" inPragmaExpression "*>" ;
```

Examples:
```
PROCEDURE Reset <*ADDR=0x12*>;
VAR memoryMappedPort : CARDINAL <*ADDR=0x100*>;
```

12.3.3.16 Optional Pragma FFI

Pragma `FFI` marks a Modula-2 definition part as the Modula-2 interface to a library implemented in another language. Procedure definitions and type declarations in the definition, part shall follow the calling convention of the specified language environment for the current target. Predefined foreign interface specifiers are `"C"`, `"Fortran"`, `"CLR"` and `"JVM"`. If pragma `FFI` is provided, at least one foreign interface shall be supported. CLR or JVM support is recommended for implementations that target the CLR or JVM, respectively.

EBNF:
```
pragmaFFI : "<*" FFI "=" foreignInterfaceName "*>" ;
foreignInterfaceName : String ;
```

The module identifier of a Modula-2 interface to a foreign library must match the name of its foreign library.

Example:
```
DEFINITION MODULE stdio <*FFI="C"*>;
FROM UNSAFE IMPORT FFI, VARGLIST;
PROCEDURE printf ( CONST format : ARRAY OF CHAR; arglist : VARGLIST );
```


12.3.3.17 Optional Pragma FFIDENT

Pragma `FFIDENT` maps a Modula-2 identifier of a foreign procedure or variable definition to its respective identifier in the foreign library. It shall be used when the foreign identifier conflicts with Modula-2 reserved words or reserved identifiers. The pragma may only be used within a foreign function interface module.

EBNF:
```
pragmaFFIDENT : “<*” IDENT “=” StringLiteral “*>” ;
```

Examples:
```
PROCEDURE Length ( s : ARRAY OF CHAR ) : INTEGER <*FFIDENT=”LENGTH”*>;
VAR rwMode : [0..3] OF CARDINAL <*VOLATILE*> <*FFIDENT=”foobarlib_rw_mode”*>;
```

12.4 Implementation Defined Pragmas

Implementation defined pragmas are compiler specific and non-portable.

EBNF:
```
implDefinedPragma :
    "<*" pragmaSymbol ( “=” inPragmaExpression )?
         “|” ( INFO | WARN | ERROR | FATAL) "*>" ;
pragmaSymbol : Ident ; (* all-lowercase or mixed case *)
```

An implementation defined pragma starts with a pragma symbol, which may be followed by a value assignment. The pragma ends with a mandatory incognito clause. The pragma symbol is an implementation defined name which shall be all-lowercase or mixed case. The value assignment follows if and only if the pragma is defined to hold a value. Such a value may be either a boolean value or a whole number.

Example:
```
<*UnrollLoops=TRUE|WARN*> (* turn loop-unrolling on, ignore but warn if unknown *)
```

The incognito clause specifies how the pragma shall be treated by implementations that do not recognise it. Modes `INFO` and `WARN` mandate it shall be ignored with an informational or warning message respectively. Modes `ERROR` and `FATAL` mandate it shall cause a compile time or fatal compile time error respectively.

12.5 Incorrect Pragma Use and Unrecognised Pragmas

A pragma is incorrectly used if it is malformed, misplaced or any other rule for its use set out in section 12 is not met. Any incorrect use of a mandatory pragma or a supported optional pragma shall cause a compile time error. Use of an unsafe pragma that is not supported or has not been enabled shall cause a compile time error.

Use of a safe optional pragma that is not supported shall cause a promotable soft compile time warning. An unsupported or unrecognised encoding specifier in pragma `ENCODING` shall cause a fatal compile time error. A code point sample list within pragma `ENCODING` shall be ignored if encoding verification is not supported. If a code point sample list or any excess samples are ignored a soft compile time warning shall be emitted. An unsupported or unrecognised language specifier in pragma `FFI` shall cause a compile time error.

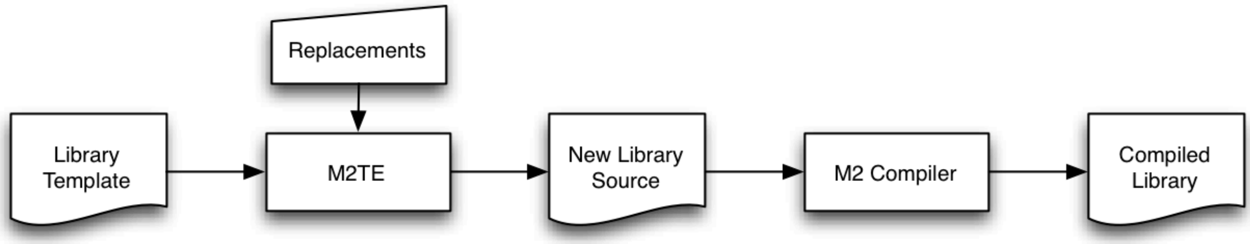
An unrecognised implementation defined pragma shall be treated as specified within its pragma body. An implementation defined pragma without such specifier is malformed and shall cause a compile time error.

13 Generics

Generic programming is not part of the core language, but is instead provided in form of a text template facility that is external to the language but may be invoked from within a Modula-2 source file.

13.1 The Modula-2 Template Engine

The Modula-2 Template Engine (M2TE) is a simple text template facility that recursively expands one or more marked placeholders in an input text file called a template to their respective replacements to produce Modula-2 library source text as its output. It may be invoked either manually prior to compilation, or automatically on an on-demand basis using the `GENLIB` directive within a Modula-2 source file. The output generated by the M2TE utility may then be compiled like any other Modula-2 source text.



Unlike built-in generics commonly used in other languages, the M2TE utility does not verify whether its input files conform to the syntax of the source language. Doing so would be unnecessary duplication because the generated output will be verified anyhow by a compiler when it is later compiled.

Moving the generics facility out of the language core and compiler into an external utility reduces complexity both in the core language and in the compiler. Allowing placeholders to appear anywhere in a template and their replacements to be free form text results in more flexibility. The output generated by the M2TE utility may be thought of as a third party supplied source library and can be examined like one. Thus, the programmer sees what compiler and debugger see, thereby leading to better transparency.

13.2 Template Format

The M2TE utility recognises the following symbols within a template:

Category	Symbol	Usage	Lexical Scope
Template Symbols	`<#   #>`	Template Engine Directive Delimiters	outside of strings and all comments
	`##   @@`	Template Placeholder Delimiters	outside of strings and template comments
	`/*   */`	Template Comment Delimiters	outside of strings and Modula-2 comments

13.2.1 Template Directives

The M2TE utility interprets a character sequence enclosed by symbols <# and #> as a directive. Similar to pragmas in the core language, template engine directives may be used to influence or control template expansion. Template directives and any trailing empty lines are not copied into the output.

13.2.2 Template Placeholders

The M2TE utility interprets an alphanumeric Modula-2 identifier enclosed by a leading and trailing ## or `@@` symbol as a placeholder. It recursively expands all placeholders to their respective replacements in the output.

13.2.3 Template Comments

The M2TE utility interprets a character sequence enclosed by a leading `/*` and trailing `*/` symbol as a template comment. Template comments and any trailing empty lines are not copied into the output.

A full description of the template engine grammar is given in the appendix.

13.3 Invoking Template Expansion

13.3.1 Manual Invocation

Manual invocation of the M2TE utility is implementation dependent. An implementation may have a command line or graphical user interface, or it may be integrated into an IDE, or a combination thereof.

13.3.2 Automatic Invocation Using the GENLIB Directive

The `GENLIB` directive may be used within the import section of a Modula-2 client library or program to invoke the M2TE utility to generate the source files for a new library module from a library template on-demand during compilation of said client library or program. The generated library is subsequently available for import.

The `GENLIB` directive contains the module identifier for the new library, the name of the template to be used and one or more placeholder/replacement pairs required for the expansion of the template.

EBNF:
```
genLibDirective :
    GENLIB libraryIdent FROM templateIdent FOR templateParamList END ;
templateParamList :
    templateParam ( ";" templateParam )*
templateParam :
    placeholder "=" replacementText ;
libraryIdent, templateIdent, placeholder : Ident ;
replacementText : StringLiteral ;
```

Example:
```
MODULE FooApp;
GENLIB EmployeeTable FROM AssocArrays FOR ValueType="Employee" END;
IMPORT EmployeeTable;
```

The example shows how a client module `FooApp` uses the `GENLIB` directive to invoke the M2TE utility to generate a new library called `EmployeeTable` from library template `AssocArrays` passing the string “Employee” as a replacement text for placeholder `ValueType`.

Typically, a software project will generate all its libraries within one or more aggregator modules.

Example:
```
DEFINITION MODULE ProjectLibraries;
(* Generate project libraries from templates *)
GENLIB EmployeeID FROM IdNumbers FOR Length="10"; CheckSum="TRUE" END;
GENLIB Employee FROM EmployeeRecords FOR RecordKey="EmployeeID" END;
GENLIB EmployeeTable FROM AssocArrays FOR ValueType="Employee" END;
GENLIB EmployeeSet FROM DynamicSets FOR ValueType="Employee" END;
(* Import to re-export *)
IMPORT EmployeeID+, Employee+, EmployeeTable+, EmployeeSet+;
END ProjectLibraries.
```

13.5 Recording the Template, Date and Time

Pragma `GENERATED` is inserted into the output after the line feed following the first semicolon in the library.

13.6 Standard Library Templates

The standard library provides a set of generic templates for commonly used abstract data types. By convention, the names of library templates are always in the plural, while library names are in the singular. A list of library templates and their brief descriptions can be found in section 14.15.

L Standard Library (TO DO: Revise and Merge with Section 14)

L.1 Library Defined Blueprints

The standard library provides a set of blueprint definitions to allow the construction of library defined abstract data types with the same semantics as predefined types and transparent data types defined using type constructor syntax. To require an ADT to conform to a blueprint, the library that defines the ADT must specify the blueprint identifier in the module header of its definition part.

In order to ensure the semantic compatibility of library defined types with built-in counterparts as well as the integrity of the standard library itself, all standard library blueprint definitions are immutable and their immutability is compiler enforced. Any attempt to use a standard library blueprint that has been modified shall cause a compilation error.

The standard library provides three kinds of blueprints:

- blueprints to construct numeric ADTs
- blueprints to construct collection ADTs
- blueprints to construct date-time ADTs

L.1.1 Blueprints to Construct Numeric ADTs

The standard library provides a hierarchical set of blueprints for the construction of numeric types.

L.1.1.1 Blueprint ProtoNumeric

The standard library provides numeric root blueprint `ProtoNumeric` for the construction of numeric blueprints. It defines a subset of properties common to all numeric types.

L.1.1.2 Blueprint ProtoScalar

The standard library provides numeric blueprint `ProtoScalar` derived from `ProtoNumeric` for the construction of numeric scalar blueprints. It defines a subset of properties common to all numeric scalar types.

L.1.1.3 Blueprint ProtoNonScalar

The standard library provides numeric blueprint `ProtoNonScalar` derived from `ProtoNumeric` for the construction of non-scalar blueprints. It defines a subset of properties common to all numeric non-scalar types.

L.1.1.4 Blueprint ProtoCardinal

The standard library provides derived numeric scalar blueprint `ProtoCardinal` for the construction of library-defined unsigned whole number types. For semantic compatibility, its definition matches the semantics of the built-in types `CARDINAL` and `LONGCARD`.

L.1.1.5 Blueprint ProtoInteger

The standard library provides derived numeric scalar blueprint `ProtoInteger` for the construction of library-defined signed whole number types. For semantic compatibility, its definition matches the semantics of the built-in types `INTEGER` and `LONGINT`.

L.1.1.6 Blueprint ProtoReal

The standard library provides derived numeric scalar blueprint `ProtoReal` for the construction of library-defined real number types. For semantic compatibility, its definition matches the semantics of the built-in types `REAL` and `LONGREAL`.

L.1.1.7 Blueprint ProtoComplex

The standard library provides derived numeric non-scalar blueprint `ProtoComplex` for the construction of library-defined complex number types. Its definition is modeled on the mathematical definition of complex numbers.

L.1.1.8 Blueprint ProtoVector

The standard library provides derived numeric non-scalar blueprint `ProtoVector` for the construction of library-defined numeric vector types. Its definition is modeled on the mathematical definition of numeric vectors.

L.1.1.9 Blueprint ProtoTuple

The standard library provides derived numeric non-scalar blueprint `ProtoTuple` for the construction of library-defined numeric tuple types. Its definition is modeled on the mathematical definition of numeric tuples.

L.1.1.10 Blueprint ProtoRealArray

The standard library provides derived numeric non-scalar blueprint `ProtoRealArray` for the construction of library-defined numeric array types with real number components.

L.1.1.11 Blueprint ProtoComplexArray

The standard library provides derived numeric non-scalar blueprint `ProtoComplexArray` for the construction of library-defined numeric array types with complex number components.

L.1.2 Collection Blueprints

The standard library provides a hierarchical set of blueprints for the construction of collection types.

L.1.2.1 Blueprint ProtoCollection

The standard library provides collection root blueprint `ProtoCollection` for the construction of collection blueprints. It defines a subset of properties common to all collection types.

L.1.2.2 Blueprint ProtoStaticSet

The standard library provides derived collection blueprint `ProtoStaticSet` for the construction of library-defined static ordered set types. For semantic compatibility, its definition matches the built-in types `BITSET` and `LONGBITSET`.

L.1.2.3 Blueprint ProtoStaticArray

The standard library provides derived collection blueprint `ProtoStaticArray` for the construction of library-defined static array types. For semantic compatibility, its definition matches the semantics of arrays created with the built-in `ARRAY OF` type constructor.

L.1.2.4 Blueprint ProtoStaticString

The standard library provides derived collection blueprint `ProtoStaticString` for the construction of library-defined static string types. For semantic compatibility, its definition matches the semantics of character arrays created with the built-in `ARRAY OF CHAR` type constructor.

L.1.2.5 Blueprint ProtoSet

The standard library provides derived collection blueprint `ProtoSet` for the construction of library-defined dynamic unordered set types. Its definition is modeled on the mathematical definition of sets.

L.1.2.6 Blueprint ProtoOrderedSet

The standard library provides derived collection blueprint `ProtoOrderedSet` for the construction of library-defined dynamic ordered set types. It is derived from blueprint `ProtoSet`.

L.1.2.7 Blueprint ProtoArray

The standard library provides derived collection blueprint `ProtoArray` for the construction of library-defined dynamic array types with numeric or ordinal indices.

L.1.2.8 Blueprint ProtoString

The standard library provides derived collection blueprint `ProtoString` for the construction of library-defined dynamic character string types.

L.1.2.9 Blueprint ProtoDictionary

The standard library provides derived collection blueprint `ProtoDictionary` for the construction of library-defined dynamic unordered associative array types such as hash tables.

L.1.2.10 Blueprint ProtoOrderedDict

The standard library provides derived collection blueprint `ProtoOrderedDict` for the construction of library-defined dynamic ordered associative array types.

L.1.3 Date-Time Blueprints

L.1.3.1 Blueprint ProtoDateTime

The standard library provides blueprint `ProtoDateTime` for the construction of library defined static date-time types.

L.1.3.2 Blueprint ProtoInterval

The standard library provides blueprint `ProtoInterval` for the construction of library defined static date-time interval types.

L.1.4 User Defined Blueprints

User libraries may provide their own blueprint definitions for their own custom designed abstract data types. User defined blueprints may be derived from standard library blueprints.

L.2 Abstract Data Types

Opaque pointer types and opaque record types are predominantly intended to define abstract data types or ADTs. An ADT is a data type whose internal structure and semantics are hidden from the user of the type and have their semantics defined by the library module that defines the ADT.

A library module that defines an abstract data type with the same name as its own module identifier is called an ADT library module.

Example:
```
DEFINITION MODULE BCD; (* ADT Library Module *)
TYPE BCD = OPAQUE; (* type identifier same as module *)
```

An ADT library module may specify a blueprint in its module header. This represents a promise to conform to the common set of semantics defined by the blueprint. An ADT defined to conform to a blueprint must bind its own library defined procedures to those operators and predefined procedures that are required by the blueprint. Static conformance is compiler enforced. No other bindings than those defined by the blueprint are permitted except for bindings to the conversion operator.

Example:
```
DEFINITION MODULE BCD [ProtoReal]; (* must conform to blueprint ProtoReal *)
TYPE BCD = OPAQUE RECORD value : ARRAY 8 OF OCTET END;
(* define bindings to built-ins and operators required by ProtoReal *)
PROCEDURE [VAL] fromSXF( VAR bcd : BCD; CONST sxf : ARRAY OF OCTET );
PROCEDURE [+] add ( a, b : BCD ) : BCD;
...
(* define IO procedures for use by READ, WRITE and WRITEF *)
PROCEDURE [WRITE] Write ( f : File; b : BCD );
...
END BCD.
```

Defining an ADT with a blueprint specified and providing appropriate bindings in the public interface of the ADT will cause the compiler to check static conformance of the public interface with the specified blueprint's definition. If conformant, this will have the following effects:

- Literals defined to be compatible with the ADT may be assigned to variables of the ADT or passed-in as arguments for formal parameters of the ADT.
- ADT values may be used in infix expressions using the ADT's bound operators.
- Bound-to predefined procedures may be called with ADT values passed as arguments.
- The compiler will replace any infix expressions with calls to the corresponding procedures defined in the ADT library module.
- The compiler will replace any invocations of bound-to predefined procedures with calls to the corresponding procedures defined in the ADT library module.

Example:
```
IMPORT BCD;
VAR a, b, sum : BCD;
a := 1.5; (* via intermediate conversion: 1.5 -> SXF -> BCD *)
b := 2.75; (* via intermediate conversion: 2.75 -> SXF -> BCD *)
sum := a + b; (* replaced by sum := BCD.add(a, b); *)
WRITE(sum); (* replaced by BCD.Write(stdOut, sum); *)
```

L.3 Library Defined ADTs Using Blueprints and Bindings

The standard library provides a rich set of `ALIAS` types and library defined ADTs using bindings and are practically indistinguishable from predefined types and transparent data types defined using type constructor syntax.

L.3.1 Standard Library Defined Bitset Types

The standard library defines a set of `ALIAS` types and ADT implementations of bitset types. The `ALIAS` types are provided for public use while the ADTs represent a private implementation layer intended for internal use by the standard library only.

L.3.1.1 Alias Types for Bitset Types

The standard library defines a set of `ALIAS` types of bitset types of different bit widths. Their identifiers are `BITSET16`, `BITSET32`, `BITSET64` and `BITSET128`, indicating the bit widths of their respective implementations. The aliases are provided in module `Bitsets`.

Example:
```
IMPORT Bitsets;
VAR set, union : BITSET16;
set := { 0, 7, 15 };  union := set + { 1, 2, 4 };  set[7] := FALSE;
```


Module `Bitsets` provides `ALIAS` types `SHORTBITSET` , `BITSET` , `LONGBITSET` and `LONGLONGBITSET`. Their bit widths are dependent on the bit widths of cardinal types:

