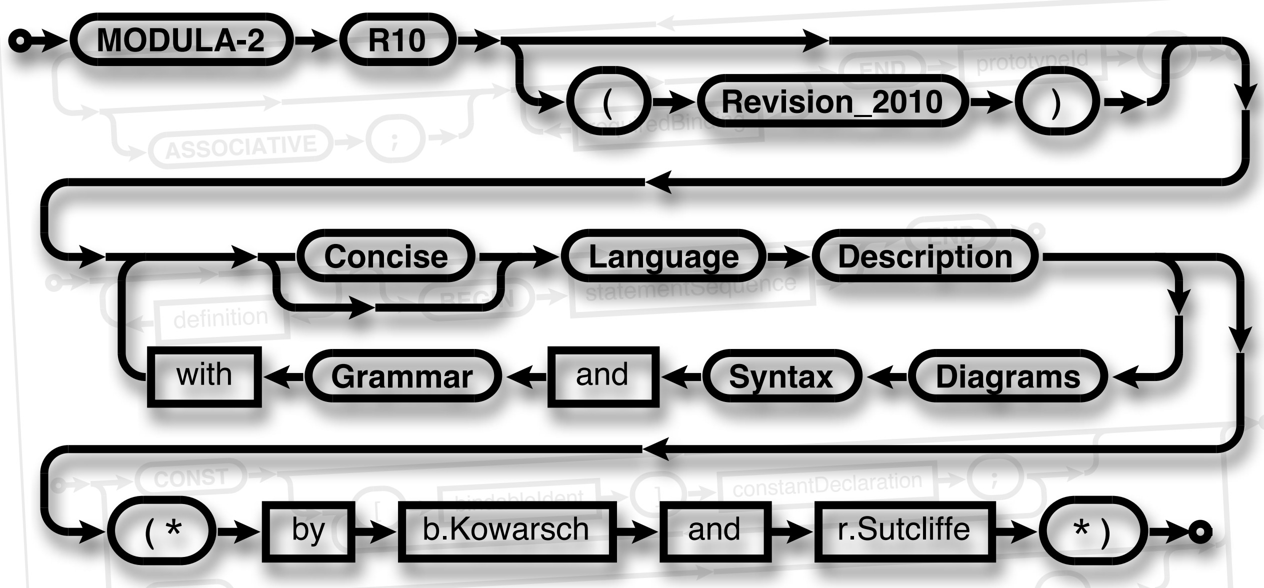
Modula-2 R10

Revision 2010

Last edited: December 12, 2012



Synopsis

This document contains the authoritative language description 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 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 finalised in 2011 and 2012. Design work on Phase I of the language definition is now complete. A general programming part will still be added to prepare for publishing the language report in book format.

Abbreviations

ADT	Abstract Data Type	EBNF	Extended Backus-Naur Formalism
API	Application Programming Interface	SXF	Scalar Exchange Format
ASCII	ISO464-US 7-bit character set	UCS	Unicode Character Set
BCD	Binary Coded Decimals	UTF8	UCS Transformation Format 8-bit
BOM	Byte Order Mark	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

Reference Compiler

Work on the M2R10 reference compiler front end for LLVM will resume on the basis of Phase I after the editorial review has been completed. It will later be extended with Phase II work items.

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

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, Gaius Mulley, Frode Odegard, Michael T. Richter, Diego Sardina and Marco van de Voort for their valuable feedback, further Günther Blaschek, Jürg Gutknecht, Christian Maurer, Kees Pronk, 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[1] to accept any constant or variable of any data-type and to cast a passed-in actual parameter to the data type of the formal parameter is called auto-casting[2]. Such a parameter is called an auto-casting formal parameter. In Modula-2 auto-casting formal parameters are always open array parameters.

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 two kinds of bindings that are user-definable.

0.4.1 Binding to an Operator

A library that implements an abstract data type may define a procedure and bind it to an operator. The abstract data type may then be used in infix expressions using the bound-to operator.

0.4.2 Binding to a Pervasive

A library that implements an abstract data type may define a procedure and bind it to a pervasive 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 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 classified as a fully compliant implementation. A fully compliant implementation that provides any additional syntax, operators, pervasives, pseudo-modules or language pragmas is classified as a fully compliant superset. Such a superset may be a domain specific superset.

0.6.2 Partial Compliance

An implementation that omits any syntax, operators, pervasives, pseudo-modules or language pragmas, but complies with the specification in those parts that it implements is classified as a partially compliant implementation or subset. Such a subset may be a domain specific subset.

---

[1] For a definition of formal parameter, see Parameter.

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

0.6.3 Non-Compliant Derivative

An implementation that provides any modified syntax, operators, pervasives, 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, pervasives or language pragmas have been reserved in the language specification for exclusive use by the superset is a coordinated 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.

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. The name 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

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.

0.13.2 Actual 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 Pervasive, Pervasive Identifier

A constant, a data type or a procedure that is visible by default in any module scope without the need to be imported prior to its use is called a pervasive entity, or in short a pervasive. Its identifier is called a pervasive identifier. A pervasive identifier is not a reserved word and may therefore be redefined. All pervasives are language defined. A library may not define a pervasive.

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 Regular Procedure

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

0.16.2 Function Procedure

A function procedure is a procedure that returns 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. The signature of a procedure 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 that has special properties different from those of the corresponding regular entities. There are three kinds.

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. Pseudo-modules may not be modified.

0.17.2 Pseudo-Procedure

A pseudo-procedure is a built-in intrinsic that acts and looks like a procedure but cannot be passed as a procedure type parameter or assigned to a procedure type variable.

0.17.3 Pseudo-Type

A pseudo-type is a built-in type whose use is restricted to a specific use case or a limited number of specific use cases, such as use as a formal type in a formal parameter list.

0.18 Type Equivalence

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

0.18.1 Name Equivalence

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

---

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

0.18.1.1 Loose Name Equivalence

Under loose name equivalence, an alias of a type is always considered equivalent to the aliased type, whether intended or unintended. No facility is provided to distinguish between intended and unintended equivalence of types.

0.18.1.2 Strict Name Equivalence

Under strict name equivalence an alias of a type is not considered equivalent to the aliased type. However, a facility may be provided to distinguish between intended and unintended alias types, in which case intended alias types are considered equivalent to their aliased type, while unintended alias types are not.

0.18.2 Structural Equivalence

Under structural equivalence, any two types are considered equivalent if their structures match. No facility is provided to distinguish between intended and unintended equivalence of types.

0.19 Type Transfer

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

0.19.1 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.

0.19.2 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 the 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.

0.20 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.

0.21 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 of Wirthian macros first appeared with the introduction of `NEW` and `DISPOSE` in classic Modula-2 but without definition of any associated terminology. The authors chose this term in honour of Niklaus Wirth.

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

Special symbols are non-alphanumeric characters or sequences of two non-alphanumeric characters that have special meaning in the language.

List of Special Symbols

`#` not-equal operator
`*` multiplication and set intersection operator
`+` addition and set union operator
`++` suffix for increment statement
`,` punctuation, used as a separator in item lists
`-` subtraction and set difference operator
`--` suffix for decrement statement
`.` punctuation, used as a separator, decimal point and module terminator
`..` range constructor
`/` division and symmetric set difference operator
`:` punctuation, used as a separator between identifiers and formal types
`::` type conversion operator
`:=` assignment operator
`;` punctuation, used as a separator in statement sequences
`<` less-than and true-subset relational operator
`<=` less-than-or-equal and subset relational operator
`=` equal operator
`>` greater-than and true-superset relational operator
`>=` greater-than-or-equal and superset relational operator
`^` pointer dereferencing operator
`|` punctuation, used as a separator in case label lists

`'` single quote, used as a string delimiter
`"` double quote, used as a string delimiter
`\` escape symbol within quoted literals

`[ ]` brackets, used as index operator and to delimit special syntax
`( )` parentheses, used to group expressions and to delimit argument lists
`{ }` braces, used to delimit structured values

`!` pseudo-operator used to define storage mutator binding
`~` pseudo-operator used to define removal mutator binding
`?` pseudo-operator used to define retrieval accessor binding

`<* *>` opening and closing delimiters for pragmas
`?` pragma value query prefix within console message pragmas

`//` prefix for single-line comment
`(* *)` opening and closing delimiters for multi-line comments

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 : "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:

```
0x80, 0xFF, 0xDEADBEAF (* without digit separator *)
0x00’00‘FF’FF, 0xCAFE’D00D (* 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 : "\" ( "0" | "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, 3.1415926, "abc" }
{ { 1, 2, 3 }, { "a", "b", "c" }, 42 }
{ 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	DESCENDING	IF	OF	RETURN
AND	DIV	IMPLEMENTATION	OPAQUE	SET
ARRAY	DO	IMPORT	OR	THEN
ASSOCIATIVE	ELSE	IN	PLACEHOLDERS	TO
BEGIN	ELSIF	INDETERMINATE	POINTER	TYPE
BY	END	LOOP	PROCEDURE	UNTIL
CASE	EXIT	MOD	PROTOTYPE	VAR
CONST	FOR	MODULE	RECORD	VARIADIC
DEFINITION	FROM	NOT	REPEAT	WHILE

1.5 Identifiers

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

EBNF:

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

The use of the lowline 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.

Examples:

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


There are two categories of identifiers:

- reserved identifiers
- user-definable identifiers

1.5.1 Reserved Identifiers

Reserved identifiers are predefined identifiers that may not be redefined. There are four categories:

- pervasive identifiers
- non-pervasive bindable identifiers
- module identifiers of pseudo-modules
- reserved identifiers in pseudo-module SYSTEM

1.5.1.1 Pervasive Identifiers

```
NIL, TRUE, FALSE, BOOLEAN, OCTET, CHAR, UNICHAR, BITSET, LONGBITSET, CARDINAL,
LONGCARD, INTEGER, LONGINT, REAL, LONGREAL, NEW, DISPOSE, RETAIN, RELEASE, READ,
WRITE, WRITEF, ABS, NEG, ODD, SUCC, PRED, ORD, CHR, COUNT, LENGTH, SIZE, NEXTV,
TMIN, TMAX, TSIZE, TLIMIT
```

1.5.1.2 Non-Pervasive Bindable Identifiers

```
TSIG, TEXP, SXF, VAL
```

1.5.1.3 Module Identifiers Of Pseudo-Modules

```
ATOMIC, SYSTEM, CONVERSION, COMPILER, RUNTIME, COROUTINE, ACTOR, ASSEMBLER
```

1.5.1.4 Reserved Identifiers In Pseudo-Module SYSTEM

```
ADDRESS, BYTE, WORD, MACHINEBYTE, MACHINEWORD, ARGCOUNT, UNSAFEARGLIST, ADR, CAST
```

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 text. Pragmas consist of an opening pragma delimiter `<*`, a pragma body and a closing pragma delimiter `*>`. The pragma body consists of a pragma word optionally followed by a sequence of symbols and further pragma words. A pragma word consists of a letter followed by any number of letters and digits.

EBNF:

```
pragma : "<*" pragmaWord ( pragmaWord | otherSymbol )* "*>" ;
pragmaWord : Letter ( Letter | Digit )* ;
```


There are two types of pragmas:

- language defined pragmas
- implementation defined pragmas

1.6.1.1 Language Defined Pragmas

Language defined pragmas use all-uppercase words as pragma symbols. The symbol names of language defined pragmas are reserved.

List of Language Defined Pragma Words:

ADDR	ENCODING	FROM	NOINLINE	TYPE
ALIGN	ERROR	IF	PADBITS	VOLATILE
ELSE	FATAL	INFO	PURITY	WARN
ELSIF	FFI	INLINE	REG	
ENDIF	FORWARD	MSG	SINGLEASSIGN	

1.6.1.2 Implementation Defined Pragmas

Any implementation may define its own set of pragmas, specific to the compiler. Implementation defined pragmas are therefore not-portable between different implementations. The symbol names of implementation defined pragmas are not reserved and may not be all-uppercase.

Examples:

```
<* BoundsChecking = FALSE *>

<* UnrollLoops = TRUE *>
```

Implementation defined pragmas are ignored by implementations that do not support them. A compile time warning is emitted when an unrecognised pragma is encountered. However, a compiler option may be provided to suppress such warnings.

1.6.2 Comments

Comments are ignored by the compiler but intended for a human reader. There are two types of comments:

- single-line comments
- multi-line comments

1.6.2.1 Single-Line Comments

Single-line comments start with a double-slash symbol and are terminated by an end-of-line marker.

EBNF:

```
SingleLineComment : "//" ~( EndOfLine )* EndOfLine ;

EndOfLine : CR LF? | LF CR? ;
```

They are intended for copyright and license information and for in-source module documentation.

The prefix for single-line comments has been chosen to make them visually stand out from source code annotations and they are therefore not intended for annotating source code.

Example:

```
// ----------------------------------------------------------------------------
// Procedure Increment( x )
// ----------------------------------------------------------------------------
// Procedure Increment increments its integer operand x by one.
// pre-conditions: operand x must be smaller than TMAX(INTEGER).
// post-conditions: the new value of x is the previous value of x + 1.
// error-conditions: if x = TMAX(INTEGER) a runtime overflow error occurs.
PROCEDURE Increment ( VAR x : INTEGER );
```

1.6.2.2 Multi-Line Comments

Multi-line comments are delimited by opening and closing comment delimiters.

EBNF:

```
MultiLineComment : "(*" ( ~( "*)" )* MultiLineComment? )* "*)" ;
```

Multi-line comments are intended for annotating source code. 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^.successor = NIL THEN
(* This is a comment (* and a comment within *) *)
```

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? | 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`, permitted only at the very beginning of a source file

Any other control codes within Modula-2 source text are illegal and shall result in a compile time error. Any control code within a string literal is illegal and shall result in a compile time error.

Implementations that support encoding schemes other than UTF8 may allow other BOMs to appear at the very beginning of a source file in order to identify the respective encoding and byte order. However, support for additional encoding schemes is implementation defined and constitutes a non-portable language extension. An unsupported BOM shall result in a compile time error.

1.8 Symbols Reserved for Language Extensions and External Utilities

Although not part of the language, certain symbols are reserved for exclusive use by language extensions and external source code processing utilities.

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`
	Reserved Pervasives	`NO OBJECT YES`
	Reserved Pragmas	`ACTION FRAMEWORK OUTLET QUALIFIED`
Parallel Modula-2	Reserved Words	`ALL PARALLEL SYNC`
	Reserved Pragmas	`LOCAL SPREAD CYCLE SBLOCK CBLOCK`
Single-Pass Compilers	Reserved 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 pervasive identifiers, reserved words and language pragmas as long as they start with a single ampersand character `&`.

Examples:

```
&ON &OFF &TRY &CATCH (* pervasives and reserved words *)
<* &OPT *> <* &PURE *> (* language pragmas *)
```

1.8.3 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	`!!` `??`

1.8.4 Symbols Reserved for Phase II

Identifiers `COROUTINE` and `ACTOR`, and pragma symbol `PRIORITY` are reserved for Phase II.

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 compile time warning shall be emitted. However, a compiler option may be provided to suppress such warnings.

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 occurs.

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 truncated a compile time warning shall be emitted. However, a compiler option may be provided to suppress such warnings.

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 single line comments, 250 characters
- for multi-line 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 compile time warning shall be emitted. However, a compiler option may be provided to suppress such warnings.

Furthermore, if a nested multi-line comment is truncated, an implementation shall insert all closing comment delimiters that would 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 compile time warning shall be emitted 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
- a prototype definition
- the definition[1] part of a library module
- the implementation part of a library module

EBNF:

```
compilationUnit :
    programModule | implementationOfModule | definitionOfModule | prototype ;
```

2.1 Program Modules

A Modula-2 program consists of exactly one program module and zero or more prototype definitions and library modules. A program module does not export any identifiers. The body of a program module corresponds to the `main()` function in a C program.

EBNF:

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

moduleIdent : Ident ;
```

2.2 Prototype Definitions

A prototype definition represents a common set of semantics to which abstract data types (ADTs) may be required to conform. A prototype determines how a conforming ADT may be declared, what type of literal it may use, and which bindings to operators and built-in procedures it shall define.

EBNF:

```
prototype :
    PROTOTYPE prototypeId ";"
    TYPE "=" ( RECORD | OPAQUE RECORD? ) ( ":=" protoliteral )? ";"
    ( requiredBinding “;” )*
    END prototypeId "." ;

prototypeId : Ident ;

protoliteral : simpleProtoliteral | structuredProtoliteral ;

simpleProtoliteral : Ident ;

structuredProtoliteral :
   “{“ ( VARIADIC OF simpleProtoliteral ( “,” simpleProtoliteral )* ) |
       ( structuredProtoliteral ( “,” structuredProtoliteral )+ ) “}” ;

requiredBinding : procedureHeader;
```

Example:

```
PROTOTYPE CTYPE; (* complex numbers *)

TYPE = OPAQUE RECORD := { REAL, REAL };

PROCEDURE [+] add ( a, b : CTYPE ) : CTYPE; (* require binding to + *)
PROCEDURE [-] sub ( a, b : CTYPE ) : CTYPE; (* require binding to - *)
```

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

2.3 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:

```
definitionOfModule :
    DEFINITION MODULE moduleId ( "[" prototypeId "]" )? ";"
    importList* definition*
    END moduleId "." ;
```

2.4 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.

EBNF:

```
implementationOfModule :
    IMPLEMENTATION programModule ;
```

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:

```
importList :
    ( FROM moduleIdent IMPORT ( identList | "*" ) |
    IMPORT moduleIdent "+"? ( "," moduleIdent "+"? )* ) ";"
```

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. When importing identically named identifiers from different modules in this way, the name conflict shall cause a compile time error. This facility should therefore be used with caution.

As a general guideline unqualified import should be avoided. However, its use may be justifiable when using certain uppercase identifiers of pseudo-modules as it can reduce clutter and improve readability. Furthermore, uppercase identifiers are less likely to clash with user-defined identifiers.

Example:

```
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 library module by using an asterisk as a wildcard. This facility should be used with the utmost caution because it increases the likelihood of name conflicts. However, its use is recommended with pseudo-module `ASSEMBLER`.