```
TSIZE(SHORTBITSET) = TSIZE(SHORTCARD)
TSIZE(BITSET) = TSIZE(CARDINAL)
TSIZE(LONGBITSET) = TSIZE(LONGCARD)
TSIZE(LONGLONGBITSET) = TSIZE(LONGLONGCARD)
```

The mappings are defined automatically in the standard library using conditional compilation pragmas. Since the bit widths of the cardinal types are target dependent and implementation defined, the mapping of `ALIAS` types `SHORTBITSET`, `BITSET`, `LONGBITSET`, and `LONGLONGBITSET` to their respective bitset ADTs are also target dependent and implementation defined.

L.3.1.2 ADT Implementations of Bitset Types

The standard library defines a set of ADT implementations of bitset types of different bit widths. Their identifiers are `BS16`, `BS32`, `BS64` and `BS128`, indicating their respective bit widths. The ADTs conform to standard library defined blueprint `ProtoStaticSet` and their semantics match those of set types declared using the built-in `SET OF` type constructor.

L.3.2 Standard Library Defined Unsigned Integer Types

The standard library defines a set of `ALIAS` types and ADT implementations of unsigned integer types. The `ALIAS` types are provided for public use while the ADTs represent a private implementation layer intended for internal use by the standard library only.

L.3.2.1 Alias Types for Unsigned Integer Types

The standard library defines a set of `ALIAS` types of unsigned integer types.

<table>
<tr><th>Identifier</th><th>Defined as ALIAS OF CARDINAL</th><th>Else as ALIAS OF LONGCARD</th><th>Otherwise as</th></tr>
<tr><td>SHORTCARD</td><td>if the bit width of CARDINAL is 16</td><td>never</td><td>ALIAS OF CARD16</td></tr>
<tr><td>CARDINAL16</td><td rowspan="3">if the bit width indicated in the identifier matches the bit width of predefined type CARDINAL</td><td>never</td><td>ALIAS OF CARD16</td></tr>
<tr><td>CARDINAL32</td><td rowspan="3">if the bit width indicated in the identifier matches the bit width of predefined type LONGCARD</td><td>ALIAS OF CARD32</td></tr>
<tr><td>CARDINAL64</td><td>ALIAS OF CARD64</td></tr>
<tr><td>CARDINAL128</td><td>never</td><td>ALIAS OF CARD128</td></tr>
<tr><td>LONGLONGCARD</td><td>never</td><td>if the bit width of LONGCARD is 128</td><td>ALIAS OF CARD128</td></tr>
</table>

The standard library defines a set of `ALIAS` types of unsigned integer types of different bit widths. Their identifiers are `CARDINAL16`, `CARDINAL32`, `CARDINAL64` and `CARDINAL128`, indicating the bit widths of their respective implementations. The aliases are provided in module `Cardinals`.

Example:
```
IMPORT Cardinals;
VAR a, sum : CARDINAL16;
a := 123;  sum := a + 456;  WRITE(sum)
```

The `ALIAS` type whose bit width matches that of predefined type `CARDINAL` is defined as an alias of `CARDINAL`. The `ALIAS` type whose bit width matches that of predefined type `LONGCARD` is defined as an alias of `LONGCARD`.

The `ALIAS` types whose bit width does not match that of predefined types `CARDINAL` or `LONGCARD` are defined as aliases of the matching standard library defined unsigned integer implementation types `CARD16`, `CARD32`, `CARD64` and `CARD128`.

Module `Cardinals` provides two additional `ALIAS` types `SHORTCARD` and `LONGLONGCARD`. The relationships between bit widths of unsigned integer types is as follows:

```
TSIZE(SHORTCARD) <= TSIZE(CARDINAL)
TSIZE(CARDINAL) < TSIZE(LONGCARD) < TSIZE(LONGLONGCARD)
```

The mappings are defined automatically in the standard library using conditional compilation pragmas. Since the bit widths of predefined types `CARDINAL` and `LONGCARD` are target dependent and implementation defined, which `ALIAS` types map to predefined types and which map to standard library implementations is also target dependent and implementation defined.

L.3.2.2 ADT Implementations of Unsigned Integer Types

The standard library defines a set of ADT implementations of unsigned integer types of different bit widths. Their identifiers are `CARD16`, `CARD32`, `CARD64` and `CARD128`, indicating their respective bit widths. The ADTs conform to standard library defined blueprint `ProtoCardinal` and their semantics match those of predefined types `CARDINAL` and `LONGCARD`.

L.3.3 Standard Library Defined Signed Integer Types

The standard library defines a set of `ALIAS` types and ADT implementations of signed integer types. The `ALIAS` types are provided for public use while the ADTs represent a private implementation layer intended for internal use by the standard library only.

L.3.3.1 Alias Types for Signed Integer Types

The standard library defines a set of `ALIAS` types of signed integer types of different bit widths. Their identifiers are `INTEGER16`, `INTEGER32`, `INTEGER64` and `INTEGER128`, indicating the bit widths of their respective implementations. The aliases are provided in module `Integers`.

Example:

```
IMPORT Integers;
VAR a, sum : INTEGER16;
a := 123;  sum := a - 456;  WRITE(sum)
```

The `ALIAS` type whose bit width matches that of predefined type `INTEGER` is defined as an alias of `INTEGER`. The `ALIAS` type whose bit width matches that of predefined type `LONGINT` is defined as an alias of `LONGINT`.

The `ALIAS` types whose bit width does not match that of predefined types `INTEGER` or `LONGINT` are defined as aliases of the matching standard library defined signed integer implementation types `INT16`, `INT32`, `INT64` and `INT128`.

Module `Integers` provides two additional `ALIAS` types `SHORTINT` and `LONGLONGINT`. The relationships between bit widths of signed integer types is as follows:

```
TSIZE(SHORTINT) <= TSIZE(INTEGER)
TSIZE(INTEGER) < TSIZE(LONGINT) < TSIZE(LONGLONGINT)
```

The mappings are defined automatically in the standard library using conditional compilation pragmas. Since the bit widths of predefined types `INTEGER` and `LONGINT` are target dependent and implementation defined, which `ALIAS` types map to predefined types and which map to standard library implementations is also target dependent and implementation defined.

L.3.3.2 ADT Implementations of Signed Integer Types

The standard library defines a set of ADT implementations of signed integer types of different bit widths. Their identifiers are `INT16`, `INT32`, `INT64` and `INT128`, indicating their respective bit widths. The ADTs conform to standard library defined blueprint `ProtoInteger` and their semantics match those of predefined types `INTEGER` and `LONGINT`.

L.3.4 Standard Library Defined BCD Real Number ADTs

The standard library provides Binary Coded Decimal (BCD) real number ADTs `BCD` and `LONGBCD`, whose semantics match those of the predefined types `REAL` and `LONGREAL`.

Example:
```
IMPORT BCD;
VAR a, amount : BCD;
a := 123.45;  amount := a * 1.05;  WRITE(amount);
```

L.3.5 Standard Library Defined Complex Number ADTs

The standard library provides complex number ADTs `COMPLEX` and `LONGCOMPLEX`, whose semantics conform to blueprint `ProtoComplex`.

Example:
```
IMPORT COMPLEX;
VAR z, zsum : COMPLEX;
z := { 1.23, 4.56 };  zsum := z + { 1.0, 0.5 };  WRITE(zsum);
```

L.3.6 Standard Library Defined Character Set ADTs

The standard library provides a character set ADT `CHARSET`, whose semantics conform to blueprint `ProtoOrderedSet`.

Example:
```
IMPORT CHARSET;
VAR delimiters : CHARSET; counter : CARDINAL;
delimiters := { ":", ",", "." }; counter := 0;
FOR char IN "foo:bar.baz,bam" DO
  IF char IN delimiters THEN counter++ END
END;
```

L.3.7 Standard Library Defined Character String ADTs

The standard library provides two dynamic string ADTs `STRING` and `UNISTRING`, whose semantics conform to blueprint `ProtoString`.

Example:
```
IMPORT STRING;
VAR s : STRING;
NEW(s);  s := "quick brown fox";  WRITE(s);  RELEASE(s);
```

L.3.8 Standard Library Defined DateTime ADTs

The standard library provides ADTs `DateTime` and `Time` that conform to blueprint `ProtoDateTime`.

Example:
```
IMPORT DateTime;
VAR date, diff : DateTime;
date := { 1979, Month.Oct, 31, 0, 0, 0.0 };
diff := date - { 1970, Month.Jan, 1, 0, 0, 0.0 };  WRITE(diff);
```


14 Standard Library

The public repository with the complete definition parts of the standard library is available at:

http://bitbucket.org/trijezdci/m2r10stdlib/src

A list of modules with a brief description for each module is given below.

14.1 Pseudo Modules and Documentation Modules

Pseudo modules provide interfaces to the system or the compiler itself and are therefore built-in. However, the identifiers they provide need to be explicitly imported to be available. Documentation modules are for the sole purpose of documenting built-in features.

There are six mandatory pseudo modules, one optional pseudo module and one documentation module:

`ATOMIC.def`	provides atomic intrinsics
`UNSAFE.def`	access to system dependent resources
`COROUTINE.def`	access to built-in coroutines (Phase II)
`RUNTIME.def`	interface to the Modula-2 runtime system
`COMPILER.def`	interface to the Modula-2 compile-time system
`CONVERSION.def`	interface to intermediate scalar conversion intrinsics
`ASSEMBLER.def`	access to target dependent inline assembler (optional)
`PREDEFINED.def`	documents predefined constants, types and procedures

14.2 IO Modules for Predefined Types

`IOSupport.def`	aggregator module to import the IO modules for all predefined types
`BooleanIO.def`	IO module for type `BOOLEAN`
`CharIO.def`	IO module for type `CHAR`
`UnicharIO.def`	IO module for type `UNICHAR`
`OctetIO.def`	IO module for type `OCTET`
`CardinalIO.def`	IO module for type `CARDINAL`
`LongCardIO.def`	IO module for type `LONGCARD`
`IntegerIO.def`	IO module for type `INTEGER`
`LongIntIO.def`	IO module for type `LONGINT`
`RealIO.def`	IO module for type `REAL`
`LongRealIO.def`	IO module for type `LONGREAL`

14.3 IO Modules for UNSAFE Types

`UnsafeTypeIO.def`	aggregator module to import the IO modules for types `BYTE`, `WORD` and `ADDRESS`.
`ByteIO.def`	IO module for type `BYTE`
`WordIO.def`	IO module for type `WORD`
`AddressIO.def`	IO module for type `ADDRESS`

14.4 Modules Defining Alias Types

`Bitsets.def`	`ALIAS` types for bitsets with guaranteed widths
`Cardinals.def`	`ALIAS` types for unsigned integers with guaranteed widths
`Integers.def`	`ALIAS` types for signed integers with guaranteed widths
`BITSET.def`	`ALIAS` type for bitset with same width as type `CARDINAL`
`LONGBITSET.def`	`ALIAS` type for bitset with same width as type `LONGCARD`
`SHORTBITSET.def`	`ALIAS` type for bitset with smallest width
`LONGLONGBITSET.def`	`ALIAS` type for bitset with largest width
`SHORTCARD.def`	`ALIAS` type for unsigned integers with smallest width
`LONGLONGCARD.def`	`ALIAS` type for unsigned integers with largest width
`SHORTINT.def`	`ALIAS` type for signed integers with smallest width
`LONGLONGINT.def`	`ALIAS` type for signed integers with largest width

14.5 Library Modules Implementing Basic Types

`BS16.def`	16-bit bitset type
`BS32.def`	32-bit bitset type
`BS64.def`	64-bit bitset type
`BS128.def`	128-bit bitset type
`CARD16.def`	16-bit unsigned integer type
`CARD32.def`	32-bit unsigned integer type
`CARD64.def`	64-bit unsigned integer type
`CARD128.def`	128-bit unsigned integer type
`INT16.def`	16-bit signed integer type
`INT32.def`	32-bit signed integer type
`INT64.def`	64-bit signed integer type
`INT128.def`	128-bit signed integer type
`BCD.def`	single precision binary coded decimals
`LONGBCD.def`	double precision binary coded decimals
`COMPLEX.def`	single precision complex number type
`LONGCOMPLEX.def`	double precision complex number type
`CHARSET.def`	character set type
`STRING.def`	dynamic ASCII strings
`UNISTRING.def`	dynamic unicode strings

14.6 Modules Providing Math for Basic Types

`CardinalMath.def`	mathematic functions for type `CARDINAL`
`LongCardMath.def`	mathematic functions for type `LONGCARD`
`IntegerMath.def`	mathematic functions for type `INTEGER`
`LongIntMath.def`	mathematic functions for type `LONGINT`
`RealMath.def`	mathematic constants and functions for type `REAL`
`LongRealMath.def`	mathematic constants and functions for type `LONGREAL`
`BCDMath.def`	mathematic constants and functions for type `BCD`
`LongBCDMath.def`	mathematic constants and functions for type `LONGBCD`
`ComplexMath.def`	mathematic constants and functions for type `COMPLEX`
`LongComplexMath.def`	mathematic constants and functions for type `LONGCOMPLEX`

14.7 Memory Management Modules

`Storage.def`	dynamic memory allocator

14.8 Modules for Exception Handling and Termination

`Exceptions.def`	exception handling
`Termination.def`	termination handling

14.9 File System Modules

`Filesystem.def`	file system operations using absolute paths
`DefaultDir.def`	file system operations relative to a working directory
`Pathnames.def`	operating system independent pathname operations

14.10 File IO Modules

`FileIO.def`	file oriented input and output
`TextIO.def`	line oriented input and output
`RegexIO.def`	regular expression based input and output
`Scanner.def`	primitives for scanning text files
`Terminal.def`	terminal based input and output

14.11 Modules Providing Primitives for Text Handling

`ASCII.def`	mnemonics and macro-functions for ASCII characters

`Regex.def`	Modula-2 regular expression library
`RegexConv.def`	conversion library for regular expression syntax

14.12 Modules for Date and Time Handling

`TZ.def`	time zone offsets and abbreviations
`Time.def`	compound time with day, hour, minute, sec/msec components
`DateTime.def`	compound calendar date and time
`TimeUnits.def`	date and time base units
`SysClock.def`	interface to the system clock

14.13 Miscellaneous Modules

`Hashes.def`	selected hash functions
`LexParams.def`	constants with lexical parameters of the compiler

14.14 Blueprint Library

Blueprints which specify common semantics that libraries may be required to conform to:

`ProtoNumeric.def`	common to all numeric blueprints
`ProtoScalar.def`	common to all numeric scalar blueprints
`ProtoNonScalar.def`	common to all numeric non-scalar blueprints
`ProtoCardinal.def`	for unsigned whole number ADTs
`ProtoInteger.def`	for signed whole number ADTs
`ProtoReal.def`	for real number ADTs
`ProtoComplex.def`	for complex number ADTs
`ProtoVector.def`	for numeric vector ADTs
`ProtoTuple.def`	for numeric tuple ADTs
`ProtoCollection.def`	common to all collection blueprints
`ProtoStaticSet.def`	for static set ADTs
`ProtoStaticArray.def`	for static array ADTs
`ProtoStaticString.def`	for static character string ADTs
`ProtoSet.def`	for dynamic set ADTs
`ProtoArray.def`	for dynamic array ADTs
`ProtoString.def`	for dynamic character string ADTs
`ProtoDictionary.def`	for dynamic associative array ADTs
`ProtoDateTime.def`	for date-time ADTs
`ProtoInterval.def`	for date-time interval ADTs
`ProtoIO.def`	for IO modules
`ProtoWholeMath.def`	for whole number math modules
`ProtoRealMath.def`	for real number math modules

14.15 Template Library

`AssocArrays.def`	generic associative array template
`DynamicArrays.def`	generic dynamic array template
`StaticArrays.def`	generic static array template
`DynamicSets.def`	generic dynamic set template
`StaticSets.def`	generic static set template
`Stacks.def`	generic stack template
`Queues.def`	generic queue template
`DEQs.def`	generic double ended queue template
`PriorityQueues.def`	generic priority queue template
`AATrees.def`	generic AA tree template
`SplayTrees.def`	generic Splay tree template
`PatriciaTries.def`	generic Patricia trie template

Appendix A: Grammar in EBNF

A.1 Non-Terminal Symbols

Compilation Units

#1 Compilation Unit