Example:

```
(* import all identifiers from module ASSEMBLER *)
FROM ASSEMBLER IMPORT *;
(* then use unqualified ... *)
CODE ( mov, eax, 0x80000 );
(* instead of qualified, which would cause clutter ... *)
ASSEMBLER.CODE ( ASSEMBLER.mov, ASSEMBLER.eax, 0x80000 );
```

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 is permissible. However, all imports but the first import are redundant and shall be ignored.

3.3.2 Unqualified Import of an Already Imported Identifier

Unqualified import of an identifier that has already been imported unqualified from the same module is permissible. However, all imports but the first import are redundant and shall be ignored.

3.3.3 Qualified and Unqualified Import of an Identifier

Unqualified import of an identifier that is also imported qualified in the same scope is permissible. The imported entity may then be referenced both qualified and unqualified.

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 the same scope results in a name conflict and causes 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 twelve language predefined pervasive data types:

Pervasive Types:

```
BOOLEAN, BITSET, LONGBITSET, CHAR, UNICHAR, OCTET,
CARDINAL, LONGCARD, INTEGER, LONGINT, REAL, LONGREAL
```

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 using the opaque type constructor.

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

4.1 Type Compatibility

Modula-2 R10 uses strict name equivalence. Two types with different names are incompatible unless they are specifically compatible under `ALIAS` type or subrange type compatibility rules.

4.1.1 Alias Type Compatibility

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 values of type `A` are compatible with `T` and 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.

4.1.2 Subrange Type Compatibility

A subrange type is a derived type that has been defined using the `[n .. m] OF` subrange type constructor. 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.

4.1.3 Compatibility of Literals

Whole number literals are assignment compatible with:

- system types `SYSTEM.BYTE` , `SYSTEM.WORD` and `SYSTEM.ADDRESS`
- pervasive 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 defines a procedure to bind to the assignment operator

Real number literals are assignment compatible with:

- pervasive 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:

- pervasive 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 not a mutable variable designator and `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:

- `e` is not 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`

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 SYSTEM.BYTE`, or `CAST ARRAY OF SYSTEM.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 SYSTEM.BYTE`, or `CAST ARRAY OF SYSTEM.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 `SYSTEM.UNSAFEARGLIST`

- 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 `SYSTEM.UNSAFEARGLIST`
- 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 Ordinal Types

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

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

4.2.2 Convertibility of Pervasive Numeric Types

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

A value `v1` of a pervasive whole number type is convertible to an equivalent value of an 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 pervasive numeric 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 pervasive numeric type
- `T` is an ADT that provides a conversion primitive from scalar exchange format

4.2.10 Non-Convertibility of SYSTEM Types

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

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 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 ordinal`)
- equal (=)
- not-equal (#)
- less (<)
- less-or-equal (<=)
- greater (>)
- greater-or-equal (>=)

Pervasive data types `BOOLEAN`, `CHAR` and `UNICHAR`, and all enumeration types are 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 an ordinal type with two values, interpreted as boolean truth values, represented by the pervasive constants `TRUE` and `FALSE`, where `TRUE > FALSE`. Further to the operations defined for ordinal types, three additional operations are defined for the boolean type:

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

Pervasive 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 (#)
- true subset (<)
- true superset (>)
- subset (<=)
- superset (>=)

Pervasive data types `BITSET` and `LONGBITSET`, and 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 subranges of the mathematical set of integers (Z), always 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`)
- 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 (>=)

Pervasive 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 approximations to the mathematical set of real numbers (R), always with a finite number of values.

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 (>=)

Pervasive 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`)
- deallocation (`DISPOSE`)
- 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 (`DISPOSE`)
- dereference (`^`)
- equal (=)
- not-equal (#)

The invalid pointer value `NIL` may not be dereferenced. An attempt to do so shall result in a run time error.

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 (`DISPOSE`)

- assignment (`:=`) of `NIL` and entities of the same type only
- 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 SYSTEM Types

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

4.4 Library Defined Prototypes

The standard library provides a set of prototype definitions to allow the construction of library defined abstract data types with the same semantics as pervasive types and transparent data types defined using type constructor syntax. To require an ADT to conform to a prototype, the library that defines the ADT must specify the prototype 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 prototype definitions are immutable and their immutability is compiler enforced. Any attempt to use a standard library prototype that has been modified shall cause a compilation error.

The standard library provides three kinds of prototypes:

- numeric prototypes
- collection prototypes
- date-time prototypes.

4.4.1 Numeric Prototypes

The standard library provides a hierarchical set of prototypes for the construction of numeric types.

4.4.1.1 Prototype ProtoNumeric

The standard library provides numeric root prototype `ProtoNumeric` for the construction of numeric prototypes. It defines a subset of properties common to all numeric types.

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

4.4.1.2 Prototype ProtoScalar

The standard library provides numeric prototype `ProtoScalar` derived from `ProtoNumeric` for the construction of numeric scalar prototypes. It defines a subset of properties common to all numeric scalar types.

4.4.1.3 Prototype ProtoNonScalar

The standard library provides numeric prototype `ProtoNonScalar` derived from `ProtoNumeric` for the construction of non-scalar prototypes. It defines a subset of properties common to all numeric non-scalar types.

4.4.1.4 Prototype ProtoCardinal

The standard library provides derived numeric scalar prototype `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`.

4.4.1.5 Prototype ProtoInteger

The standard library provides derived numeric scalar prototype `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`.

4.4.1.6 Prototype ProtoReal

The standard library provides derived numeric scalar prototype `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`.

4.4.1.7 Prototype ProtoComplex

The standard library provides derived numeric non-scalar prototype `ProtoComplex` for the construction of library-defined complex number types. Its definition follows the mathematical definition of complex numbers.

4.4.1.8 Prototype ProtoVector

The standard library provides derived numeric non-scalar prototype `ProtoVector` for the construction of library-defined numeric vector types. Its definition follows the mathematical definition of numeric vectors.

4.4.1.9 Prototype ProtoTuple

The standard library provides derived numeric non-scalar prototype `ProtoTuple` for the construction of library-defined numeric tuple types. Its definition follows the mathematical definition of numeric tuples.

4.4.1.10 Prototype ProtoRealArray

The standard library provides derived numeric non-scalar prototype `ProtoRealArray` for the construction of library-defined numeric array types with real number components.

4.4.1.11 Prototype ProtoComplexArray

The standard library provides derived numeric non-scalar prototype `ProtoComplexArray` for the construction of library-defined numeric array types with complex number components.

4.4.2 Collection Prototypes

The standard library provides a hierarchical set of prototypes for the construction of collection types.

4.4.2.1 Prototype ProtoCollection

The standard library provides collection root prototype `ProtoCollection` for the construction of collection prototypes. It defines a subset of properties common to all collection types.

4.4.2.2 Prototype ProtoStaticSet

The standard library provides derived collection prototype `ProtoStaticSet` for the construction of library-defined static ordered set types. For semantic compatibility, its definition matches the built-in types `BITSET` and `LONGBITSET`.

4.4.2.3 Prototype ProtoStaticArray

The standard library provides derived collection prototype `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.

4.4.2.4 Prototype ProtoStaticString

The standard library provides derived collection prototype `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.

4.4.2.5 Prototype ProtoSet

The standard library provides derived collection prototype `ProtoSet` for the construction of library-defined dynamic unordered set types. Its definition follows the mathematical definition of sets.

4.4.2.6 Prototype ProtoOrderedSet

The standard library provides derived collection prototype `ProtoOrderedSet` for the construction of library-defined dynamic ordered set types. It is derived from prototype `ProtoSet`.

4.4.2.7 Prototype ProtoArray

The standard library provides derived collection prototype `ProtoArray` for the construction of library-defined dynamic array types with numeric or ordinal indices.

4.4.2.8 Prototype ProtoString

The standard library provides derived collection prototype `ProtoString` for the construction of library-defined dynamic character string types.

4.4.2.9 Prototype ProtoDictionary

The standard library provides derived collection prototype `ProtoDictionary` for the construction of library-defined dynamic unordered associative array types such as hash tables.

4.4.2.10 Prototype ProtoOrderedDict

The standard library provides derived collection prototype `ProtoOrderedDict` for the construction of library-defined dynamic ordered associative array types.

4.4.3 Date-Time Prototypes

4.4.3.1 Prototype ProtoDateTime

The standard library provides prototype `ProtoDateTime` for the construction of library defined static date-time types.

4.4.3.2 Prototype ProtoInterval

The standard library provides prototype `ProtoInterval` for the construction of library defined static date-time interval types.

4.4.4 User Defined Prototypes

User libraries may provide their own prototype definitions for their own custom designed abstract data types. User defined prototypes may be derived from standard library prototypes.

4.5 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 prototype in its module header. This represents a promise to conform to the common set of semantics defined by the prototype. An ADT defined to conform to a prototype must bind its own library defined procedures to those operators and pervasive procedures that are required by the prototype. Static conformance is compiler enforced. No other bindings than those defined by the prototype are permitted except for bindings to the conversion operator.

Example:

```
DEFINITION MODULE BCD [ProtoReal]; (* must conform to prototype ProtoReal *)
TYPE BCD = OPAQUE RECORD value : ARRAY 8 OF OCTET END;
(* define bindings to pervasives and operators required by RTYPEProtoReal *)
PROCEDURE [VAL] fromSXF( VAR bcd : BCD; CONST sxf : ARRAY OF OCTET );
PROCEDURE [+] add ( a, b : BCD ) : BCD;
...
(* define IO procedures for use by READ, READF, WRITE and WRITEF *)
PROCEDURE Write ( f : File; b : BCD );
...
END BCD.
```

Defining an ADT with a prototype 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 prototype'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 pervasive 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 calls to bound pervasive 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(stdOut, sum); (* replaced by BCD.Write(stdOut, sum); *)
```

4.6 Library Defined ADTs Using Prototypes and Bindings

The standard library provides a rich set of `ALIAS` types and library defined ADTs using bindings and are practically indistinguishable from pervasive types and transparent data types defined using type constructor syntax.

4.6.1 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.

4.6.1.1 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;
BEGIN set := { 0, 7, 15 }; union := set + { 1, 2, 4 }; set[7] := FALSE END.
```

The `ALIAS` type whose bit width matches that of pervasive type `BITSET` is defined as an alias of `BITSET`. The `ALIAS` type whose bit width matches that of pervasive type `LONGBITSET` is defined as an alias of `LONGBITSET`.

The `ALIAS` types whose bit width does not match that of pervasive types `BITSET` or `LONGBITSET` are defined as aliases of the matching standard library defined bitset implementation types `BS16`, `BS32`, `BS64` and `BS128`.

Module `Bitsets` provides two additional `ALIAS` types `SHORTBITSET` and `LONGLONGBITSET`. The relationships between bit widths of bitset types is as follows:

```
TSIZE(SHORTBITSET) <= TSIZE(BITSET)
TSIZE(BITSET) < TSIZE(LONGBITSET) < TSIZE(LONGLONGBITSET)
```

The mappings are defined automatically in the standard library using conditional compilation pragmas. Since the bit widths of pervasive types `BITSET` and `LONGBITSET` are target dependent and implementation defined, which `ALIAS` types map to pervasive types and which map to standard library implementations is also target dependent and implementation defined.

4.6.1.2 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 prototype ~~`SetType`~~`ProtoStaticSet` and their semantics match those of pervasive types `BITSET` and `LONGBITSET`.

4.6.2 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.

4.6.2.1 Alias Types for Unsigned Integer Types

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;
BEGIN a := 123; sum := a + 456; WRITE(stdOut, sum) END.
```

The `ALIAS` type whose bit width matches that of pervasive type `CARDINAL` is defined as an alias of `CARDINAL`. The `ALIAS` type whose bit width matches that of pervasive type `LONGCARD` is defined as an alias of `LONGCARD`.

The `ALIAS` types whose bit width does not match that of pervasive 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 pervasive types `CARDINAL` and `LONGCARD` are target dependent and implementation defined, which `ALIAS` types map to pervasive types and which map to standard library implementations is also target dependent and implementation defined.

4.6.2.2 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 prototype ~~`ZTYPE`~~`ProtoCardinal` and their semantics match those of pervasive types `CARDINAL` and `LONGCARD`.

4.6.3 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.

4.6.3.1 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;
BEGIN a := 123; sum := a - 456; WRITE(stdOut, sum) END.
```

The `ALIAS` type whose bit width matches that of pervasive type `INTEGER` is defined as an alias of `INTEGER`. The `ALIAS` type whose bit width matches that of pervasive type `LONGINT` is defined as an alias of `LONGINT`.

The `ALIAS` types whose bit width does not match that of pervasive 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 pervasive types `INTEGER` and `LONGINT` are target dependent and implementation defined, which `ALIAS` types map to pervasive types and which map to standard library implementations is also target dependent and implementation defined.

4.6.3.2 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 prototype ~~`ZTYPE`~~`ProtoInteger` and their semantics match those of pervasive types `INTEGER` and `LONGINT`.

4.6.4 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 pervasive types `REAL` and `LONGREAL`.

Example:

```
IMPORT BCD;
VAR a, amount : BCD;
BEGIN a := 123.45; amount := a * 1.05; WRITE(stdOut, amount) END.
```

4.6.5 Standard Library Defined Complex Number ADTs

The standard library provides complex number ADTs `COMPLEX` and `LONGCOMPLEX`, whose semantics conform to prototype ~~`CTYPE`~~`ProtoComplex`.

Example:

```
IMPORT COMPLEX;

VAR z, zsum : COMPLEX;

BEGIN z := { 1.23, 4.56 }; zsum := z + { 1.0, 0.5 }; WRITE(stdOut, zsum) END.
```

4.6.6 Standard Library Defined Character Set ADTs

The standard library provides a character set ADT `CHARSET`, whose semantics conform to prototype ~~`SetType`~~`ProtoOrderedSet`.

Example:

```
IMPORT CHARSET;

VAR delimiters : CHARSET; counter : CARDINAL;

BEGIN delimiters := { ":", ",", "." }; counter := 0;

FOR char IN "foo:bar.baz,bam" DO
    IF char IN delimiters THEN counter++ END
END END.
```

4.6.7 Standard Library Defined Character String ADTs

The standard library provides two dynamic string ADTs `STRING` and `UNISTRING`, whose semantics conform to prototype ~~`StringType`~~`ProtoString`.

Example:

```
IMPORT STRING;

VAR s : STRING;

BEGIN
    NEW(s, 20);
    s := "quick brown fox";
    WRITE(stdOut, s);
    RELEASE(s)
END.
```

4.6.8 Standard Library Defined DateTime ADTs

The standard library provides two date-time ADTs `DateTime` and `Time` that conform to prototype ~~`DateType`~~`ProtoDateTime`.

Example:

```
IMPORT DateTime;

VAR date, diff : DateTime;

BEGIN date := { 1979, Month.Oct, 31, 0, 0, 0.0 };
    diff := date - { 1970, Month.Jan, 1, 0, 0, 0.0 }; WRITE(stdOut, diff) END.
```


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 it may not be defined or declared as an alias of a module, a variable, a type or a procedure.

EBNF:

```
constDefinition :
    CONST ( ( "[" bindableIdent "]" )? Ident "=" constExpression ";" )* ;

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

Examples:

```
CONST zero = 0;

CONST maxInt = TMAX(INTEGER);
```

5.2 Variable Definitions and Declarations

A variable is an entity whose value is determined at runtime and may change at runtime. A variable always has a type which is determined at compile time.

EBNF:

```
varDefinition :
    VAR identList : ( ARRAY constComponentCount OF )? namedType ";" )* ;

varDeclaration : varDefinition ;
```

Examples:

```
VAR x, y : REAL;

VAR str100 : ARRAY 100 OF CHAR;
```

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 within modules that import it.

A variable that is 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.

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 | 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.

Example:

```
TYPE INT = ALIAS OF INTEGER;

VAR i : INT; j : INTEGER;

i := j; (* i and j are compatible *)
```


5.3.3 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

5.3.3.1 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 *)
```

5.3.3.2 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 *)
```

5.3.4 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, then compatibility cannot be determined.

For this reason, anonymous types are of very limited use in languages with name equivalence. However, in order to facilitate the construction of VLAs, Modula-2 R10 permits anonymous one-dimensional arrays within indeterminate record field declarations. Any other use of anonymous types shall result in a compile time error.

5.3.5 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 ) | ident )
        ( "," ( ( "+" namedType ) | ident ) )* ")" ;
```

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

Example:

```
TYPE Colour = ( red, green, blue );
VAR colour : Colour;
colour := Colour.green; (* qualified identifier of value green *)
```


The list of identifiers that define the legal values of an enumeration type may contain references to other enumeration types. 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.

Example:

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

5.3.6 Array Types

5.3.6.1 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 must be of an unsigned whole number type. The number of components cannot be zero.

Array types are defined using the `ARRAY` type constructor.

EBNF:

```
indexedArray :
    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(stdOut, array);
```

5.3.6.2 Associative Array Types

An associative array type is a dynamic collection type for an arbitrary number of key/value pairs. Its keys are of type `ARRAY OF CHAR`, its values are of arbitrary type. All values have the same type. Associative array types are defined using the `ASSOCIATIVE ARRAY` type constructor.

EBNF:

```
associativeArray :
    ASSOCIATIVE ARRAY OF namedType ;
```

Example:

```
TYPE AA = ASSOCIATIVE ARRAY OF INTEGER;
VAR array : AA;
NEW(array); array["foo"] := 0; array["bar"] := -123; array["baz"] := 456;
```

5.3.7 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.

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;
```

5.3.8 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.

5.3.8.1 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 *)
```

5.3.8.2 Allocating Indeterminate Records

Records of an indeterminate type may only be allocated dynamically at runtime using pervasive 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 the value of its allocation size without the size of 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;
```

5.3.8.3 Immutability of the Discriminant Field

The discriminant field of a record of indeterminate type is automatically initialised when it is allocated. After initialisation the discriminant field becomes immutable and the compiler enforces its immutability 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 -- *)
```

5.3.8.4 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.

5.3.8.5 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 SYSTEM.BYTE` or `CAST ARRAY OF SYSTEM.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.

5.3.8.7 Deallocating Indeterminate Records

Records of indeterminate type may only be deallocated using pervasive procedure `DISPOSE`.

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

Two kinds of set types are provided by built-in facilities: bitset types and enumerated set types.

5.3.9.1 Bitset Types

A bitset type is a static ordered set type whose value space is defined by the subrange type `[0 .. 8*TSIZE(BitsetType)-1] OF CARDINAL`. The values are called bits. A bit whose membership in a bitset is `TRUE` is said to be *set*, a bit whose membership is `FALSE` is said to be *cleared*.

Two bitset types are language predefined: `BITSET` and `LONGBITSET`.

Examples:

```
VAR bitset : BITSET; (* declaring a bitset variable *)
bitset := { 0, 1, 2, 7 }; (* assigning a literal value *)
bitset[3] := TRUE; bitset[3] := FALSE; (* setting and clearing a bit *)
bool := bitset[4]; IF 4 IN bitset THEN ... (* testing a bit *)
FOR bit IN bitset DO bitCounter++ END; (* iterating over set bits *)
```

5.3.9.2 Enumerated Set Types

An enumerated set type is a static ordered set type whose value space is defined by an enumeration type. Enumerated set types are defined using the `SET OF` type constructor.

EBNF:

```
setType :
    SET OF ( enumTypeIdent | "(" identList ")" ) ;
enumTypeIdent : typeIdent ;
```

Example:

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

5.3.10 Pointer Types

A pointer type is a container for a reference to a storage location of given type. The type of the storage location 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 ImmutablePtr = POINTER TO CONST INTEGER;
VAR intPtr: IntPtr; immPtr : ImmutablePtr; int : INTEGER;
intPtr := int; immPtr := int; int := 0;
intPtr^ := 0; (* OK, modifying a mutable entity *)
immPtr^ := 0; (* compile time error: attempt to modify an immutable entity *)
```

5.3.11 Procedure Types

A procedure type is a special container type for 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 );
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 pervasive procedure, the procedure's formal parameter list and its return type.

A procedure header may only define a binding to an operator or pervasive 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 pervasive procedure must conform to the language defined signature for the respective operator. A list of procedure signatures is provided in documentation module `BINDINGS.def`.

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 any local variable declarations, any 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 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

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 the pass-by-value convention is used, a copy of the value parameter is passed to the called procedure and the scope of the copy is the procedure's ~~body~~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 by mutable-reference is called a `VAR` parameter. When the pass-by-mutable-reference convention is used, a mutable reference to the parameter is passed to the called procedure and the procedure may modify the value of the passed-in variable.

Immutable entities may not be passed by mutable-reference.

Example:

```
PROCEDURE Increment ( VAR x : INTEGER );
(* VAR => pass-by-reference, mutable *)
BEGIN
    x++; (* modifies original *)
    RETURN
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 `SYSTEM` 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 `VARIADIC`.

EBNF:

```
variadicFormalParams :
    VARIADIC ( "[" variadicTerminator "]" )? OF
    ( simpleFormalType |
      "{" simpleFormalParams ( ";" simpleFormalParams )* "}" ) ;
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 : VARIADIC 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. Pervasive function `COUNT` returns the value of the variadic counter.

Example:

```
PROCEDURE Average ( v : VARIADIC 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 : VARIADIC [terminator] OF INTEGER );
```

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 : VARIADIC [terminator] OF { key : Str; val : REAL } );
```

Within the procedure, pervasive function `NEXTV` may be used to traverse the list of arguments in the variadic list. Each time `NEXTV` is called it returns a pointer to the next argument of the variadic argument list or `NIL` if the end of the argument list has been reached.

Example:

```
CONST terminator = -1; (* variadic list terminator *)
PROCEDURE Variadic ( v : VARIADIC [terminator] OF INTEGER );
VAR item : POINTER TO INTEGER;
BEGIN
    item := NEXTV(v);
    WHILE item # NIL DO
        WRITE(f, item^); item := NEXTV(v)
    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;
                   v : VARIADIC 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; v : VARIADIC OF { key : Key; value : 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 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. Pervasive 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;
```

5.4.3.11 Auto-Casting Open Array Parameters

A formal parameter may be declared as an auto-casting open array parameter with component type `OCTET`, `SYSTEM.BYTE` or `SYSTEM.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 `SYSTEM` to declare an auto-casting formal parameter.

Example:

```
FROM SYSTEM 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 pervasive procedure in respect of an abstract data type defined to conform to a prototype. Except for bindings to the conversion operator which are always permitted, only bindings required by the prototype the ADT conforms to may be defined.

Example:

```
(* Module BCD is required to conform to prototype 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.

EBNF:

```
assignment : designator ":=" expression ;

designator : qualident ( ( "[" expressionList "]" | "^" ) ( "." 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 Post-Increment and Post-Decrement Statements

A post-increment adds one to, a post-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. Calls may be recursive, that is, a procedure may call itself.

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(stdOut, "Positive");
ELSIF i = 0 THEN WRITE(stdOut, "Zero");
ELSE WRITE(stdOut, "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(stdOut, "Red");
|   Colour.green : WRITE(stdOut, "Green");
|   Colour.blue  : WRITE(stdOut, "Blue");
ELSE
    SYSTEM.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 or sequence of statements indefinitely unless explicitly terminated by an `EXIT` statement.

EBNF:

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

Example:

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

6.9 FOR Statements

The `FOR` statement is used to repeatedly execute a statement or statement sequence a given number of times, depending on a control variable. The control variable is declared in the loop header and its scope is the loop. It no longer exists after the loop has exited. The control variable is treated as a mutable variable inside the loop header and as an immutable variable or runtime constant 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 to any procedure as a `VAR` parameter
- it may not be assigned to any pointer other than a `POINTER TO CONST` pointer

6.9.1 FOR IN Statements

`FOR IN` statements iterate over all values of an ordered value set. By default the order is ascending. If the `DESCENDING` attribute is specified, the order is descending. The value set may be a designator of an ordinal or whole number type, or a subrange of an ordinal or whole number type, or an array, a set, or an ordered collection. The control variable is assigned a value of the value set at each iteration and its type therefore depends on that of the value set:

- If the value set is an ordinal or whole number type or a subrange thereof then the control variable is of the ordinal or whole number type and the loop will iterate over all legal values of the ordinal or whole number type.
- If the value set is an array then the control variable is of the component type of the array and the loop will iterate over all the components of the array.
- If the value set is a set then the control variable is of the element type of the set and the loop will iterate over all the elements present in the set.
- If the value set is an ordered collection then the control variable is of the key type of the collection and the loop will iterate over all the keys present in the collection.

EBNF:

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

controlVariable : Ident;
```

Examples:

```
(* iterating over all values in a list of values of a common type *)
FOR char OF CHAR IN {"a".."z", "A".."Z", "0".."9"} DO WRITE(file, char) END;

(* iterating over all values of a subrange type *)
FOR i IN [0..9] OF CARDINAL DO array[2*i+1] := foo END;

(* iterating over all values of a whole number type *)
FOR number IN CARDINAL DO BottlesOfBeer(number) END;

(* iterating over all values of an ordinal type *)
FOR colour IN Colours DO WRITE(file, NameOfColour(colour)) END;

(* iterating over all elements of a set *)
FOR colour IN ColourSet DO counter++ END;

(* iterating over all components of an array *)
FOR char IN string DO WRITE(file, char) END;

(* iterating over all keys in a collection *)
FOR key IN dictionary DO WRITE(file, dictionary[key]) END;
```

6.10 EXIT Statements

An `EXIT` statement in the body of a `WHILE`, `REPEAT`, `LOOP` or `FOR` 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` 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. A value may be returned.

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, operators and optional parentheses. Operands may be constant or variable operands. An operand that is also an expression is called a sub-expression. Pairs of matching parentheses may be used in an expression to control the order in which its sub-expressions are evaluated.

EBNF:

```
expression :
    simpleExpression ( relation simpleExpression )? ;

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

term :
    factor ( mulOperator factor )* ;

factor :
    ( NumberLiteral | StringLiteral | structuredValue |
      designatorOrProcedureCall | "(" expression ")" |
      CAST "(" namedType "," expression ")" )
    ( "::" namedType )? | NOT factor ;

relation : relationalOperator ;
```

Examples:

```
VAR x, y : CARDINAL; i : INTEGER; r : REAL;
x + y * 5; NOT (ORD(x) > 10); { r, 1.5 }; i :: REAL;
```

An expression in which only constant operands are permitted is called a constant expression. Constant expressions are always evaluated at compile time.

EBNF:

```
constExpression :
    simpleConstExpr ( relation simpleConstExpr )? ;

simpleConstExpr :
    ( "+" | "-" )? constTerm ( addOperator constTerm )* ;

constTerm :
    constFactor ( mulOperator constFactor )* ;

constFactor :
    ( NumberLiteral | StringLiteral | constQualident | constStructuredValue |
      "(" constExpression ")" | CAST "(" namedType "," constExpression ")" )
    ( "::" namedType )? | NOT constFactor ;
```

Examples:

```
3.1415926; "foobar"; 1 + 3 * 5; NOT (TSIZE(T) > 10); { 0, 1.5 }; 100 :: REAL;
```

7.1 Operands

Operands are denoted by literals, designators or sub-expressions. Designators consist of an identifier that refers to a constant, a variable or a function procedure, followed by an optional designator tail that consists of one or more selectors. The designator's identifier may be qualified. A selector may denote the index of an array, the element of a set, the key of a collection, the dereference symbol following a pointer, a field of a record or a procedure's actual parameter list. An actual parameter list is a list of expressions separated by commas, enclosed in parentheses. The list of expressions within an actual parameter list may be empty.

EBNF:

```
designatorOrProcedureCall :
    qualident designatorTail? actualParameters? ;

actualParameters :
    "(" expressionList? ")" ;

designator :
    qualident designatorTail? ;

designatorTail :
    ( ( "[" expressionList "]" | "^" ) ( "." ident )* )+ ;

expressionList :
    expression ( "," expression )* ;
```

Examples:

```
array[index], multiDimArray[i, j, k], pointer^, record.field, write("a")
```

7.2 Operators

Operators are special symbols or reserved words that represent an operation. Each operator has an associated precedence level that determines the order of evaluation for expressions consisting of multiple sub-expressions with diverse operations. Operators with higher precedence are evaluated before operators with lower precedence. There are five different precedence levels:

- level 1: `::`
- level 2: `NOT`
- level 3: `*`, `/`, `DIV`, `MOD`, `AND`
- level 4: `+`, `-`, `OR`
- level 5: =, #, <, <=, >, >=, `IN`

Level 1 represents highest precedence, level 5 represents lowest precedence.

EBNF:

```
typeConversionOperator : "::" ;

notOperator : NOT ;

mulOperator : "*" | "/" | DIV | MOD | AND ;

addOperator : "+" | "-" | OR ;

relationalOperator : "=" | "#" | "<" | "<=" | ">" | ">=" | IN ;
```

7.3 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 Pervasive Identifiers

Pervasive identifiers are predefined identifiers that are available in every module scope of a program without having to import them. Pervasive identifiers are reserved identifiers, they may not be redefined. There are four groups of pervasive identifiers:

- 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 twelve predefined types:

`BOOLEAN`	boolean type
`BITSET`	bitset type of same size as `CARDINAL`
`LONGBITSET`	bitset type of same size as `LONGCARD`
`CHAR`	7-bit character type, subset of UTF-8
`UNICHAR`	4-octet character type, full UTF-32 set
`OCTET`	unsigned 8-bit integer type, smallest unit
`CARDINAL`	unsigned integer type, $2n$ octets for $n \geq 1$
`LONGCARD`	unsigned integer type, $2n$ octets for $n \geq 1$
`INTEGER`	signed integer type of same size as `CARDINAL`
`LONGINT`	signed integer type of same size as `LONGCARD`
`REAL`	real number type with implementation defined precision
`LONGREAL`	real number type with a precision equal to or higher than that of `REAL`

Although these types are predefined, their IO operations are not. The IO operations corresponding to `READ`, `WRITE` and `WRITEF` for pervasive types are provided in the standard library and need to be imported to become available. For each pervasive type, the standard library provides one library module whose module identifier matches the type identifier of its corresponding type. Library modules may therefore reuse pervasive type identifiers as their module identifiers. Reuse of a pervasive type identifier as a module identifier does not redefine the corresponding type and it does not change its pervasiveness.

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 calls to them are replaced by the compiler with a predefined statement or statement sequence or a call to a corresponding library procedure. There are five predefined procedures:

```
NEW DISPOSE READ WRITE WRITEF
```


8.3.1 Procedure NEW

Procedure `NEW` is used to dynamically allocate 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`.

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 DISPOSE

Procedure `DISPOSE` is used to deallocate storage that was earlier allocated by a call to procedure `NEW`. Its pseudo-definition is:

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

A call to procedure `DISPOSE` is replaced by the compiler with a call to library procedure `DEALLOCATE` which must be imported before `DISPOSE` can be used. The standard library provides a procedure `DEALLOCATE` in module Storage. Procedure `DISPOSE` always requires a single parameter only.

Examples:

```
DISPOSE(fooptr); (* replaced by DEALLOCATE(fooptr, TSIZE(Foo)); *)
DISPOSE(v); (* replaced by
               DEALLOCATE(v, TSIZE(VLA) + v^.items*TSIZE(INTEGER)); *)
```

8.3.3 Procedure RETAIN

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

```
PROCEDURE ~~DISPOSE~~RETAIN ( VAR p : <AnyPointerType> );
```

A call to procedure `RETAIN` is replaced by the compiler with a call to the procedure that has been bound to pervasive procedure `RETAIN` for the type of its argument.

Examples:

```
VAR str : STRING;~~DISPOSE~~
NEW(str); ... RETAIN(~~fooptr~~str); (* replaced by ~~DEALLOCATE(fooptr~~STRING.~~,~~
```

8.3.4 Procedure RELEASE

Procedure `RELEASE` is used to release a variable of a reference counted type. When a variable is released it cancels a previous `RETAIN`. When all prior calls to `RETAIN` have been canceled then the variable is automatically disposed by calling procedure `DISPOSE`. Its pseudo-definition is:

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

A call to procedure `RELEASE` is replaced by the compiler with a call to the procedure that has been bound to pervasive procedure `RELEASE` for the type of its argument.

Examples:

```
~~NEW(str); ... RETAIN~~RELEASE(str); (* replaced by STRING.re~~tain~~lease(str); *)
```

8.3.5 Procedure READ

Procedure `READ` is used to read 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 pervasive type in a corresponding module. The IO modules for all pervasive types may be imported at once by importing aggregator module `PervasiveIO`.

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 PervasiveIO;
VAR n : CARDINAL;
READ(stdIn, n); (* replaced by CARDINAL.Read(stdIn, n); *)

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

8.3.6 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 pervasive type in a corresponding module. The IO modules for all pervasive types may be imported at once by importing aggregator module `PervasiveIO`.

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 PervasiveIO;
VAR n : CARDINAL;
WRITE(stdOut, n); (* replaced by CARDINAL.Write(stdOut, n); *)

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

8.3.7 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 : VARIADIC 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 pervasive type in a corresponding module. The IO modules for all pervasive types may be imported at once by importing aggregator module `PervasiveIO`.