```
compilationUnit :
    IMPLEMENTATION? programModule | definitionModule | blueprint ;
```

#2 Program Module

```
programModule :
    MODULE moduleIdent ";" importList* block moduleIdent "." ;
```

#2.1 Module Identifier

```
moduleIdent : Ident ;
```

#3 Definition Module

```
definitionModule :
    DEFINITION MODULE moduleIdent
    ( "[" blueprintToObey "]" )? ( FOR typeToExtend )? ";"
    importList* definition*
    END moduleIdent "." ;
```

#3.1 Blueprint to Obey

```
blueprintToObey : blueprintIdent ;
```

#3.2 Blueprint Identifier

```
blueprintIdent : Ident ;
```

#3.3 Type to Extend

```
typeToExtend : Ident ;
```

#4 Blueprint

```
blueprint :
    BLUEPRINT blueprintIdent
    ( "[" blueprintToRefine "]" )? ( FOR blueprintForTypeToExtend )? ";"
    ( REFERENTIAL identList ";" )? moduleTypeRequirementOrImpediment ";"
    ( requiredConstant ";" )* ( reqProcedureOrProcType ";" )*
    END blueprintIdent "." ;
```

#4.1 Blueprint to Refine

```
blueprintToRefine : blueprintIdent ;
```

#4.2 Blueprint for Type to Extend

```
blueprintForTypeToExtend : blueprintIdent ;
```

#5 Module Type Requirement or Impediment

```
moduleTypeRequirementOrImpediment:
   TYPE "=" ( permittedTypeDefinition ( "|" permittedTypeDefinition )*
    ( ":=" protoliteral ( "|" protoliteral )* )? ) | NIL | "*" ;
```

#5.1 Proto-Literal

```
protoliteral :
    simpleProtoliteral | structuredProtoliteral ;
```

#5.2 Simple Proto-Literal

```
simpleProtoliteral : Ident ;
```

#6 Structured Proto-Literal

```
structuredProtoliteral :
    "{" ( Ident ( "," identList | BY repeatFactor ) |
         "{" identList "}" BY repeatFactor ) "}" ;
```

#6.1 Repeat Factor

```
repeatFactor : Ident | "*" ;
```


#7 Required Constant

```
requiredConstant :
    CONST ( "[" constBindableProperty "]" )? Ident
    ( ":" ( range OF )? predefOrRefTypeIdent | "=" constExpression ) ;
```

#7.1 Constant-Bindable Property

```
constBindableProperty :
    ":=" | DESCENDING |
    constBindableIdent ;
```

#7.2 Constant-Bindable Identifier

```
constBindableIdent[1] : Ident ;
```

#7.3 Predefined or Referential Type Identifier

```
predefOrRefTypeIdent : Ident ;
```

#7.4 Constant Expression

```
constExpression : expression ;
```

#8 Permitted Type Definition

```
permittedTypeDefinition :
    RECORD | OPAQUE RECORD?
```

#9 Required Procedure Or Procedure Type

```
reqProcedureOrProcType :
    procedureHeader | TYPE Ident "=" procedureType ;
```

Import Lists, Blocks, Definitions and Declarations

#10 Import List

```
importList :
    ( libGenDirective | importDirective ) ";" ;
```

#11 Library Generation Directive

```
libGenDirective :
    GENLIB libIdent FROM template FOR templateParamList END ;
```

#11.1 Library Identifier

```
libIdent : Ident ;
```

#11.2 Template Identifier

```
template : Ident ;
```

#12 Template Parameter List

```
templateParamList :
    placeholder "=" replacement ( ";" placeholder "=" replacement )* ;
```

#12.1 Placeholder

```
placeholder : Ident ;
```

#12.2 Replacement

```
replacement : StringLiteral ;
```

#13 Import Directive

```
importDirective :
    IMPORT moduleIdent importMode? ( "," moduleIdent importMode? )* |
    FROM moduleIdent IMPORT ( identList | "*" ) ;
```

#13.1 Import Mode

```
importMode : "+" | "-" ;
```

#14 Block

```
block :
    declaration* ( BEGIN statementSequence )? END ;
```

#15 Definition

```
definition :
    CONST ( Ident "=" constExpression ";" )+ |
    TYPE ( typeDefinitionOrDeclaration ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    restrictedExport? procedureHeader ";" ;
```

#15.1 Restricted Export

```
restrictedExport : "*" ;
```

[1] Constant-bindable identifiers are `TLIMIT`, `TSIGNED`, `TBASE`, `TPRECISION`, `TMINEXP`, and `TMAXEXP`.

#16 Type Definition Or Declaration

```
typeDefinitionOrDeclaration :
    Ident "=" ( OPAQUE recordType? | type ) ;
```

#17 Declaration

```
declaration :
    CONST ( constDeclaration ";" )+ |
    TYPE ( Ident "=" type ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureHeader ";" block Ident ";" ;
```

#18 Constant Declaration

```
constDeclaration :
    Ident "=" constExpression |
    FOR "*" IN enumTypeIdent ;
```

Types

#19 Type

```
type :
    ( ( ALIAS | SET | range ) OF )? typeIdent |
    enumType | arrayType | recordType | pointerType | procedureType ;
```

#19.1 Type Identifier

```
typeIdent : qualident ;
```

#20 Range

```
range :
    "[" ">"? constExpression ".." "<"? constExpression "]" ;
```

#21 Enumeration Type

```
enumType :
    "(" ( "+" enumBaseType "," )? identList ")" ;
```

#21.1 Enumeration Base Type

```
enumBaseType : enumTypeIdent ;
```

#21.2 Enumeration Type Identifier

```
enumTypeIdent : typeIdent ;
```

#22 Array Type

```
arrayType :
    ARRAY componentCount ( "," componentCount )* OF typeIdent ;
```

#22.1 Component Count

```
componentCount : constExpression ;
```

#23 Record Type

```
recordType :
    RECORD ( fieldList ( ";" fieldList )* indeterminateField? |
    "(" recBaseType ")" fieldList ( ";" fieldList )* ) END ;
```

#23.1 Field List

```
fieldList : variableDeclaration ;
```

#23.2 Record Base Type

```
recBaseType : recTypeIdent ;
```

#23.3 Record Type Identifier

```
recTypeIdent : typeIdent ;
```

#24 Indeterminate Field

```
indeterminateField :
    INDETERMINATE Ident ":" ARRAY discriminantFieldIdent OF typeIdent ;
```

#24.1 Discriminant Field Identifier

```
discriminantFieldIdent : Ident ;
```


#25 Pointer Type

```
pointerType :
    POINTER TO CONST? typeIdent ;
```

#26 Procedure Type

```
procedureType :
    PROCEDURE ( "(" formalTypeList ")" )? ( ":" returnedType )? ;
```

#26.1 Returned Type

```
returnedType : typeIdent ;
```

#27 Formal Type List

```
formalTypeList :
    formalType ( "," formalType )* ;
```

#27.1 Formal Type

```
formalType :
    attributedFormalType | variadicFormalType ;
```

#28 Attributed Formal Type

```
attributedFormalType :
    ( CONST | VAR )? simpleFormalType ;
```

#29 Simple Formal Type

```
simpleFormalType :
    CAST? ( ARRAY OF )? typeIdent ;
```

#30 Variadic Formal Type

```
variadicFormalType :
    ARGLIST numberOfArgumentsToPass? OF
    ( attributedFormalType | "{" attributedFormalTypeList "}" )
    ( "|" variadicTerminator )? ;
```

#30.1 Attributed Formal Type List

```
attributedFormalTypeList :
    attributedFormalType ( "," attributedFormalType )* ;
```

Variables

#31 Variable Declaration

```
variableDeclaration :
    identList ":" ( range OF )? typeIdent ;
```

Procedures

#32 Procedure Header

```
procedureHeader :
    PROCEDURE ( "[" procBindableEntity "]" )? Ident
    ( "(" formalParamList ")" )? ( ":" returnedType )? ;
```

#32.1 Procedure-Bindable Entity

```
procBindableEntity :
    "+" | "-" | "*" | "/" | "=" | "<" | ">" | "::" | ":=" | ".." |
    DIV | MOD | FOR | IN | procBindableIdent;
```

#32.2 Procedure-Bindable Identifier

```
procBindableIdent[2] : Ident;
```

[2] Bindable-predefined identifiers are ABS, NEG, DUP, COUNT, LENGTH, CONCAT, STORE, INSERT, REMOVE, RETRIEVE, NEW, RETAIN, RELEASE, SUBSET, READ, WRITE, WRITEF, TMIN and TMAX. Bindable-primitives are SXF and VAL.

#33 Formal Parameter List
```
formalParamList :
    formalParams ( ";" formalParams )* ;
```
#33.1 Formal Parameters
```
formalParams :
    simpleFormalParams | variadicFormalParams ;
```
#34 Simple Formal Parameters
```
simpleFormalParams :
    ( CONST | VAR )? identList ":" simpleFormalType ;
```
#35 Variadic Formal Parameters
```
variadicFormalParams :
    ARGLIST numberOfArgumentsToPass? OF
    ( simpleFormalType | "{" simpleFormalParams ( ";" simpleFormalParams )* "}" )
    ( "|" variadicTerminator )? ;
```
#35.1 Number Of Arguments To Pass
```
numberOfArgumentsToPass : constExpression ;
```
#35.2 Variadic Terminator
```
variadicTerminator : constExpression ;
```

Statements

#36 Statement
```
statement :
    ( assignmentOrProcedureCall | ifStatement | caseStatement |
      whileStatement | repeatStatement | loopStatement |
      forStatement | RETURN expression? | EXIT )? ;
```
#37 Statement Sequence
```
statementSequence :
    statement ( ";" statement )* ;
```
#38 Assignment Or Procedure Call
```
assignmentOrProcedureCall :
    designator ( ":=" expression | "++" | "--" | actualParameters )? ;
```
#39 IF Statement
```
ifStatement :
    IF expression THEN statementSequence
    ( ELSIF expression THEN statementSequence )*
    ( ELSE statementSequence )?
    END ;
```
#40 CASE Statement
```
caseStatement :
    CASE expression OF case ( "|" case )+ ( ELSE statementSequence )? END ;
```
#41 Case
```
case :
    caseLabels ( "," caseLabels )* ":" statementSequence ;
```
#42 Case Labels
```
caseLabels :
    constExpression ( ".." constExpression )? ;
```
#43 WHILE Statement
```
whileStatement :
    WHILE expression DO statementSequence END ;
```


#44 REPEAT Statement
```
repeatStatement :
    REPEAT statementSequence UNTIL expression ;
```

#45 LOOP Statement
```
loopStatement :
    LOOP statementSequence END ;
```

#46 FOR Statement
```
forStatement :
    FOR DESCENDING? controlVariable
    IN ( designator | range OF typeIdent )
    DO statementSequence END ;
```

#46.1 Control Variable
```
controlVariable : Ident ;
```

#47 Designator
```
designator :
    qualident designatorTail? ;
```

#48 Designator Tail
```
designatorTail :
    ( ( "[" exprListOrSlice "]" | "^" ) ( "." Ident )* )+ ;
```

Expressions

#49 Expression List Or Slice
```
exprListOrSlice :
    expression ( ( "," expression )+ | ".." expression )?  ;
```

#50 Expression
```
expression :
    simpleExpression ( relOp simpleExpression )? ;
```

#50.1 Relational Operator
```
relOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" | "==" | IN ;
```

#51 Simple Expression
```
simpleExpression :
    ( "+" | "-" )? term ( addOp term )* ;
```

#51.1 Add Operator
```
addOp :
    "+" | "-" | OR ;
```

#52 Term
```
term :
    factorOrNegation ( mulOp factorOrNegation )* ;
```

#52.1 Multiply Operator
```
mulOp :
    "*" | "/" | DIV | MOD | AND ;
```

#53 Factor Or Negation
```
factorOrNegation :
    NOT? factor ;
```

#54 Factor
```
factor :
    simpleFactor  ( "::" typeIdent )? ;
```

#55 Simple Factor