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 PervasiveIO;
VAR n1, n2, n3 : CARDINAL;
WRITEF(stdOut, "", n1, n2, n3); (* replaced by
                                   CARDINAL.WriteF(stdOut, "", n1, n2, n3); *)

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

8.4 Predefined Functions

Predefined functions act and look like library defined functions but they may not be assigned to procedure variables, may not be passed to a procedure as parameters and calls to them are typically replaced by the compiler with an expression rather than a call to a corresponding function. There are 15 predefined functions:

```
ABS NEG ODD PRED SUCC ORD CHR COUNT LENGTH SIZE NEXTV 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 always the same as 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 always the same as 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 ~~Z-Type~~whole number type. Its return type is the boolean type. Its pseudo-definition is:

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

8.4.4 Function PRED

Function `PRED` returns the n-th predecessor of its first operand where n is the second operand. Its first operand may be of any ordinal type and its second operand may be of any unsigned type. Its return type is always the same as the type of its first operand. Its pseudo-definition is:

```
PROCEDURE PRED ( x : <OrdinalType>; n : <UnsignedType> ) : <typeOf(x)> ;
```

8.4.5 Function SUCC

Function `SUCC` returns the n-th successor of its first operand where n is the second operand. Its first operand may be of any ordinal type and its second operand may be of any unsigned Z-Type. Its return type is always the same as the type of its first operand. Its pseudo-definition is:

```
PROCEDURE SUCC ( x : <OrdinalType>; n : <UnsignedType> ) : <typeOf(x)> ;
```

8.4.6 Function ORD

Function `ORD` returns the ordinal value of its operand. Its operand may be of any ordinal type. Its return type is the Z-Type. Its pseudo-definition is:

```
PROCEDURE ORD ( x : <OrdinalType> ) : <Z-Type> ;
```

8.4.7 Function CHR

Function `CHR` returns the character whose code point is its operand. Its operand may be of any unsigned ~~Z-Type~~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 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 variadic parameter or variadic parameter list. Its return type is `LONGCARD`. Its pseudo-definition is:

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

8.4.9 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 : <CharacterStringArrayOrStringADT> ) : LONGCARD ;
```

8.4.10 Function SIZE

Function `SIZE` returns the allocation size of its operand. The value returned represents the number of octets allocated for its operand. Its operand may be a variable of any type. Its return type is `LONGCARD`. Its pseudo-definition is:

```
PROCEDURE SIZE ( variable : <AnyType> ) : LONGCARD ;
```

~~8.4.10 Function HIGH~~

~~Function HIGH returns the highest subscript of its operand. Its operand may be a variable of an indexed array type, or the identifier of an open array parameter. Its return type is LONGCARD. Its pseudo-definition is:~~

```
PROCEDURE HIGH ( ident : <ArrayOrOpenArray> ) : LONGCARD ;
```

8.4.11 Function NEXTV

Function `NEXTV` returns a pointer to the next item in a value-terminated variadic parameter list or `NIL` if the end of the parameter list has been reached. Its operand is an identifier of a variadic parameter of a value-terminated variadic parameter list. Its return type is a pointer to the type of its operand. Its pseudo-definition is:

```
PROCEDURE NEXTV ( ident : <VariadicParam> ) : <PointerToVariadicParam> ;
```

8.4.12 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> ;
```

8.4.13 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> ;
```


8.4.14 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 ;
```

8.4.15 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 numeric type into the equivalent value of another numeric type. A library that defines and implements a numeric type as an abstract data type may provide one or more conversion procedures to convert values of its type to other numeric types. When such a conversion procedure is defined, a direct conversion path exists from the source type to the target type. However, it is impractical to provide conversion procedures for all possible conversion targets as the number of conversions grows exponentially with the number of types.

In order to be able to support conversions from any numeric type to any other numeric type without having to define and implement conversions for all possible conversion targets, a language defined intermediate representation of scalar values is provided. This intermediate representation is called Scalar Exchange Format, or SXF. For any two numeric types `T1` and `T2` that provide conversions to and from SXF, values of `T1` can be converted to values of `T2` and vice versa even if no direct conversion path exists between `T1` and `T2`. This is called an intermediate conversion path.

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

9.1 Scalar Exchange Format

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 exponent, valid range 0 to 63 *)

digitRadix :
    "D" | "H" ; (* digit radix, D for base 10, H for base 16 *)

sigSign :
    "+" | "-" ; (* sign of the significand *)

sigDigits :
    digitB10+ | digitB16+ ; (* digits of the significand *)

expSign :
    "+" | "-" ; (* sign of the exponent *)

expDigits :
    digitB10+ | digitB16+ ; (* digits of the exponent *)

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) ;
```


9.2 Built-in Scalar Conversion Primitives

Module `CONVERSION` provides an interface to two built-in scalar conversion primitives, used internally to convert values of built-in numeric types to and from scalar exchange format as an intermediate step when converting to and from library defined numeric types if no direct conversion path exists.

9.2.1 SXF Version

Module `CONVERSION` provides a constant indicating the version of the scalar exchange format used:

`SXFVersion`	whole number constant denoting the SXF version used by the compiler

9.2.2 Macro SXFSizeForType

An invocation of macro `SXFSizeOfType` is replaced by the allocation size in octets required to represent values of the given numeric type in scalar exchange format. An invocation of this macro results in a compile time expression.

Pseudo Definition:

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

9.2.3 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, an overflow runtime error is raised.

9.2.4 Primitive VAL

The `VAL` primitive converts a value in scalar exchange format to an equivalent 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, an overflow runtime error is raised.

10 Interfaces to Compiler and Runtime System

Modula-2 R10 provides facilities to interface and communicate with the compiler and its runtime system. 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 constants to indicate support for optional capabilities. A constant with a value of `TRUE` indicates support while a value of `FALSE` indicates no support:

~~`SupportsForToByLoop`~~	~~indicating availability of the `FOR TO BY` loop statement~~
`SupportsInlineAssembler`	indicating availability of pseudo-module `ASSEMBLER`
`SupportsUTF8EncodedSource`	indicating availability of support for UTF8 encoded source
`SupportsCFFI`	indicating availability of a foreign function interface to C
`SupportsFortranFFI`	indicating availability of a foreign function interface to Fortran
`SupportsAlignmentControl`	indicating availability of pragma `ALIGN`
`SupportsBitPadding`	indicating availability of pragma `PADBITS`
`SupportsAddressMapping`	indicating availability of pragma `ADDR`
`SupportsRegisterMapping`	indicating availability of pragma `REG`
`SupportsPurityAttribute`	indicating availability of pragma `PURITY`
`SupportsVolatileAttribute`	indicating availability of pragma `VOLATILE`

Implementations may not add any implementation defined capability constants to pseudo-module `COMPILER`. To allow testing of implementation defined extensions and pragmas, an implementation shall provide boolean capability pragmas instead. For consistency, the names of such pragmas shall follow the naming convention of capability constants and be prefixed with `Supports`.

10.1.3 Information About The Implementation Model of REAL and LONGREAL

Module `COMPILER` provides an enumeration type and two constants to provide information about the implementation defined implementation model of pervasive 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`

10.1.4 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
`LINENO`	expands to a whole number constant representing the line number being compiled

10.1.5 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.

10.1.5.1 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;
```

10.1.5.2 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;
```

10.1.5.3 Macro IsExtensionOfType

An invocation of macro `IsExtensionOfType` is replaced by boolean value `TRUE` if the identifier passed as its first argument is a type extension of the type passed as its second argument. Otherwise 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;
```

10.1.5.4 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 pervasive 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;
```


10.1.5.5 Macro ConformsToPrototype

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

Pseudo Definition:

```
PROCEDURE ConformsToPrototype ( <qualident>; <Prototype> ) : Boolean;
```

10.1.6 Smallest And Largest Value Within A List Of Constants

Module `COMPILER` provides two lexical macros to determine the smallest and largest value within a list of constants during compile time. The macros only accept argument lists of constants or constant expressions of an enumerated type or of a pervasive whole number or real number 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 pervasive whole number or real number type

10.1.6.1 Macro MIN

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

Pseudo Definition:

```
PROCEDURE MIN ( constant : VARIADIC 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
```

10.1.6.2 Macro MAX

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

Pseudo Definition:

```
PROCEDURE MAX ( constant : VARIADIC 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
```

---

[1] It should be noted that library defined conversions may not be used within constant expressions.

10.1.7 Powers Of Two Of An Unsigned Number Constant

Module `COMPILER` provides a lexical macro to calculate powers of two ~~during~~at compile time.

10.1.7.1 Macro EXP2

An invocation of macro `EXP2` is replaced by two raised to the power of its argument. Its argument is an unsigned whole number constant or constant expression.

Pseudo Definition:
```
PROCEDURE EXP2 ( n : <UnsignedPervasiveType> ) : <OperandType>;
```

10.1.8 Built-in Hash Function

Module `COMPILER` provides an interface to the compiler's built-in hash function. The underlying hash algorithm is implementation dependent. It may also differ between different releases of the same implementation. Hash values may therefore differ between implementations and releases.

10.1.8.1 Function HASH

Function `HASH` calculates and returns the hash value of a string passed as its argument, using the compiler's built-in hash function. A call to this function with a string literal passed as parameter results in a compile time expression. The enclosing quotation marks of the string literal do not have any influence on the calculated hash value because they are not part of the string.

```
PROCEDURE HASH ( CONST s : ARRAY OF CHAR ) : LONGCARD;
```


10.2 Pseudo-Module RUNTIME

Pseudo-module `RUNTIME` provides an interface to the runtime system.

10.2.1 Runtime Exceptions

Runtime system exceptions are defined as an enumerated type `Exception`. Its definition is:

```
TYPE Exception = ( DivByZero, DerefNil, TypeOverflow, IndexOutOfBounds,
                   StringCapacityExceeded, StackOverflow, OutOfMemory,
                   UserAbort );
```

10.2.1.1 Raising Runtime Exceptions

Module `RUNTIME` provides procedure `RaiseError` to raise a runtime system exception of type `e`. This procedure never returns. Its definition is:

```
PROCEDURE RaiseError ( e : Exception );
```

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 notification handler 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 ( SYSTEM.ADDRESS, SYSTEM.ADDRESS );
```

10.2.2.1.1 Installing a Notification Handler

Procedure `InstallNotificationHandler` installs a user-defined notification handler `p` for runtime exceptions of type `e`. Its definition is:

```
PROCEDURE InstallNotificationHandler ( e : Exception;
                                       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 three mandatory and one optional low-level pseudo-modules:

- pseudo-module `SYSTEM`
- pseudo-module `ATOMIC`
- pseudo-module `COROUTINE` (Phase II)
- pseudo-module `ASSEMBLER` (optional)

11.1 Pseudo-Module SYSTEM

Pseudo-module `SYSTEM` provides implementation- and target-dependent facilities.

11.1.1 SYSTEM Constants

Module `SYSTEM` provides four whole number constants > 0 pertaining to byte and word sizes:

`OctetsPerByte`	implementation defined size of a byte
`BytesPerWord`	implementation defined size of a word
`BitsPerMachineByte`	target dependent size of a machine byte
`MachineBytesPerMachineWord`	target dependent size of a machine word

Module `SYSTEM` provides the following constants pertaining to endianness and addressing:

`TargetName`	target dependent string with the name of the target architecture
`BigEndian`	3-2-1-0 byte order, also known as big endian
`LittleEndian`	0-1-2-3 byte order, also known as little endian
`BigLittleEndian`	2-3-0-1 byte order, also known as big-little endian
`LittleBigEndian`	2-1-0-3 byte order, also known as little-big endian
`TargetByteOrder`	enumerated value indicating byte order of the target architecture
`TargetIsBiEndian`	target dependent boolean value, `TRUE` if target is bi-endian, otherwise `FALSE`
`ByteBoundaryAddressing`	target dependent boolean value, `TRUE` if every machine byte is addressable, otherwise `FALSE`
`MaxWordsPerOperand`	implementation defined whole number value ≥ 4 denoting the size of the largest supported operand in system intrinsics

11.1.2 SYSTEM Types

Module `SYSTEM` provides the following system 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 `SYSTEM`, their respective IO operations are not. The IO operations corresponding to `READ`, `WRITE` and `WRITEF` for `SYSTEM` types are provided by the standard library and need to be imported to become available. For any target architecture where the bit width of a machine byte is eight, `BYTE` and `MACHINEBYTE` are `ALIAS` types of type `OCTET`.

11.1.2.1 System 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 System Type WORD

`WORD` is a system type whose size is implementation defined. Its pseudo-definition is:

```
TYPE WORD = OPAQUE RECORD value : ARRAY BytesPerWord OF SYSTEM.BYTE END;
```

11.1.2.3 System 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 System Type MACHINEWORD

`WORD` is a system type whose size is target dependent value representing the actual word size of a machine word of the target architecture. Its pseudo-definition is:

```
TYPE MACHINEWORD = OPAQUE RECORD
    value : ARRAY MachineBytesPerMachineWord OF SYSTEM.MACHINEBYTE END;
```

11.1.2.5 System Type ADDRESS

`ADDRESS` is a system type representing references to memory locations. Its pseudo-definition is:

```
<* IF ByteBoundaryAddressing *> TYPE ADDRESS = POINTER TO SYSTEM.MACHINEBYTE;
<* ELSE *> TYPE ADDRESS = POINTER TO SYSTEM.MACHINEWORD; <* ENDIF *>
```

11.1.3 SYSTEM Intrinsics

`SYSTEM` 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>;
```

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 Mapping to Unsafe Variadic Procedures in Foreign APIs

Module `SYSTEM` provides two pseudo-types to facilitate the definition of formal parameter lists for mapping to unsafe variadic parameter lists in foreign API functions:

`UNSAFEARGLIST`	pseudo-type to define formal parameter for unsafe variadic parameter
`ARGCOUNT`	pseudo-type to define formal parameter for variadic argument counter

11.1.4.1 Pseudo-Type UNSAFEARGLIST

Pseudo-type `UNSAFEARGLIST` may be used as a formal type within the formal parameter list of a procedure definition in order to define an unsafe variadic parameter. An unsafe variadic parameter is a formal parameter to which any number of actual parameters of any type may be passed and whose actual parameters are not type checked.

Example:
```
FROM SYSTEM IMPORT UNSAFEARGLIST;
PROCEDURE printf( fmt : ARRAY OF CHAR; args : UNSAFEARGLIST ) <* FFI="C" *>;
```

`UNSAFEARGLIST` may only be used as the formal type of the last parameter in a formal parameter list of a procedure definition. Any other uses of this pseudo-type will result in a compile-time error.

11.1.4.2 Pseudo-Type ARGCOUNT

Pseudo-type `ARGCOUNT` may be used as a formal type within the formal parameter list of an unsafe variadic procedure definition in order to define an automatic variadic counter. An automatic variadic counter is a formal parameter to which the compiler automatically passes the number of actual parameters passed for a formal variadic parameter of a variadic procedure.

Example:
```
FROM SYSTEM IMPORT ARGCOUNT, UNSAFEARGLIST;
PROCEDURE wrt( c : ARGCOUNT; f : FILE; args : UNSAFEARGLIST ) <* FFI="C" *>;
```
Invoked as:
```
wrt( file, TRUE, 42, 1.23, 0u40, "foo" );
```
Compiled as:
```
wrt( 5, file, TRUE, 42, 1.23, 0u40, "foo" ); (* argument count inserted *)
```

`ARGCOUNT` may only be used as a formal type in a formal parameter list whose last parameter is defined as an `UNSAFEARGLIST` and it may only occur once in the formal parameter list. `ARGCOUNT` is a formal whole number type whose size, minimum and maximum values are target dependent. Its actual size, minimum and maximum values may be obtained using pervasive functions `TSIZE`, `TMIN` and `TMAX`. Any other uses of this pseudo-type will 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. Its definition is:

```
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 Pseudo-Module ASSEMBLER (optional)

Pseudo-module `ASSEMBLER` is an optional module that provides an inline-assembler facility for one or more target architectures.

11.5.1 Testing The Availability Of The Inline Assembler

The availability of module `ASSEMBLER` and its inline assembler facility is both implementation and target dependent. For the purpose of testing the availability of optional compile time capabilities module `COMPILER` provides boolean constant `SupportsInlineAssembler`. It may be used to test whether an inline assembler is available for the current target architecture.

Example:

```
IMPORT COMPILER;

<* IF NOT COMPILER.SupportsInlineAssembler *>
<* MSG=FATAL : “No inline assembler is available for this target.” *>
<* ENDIF *>
```

11.5.2 Mnemonics Of The Target Architecture

Module `ASSEMBLER` provides two enumeration types defining mnemonics of opcodes and registers of the target architecture:

`OPCODE`	target dependent enumeration of opcode mnemonics
`REGISTER`	target dependent enumeration of register mnemonics

Mnemonics are architecture dependent. The identifiers of enumeration elements are determined from lowercase transliterations of the target architecture’s official mnemonics. The element order is implementation defined.

Pseudo-Definition Example:

```
TYPE

(* Opcode mnemonics *)
    OPCODE = ( add, adc, and, ... );

(* Register mnemonics *)
    REGISTER = ( eax, ebx, ecx, edx, ... );
```

For convenience, the module also provides constants for the unqualified use of the mnemonics defined by types `OPCODE` and `REGISTER`:

- constants for unqualified opcode mnemonics
- constants for unqualified register mnemonics

Pseudo-Definition Example:

```
CONST
(* unqualified opcode mnemonics *)
    add = OPCODE.add;
    adc = OPCODE.adc;
    ...

(* unqualified register mnemonics *)
    eax = REGISTER.eax;
    ebx = REGISTER.ebx;
    ...
```


11.5.3 Inline Assembler Intrinsic CODE

Module `ASSEMBLER` provides a single intrinsic `CODE`.

Invocations of the intrinsic are translated by the compiler's target specific inline assembler into a sequence of inlined machine instructions generated from an instruction list specified by the actual parameters of the invocation.