```
simpleFactor :
    NumericLiteral | StringLiteral | structuredValue |
    designatorOrFunctionCall | "(" expression ")" ;
```

#56 Designator Or Function Call

```
designatorOrFunctionCall :
    designator actualParameters? ;
```

#57 Actual Parameters

```
actualParameters :
    "(" expressionList? ")" ;
```

#58 Expression List

```
expressionList :
    expression ( "," expression )* ;
```

Value Constructors

#59 Structured Value

```
structuredValue :
    "{" ( valueComponent ( "," valueComponent )* )? "}" ;
```

#60 Value Component

```
valueComponent :
    expression ( ( BY | ".." ) constExpression )? ;
```

Identifiers

#61 Qualified Identifier

```
qualident :
    Ident ( "." Ident )* ;
```

#62 Identifier List

```
identList :
    Ident ( "," Ident )* ;
```

Optional Language Facilities

Architecture Specific Implementation Module Selection

Replacement For Production #2

```
programModule :
    MODULE moduleIdent ( "(" archSelector ")" )? ";"
    importList* block moduleIdent "." ;
```

Architecture Selector

```
archSelector : Ident ;
```

Register Mapping

Replacement For Production #29

```
simpleFormalType :
    CAST? ( ARRAY OF )? typeIdent regAttribute? ;
```

Register Mapping Attribute

```
regAttribute :
    IN REG ( registerNumber | registerMnemonic ) ;
```

Register Number

```
registerNumber : constExpression ;
```

Register Mnemonic

```
registerMnemonic : qualident ;
```

Symbolic Assembly Inlining

Replacement For Production #36

```
statement :
    ( assignmentOrProcedureCall | ifStatement | caseStatement |
      whileStatement | repeatStatement | loopStatement | forStatement |
      assemblyBlock | RETURN expression? | EXIT )? ;
```

Assembly Block

```
assemblyBlock :
    ASM assemblySourceCode END ;
```

Assembly Source Code

```
assemblySourceCode : <implementation defined syntax> ;
```

A.2 Terminal Symbols

#1 Reserved Words

```
ReservedWord :
    ALIAS | AND | ARGLIST | ARRAY | BEGIN | BLUEPRINT | BY | CASE | CONST |
    DEFINITION | DESCENDING | DIV | DO | ELSE | ELSIF | END | EXIT | FOR | FROM |
    GENLIB | IF | IMPLEMENTATION | IMPORT | IN | INDETERMINATE | LOOP | MOD |
    MODULE | NOT | OF | OPAQUE | OR | POINTER | PROCEDURE | RECORD | REFERENTIAL |
    REPEAT | RETURN | SET | THEN | TO | TYPE | UNTIL | VAR | WHILE ;
```

#2 Identifier

```
Ident :
    IdentLeadChar IdentTail? ;
```

#2.1 Identifier Lead Character

```
IdentLeadChar :
    "_" | "$" | Letter ;
```

#2.2 Identifier Tail

```
IdentTail :
    ( IdentLeadChar | Digit )+ ;
```

#3 Numeric Literal

```
NumericLiteral :
    "0"
       ( RealNumberTail |
         "b" Base2DigitSeq |
         "x" Base16DigitSeq |
         "u" Base16DigitSeq )?
    | "1" .. "9" DecimalNumberTail? ;
```

#3.1 Decimal Number Tail

```
DecimalNumberTail :
    DigitSep? DigitSeq RealNumberTail? | RealNumberTail ;
```

#3.2 Real Number Tail

```
RealNumberTail :
    "." DigitSeq ( "e" ( "+" | "-" )? DigitSeq )? ;
```

#3.3 Digit Sequence

```
DigitSeq :
    Digit+ ( DigitSep Digit+ )* ;
```

#3.4 Base-2 Digit Sequence

```
Base2DigitSeq :
    Base2Digit+ ( DigitSep Base2Digit+ )* ;
```

#3.5 Base-16 Digit Sequence

```
Base16DigitSeq :
    Base16Digit+ ( DigitSep Base16Digit+ )* ;
```

#3.6 Digit

```
Digit :
    Base2Digit | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

#3.7 Base-16 Digit

```
Base16Digit :
    Digit | "A" | "B" | "C" | "D" | "E" | "F" ;
```

#3.8 Base-2 Digit

```
Base2Digit :
    "0" | "1" ;
```

#3.9 Digit Separator

```
DigitSep : "'" ;
```

#4 String Literal
```
StringLiteral :
    SingleQuotedString | DoubleQuotedString ;
```

#4.1 Single Quoted String
```
SingleQuotedString :
    "'" ( QuotableCharacter | '"' )* "'" ;
```

#4.2 Double Quoted String
```
DoubleQuotedString :
    '"' ( QuotableCharacter | "'" )* '"' ;
```

#4.3 Quotable Character
```
QuotableCharacter :
    Digit | Letter | Space | NonAlphaNumQuotable | EscapedCharacter ;
```

#4.4 Letter
```
Letter :
    "A" .. "Z" | "a" .. "z" ;
```

#4.5 Space
```
Space : " " ;
```

#4.6 Non-Alphanumeric Quotable Character
```
NonAlphaNumQuotable :
    "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" | "," |
    "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" |
    "[" | "]" | "^" | "_" | "`" | "{" | "|" | "}" | "~" ;
```

#4.7 Escaped Character
```
EscapedCharacter :
    "\" ( "n" | "t" | "\" ) ;
```

A.3 Ignore Symbols

#1 Whitespace
```
Whitespace :
    Space | ASCII_TAB ;
```

#2 Line Comment
```
LineComment :
    "//" ~( EndOfLine )* EndOfLine ;
```

#3 Block Comment
```
BlockComment :
    "(*" ( ~( "(*" | "*)" )* ( BlockComment | EndOfLine )? )* "*)" ;
```

#4 End Of Line Marker
```
EndOfLine :
    ASCII_LF | ASCII_CR ASCII_LF? ;
```

A.4 Control Codes

#1 Horizontal Tab
```
ASCII_TAB : CHR(8) ;
```

#2 Line Feed
```
ASCII_LF : CHR(10) ;
```

#3 Carriage Return
```
ASCII_CR : CHR(13) ;
```

#4 UTF8 BOM
UTF8_BOM[3] : { 0uFE, 0uBB, 0uBF } ;

[3] BOM support is optional. If supported, a BOM may only occur at the very beginning of a file.

A.5 Pragma Grammar

#1 Pragma

```
pragma :
    “<*” pragmaBody “*>” ;
```

#1.1 Pragma Body

```
pragmaBody :
    pragmaMSG | pragmaIF | procAttrPragma | pragmaPTW | pragmaFORWARD |
    pragmaENCODING | pragmaALIGN | pragmaPADBITS | pragmaPURITY |
    varAttrPragma | pragmaDEPRECATED | pragmaGENERATED | pragmaADDR |
    pragmaFFI | pragmaFFIDENT | implDefinedPragma ;
```

#2 Body Of Compile Time Message Pragma

```
pragmaMSG :
    MSG “=” messageMode “:”
    compileTimeMsgComponent ( “,” compileTimeMsgComponent )* ;
```

#2.1 Message Mode

```
messageMode :
    INFO | WARN | ERROR | FATAL ;
```

#3 Compile Time Message Component

```
compileTimeMsgComponent :
    StringLiteral | ConstQualident |
    “@” ( ALIGN | ENCODING | implDefinedPragmaSymbol ) ;
```

#3.1 Constant Qualified Identifier

```
constantQualident : qualident ;
```

#3.2 Implementation Defined Pragma Symbol

```
implDefinedPragmaSymbol : Ident ;
```

#4 Body Of Conditional Compilation Pragma

```
pragmaIF :
    ( IF | ELSIF ) inPragmaExpression | ELSE | ENDIF ;
```

#5 Body Of Procedure Attribute Pragma

```
procAttrPragma :
    INLINE | NOINLINE | NORETURN ;
```

#6 Body Of Promise-To-Write Pragma

```
pragmaPTW :
    PTW ;
```

#7 Body Of Forward Declaration Pragma

```
pragmaFORWARD :
    FORWARD ( TYPE identList | procedureHeader ) ;
```

#8 Body Of Character Encoding Pragma

```
pragmaENCODING :
    ENCODING “=” ( ”ASCII” | ”UTF8” ) ( “:” codePointSampleList )? ;
```

#9 Code Point Sample List

```
codePointSampleList :
    quotedChar “=” characterCode ( “,” quotedChar “=” characterCode )* ;
```

#9.1 Quoted Character Literal

```
quotedChar : StringLiteral ;
```

#9.2 Character Code Literal

```
characterCode : NumericLiteral ;
```

#10 Body Of Memory Alignment Pragma

```
pragmaALIGN :
   ALIGN “=” inPragmaExpression ;
```

#11 Body Of Bit Padding Pragma

```
pragmaPADBITS :
    PADBITS “=” inPragmaExpression ;
```

#12 Body Of Purity Attribute Pragma

```
pragmaPURITY :
   PURITY “=” inPragmaExpression ;
```

#13 Body Of Variable Attribute Pragma

```
varAttrPragma :
    SINGLEASSIGN | LOWLATENCY | VOLATILE ;
```

#14 Body Of Deprecation Pragma

```
pragmaDEPRECATED :
    DEPRECATED ;
```

#15 Body Of Generation Timestamp Pragma

```
pragmaGENERATED :
    GENERATED FROM template "," datestamp "," timestamp ;
```

#15.1 Date Stamp

```
datestamp :
    year "-" month "-" day ;
```

#15.2 Time Stamp

```
timestamp :
    hours ":" minutes ":" seconds "+" tz ;
```

#15.3 Year, Month, Day, Hours, Minutes, Seconds, Timezone

```
year, month, day, hours, hours, minutes, seconds, tz : wholeNumber ;
```

#16 Body Of Memory Mapping Pragma

```
pragmaADDR :
    ADDR "=" inPragmaExpression ;
```

#17 Body Of Foreign Function Interface Pragma

```
pragmaFFI :
   FFI "=" ( "C" | "Fortran" | "CLR" | "JVM" ) ;
```

#18 Body Of FFI Identifier Mapping Pragma

```
pragmaFFIDENT :
    FFIDENT "=" StringLiteral ;
```

#19 Body Of Implementation Defined Pragma

```
implDefinedPragma :
   implDefinedPragmaSymbol ( "=" inPragmaExpression )? "|" messageMode ;
```

#20 In-Pragma Expression

```
inPragmaExpression :
    inPragmaSimpleExpr ( inPragmaRelOp inPragmaSimpleExpr )? ;
```

#20.1 In-Pragma Relational Operator

```
inPragmaRelOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" ;
```

#21 In-Pragma Simple Expression

```
inPragmaSimpleExpr :
    ( "+" | "-" )? inPragmaTerm ( addOp inPragmaTerm )* ;
```

#22 In-Pragma Term

```
inPragmaTerm :
    inPragmaFactor ( inPragmaMulOp inPragmaFactor )* ;
```

#22.1 In-Pragma Multiply Operator

```
inPragmaMulOp :
    "*" | DIV | MOD | AND ;
```

#23 In-Pragma Factor

```
inPragmaFactor :
    NOT? inPragmaSimpleFactor ;
```

#24 In-Pragma Simple Factor

```
inPragmaSimpleFactor :
   wholeNumber | constQualident |
   "(" inPragmaExpression ")" |
   inPragmaCompileTimeFunctionCall ;
```

#24.1 Whole Number

```
wholeNumber : NumericLiteral ;
```

#25 In-Pragma Compile-Time Function Call

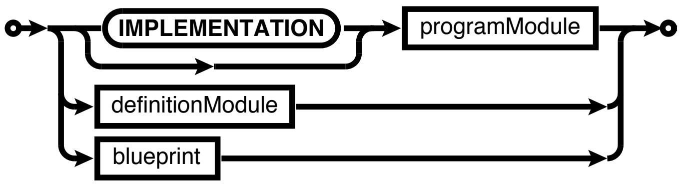
```
inPragmaCompileTimeFunctionCall :
    Ident[1] "(" inPragmaExpression ( "," inPragmaExpression )* ")" ;
```

[1] Permissible are ABS, NEG, ODD, ORD, LENGTH, TMIN, TMAX, TSIZE, TLIMIT and macros from module COMPILER.

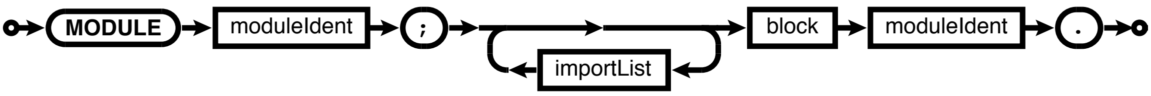
Appendix B: Syntax Diagrams

B.1 Non-Terminal Symbols

#1 Compilation Unit

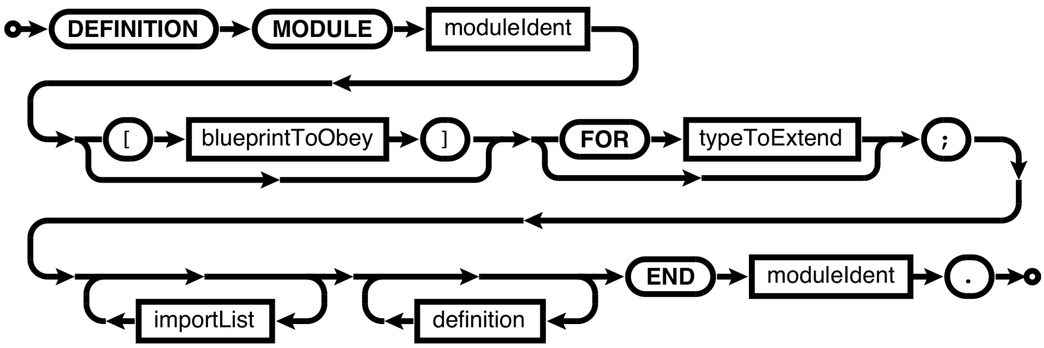


#2 Program Module



#2.1 Module Identifier

#3 Definition Module

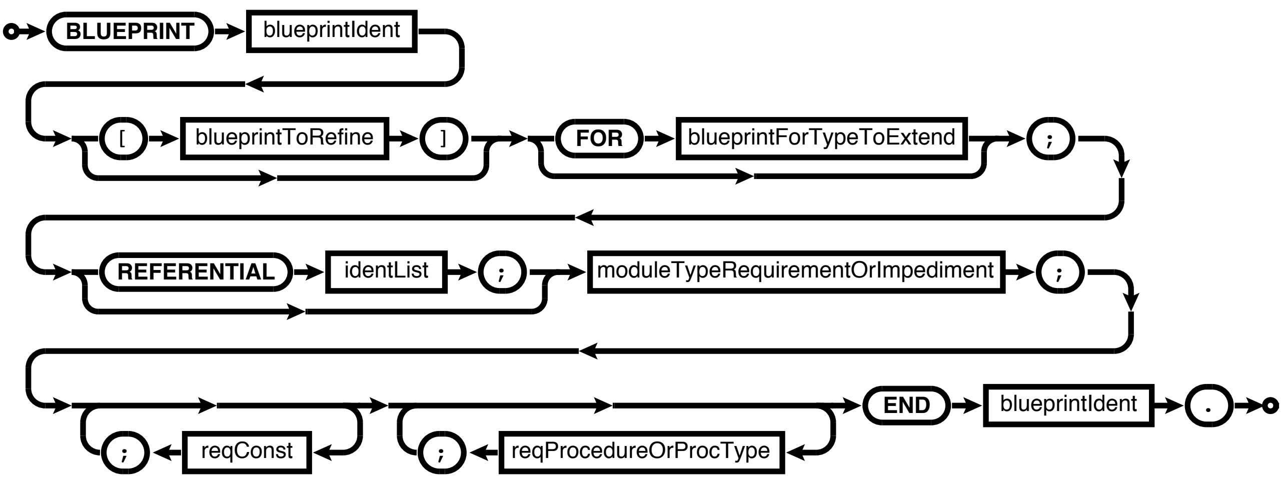


#3.1 Blueprint to Obey

#3.2 Blueprint Identifier

#3.3 Type to Extend

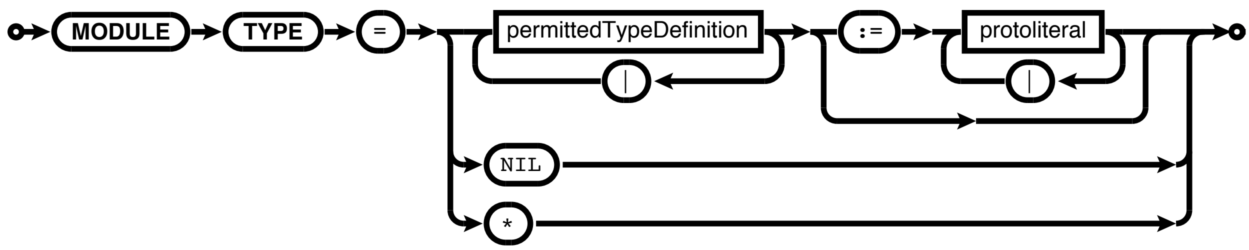
#4 Blueprint



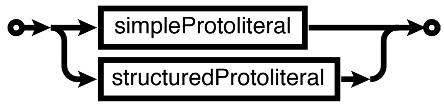
#4.1 Blueprint to Refine

#4.2 Blueprint for Type to Extend

#5 Module Type Requirement or Impediment

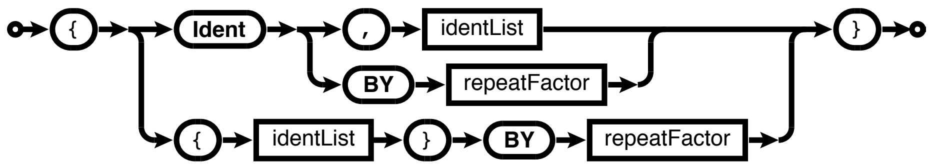


#5.1 Proto-Literal

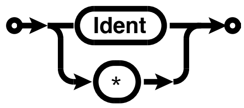


#5.2 Simple Proto-Literal

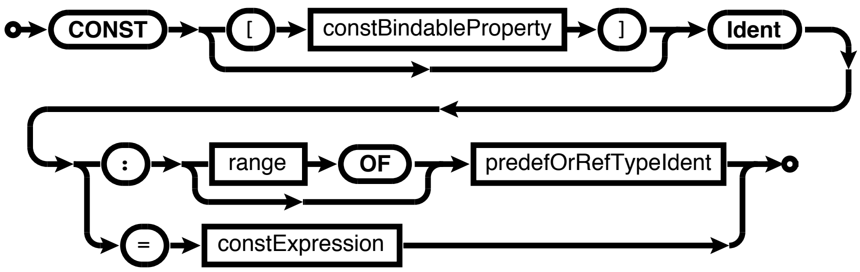
#6 Structured Proto-Literal



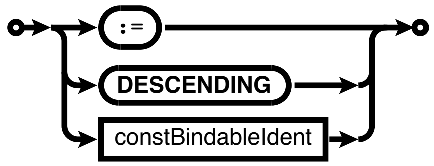
#6.1 Repeat Factor



#7 Required Constant



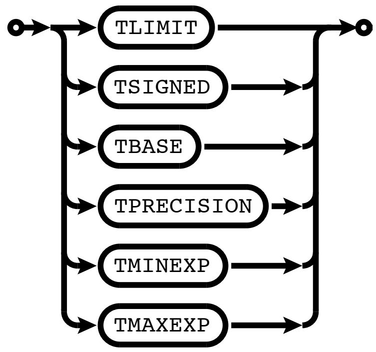
#7.1 Constant-Bindable Property



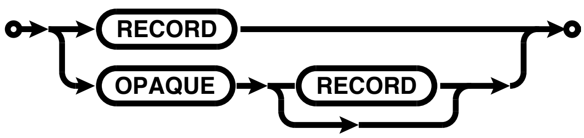
#7.3 Predefined Or Referential Type Identifier