Pseudo-Definition:

```
<*INLINE*> PROCEDURE CODE ( <instructionList> );
```

The instruction list may consist of opcode mnemonics, register mnemonics, constant expressions and variables. Its grammar is defined as:

EBNF:

```
instructionList :
    instruction ( "," instruction )* ;

instruction :
    opcode operandList ;

operandList :
    operand ( "," operand )* ;

operand :
    register | constant | variable ;

opcode : qualIdent ; (* any identifier defined by type OPCODE *)

register : qualIdent ; (* any identifier defined by type REGISTER *)

constant : constExpression ;

variable : qualIdent ;
```

The number of operands in an operand list is dependent on the target architecture.

Examples:

```
FROM ASSEMBLER IMPORT *;

CODE ( mov, eax, foobar );

CODE ( mov, ebx, 0xDEADBEEF );
```


12 Pragmas

Pragmas are directives to the compiler, used to control or influence the translation process, but they do not change the meaning of the program itself. Pragmas may be positional or non-positional. Positional pragmas may only appear at specific positions within the source text. Non-positional pragmas may appear before or after any token within the source text. Pragmas fall into four groups:

- language defined pragmas to emit compile time messages
- language defined pragmas to facilitate conditional compilation
- language defined pragmas to influence or control the compilation process
- implementation defined pragmas to influence or control the compilation process

12.1 Pragma to Emit Console Messages During Compile Time

A single pragma is provided to emit four different types of console messages during compilation:

- informational messages
- compilation warnings
- compilation error messages
- fatal compilation error messages

12.1.1 Pragma MSG

Pragma `MSG` is used to emit different types of user defined console messages during compilation. The pragma is non-positional. It may occur anywhere in the source text. A mode selector determines the message type. Console messages may consist of a quoted string literal, the value of a compile time constant, the value of a pragma that represents a compilation setting, or a combination thereof.

EBNF:

```
pMSG :
    “<*” MSG “=” ( INFO | WARN | ERROR | FATAL ) “:”
         compileTimeMsgComponent ( “,” compileTimeMsgComponent )* “*>” ;

compileTimeMsgComponent :
    String | constQualident | “?” ( ALIGN | ENCODING | implDefPragmaName ) ;
```

12.1.2 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.

Examples:

```
<* MSG=INFO : “Alignment is: ”, ?ALIGN *>

<* MSG=INFO : “Bounds checking is: ”, ?BoundsChecking *>

<* MSG=INFO : “Library documentation is available at http://foolib.com” *>
```

12.1.3 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.

Example:

```
<* MSG=WARN : “foo exceeds maximum value. A default of 100 will be used.” *>
```


12.1.4 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.

Example:

```
<* MSG=ERROR : "Value of foo is outside of its legal range of [1..100]." *>
```

12.1.5 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.

Example:

```
<* MSG=FATAL : "Unsupported target architecture." *>
```

12.2 Pragmas For Conditional Compilation

Conditional compilation pragmas may be 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 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 :
    pIF anySymbol* ( pELSIF anySymbol* )* ( pELSE anySymbol* )? pENDIF ;
```

Example:

```
<* IF TSIZE(INTEGER) = 2 *>
CONST model = Model.small;

<* ELSIF TSIZE(INTEGER) = 4 *>
CONST model = Model.large;

<* ELSIF TSIZE(INTEGER) = 8 *>
CONST model = Model.custom;

<* ELSE *>
<* FATAL "unsupported type model." *>
error := Errors.UnsupportedTypeModel;
SYSTEM.HALT(error);

<* 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 the maximum nesting level is exceeded. For better readability of source text it is recommended to avoid nesting whenever possible and keep the nesting depth to an absolute minimum when nesting cannot be avoided.

12.2.1 Pragma IF

Pragma `IF` denotes the start of the initial branch of a conditional compilation section. The source text within the initial branch of a conditional compilation section is only processed if the condition specified in the pragma denoting the start of the initial branch is true, otherwise it is ignored.

EBNF:

```
pIF : "<*" IF inPragmaExpression "*>" ;
```

This pragma causes the current conditional compilation nesting level to be incremented by one.

12.2.2 Pragma ELSIF

Pragma `ELSIF` denotes the start of an alternative branch within a conditional compilation section. The source text within an alternative branch of a conditional compilation section is only processed if the condition specified in the pragma denoting the alternative branch 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:

```
pELSIF : "<*" ELSIF inPragmaExpression "*>" ;
```

This pragma does not alter the current conditional compilation nesting level.

12.2.3 Pragma ELSE

Pragma `ELSE` denotes the start of a default branch within a conditional compilation section. The source text within the default branch of a conditional compilation section 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:

```
pELSE : "<*" ELSE "*>" ;
```

This pragma does not alter the current conditional compilation nesting level.

12.2.4 Pragma ENDIF

Pragma `ENDIF` denotes the end of a conditional compilation section.

EBNF:

```
pENDIF : "<*" ENDIF "*>" ;
```

This pragma causes the current conditional compilation nesting level to be decremented by one.

12.3 Pragmas To Control Code Generation

12.3.1 Pragma ENCODING

Pragma `ENCODING` may be used to specify the encoding of the source file and to verify the current encoding against the intended encoding. By default two encodings, `ASCII` and `UTF8` are recognised. Support for any other source file encodings is implementation dependent. Unrecognised encodings will cause a fatal compilation error and compilation will be aborted immediately.

The pragma controls whether any characters other than the printable characters of the US-ASCII character set are permitted within comments and quoted literals. In order to verify the current encoding against the intended encoding, the pragma may further specify a list of arbitrary samples with pairs of quoted characters and their respective code point values.

EBNF:

```
pENCODING :
    "<*" ENCODING "=" quotedStringLiteral ( ":" codePointSampleList )? "*>" ;

encodingControlPragma :
    "<*" ENCODING "=" quotedStringLiteral ( ":" codePointSampleList )? "*>" ;

codePointSampleList :
    codePointSample ( "," codePointSample )* ;

codePointSample :
    quotedCharacterLiteral "=" characterCodeLiteral ;
```

Examples:

```
<* ENCODING = "ASCII" *> (* force 7-bit ASCII only *)

<* ENCODING = "UTF8" : "é" = 0uE9, "©" = 0uA9, "€" = 0u20AC *>
```

When no BOM and no encoding pragma is present in the source file, only printable characters of the US-ASCII character set are permitted within comments and quoted literals. The printable characters of the US-ASCII character set are the characters whose code points are within the range of values `0u20` (whitespace) and `0u7E` (tilde). Any other characters will cause a compilation error.

When the encoding pragma is present, specifying `ASCII` as the encoding, the use of characters is forcefully restricted to printable characters of the US-ASCII character set, regardless of whether a UTF-8 BOM is present in the source file or not. Any other characters will cause a compilation error.

When a BOM and and an encoding pragma is present in the source file and a supported encoding other than `ASCII` is specified, the BOM is checked against the specified encoding. A mismatch will cause a fatal compilation error and compilation to abort immediately. If BOM and specified encoding match, printable characters of the character set associated with the encoding are permitted within comments and quoted literals. Any other characters will cause a compilation error.

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 will cause a fatal compilation error and compilation to abort immediately.

The encoding pragma is positional. If it is present in the source text, it must appear before any other token. There can only be one encoding pragma per source file. The maximum number of code point samples accepted within an encoding pragma is implementation defined but a value of at least 96 is recommended. Excess samples are ignored and cause a compile time warning.

12.3.2 Pragma GENLIB

Pragma `GENLIB` may be used to invoke the Modula-2 template engine to automatically generate the source files for a new library module from a library template during compilation of a library that wants to import the library to be generated. The Modula-2 template engine is described in detail in section 13 ("Generics"). The pragma is positional. It may only appear before an import statement.

EBNF:

```
pGENLIB :
    "<*" GENLIB moduleName "FROM" template ":" templateParamList "*>" ;
templateParamList :
    templateParam ( "," templateParam )*
templateParam :
    placeholder "=" replacement ;
moduleName : Ident ;
template : Ident ;
placeholder : Ident ;
replacement : String ;
```

Example:

```
DEFINITION MODULE FooApp;
<* GENLIB IntegerStack FROM Stack : componentType="INTEGER", maxSize="100" *>
IMPORT IntegerStack;
```

12.3.3 Pragma FFI

Pragma `FFI` may be used to specify the calling convention of procedures and procedure types to interface with programs and libraries written in other languages. Language defined specifiers for foreign function interfaces are `"C"` and `"FORTRAN"`.

Implementations that provide pragma `FFI` should always support C. Support for Fortran is optional. It is recommended that an implementation should emit a compile time error when it encounters an `FFI` pragma that specifies an unsupported foreign function interface.

The pragma is positional. Its position in the source text determines its scope and it may only appear at specific positions within a module header, a procedure definition or a procedure type declaration as defined by the EBNF grammar.

EBNF:

```
pFFI : "<*" FFI "=" foreignInterfaceName "*>" ;
foreignInterfaceName : String ;
```

Example 1:

```
(* Module scope use of FFI pragma *)
DEFINITION MODULE stdio <* FFI = "C" *>;
```

Example 2:

```
(* Procedure scope use of FFI pragma *)
PROCEDURE printf ( CONST fmt  : ARRAY OF CHAR;
                         args : UNSAFEARGLIST ) <* FFI = "C" *>;
```


12.3.4 Pragma INLINE

Pragmas `INLINE` may be used to influence the inlining of procedures. The `INLINE` pragma strongly suggests that inlining of a procedure is desirable. An informational message will be emitted if the suggestion is not followed. The pragma is positional. It may only appear after a procedure header within a procedure definition as defined by the EBNF grammar.

EBNF:

```
pINLINE : “<*” INLINE “*>” ;
```

Examples:

```
PROCEDURE Foo (bar, baz : CARDINAL) <* INLINE *>;
```

12.3.5 Pragma NOINLINE

Pragma `NOINLINE` mandates that a procedure may not be inlined. The pragma is positional. It may only appear after a procedure header within a procedure definition as defined by the EBNF grammar.

EBNF:

```
pNOINLINE : “<*” NOINLINE “*>” ;
```

Examples:

```
PROCEDURE Bar (baz, bam : CARDINAL) <* NOINLINE *>;
```

12.3.6 Pragma ALIGN

Pragma `ALIGN` may be used to set the alignment of variables and record fields. It specifies the alignment in octets. An alignment of zero specifies packing. The pragma is positional. Its position in the source text determines its scope and it may only appear at specific positions within a variable declaration section or a record type declaration as defined by the EBNF grammar.

EBNF:

```
pALIGN : “<*” ALIGN “=” inPragmaExpression “*>” ;
```

Examples:

```
VAR n : CARDINAL <* ALIGN = TSIZE(CARDINAL) * 2 *>;

TYPE Aligned = RECORD
   <* ALIGN = 2 *>
   foo, bar : INTEGER;           (* 16 bit aligned *)
   baz : INTEGER <* ALIGN = 4 *>; (* 32 bit aligned *)
   bam : INTEGER;                (* 16 bit aligned *)
END; (* Aligned *)
```


12.3.7 Pragma PADBITS

Pragma `PADBITS` may be used to insert padding bits into packed record type declarations. It specifies the number of padding bits to be inserted. The pragma is positional. It may only appear within packed record type declarations as defined by the EBNF grammar.

EBNF:

```
pPADBITS : "<*" PADBITS "=" inPragmaExpression "*>" ;
```

Examples:

```
TYPE Packed = RECORD
   <* ALIGN=0 *>
   foo : [0..1] OF OCTET; (* 1 bit *)
   <* PADBITS=2 *> (* unused 2 bits *)
   bar : [0..3] OF OCTET; (* 2 bits *)
   baz : [0..7] OF OCTET; (* 3 bits *)
END; (* Packed *)
```

12.3.8 Pragma ADDR

Pragma `ADDR` may be used to map procedures and global variables to fixed memory addresses. The pragma is positional. It may only appear at the end of a procedure definition or global variable declaration as defined by the EBNF grammar.

EBNF:

```
pADDR : "<*" ADDR "=" inPragmaExpression "*>" ;
```

Examples:

```
VAR memoryMappedPort : CARDINAL <* ADDR = 0x100 *>;
PROCEDURE Reset <* ADDR = 0x12 *>;
```

12.3.9 Pragma REG

Pragma `REG` may be used to map formal parameters in procedure definitions and procedure type declarations to machine registers. A mapped parameter will be passed in the register it is mapped to. The pragma is positional. It may only appear at specific positions within the formal parameter list of a procedure definition or procedure type declaration as defined by the EBNF grammar.

EBNF:

```
pREG : "<*" REG "=" ( registerNumber | registerMnemonic ) "*>" ;
registerNumber = Number;
registerMnemonic = String;
```

Examples:

```
TYPE FooProc = PROCEDURE (CARDINAL <* REG = 10 *>);
PROCEDURE Bar (baz : CARDINAL <* REG = REGISTER.eax *>);
```


12.3.10 Pragma PURITY

Pragma `PURITY` may be used to mark the intended purity level of a procedure as follows:

- level 0 : may read and modify global state, may call procedures of any level
- level 1 : may read but not modify global state, may call level 1 and level 3 procedures
- level 2 : may not read but modify global state, may call level 2 and level 3 procedures
- level 3 : may not read nor modify global state, may call level 3 procedures

The pragma shall cause the compiler to emit a warning if it detects any violation of the purity level.

EBNF:

```
pVOLATILE : “<*” PURITY “=” inPragmaExpression “*>” ;
```

Example:

```
PROCEDURE Foo ( a : Bar) : Baz <* PURITY=3 *>; (* pure and side-effect free *)
```

12.3.11 Pragma VOLATILE

Pragma `VOLATILE` may be used to mark a variable as volatile. The value of a volatile variable may change during the life time of a program but no write access to it can be deduced from source code analysis. Compilers may use this information during the optimisation phase, for example to avoid “optimising away” a variable that might otherwise have been considered unused.

EBNF:

```
pVOLATILE : “<*” VOLATILE “*>” ;
```

Example:

```
VAR foo : INTEGER <* VOLATILE *>;
```

12.4 Implementation Defined Pragmas

Implementation defined pragmas are compiler specific and non-portable. Their names must not be all-uppercase words. Implementation defined pragmas may be positional or non-positional. If an implementation defined pragma is not recognised by another implementation, the unrecognised pragma shall be ignored and a warning message shall be emitted.

An implementation defined pragma may either hold no value, boolean values or whole number values. Its value may be changed during compilation by assigning a new value. No value may be assigned to a pragma that is not defined to hold a value.

EBNF:

```
implementationDefinedPragma :
    "<*" ImplDefPragmaName ( "=" inPragmaExpression )? "*>" ;

ImplDefPragmaName : Ident ; (* lower case or mixed case *)
```

Examples:

```
<* UnrollLoops = FALSE *> (* turn loop unrolling off *)
<* UnrollLoops = TRUE *> (* turn loop unrolling back on *)
```


13 Generics

Modula-2 R10 does not provide any syntax for generic programming within the language itself. Instead, generic programming is supported via the Modula-2 Template Engine, or M2TE, a utility that is external to the compiler, invoked prior to compilation. The M2TE utility provides specialisation of generic modules with call by name parameters through a simple text template facility.

The utility is invoked by passing the name of a template file and one or more translations for placeholders in the template file as parameters, the latter in the form of strings. It then recursively replaces all the placeholders with their respective translations, thereby generating the source text for a library module that is then written to a set of Modula-2 source files available for import by any Modula-2 library or program.

The M2TE utility recognises any string prefixed and suffixed by `@@` as a placeholder and any line that starts with `%%` as a comment not to be copied into the output. To produce these symbols verbatim they may be escaped with backquote `` ` ``.

The M2TE utility may be invoked manually, or automatically by the compiler on a generate-on-demand basis using the `GENLIB` pragma within a Modula-2 source file.

EBNF:

```
m2teInvocation :
    "m2te" templateFilename ( placeholderName ":" translation )*
```

By convention, the first placeholder name is always `module`, standing in for the name of the module to be generated.

Example:

```
$ m2te Stack module:IntegerStack componentType:INTEGER
```

The example above invokes the M2TE utility to read template files `Stack.def` and `Stack.mod`, replace any occurrences of the passed placeholders `module` and `componentType` with identifiers `IntegerStack` and `INTEGER` respectively, and write the resulting output into Modula-2 source files named `IntegerStack.def` and `IntegerStack.mod` respectively.

The `GENLIB` pragma may be used to generate a module on demand from within the source text of a client module or program.

Example:

```
<* GENLIB IntegerStack FROM Stack : componentType=”INTEGER” *>
IMPORT IntegerStack; (* import the generated module *)
```

The standard library provides a portfolio of generic templates for commonly used abstract data types. A list of templates and their brief descriptions can be found in the standard library section of this document.

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 such as pervasives.

There are six mandatory pseudo modules, one optional pseudo module and one documentation module:

`ATOMIC.def`	provides atomic intrinsics
`SYSTEM.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)
`PERVASIVES.def`	documents pervasive constants, types and macros

14.2 Prototype Library

Prototypes which specify common semantics that data types may be required to conform to:

`ProtoNumeric.def`	defines properties common to all numeric prototypes
`ProtoScalar.def`	defines properties common to all numeric scalar prototypes
`ProtoNonScalar.def`	defines properties common to all numeric non-scalar prototypes
`ProtoCardinal.def`	defines properties for unsigned whole number ADTs
`ProtoInteger.def`	defines properties for signed whole number ADTs
`ProtoReal.def`	defines properties for real number ADTs
`ProtoComplex.def`	defines properties for complex number ADTs
`ProtoVector.def`	defines properties for numeric vector ADTs
`ProtoTuple.def`	defines properties for numeric tuple ADTs
`ProtoRealArray.def`	defines properties for numeric array ADTs of real numbers
`ProtoComplexArray.def`	defines properties for numeric array ADTs of complex numbers
`ProtoCollection.def`	defines properties common to all collection prototypes
`ProtoStaticSet.def`	defines properties for static set ADTs
`ProtoStaticArray.def`	defines properties for static array ADTs
`ProtoStaticString.def`	defines properties for static character string ADTs
`ProtoSet.def`	defines properties for dynamic unordered set ADTs
`ProtoOrderedSet.def`	defines properties for dynamic ordered set ADTs
`ProtoArray.def`	defines properties for dynamic array ADTs
`ProtoString.def`	defines properties for dynamic character string ADTs
`ProtoDictionary.def`	defines properties for dynamic unordered associative array ADTs
`ProtoOrderedDict.def`	defines properties for dynamic ordered associative array ADTs
`ProtoDateTime.def`	defines properties for date-time ADTs
`ProtoInterval.def`	defines properties for date-time interval ADTs

14.3 Memory Management Modules

`Storage.def`	dynamic memory allocator

14.4 Modules for Exception Handling and Termination

`Exceptions.def`	exception handling
`Termination.def`	termination handling

14.5 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

14.6 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

14.7 IO Modules for SYSTEM Types

`BYTE.def`	IO module for type `BYTE`
`WORD.def`	IO module for type `WORD`
`ADDRESS.def`	IO module for type `ADDRESS`

14.8 IO Modules for Pervasive Types

`PervasiveIO.def`	aggregator module to import all pervasive IO modules
`BOOLEAN.def`	IO module for type `BOOLEAN`
`BITSET.def`	IO module for type `BITSET`
`LONGBITSET.def`	IO module for type `LONGBITSET`
`CHAR.def`	IO module for type `CHAR`
`ARRAYOFCHAR.def`	IO module for `ARRAY OF CHAR` types
`UNICHAR.def`	IO module for type `UNICHAR`
`ARRAYOFUNICHAR.def`	IO module for `ARRAY OF UNICHAR` types
`OCTET.def`	IO module for type `OCTET`
`CARDINAL.def`	IO module for type `CARDINAL`
`LONGCARD.def`	IO module for type `LONGCARD`
`INTEGER.def`	IO module for type `INTEGER`
`LONGINT.def`	IO module for type `LONGINT`
`REAL.def`	IO module for type `REAL`
`LONGREAL.def`	IO module for type `LONGREAL`

14.9 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
`UnsignedReal1.def`	real number type with values from 0.0 to 1.0
`UnsignedReal60.def`	real number type with values from 0.0 to 59.999
`UnsignedReal360.def`	real number type with values from 0.0 to 359.9999999

14.10 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
`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

14.11 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`

14.12 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

14.13 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

14.14 Modules with Legacy Interfaces

`LegacyPIM.def`	selected legacy PIM functions and procedures
`LegacyISO.def`	selected legacy ISO functions and procedures

14.15 Miscellaneous Modules

`Hashes.def`	selected hash functions
`LexParams.def`	constants with lexical parameters of the compiler

14.16 Template Library

`Stack.def`	generic stack template
`Queue.def`	generic queue template
`DEQ.def`	generic double ended queue template
`PriorityQueue.def`	generic priority queue template
`AATree.def`	generic AA tree template
`SplayTree.def`	generic Splay tree template
`PatriciaTrie.def`	generic Patricia trie template
`DynamicArray.def`	generic dynamic array template
`KeyValueStore.def`	generic key value storage template
`NonZeroIndexArray.def`	generic non-zero index array type template

Appendix A: Grammar in EBNF

A.1 Non-Terminal Symbols

Compilation Units

#1 Compilation Unit

```
compilationUnit :
    IMPLEMENTATION? programModule | definitionOfModule | prototype ;
```

#2 Program Module

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

#2.1 Module Identifier

```
moduleIdent : Ident ;
```

#3 Definition Of Module

```
definitionOfModule :
    DEFINITION MODULE moduleIdent ( "[" requiredConformance "]" )? ";"
    importList* definition*
    END moduleIdent "." ;
```

#4 Prototype

```
prototype :
    PROTOTYPE prototypeIdent "[" requiredConformance "]" ";"
    ( PLACEHOLDERS identList ";" )?
    requiredTypeDefinition
    ( ";" requiredBinding )*
    END prototypeIdent "." ;
```

#4.1 Prototype Identifier

```
prototypeIdent : Ident ;
```

#4.2 RequiredConformance

```
requiredConformance : prototypeIdent ;
```

#4.3 RequiredBinding

```
requiredBinding : procedureHeader ;
```

Prototype Definitions

#5 Required Type Definition

```
requiredTypeDefinition :
   TYPE "=" permittedTypeDefinition ( "|" permittedTypeDefinition )*
    ( ":=" protoliteral ( "|" protoliteral )* )?
```

#6 Permitted Type Definition

```
permittedTypeDef :
    RECORD | OPAQUE RECORD?
```

#7 Proto-Literal

```
protoliteral :
    simpleProtoliteral | structuredProtoliteral ;
```

#7.1 Simple Proto-Literal

```
simpleProtoliteral[1] : Ident;
```

[1] Simple protoliterals are `CHAR`, `INTEGER` and `REAL`, representing any quoted literals, whole numbers and real numbers.

#8 Structured Proto-Literal

```
structuredProtoliteral :
    "{" ( VARIADIC OF simpleProtoliteral ( "," simpleProtoliteral )* |
    structuredProtoliteral ( "," structuredProtoliteral )* ) "}" ;
```

Import Lists, Blocks, Declarations and Definitions

#9 Import List

```
importList :
    ( FROM moduleIdent IMPORT ( identList | "*" ) |
    IMPORT moduleIdent "+"? ( "," moduleIdent "+"? )* ) ";" ;
```

#10 Block

```
block :
    declaration*
    ( BEGIN statementSequence )? END ;
```

#11 Declaration

```
declaration :
    CONST ( constantDeclaration ";" )+ |
    TYPE ( Ident "=" type ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureDeclaration ";" ;
```

#12 Definition

```
definition :
    CONST ( ( "[" constBindableIdent "]" )? constantDeclaration ";" )+ |
    TYPE ( Ident "=" ( type | OPAQUE recordType? ) ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureHeader ";" ;
```

#12.1 CONST Bindable Identifier

constBindableIdent[2] : Ident ;

Constant Declarations

#13 Constant Declaration

constantDeclaration :
    Ident "=" constExpression[3] ;

#13.1 Constant Expression

```
constExpression : expression ;
```

Type Declarations

#14 Type

```
type :
    ( ( ALIAS | range ) OF )? typeIdent | enumerationType |
    arrayType | recordType | setType | pointerType | procedureType ;
```

#14.1 Type Identifier

```
typeIdent : qualident ;
```

#15 Range

```
range :
    "[" constExpression ".." constExpression "]" ;
```

---

[2] CONST bindable identifiers are TSIG and TEXP.

[3] Constants may not be declared as aliases of type identifiers.

#16 Enumeration Type
```
enumerationType :
    "(" ( ( "+" enumTypeIdent ) | Ident )
          ( "," ( ( "+" enumTypeIdent ) | Ident ) )* ")" ;
```

#16.1 Enumeration Type Identifier
```
enumTypeIdent : typeIdent ;
```

#17 Array Type
```
arrayType :
    ( ARRAY componentCount ( "," componentCount )* |
      ASSOCIATIVE ARRAY ) OF typeIdent ;
```

#17.1 Component Count
```
componentCount : constExpression ;
```

#18 Record Type
```
recordType :
    RECORD ( fieldList ( ";" fieldList )* indeterminateField |
    "(" baseType ")" fieldList ( ";" fieldList )* ) END ;
```

#18.1 Field List
```
fieldList : variableDeclaration ;
```

#18.2 Base Type
```
baseType : typeIdent ;
```

#19 Indeterminate Field
```
indeterminateField :
    INDETERMINATE Ident ":" ARRAY discriminantField OF typeIdent ;
```

#19.1 Discriminant Field
```
discriminantField : Ident ;
```

#20 Set Type
```
setType :
    SET OF ( enumTypeIdent | "(" identList ")" ) ;
```

#21 Pointer Type
```
pointerType :
    POINTER TO CONST? typeIdent ;
```

#22 Procedure Type
```
procedureType :
    PROCEDURE
    ( "(" formalTypeList ")" )?
    ( ":" returnedType )? ;
```

#22.1 Returned Type
```
returnedType : typeIdent ;
```

#23 Formal Type List
```
formalTypeList :
    formalType ( "," formalType )* ;
```

#24 Formal Type
```
formalType :
    attributedFormalType | variadicFormalType ;
```

#25 Attributed Formal Type
```
attributedFormalType :
    ( CONST | VAR )? simpleFormalType ;
```


#26 Simple Formal Type
simpleFormalType :
(CAST? ARRAY OF)? namedType ;

#27 Variadic Formal Type
variadicFormalType :
VARIADIC OF
(attributedFormalType |
“{~~(~~” attributedFormalType (“,” attributedFormalType)* “}~~)~~”) ;

Variable Declarations

#28 Variable Declaration
variableDeclaration :
identList “:” (range OF)? typeIdent ;

Procedure Declarations

#29 Procedure Declaration
procedureDeclaration :
procedureHeader “;” block Ident ;

#30 Procedure Header
procedureHeader :
PROCEDURE
(“[” bindableEntity “]”)?
Ident (“(” formalParamList “)”)? (“:” returnedType)? ;

#31 Bindable Entity
bindableEntity :
DIV | MOD | FOR | DESCENDING |
“::” | “:=” | “?” | “!” | “~” | “+” | “-” | “*” | “/” | “=” | “<” | “>” |
bindableIdent;

#31.1 Bindable Identifier
bindableIdent[4] : Ident;

#32 Formal Parameter List
formalParamList :
formalParams (“;” formalParams)* ;

#33 Formal Parameters
formalParams :
simpleFormalParams | variadicFormalParams ;

#34 Simple Formal Parameters
simpleFormalParams :
(CONST | VAR)? identList “:” simpleFormalType ;

#35 Variadic Formal Parameters
variadicFormalParams :
VARIADIC (~~variadicCounter |~~ “[” variadicTerminator “]”)? OF
(simpleFormalType |
“{~~(~~” simpleFormalParams (“;” simpleFormalParams)* “}~~)~~”) ;
~~variadicCounter : Ident ;~~

[4] PROCEDURE bindable identifiers are ABS, NEG, ODD, COUNT, LENGTH, NEW, DISPOSE, RETAIN, RELEASE, TLIMIT, TMIN, TMAX, SXF and VAL.

#35.1 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 :
    ( ( "[" expressionList "]" | "^" ) ( "." Ident )* )+ ;
```

Expressions

#49 Expression List
```
expressionList :
    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 :
    factor ( mulOp factor )* ;
```

#52.1 Multiply Operator
```
mulOp :
    "*" | "/" | DIV | MOD | AND ;
```

#53 Factor
```
factor :
    ( NumericLiteral | StringLiteral | structuredValue |
      designatorOrFunctionCall | "(" expression ")" )
    ( "::" namedType )? | NOT factor ;
```

#54 Designator Or Function Call
```
designatorOrFunctionCall :
    designator actualParameters? ;
```

#55 Actual Parameters
```
actualParameters :
    "(" expressionList? ")" ;
```


Value Constructors

#56 Structured Value

```
structuredValue :
    "{" ( valueComponent ( "," valueComponent )* )? "}" ;
```

#57 Value Component

```
valueComponent :
    expression ( ( BY | ".." ) constExpression )? ;
```

Identifiers

#58 Qualified Identifier

```
qualident :
    Ident ( "." Ident )* ;
```

#59 Identifier List

```
identList :
    Ident ( "," Ident )* ;
```

A.2 Terminal Symbols

#1 Reserved Words

```
ReservedWord :
    ALIAS AND ARRAY ASSOCIATIVE BEGIN BY CASE CONST DEFINITION DESCENDING
    DIV DO ELSE ELSIF END EXIT FOR FROM IF IMPLEMENTATION IMPORT IN
    INDETERMINATE LOOP MOD MODULE NOT OF OPAQUE OR PLACEHOLDERS POINTER
    PROCEDURE PROTOTYPE RECORD REPEAT RETURN SET THEN TO TYPE UNTIL VAR
    VARIADIC WHILE ;
```

#2 Identifier

```
Ident :
    IdentLeadChar IdentTailChar* ;
```

#2.1 Identifier Leading Character

```
IdentLeadChar :
    “_” | “$” | Letter ;
```

#2.2 Identifier Tail Character

```
IdentTailChar :
    IdentLeadChar | Digit ;
```

#3 Numeric Literal

```
NumericLiteral :
    “0”
       ( DecimalNumberTail |
         “b” Base2DigitSeq |
         “x” Base16DigitSeq |
         “u” Base16DigitSeq )?
    | “1” .. “9” DecimalNumberTail? ;
```

#3.1 Decimal Number Tail

```
DecimalNumberTail :
    DigitSep? DigitSeq
    ( “.” DigitSeq ( “e” ( “+” | “-” )? DigitSeq )? )? ;
```

#3.2 Digit Sequence

```
DigitSeq :
    Digit+ ( DigitSep Digit+ )* ;
```

#3.3 Base-2 Digit Sequence

```
Base2DigitSeq :
    Base2Digit+ ( DigitSep Base2Digit+ )* ;
```

#3.4 Base-16 Digit Sequence

```
Base16DigitSeq :
    Base16Digit+ ( DigitSep Base16Digit+ )* ;
```

#3.5 Digit Separator

```
DigitSep : “‘” ;
```

#4 String Literal

```
StringLiteral :
    SingleQuotedString | DoubleQuotedString ;
```

#4.1 Single Quoted String

```
    “‘“ ( QuotableCharacter | ‘“‘ )* “‘“ ;
```

#4.2 Double Quoted String

```
    ‘“‘ ( QuotableCharacter | “‘“ )* ‘“‘ ;
```

#4.3 Quotable Character

```
QuotableCharacter :
    Digit | Letter | Space | QuotableGraphicChar | EscapedCharacter ;
```

#4.4 Digit

```
Digit :
    Base2Digit | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” ;
```

#4.5 Base-2 Digit

```
Base2Digit :
    “0” | “1” ;
```

#4.6 Base-16 Digit

```
Base16Digit :
    Digit | “A” | “B” | “C” | “D” | “E” | “F” ;
```

#4.7 Letter

```
Letter :
    “A” .. “Z” | “a” .. “z” ;
```

#4.8 Space

```
Space : “ ” ;
```

#4.9 Quotable Graphic Character

```
QuotableGraphicChar :
    “!” | “#” | “$” | “%” | “&” | “(” | “)” | “*” | “+” | “,” |
    “-” | “.” | “/” | “:” | “;” | “<” | “=” | “>” | “?” | “@” |
    “[” | “]” | “^” | “_” | “`” | “{” | “|” | “}” | “~” ;
```

#4.10 Escaped Character

```
EscapedCharacter :
    “\” ( “0” | “n” | “t” | “\” | “‘“ | ‘“‘ ) ;
```

A.3 Ignore Symbols

#1 Whitespace

```
Whitespace :
    Space | ASCII_TAB ;
```

#2 Single-Line Comment

```
SingleLineComment :
    “//” ~( EndOfLine )* EndOfLine ;
```

#3 Multi-Line Comment

```
MultiLineComment :
    “(*” ( ~( “(*” | “*)” )* ( MultiLineComment | EndOfLine )? )* “*)” ;
```

#4 End Of Line Marker

```
EndOfLine :
    ASCII_LF ASCII_CR? | 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[5] `: { 0uFE, 0uBB, 0uBF } ;`

---

[5] 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 :
    "<*" ( pragmaMSG | pragmaIF | pragmaENCODING | pragmaGENLIB | pragmaFFI |
    pragmaINLINE | pragmaALIGN | pragmaPADBITS | pragmaADDR | pragmaREG |
    pragmaPURITY | varAttrPragma | pragmaFORWARD | implDefinedPragma ) "*>" ;
```

#2 Body Of Compile Time Message Pragma

```
pragmaMSG :
    MSG "=" ( INFO | WARN | ERROR | FATAL ) ":"
    compileTimeMsgComponent ( "," compileTimeMsgComponent )* ;
```

#3 Compile Time Message Component

```
compileTimeMsgComponent :
    StringLiteral | ConstQualident |
    "?" ( ALIGN | ENCODING | implDefPragmaName ) ;
```

#3.1 Constant Qualified Identifier

```
constQualident : qualident ;
```

#3.2 Implementation Defined Pragma Name

```
implDefPragmaName : Ident ;
```

#4 Body Of Conditional Compilation Pragma

```
pragmaIF :
    ( IF | ELSIF ) inPragmaExpression | ELSE | ENDIF ;
```

#5 Body Of Character Encoding Pragma

```
pragmaENCODING :
    ENCODING "=" ( "ASCII" | "UTF8" ) ( ":" codePointSampleList )? "*>" ;
```

#6 Code Point Sample List

```
codePointSampleList :
    quotedChar "=" characterCode ( "," quotedChar "=" characterCode )* ;
```

#6.1 Quoted Character Literal

```
quotedChar : StringLiteral ;
```

#6.2 Character Code Literal

```
characterCode : NumericLiteral ;
```

#7 Library Template Expansion Pragma

```
pragmaGENLIB :
    GENLIB moduleIdent FROM template ":" templateParamList ;
```

#7.1 Template Identifier