#7.4 Constant Expression

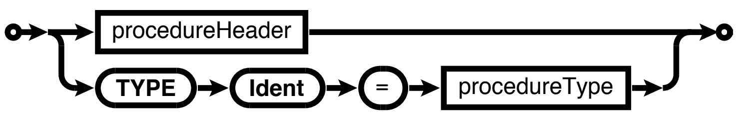
#7.2 Constant-Bindable Identifier



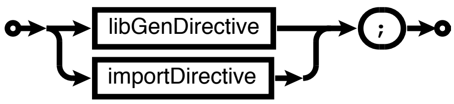
#8 Permitted Type Definition



#9 Required Procedure Or Procedure Type



#10 Import List

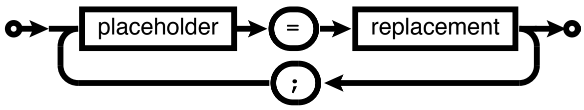


#11 Library Generation Directive

#11.1 Library Identifier

#11.2 Template

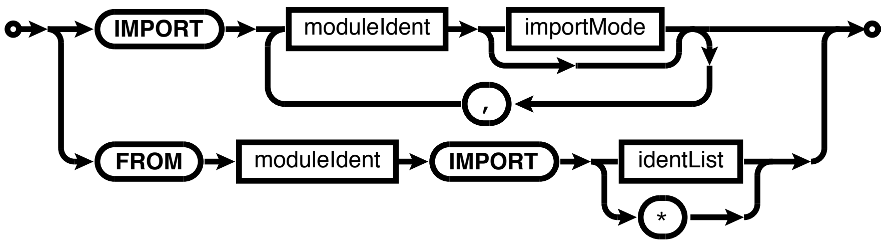
#12 Template Parameter List



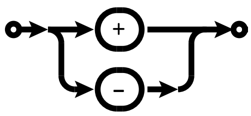
#12.1 Placeholder

#12.2 Replacement

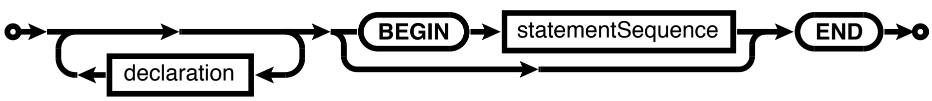
#13 Import Directive



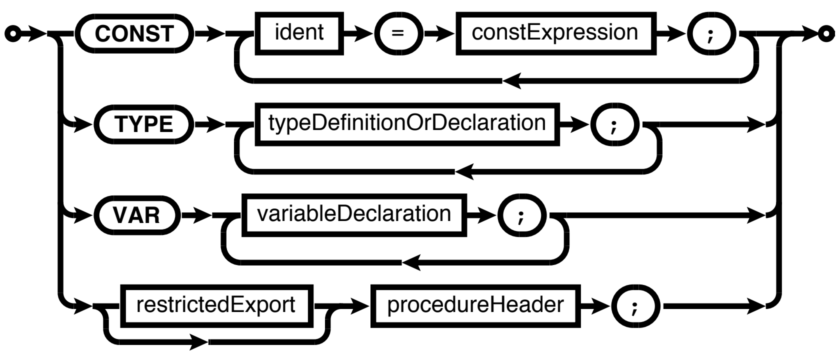
#13.1 Import Mode



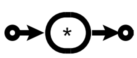
#14 Block



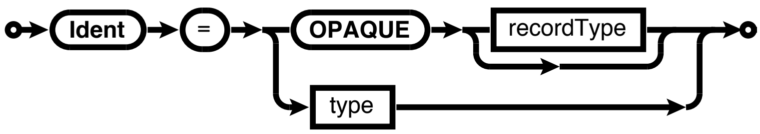
#15 Definition



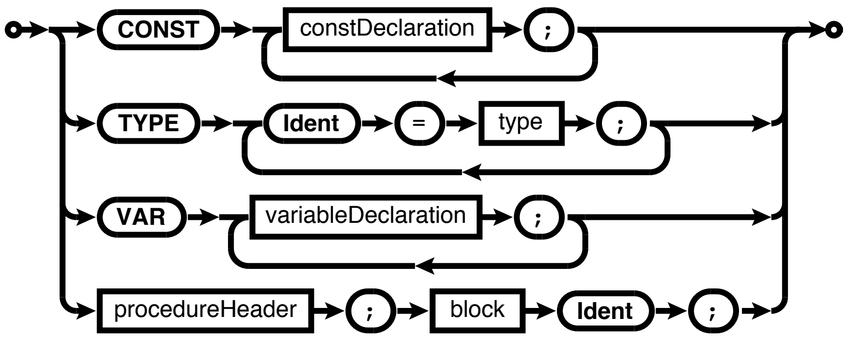
#15.1 Restricted Export



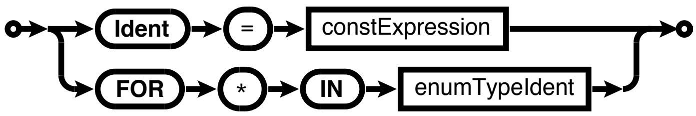
#16 Type Definition Or Declaration



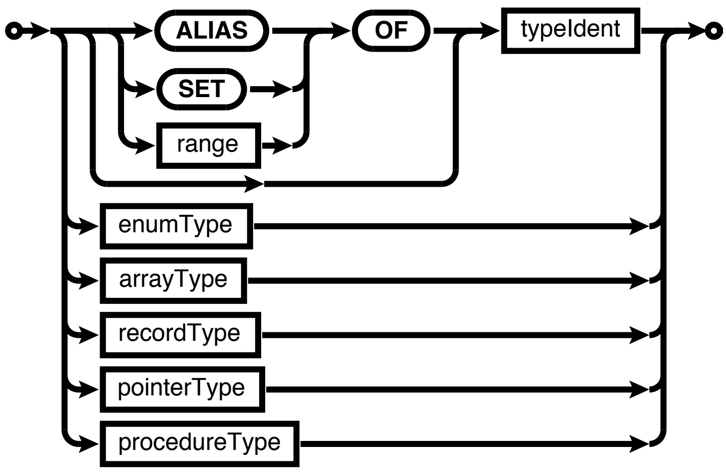
#17 Declaration



#18 Constant Declaration

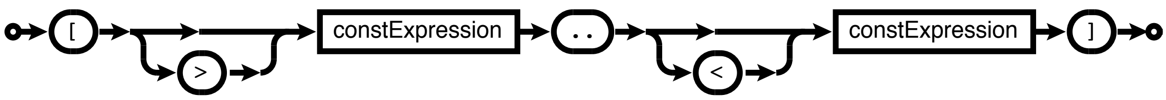


#19 Type

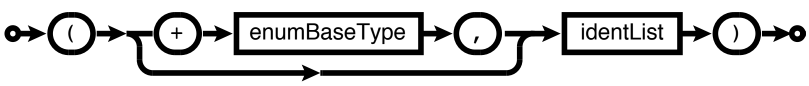


#19.1 Type Identifier

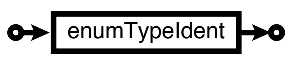
#20 Range



#21 Enumeration Type

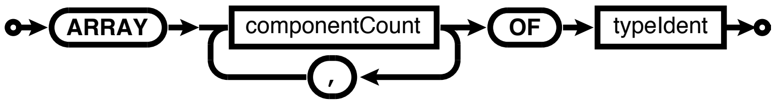


#21.1 Enumeration Base Type



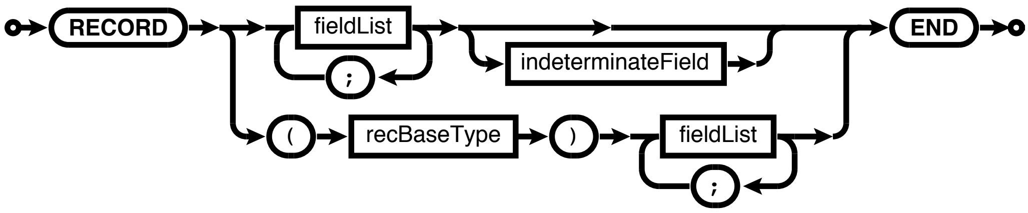
#21.2 Enumeration Type Identifier

#22 Array Type



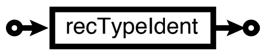
#22.1 Component Count

#23 Record Type



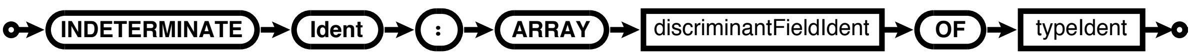
#23.1 Field List

#23.2 Record Base Type



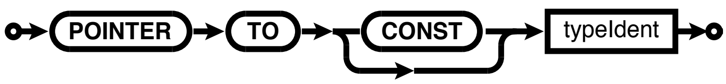
#23.3 Record Type Identifier

#24 Indeterminate Field

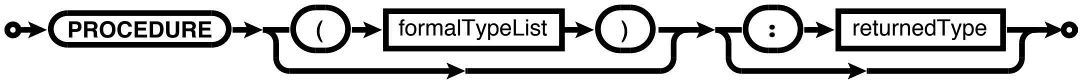


#24.1 Discriminant Field Identifier

#25 Pointer Type

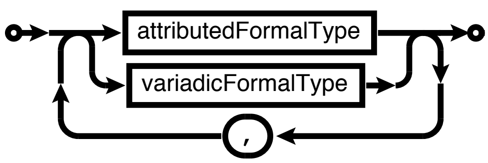


#26 Procedure Type

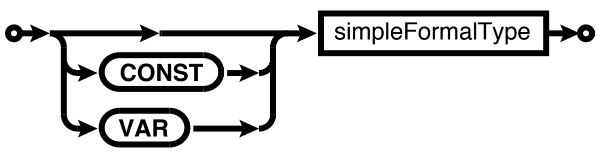


#26.1 Returned Type

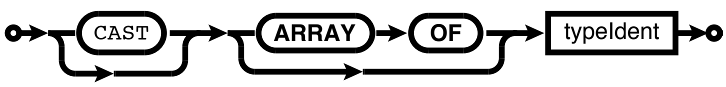
#27 Formal Type List



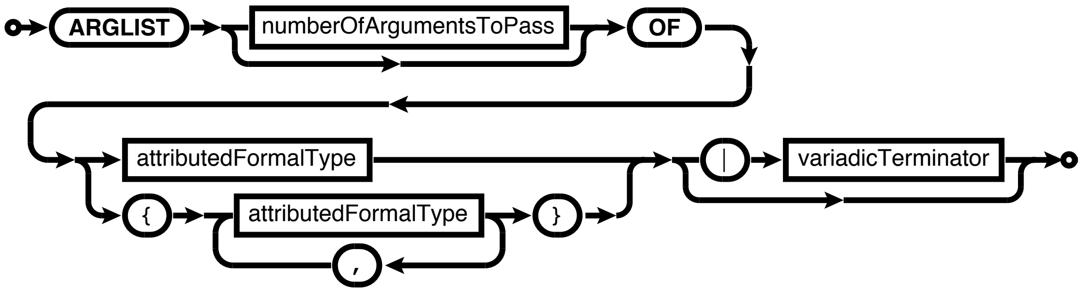
#28 Attributed Formal Type



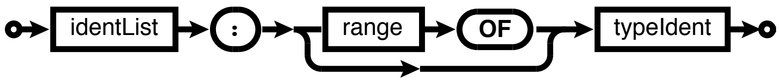
#29 Simple Formal Type



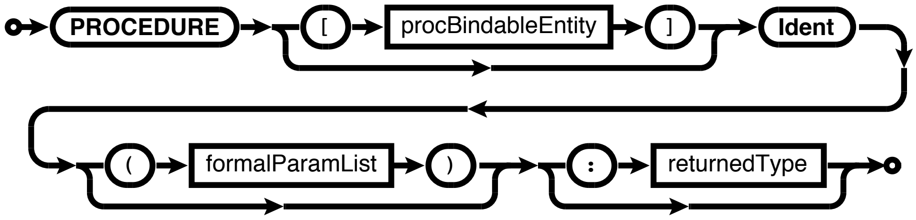
#30 Variadic Formal Type



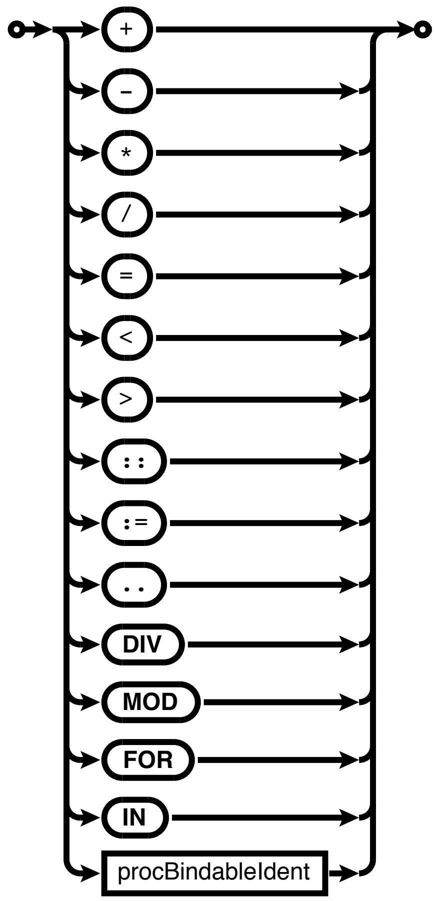
#31 Variable Declaration



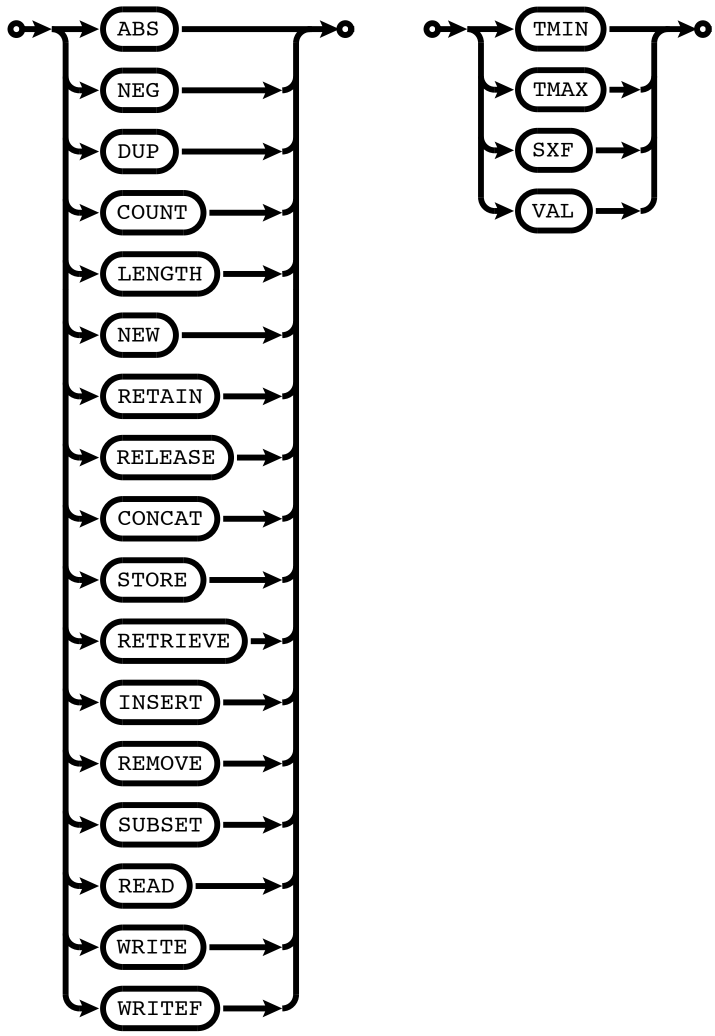
#32 Procedure Header



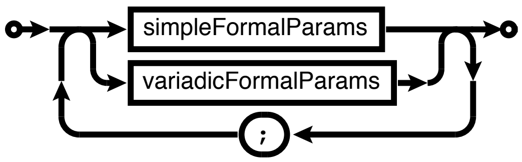
#32.1 Procedure-Bindable Entity



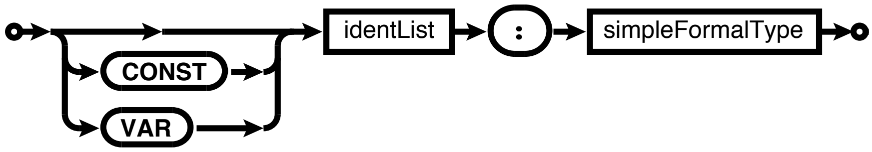
#32.2 Procedure-Bindable Identifier



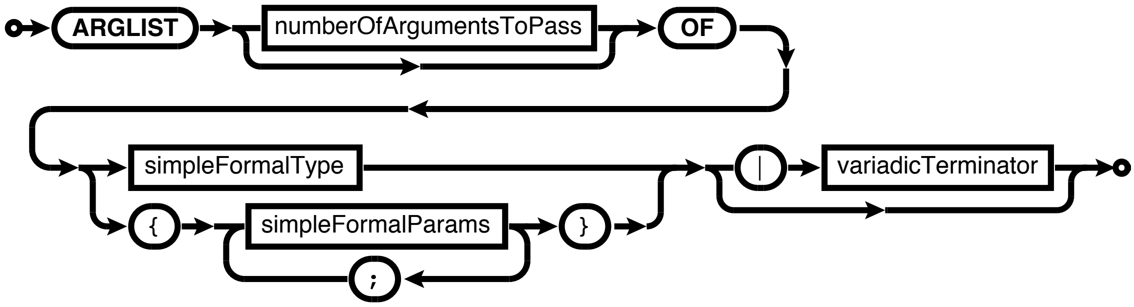
#33 Formal Parameter List



#34 Simple Formal Parameters



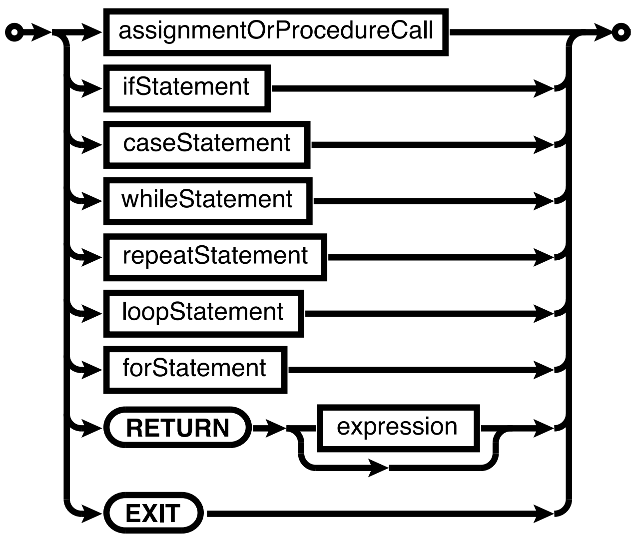
#35 Variadic Formal Parameters



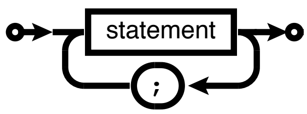
#35.1 Number Of Arguments To Pass

#35.2 Variadic Terminator

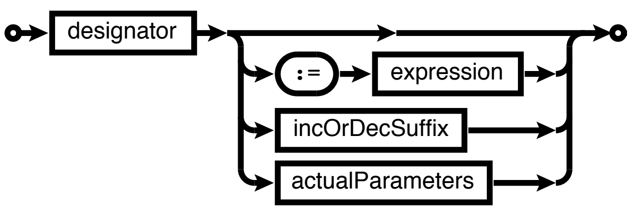
#36 Statement



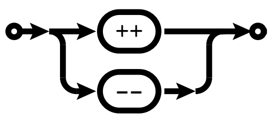
#37 StatementSequence



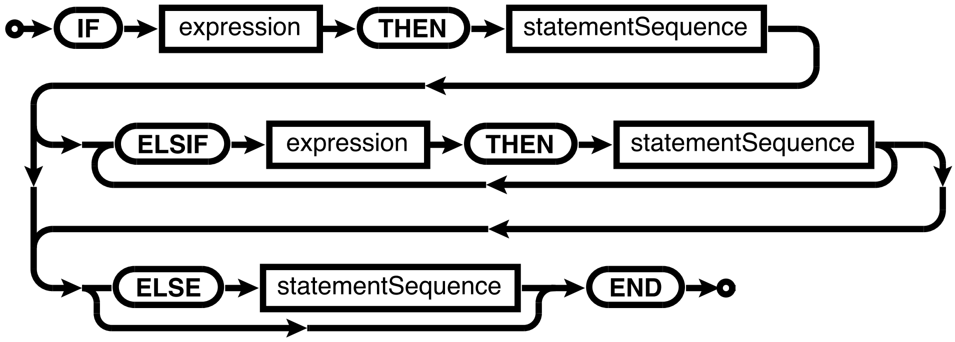
#38 Assignment Or Procedure Call



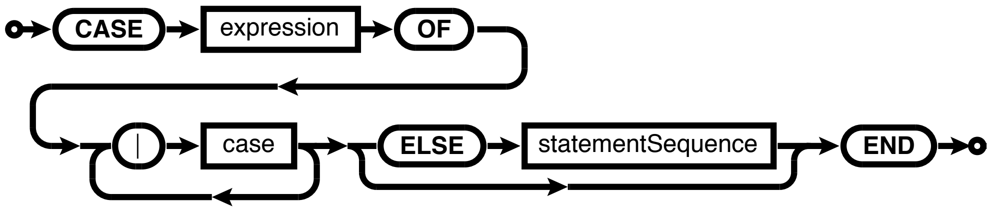
#38.1 Increment Or Decrement Suffix



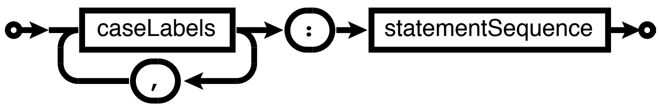
#39 IF Statement



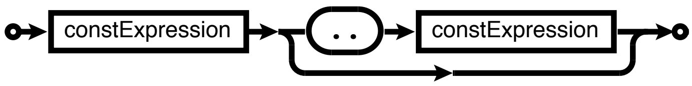
#40 CASE Statement



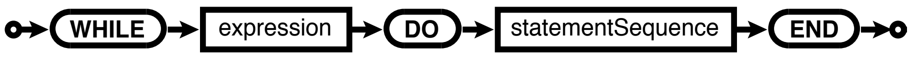
#41 Case



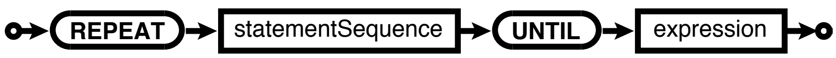