```
template : Ident ;
```

#8 Template Parameter List

```
templateParamList :
    placeholder "=" replacement ( "," placeholder "=" replacement )*
```

#8.1 Placeholder

```
placeholder : Ident ;
```

#8.2 Replacement

```
replacement : StringLiteral ;
```

#9 Body Of Foreign Function Interface Pragma

```
pragmaFFI :
    FFI "=" ( "C" | "Fortran" ) ;
```

#10 Body Of Function Inlining Pragma

```
pragmaINLINE :
    INLINE | NOINLINE ;
```

#11 Body Of Memory Alignment Pragma

```
pragmaALIGN :
    ALIGN "=" inPragmaExpression ;
```


#12 Body Of Bit Padding Pragma

```
pragmaPADBITS :
    PADBITS "=" inPragmaExpression ;
```

#13 Body Of Memory Mapping Pragma

```
pragmaADDR :
    ADDR "=" inPragmaExpression ;
```

#14 Body Of Register Mapping Pragma

```
pragmaREG :
    REG "=" inPragmaExpression ;
```

#15 Body Of Purity Attribute Pragma

```
pragmaPURITY :
    PURITY "=" inPragmaExpression ;
```

#16 Body Of Variable Attribute Pragma

```
varAttrPragma :
    SINGLEASSIGN | VOLATILE ;
```

#17 Body Of Forward Declaration Pragma

```
pragmaFORWARD :
    FORWARD ( TYPE identList | procedureHeader ) ;
```

#18 Body Of Implementation Defined Pragma

```
implDefinedPragma :
    implDefPragmaName ( "=" inPragmaExpression )? ;
```

#19 In-Pragma Expression

```
inPragmaExpression :
    inPragmaSimpleExpr ( inPragmaRelOp inPragmaSimpleExpr )? ;
```

#19.1 In-Pragma Relational Operator

```
inPragmaRelOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" ;
```

#20 In-Pragma Simple Expression

```
inPragmaSimpleExpr :
    ( "+" | "-" )? inPragmaTerm ( addOp inPragmaTerm )* ;
```

#21 In-Pragma Term

```
inPragmaTerm :
    inPragmaFactor ( inPragmaMulOp inPragmaFactor )* ;
```

#21.1 In-Pragma Multiply Operator

```
inPragmaMulOp :
    "*" | DIV | MOD | AND ;
```

#22 In-Pragma Factor

```
inPragmaFactor :
    wholeNumber | constQualident | "(" inPragmaExpression ")" |
    inPragmaCompileTimeFunctionCall | NOT inPragmaFactor ;
```

#22.1 Whole Number