#42 Case Labels



#43 WHILE Statement

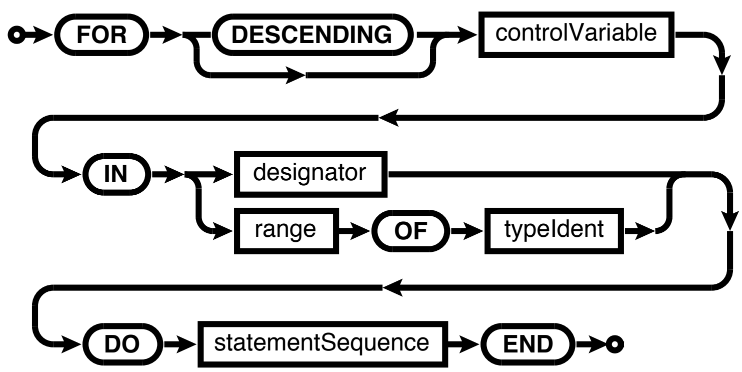


#44 REPEAT Statement



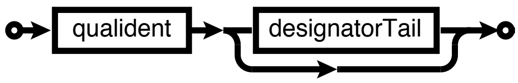
#45 LOOP Statement

#45 FOR Statement

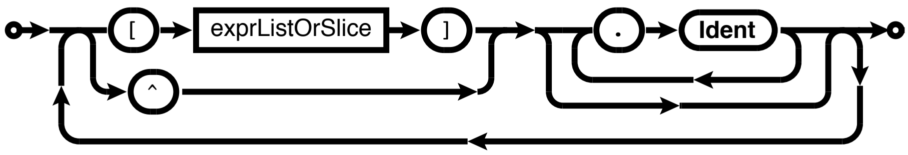


#46.1 Control Variable

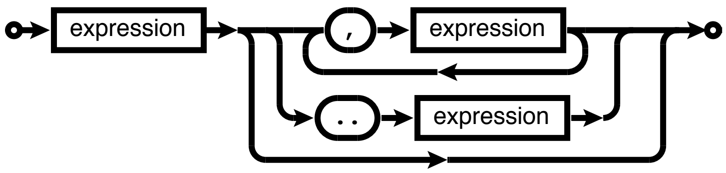
#47 Designator



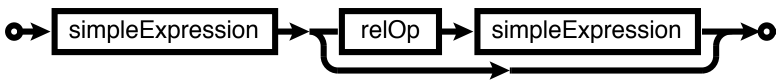
#48 Designator Tail



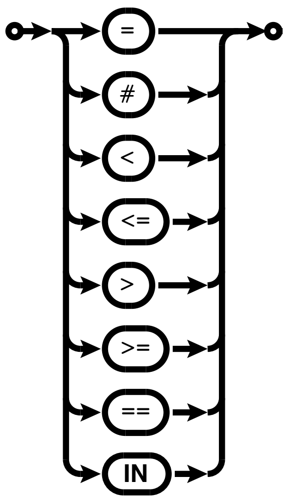
#49 Expression List Or Slice



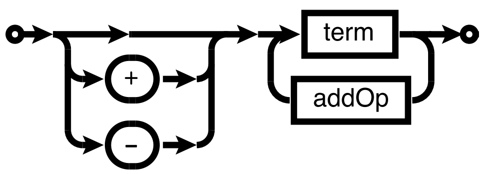
#50 Expression



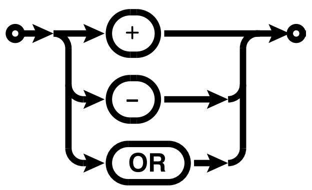
#50.1 Relational Operator



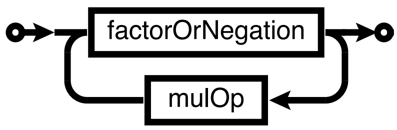
#51 Simple Expression



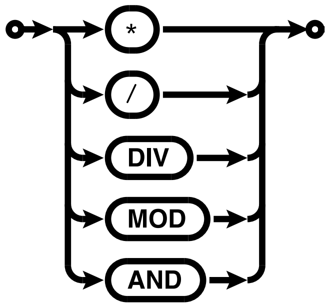
#51.1 Add Operator



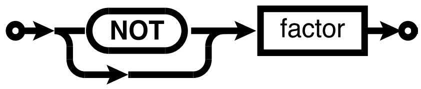
#52 Term



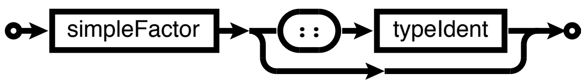
#52.1 Multiply Operator



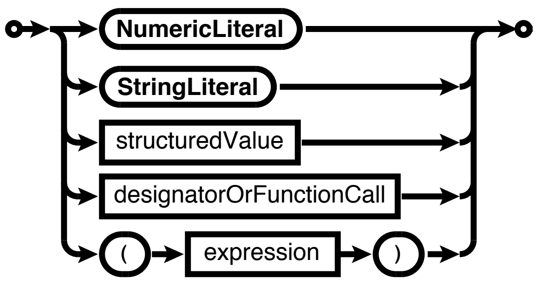
#53 Factor Or Negation



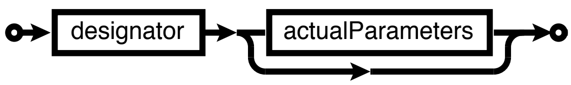
#54 Factor



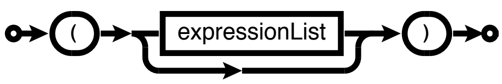
#55 Simple Factor



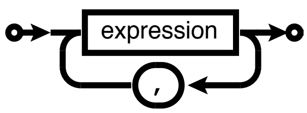
#56 Designator Or Function Call



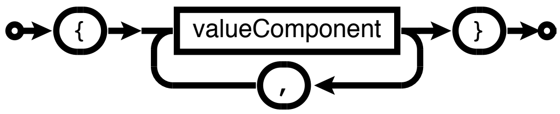
#57 Actual Parameters



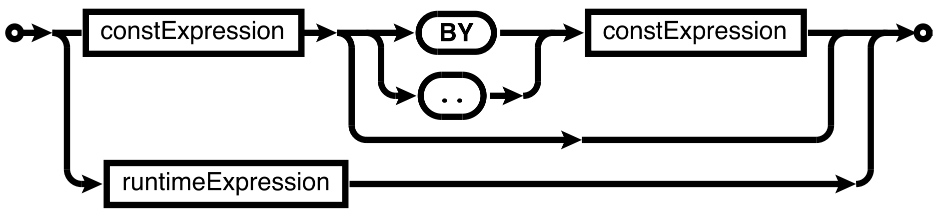
#58 Expression List



#59 Structured Value

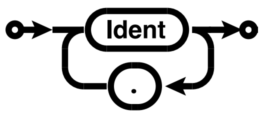


#60 Value Component

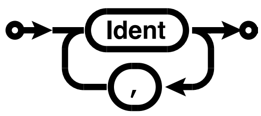


#60.1 Runtime Expression

#61 Qualified Identifier

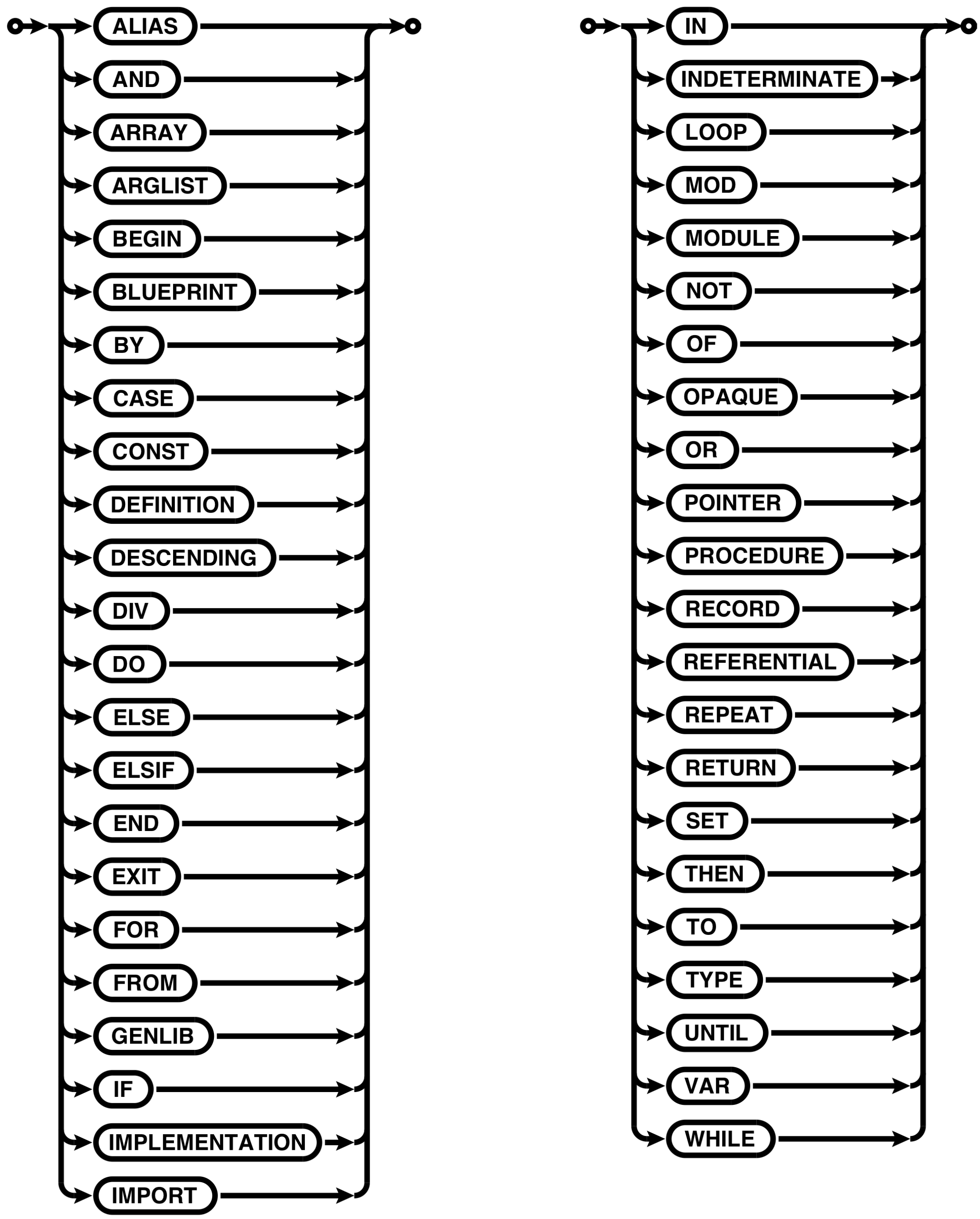


#62 Identifier List

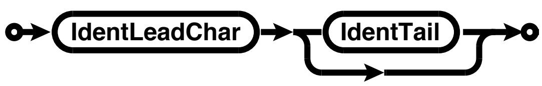


B.2 Terminal Symbols

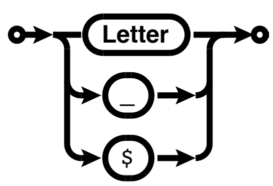
#1 Reserved Words



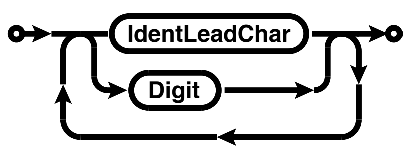
#2 Identifier



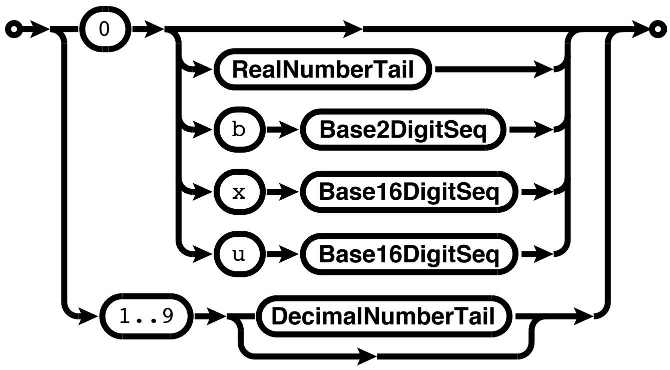
#2.1 Identifier Lead Character



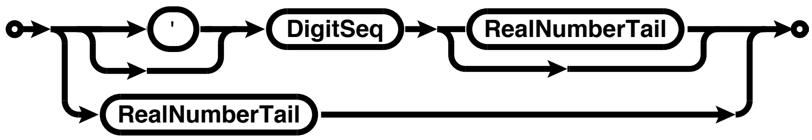
#2.2 Identifier Tail



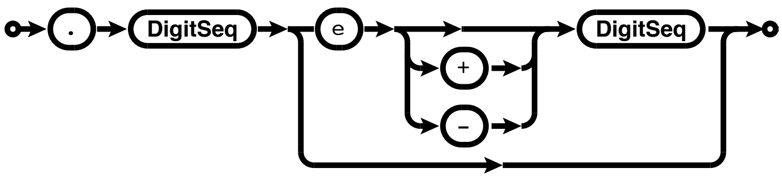
#3 Numeric Literal



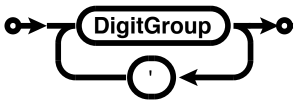
#3.1 Decimal Number Tail



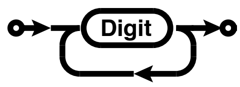
#3.2 Real Number Tail



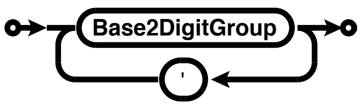
#3.3 Digit Sequence



#3.3b Digit Group

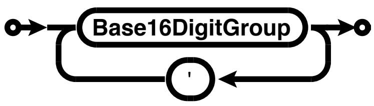


#3.4 Base-2 Digit Sequence

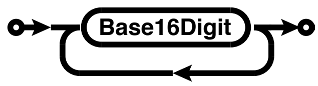


#3.4b Base-2 Digit Group

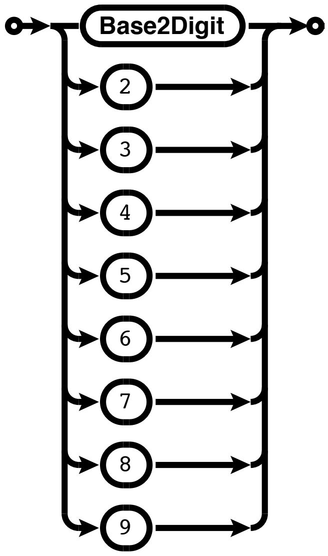
#3.5 Base-16 Digit Sequence



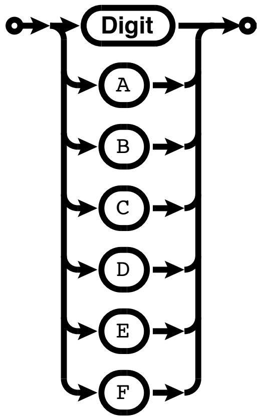
#3.5b Base-16 Digit Group



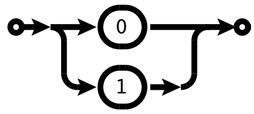
#3.6 Digit



#3.7 Base-16 Digit

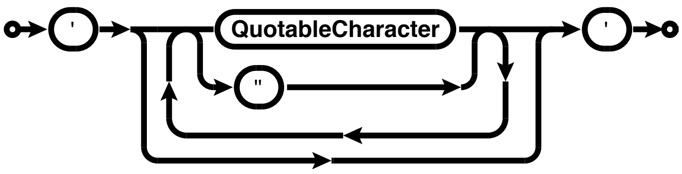


#3.8 Base-2 Digit

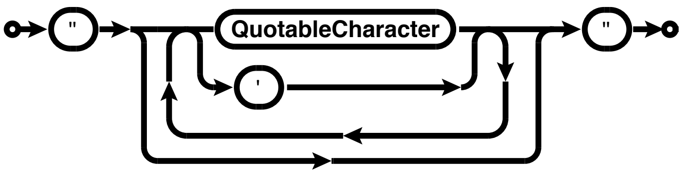


#4 String Literal

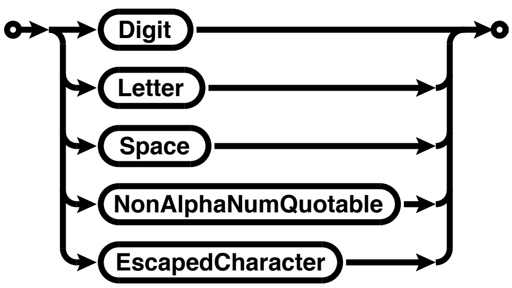
#4.1 Single Quoted String



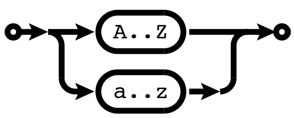
#4.2 Double Quoted String



#4.3 Quotable Character



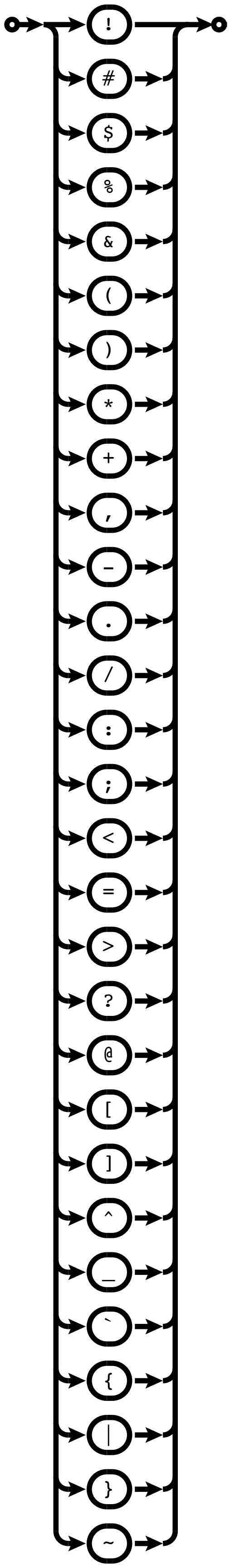
#4.4 Letter



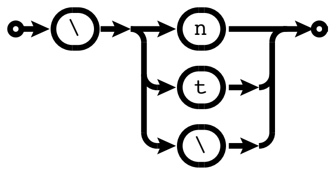
#4.5 Space

```
CONST Space = CHR(32)
```

#4.6 Non-Alphanumeric Quotable Character

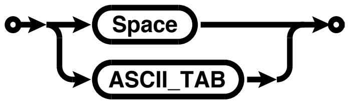


#4.7 Escaped Character



B.3 Ignore Symbols

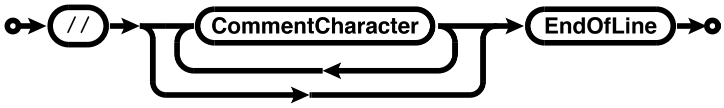
#1 Whitespace



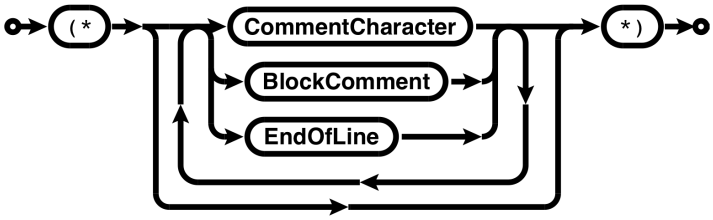
#1.1 ASCII Tabulator

```
CONST ASCII_TAB = CHR(8)
```

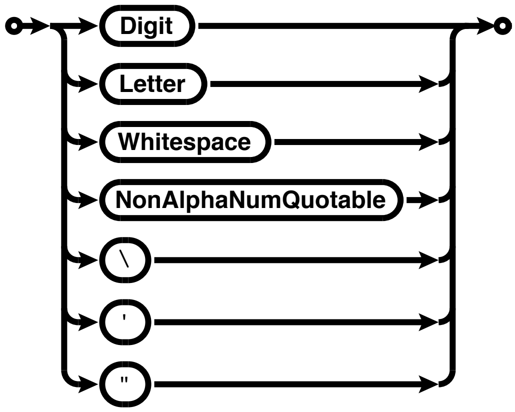
#2 Line Comment



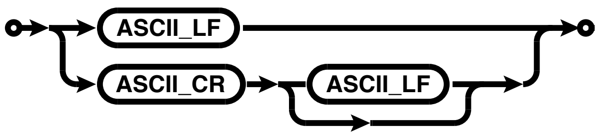
#3 Block Comment



#3.1 Comment Character



#4 End Of Line Marker



#4.1 ASCII Line Feed

```
CONST ASCII_LF = CHR(10)
```

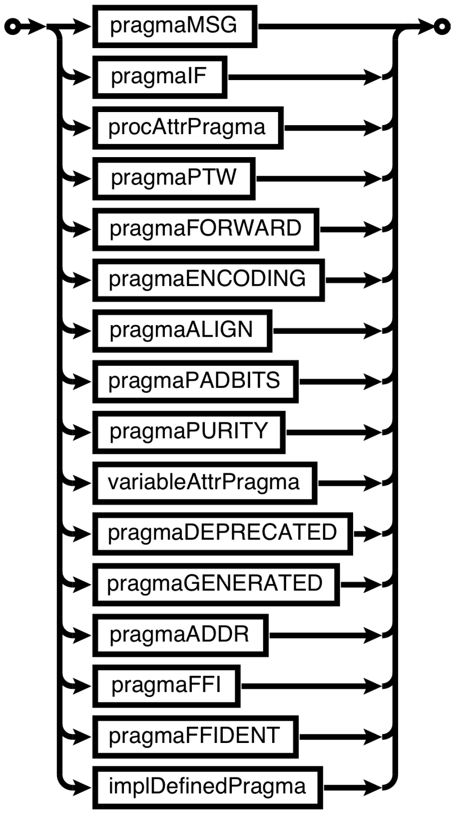
#4.2 ASCII Carriage Return

```
CONST ASCII_CR = CHR(13)
```

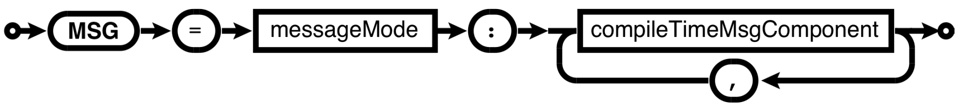

B.4 Pragma Grammar

#1 Pragma

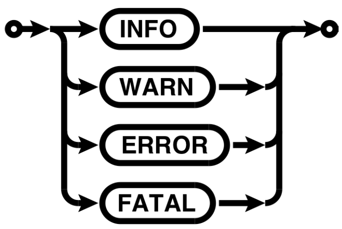
#1.1 Pragma Body



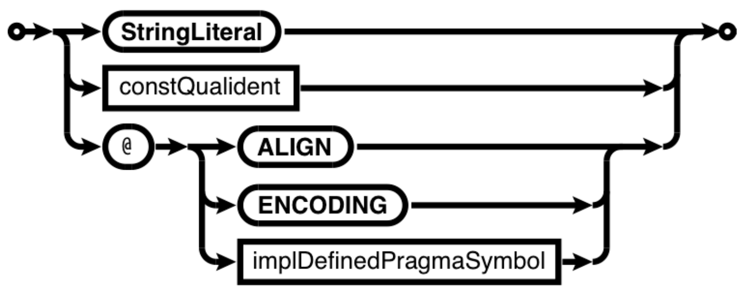
#2 Body Of Compile Time Message Pragma



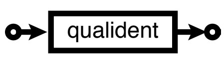
#2.1 Message Mode



#3 Compile Time Message Component

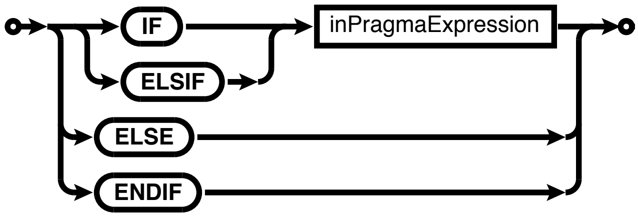


#3.1 Constant Qualified Identifier

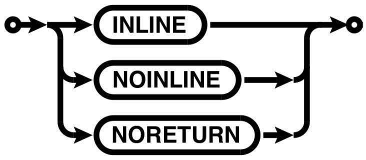


#3.2 Implementation Defined Pragma Symbol

#4 Body Of Conditional Compilation Pragma

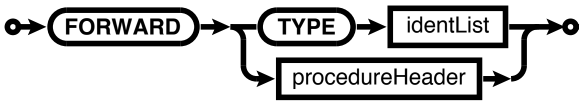


#5 Body Of Procedure Attribute Pragma

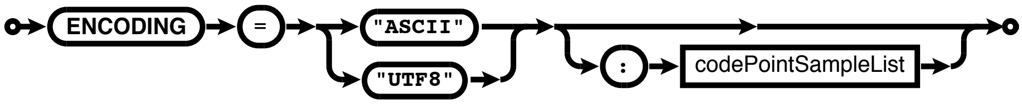


#6 Body Of Promise-To-Write Pragma

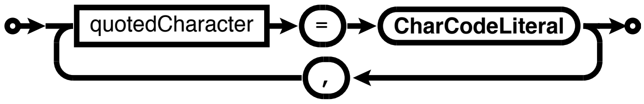
#7 Body Of Forward Declaration Pragma



#8 Body Of Character Encoding Pragma



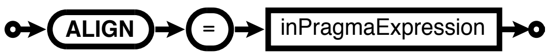
#9 Code Point Sample List



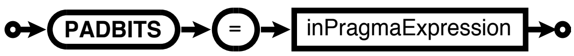
#9.1 Quoted Character

#9.2 Character Code Literal

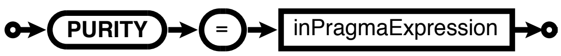
#10 Body Of Memory Alignment Pragma



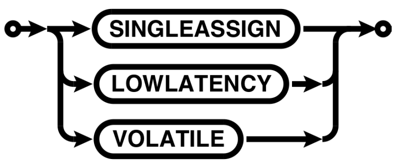
#11 Body Of Bit Padding Pragma



#12 Body Of Purity Attribute Pragma

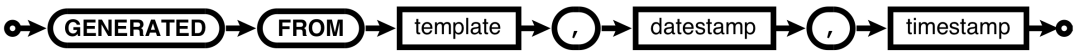


#13 Body Of Variable Attribute Pragma



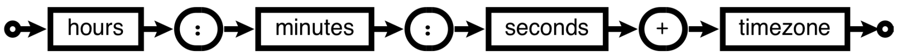
#14 Body Of Deprecation Pragma

#15 Body Of Generation Timestamp Pragma

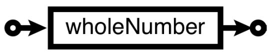


#15.1 Date Stamp

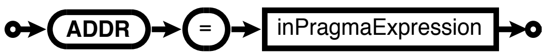
#15.2 Time Stamp



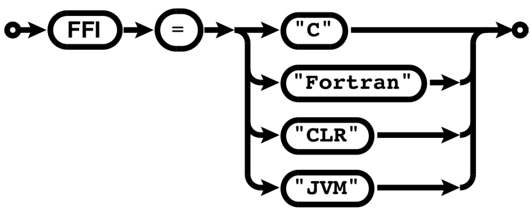
#15.3 Year, Month, Day, Hours, Minutes, Seconds, Timezone



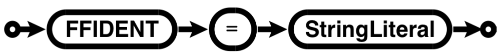
#16 Body Of Memory Mapping Pragma



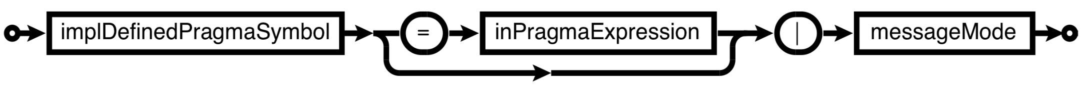
#17 Body Of Foreign Function Interface Pragma



#18 Body Of FFI Identifier Mapping Pragma

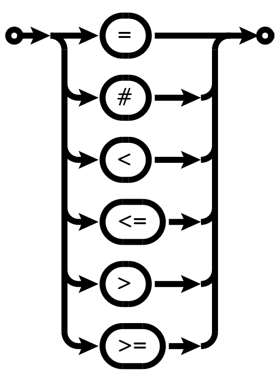


#19 Body Of Implementation Defined Pragma

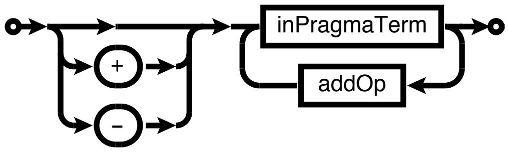


#20 In-Pragma Expression

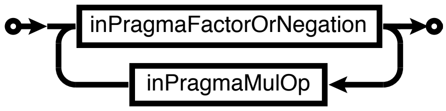
#20.1 In-Pragma Relational Operator



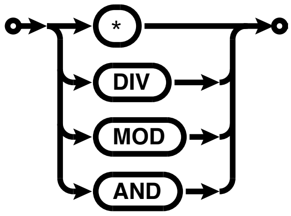
#21 In-Pragma Simple Expression



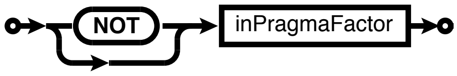
#22 In-Pragma Term



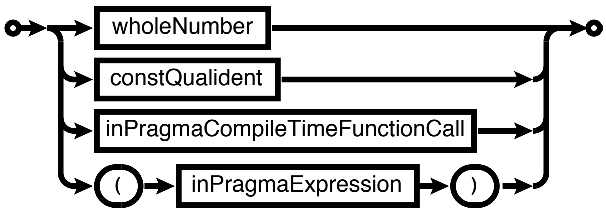
#22.1 In-Pragma Multiply Operator



#23 In-Pragma Factor Or Negation

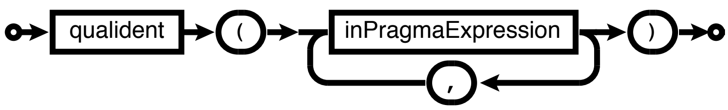


#24 In-Pragma Factor



#24.1 Whole Number

#25 In-Pragma Compile-Time Function Call



Appendix C: Compliance Report Sheet

ID	Category	Requirement	Compliance
1	**Core Language**		
1.1	Literals, Comments, Lexical Parameters	mandatory	
1.2	Compilation Units	mandatory	
1.3	`GENLIB` Directive	mandatory	
1.4	Import Directives	mandatory	
1.5	Type Constructors	mandatory	
1.6	Procedures	mandatory	
1.7	Expressions and Operators	mandatory	
1.8	Structured Value Constructors	mandatory	
1.9	Statements	mandatory	
1.10	Predefined Constants	mandatory	
1.11	Predefined Types	mandatory	
1.12	Predefined Procedures and Functions	mandatory	
1.13	Binding to built-in Syntax and Procedures	mandatory	
1.14	Scalar Conversion	mandatory	
2	**Pseudo-Modules**		
2.1	Module `COMPILER`	mandatory	
2.2	Module `RUNTIME`	mandatory	
2.3	Module `CONVERSION`	mandatory	
2.4	Module `UNSAFE`	mandatory	
2.5	Module `ATOMIC`	mandatory	
2.6	Module `ASSEMBLER`	optional	
3	**Language Pragmas**		
3.1	Compile Time Message Pragma	mandatory	
3.2	Conditional Compilation Pragmas	mandatory	
3.3	Procedure Inlining Pragmas	mandatory	
3.4	Pragma `PTW`	mandatory	
3.5	Pragma `FORWARD`	see below	
3.5.1	Forward declarations	mandatory for single-pass compilers	
3.5.2	Silently ignore and skip pragma	mandatory for multi-pass compilers	
3.6	Pragma `ENCODING`	see below	
3.6.1	ASCII encoding	mandatory	
3.6.2	UTF8 encoding	optional	
3.7	Pragma `ALIGN`	optional	
3.8	Pragma `PADBITS`	optional	
3.9	Pragma `NORETURN`	optional	
3.10	Pragma `PURITY`	optional	
3.11	Pragma `SINGLEASSIGN`	optional	
3.12	Pragma `LOWLATENCY`	optional	
3.13	Pragma `VOLATILE`	optional	
3.14	Pragma `DEPRECATED`	optional	
3.15	Pragma `GENERATED`	optional, strongly recommended	
3.16	Pragma `ADDR`	optional	
3.17	Pragma `FFI`	optional	
3.17.1	Foreign Function Interface to C	optional in combination with pragma `FFI`	
3.17.2	Foreign Function Interface to Fortran	optional in combination with pragma `FFI`	
3.17.3	Foreign Function Interface to the CLR	optional in combination with pragma `FFI`	
3.17.4	Foreign Function Interface to the JVM	optional in combination with pragma `FFI`	
3.18	Pragma `FFIDENT`	optional in combination with pragma `FFI`	
4	**Generics**		
4.1	Modula-2 Template Engine	mandatory	
5	**Standard Library**		
5.1	IO Libraries for Built-in Types	mandatory	
5.2	Standard Library Blueprints	mandatory	
5.3	Template Library	mandatory	

Appendix D: Online Resources

D.1 Differences between R10 and PIM

http://modula2.net/resources/Diff-R10-PIM.pdf

D.2 Pseudo Module Definitions

http://bitbucket.org/trijezdci/m2r10/src/tip/_PSEUDO_MODULES

D.3 Standard Library Definitions

http://bitbucket.org/trijezdci/m2r10/src/tip/_STANDARD_LIBRARY

D.4 Reference Compiler

Project Root
http://bitbucket.org/trijezdci/m2r10

Compiler Sources

http://bitbucket.org/trijezdci/m2r10/src/tip/_REFERENCE_COMPILER [1]

Appendix E: Statistics

E.1 Specification

- the language specification document has 162 pages, 45 100 words, 249 000 characters, thereof 1 cover page, 2 pages abstract and acknowledgements,12 pages TOC, 4 pages Glossary, 8 blank pages, 11 pages EBNF, 12 pages syntax diagrams, 1 compliance sheet page, 1 URL page, the actual language report including examples and 11 pages of EBNF is thus only about 120 pages long.

E.2 Base Language

- the core grammar has 62 non-terminals, 4 terminals, 4 ignore symbols
- the pragma grammar has 25 non-terminals and re-uses the terminals of the core grammar
- the language has 45 reserved words, 27 reserved pragma symbols, 40 predefined identifiers, thereof 3 built-in constants, 10 built-in types, 10 built-in procedures and 17 built-in functions; 5 operator precedence levels and 17 operators

E.3 Pseudo Modules

- module `COMPILER` provides 1 type, 22 constants and 12 lexical macros
- module `RUNTIME` provides 2 types, 8 procedures and 7 functions
- module `CONVERSION` provides 3 constants, 3 lexical macros and 4 primitives
- module `UNSAFE` provides 13 constants, 5 types, 2 pseudo-types and 26 intrinsics
- module `ATOMIC` provides 1 type, 1 function, 8 atomic intrinsics

[1] The reference compiler is work in progress.