```
wholeNumber : NumericLiteral ;
```

#23 In-Pragma Compile-Time Function Call

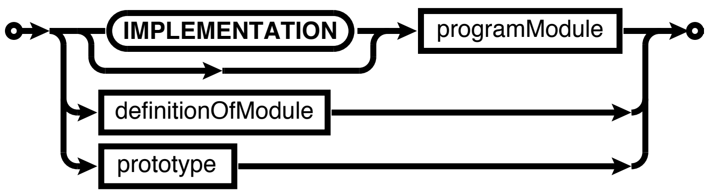
```
inPragmaCompileTimeFunctionCall :
    Ident[1] "(" inPragmaExpression ( "," inPragmaExpression )* ")" ;
```

[1] Permissible are `ABS`, `NEG`, `ODD`, `ORD`, `LENGTH`, `TMIN`, `TMAX`, `TSIZE`, `TLIMIT` and macros in module `COMPILER`.

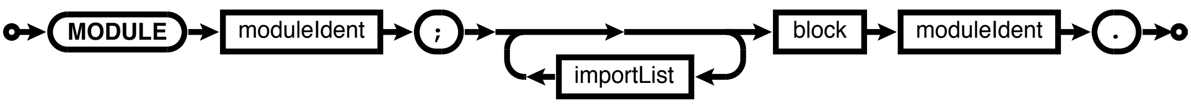
Appendix B: Syntax Diagrams

B.1 Non-Terminal Symbols

#1 Compilation Unit

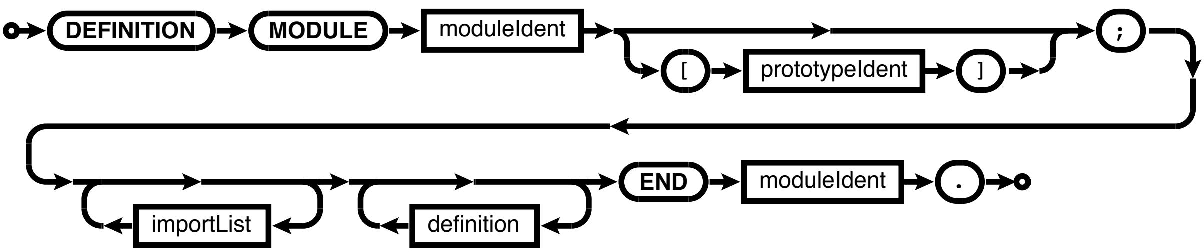


#2 Program Module

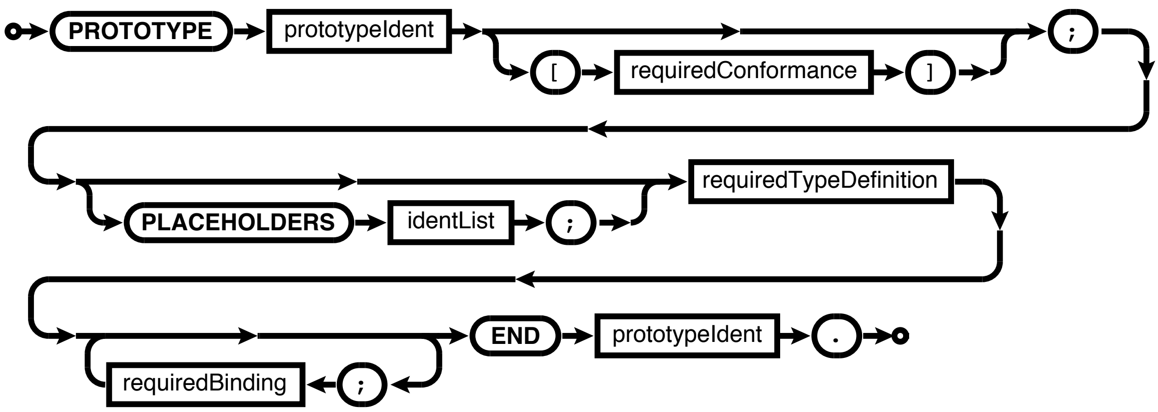


#2.1 Module Identifier

#3 Definition Of Module



#4 Prototype Definition

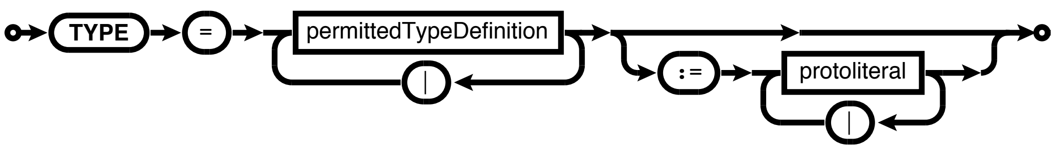


#4.1 Prototype Identifier

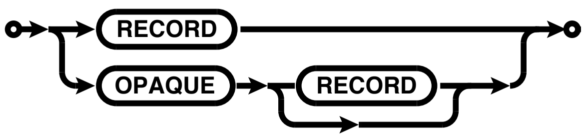
#4.2 Required Conformance

#4.3 Required Binding

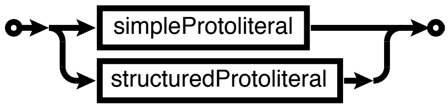
#5 Required Type Definition



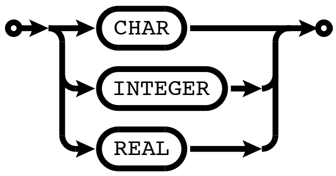
#6 Permitted Type Definition



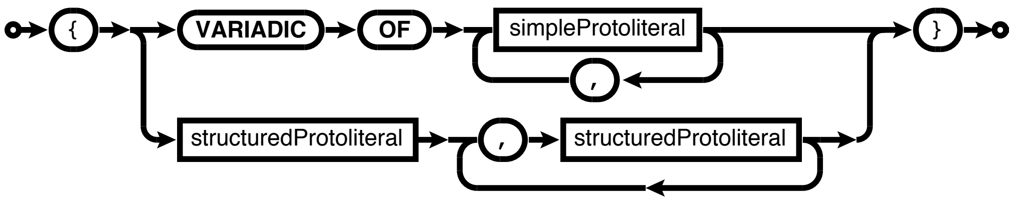
#7 Proto-Literal



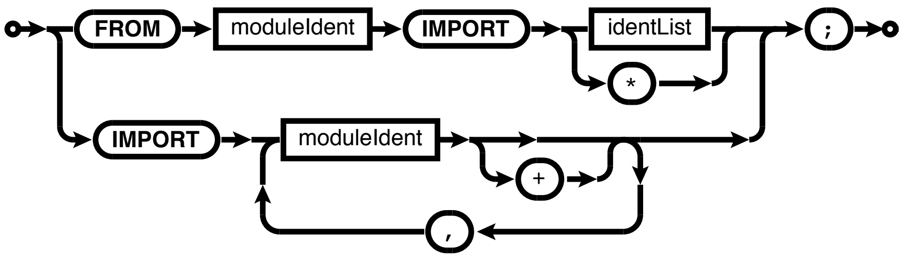
#7.1 Simple Proto-Literal



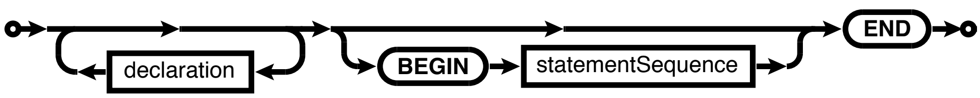
#8 Structured Proto-Literal



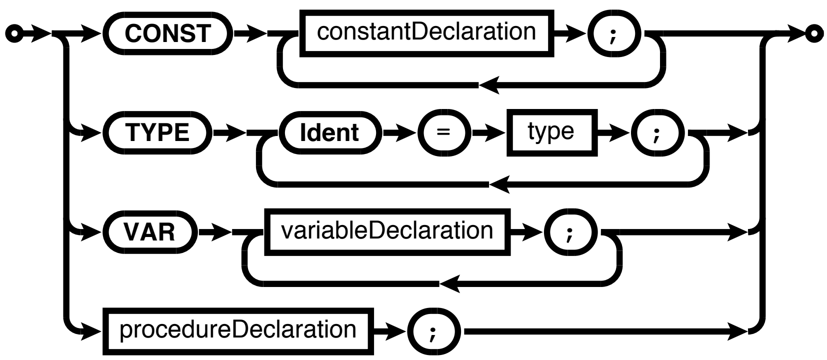
#9 Import List



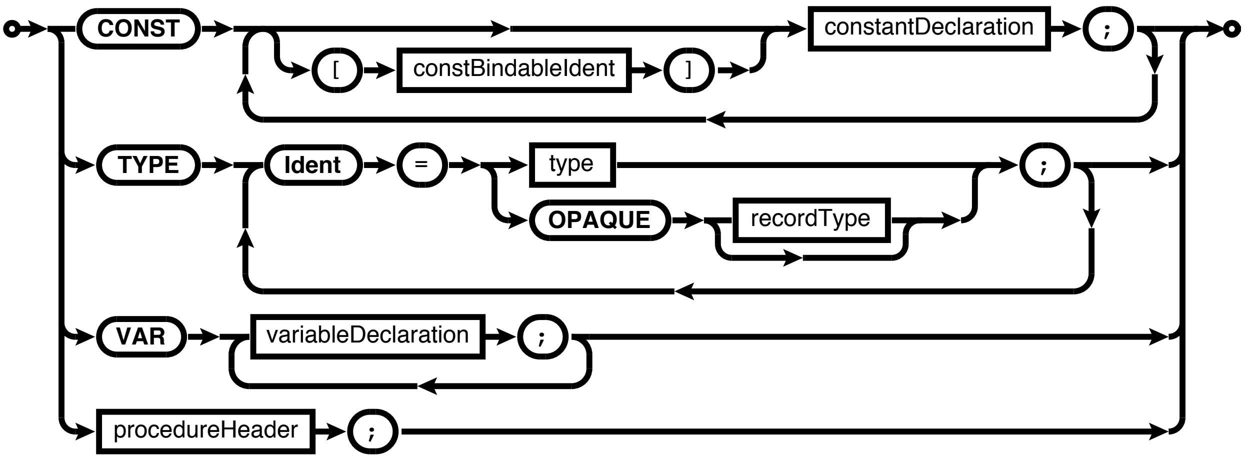
#10 Block



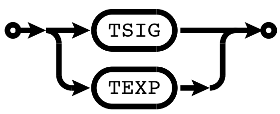
#11 Declaration



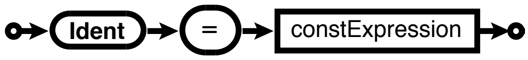
#12 Definition



#12.1 CONST Bindable Identifier

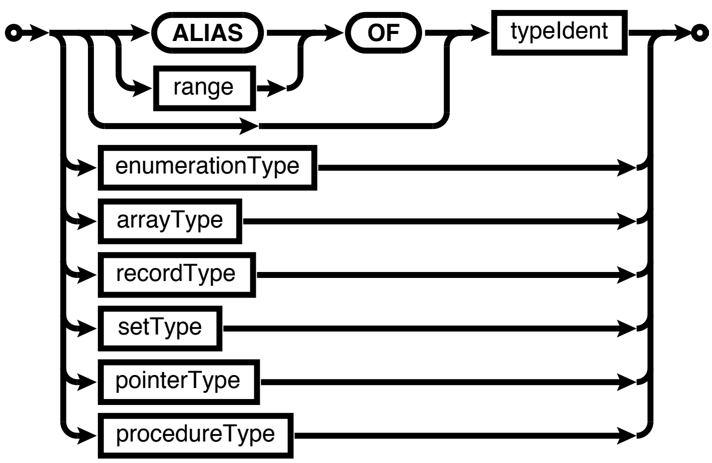


#13 Constant Declaration

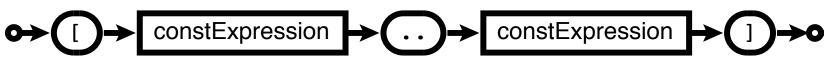


#13.1 Constant Expression

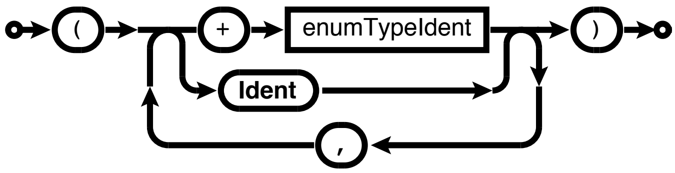
#14 Type



#15 Range



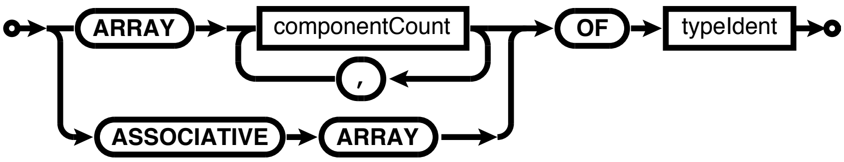
#16 Enumeration Type



#16.1 Enumeration Type Identifier

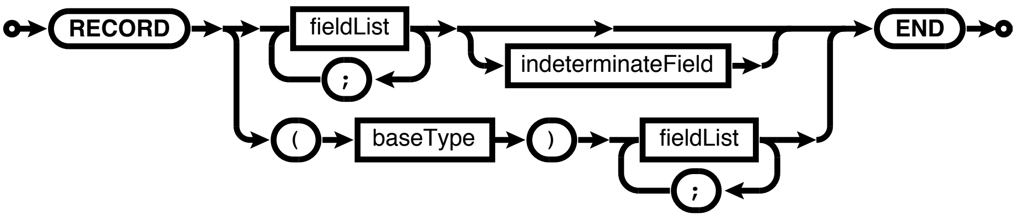
#16.2 Type Identifier

#17 Array Type



#17.1 Component Count

#18 Record Type



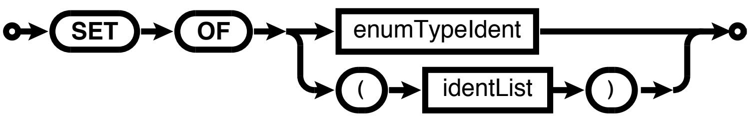
#18.1 Field List

#18.2 Base Type

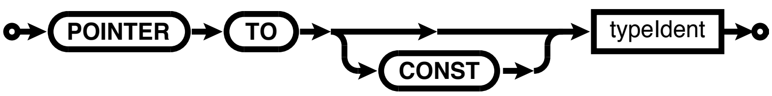
#19 Indeterminate Field

#19.1 Discriminant Field

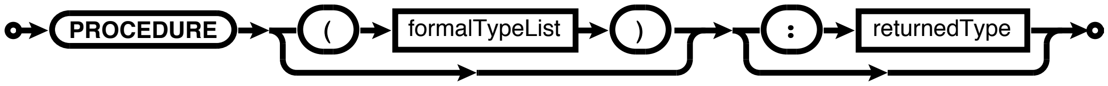
#20 Set Type



#21 Pointer Type

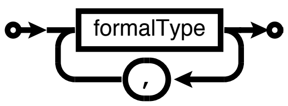


#22 Procedure Type

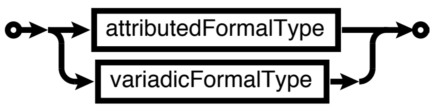


#22.1 Returned Type

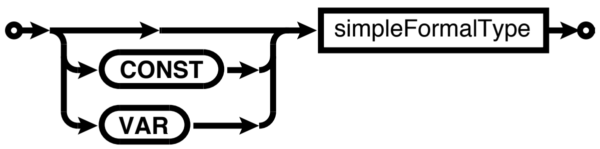
#23 Formal Type List



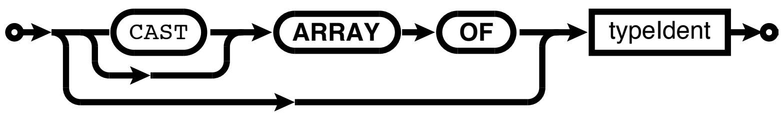
#24 Formal Type



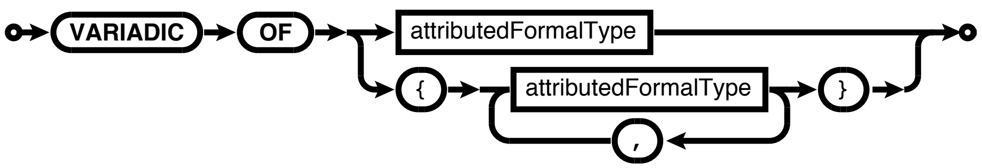
#25 Attributed Formal Type



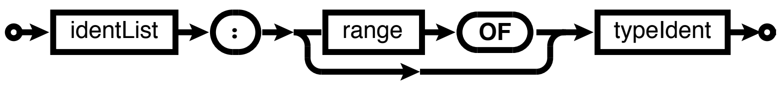
#26 Simple Formal Type



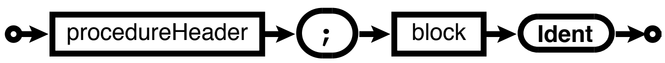
#27 Variadic Formal Type



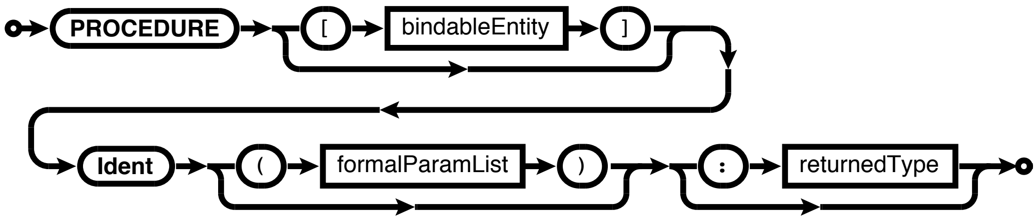
#28 Variable Declaration



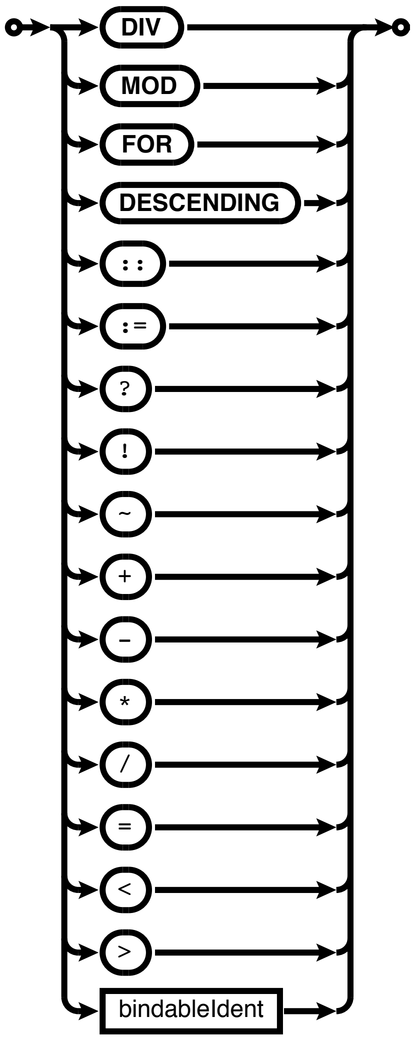
#29 Procedure Declaration



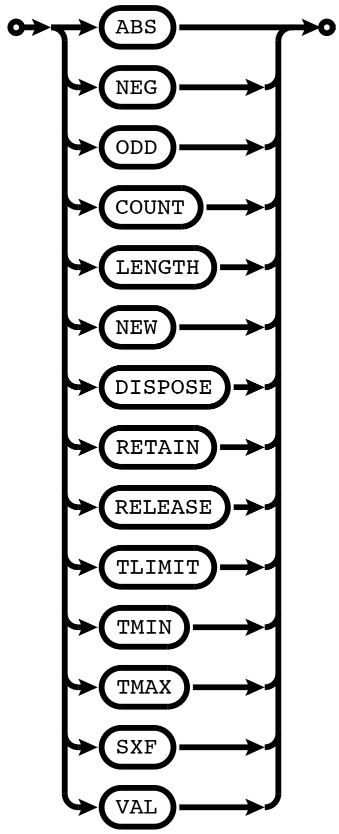
#30 Procedure Header



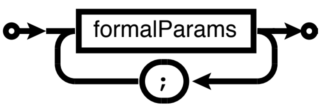
#31 Bindable Entity



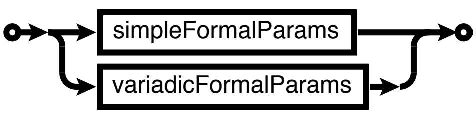
#31.1 Bindable Identifier



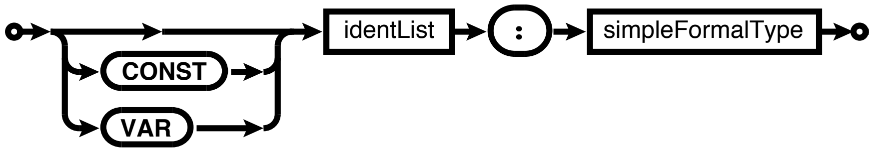
#32 Formal Parameter List



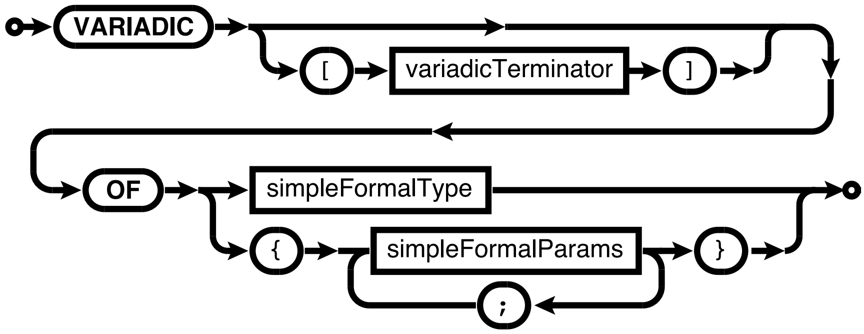
#33 Formal Parameters



#34 Simple Formal Parameters

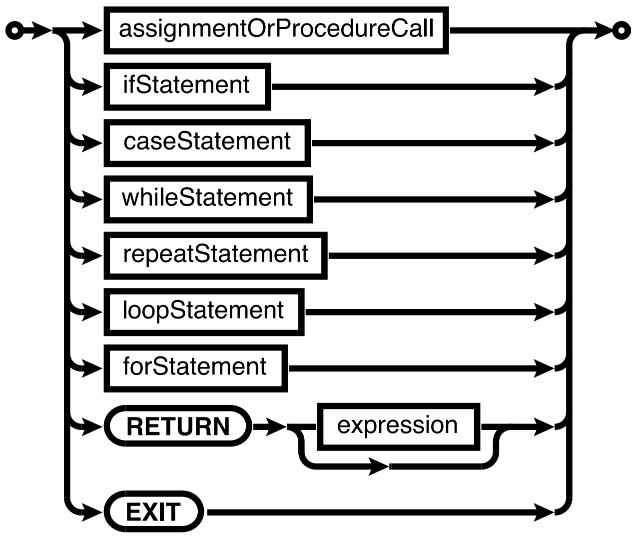


#35 Variadic Formal Parameters

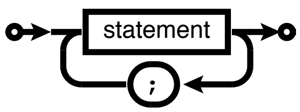


#35.1 Variadic Terminator

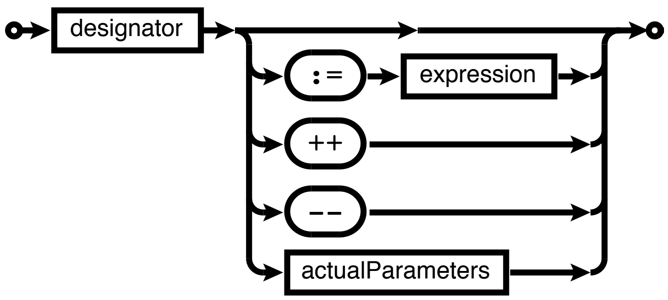
#36 Statement



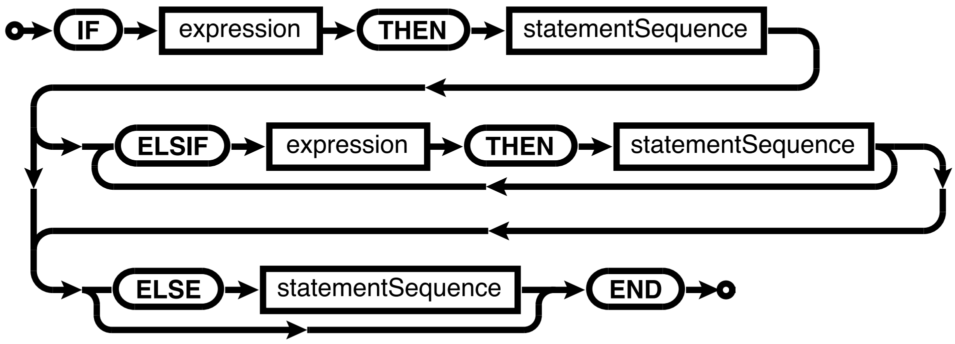
#37 StatementSequence



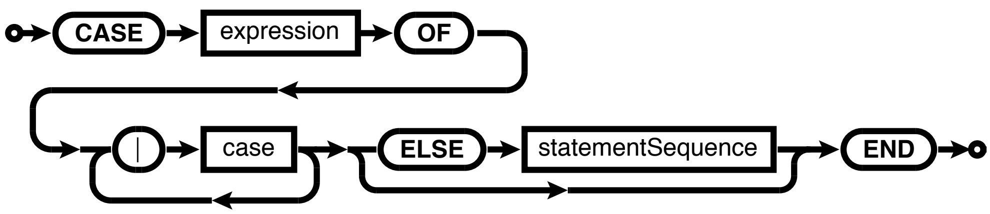
#38 Assignment Or Procedure Call



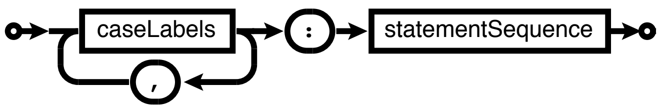
#39 IF Statement



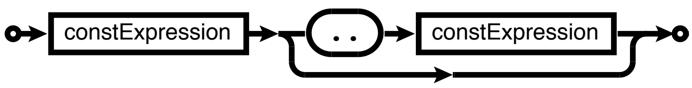
#40 CASE Statement



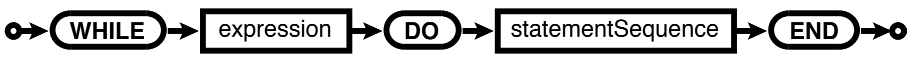
#41 Case



#42 Case Labels



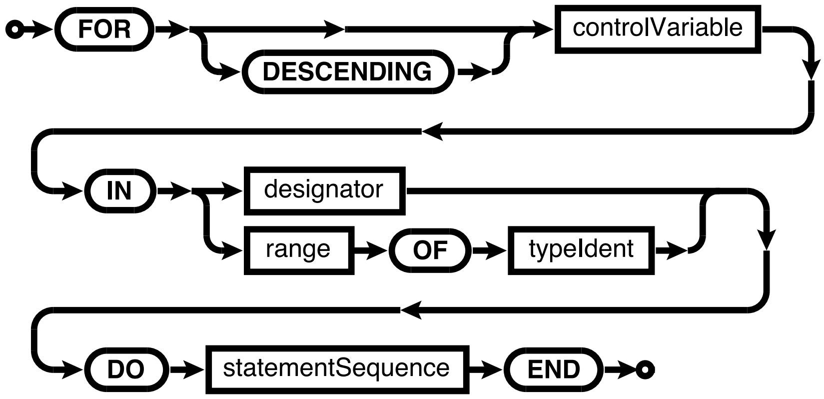
#43 WHILE Statement



#44 REPEAT Statement

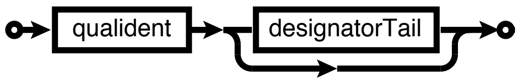
#45 LOOP Statement

#46 FOR Statement

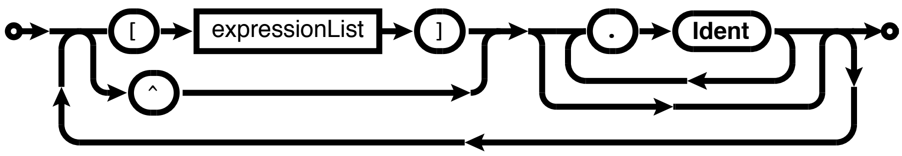


#46.1 Control Variable

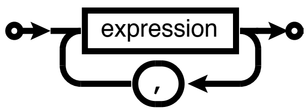
#47 Designator



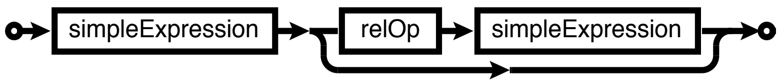
#48 Designator Tail



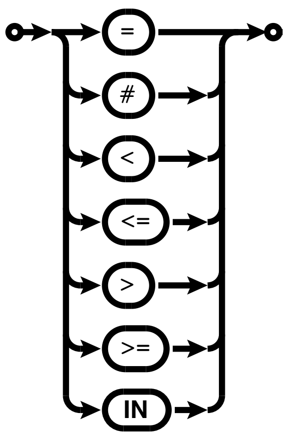
#49 Expression List



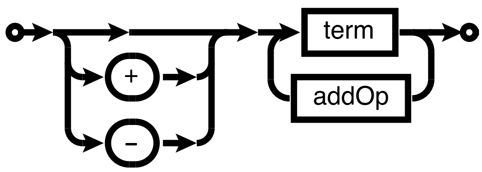
#50 Expression



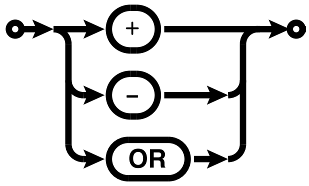
#50.1 Relational Operator



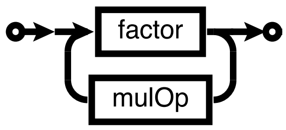
#51 Simple Expression



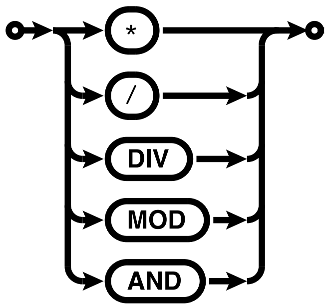
#51.1 Add Operator



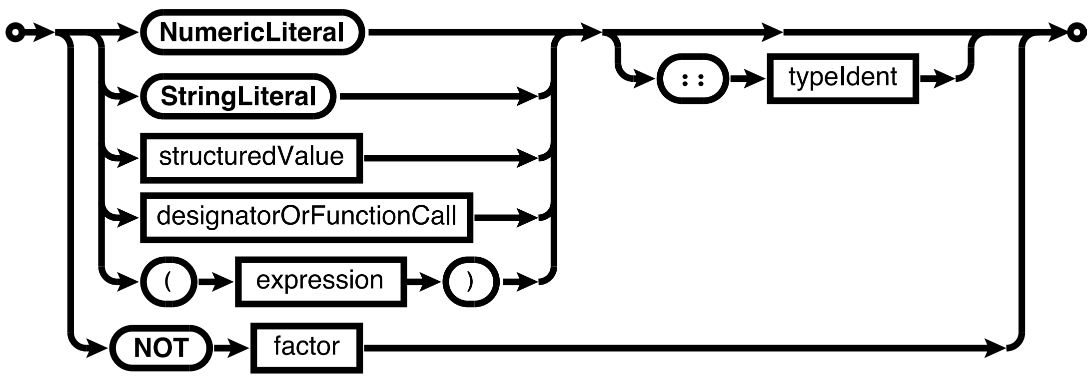
#52 Term



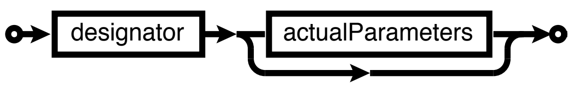
#52.1 Multiply Operator



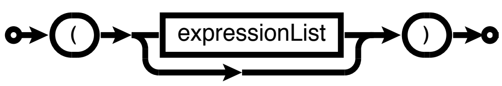
#53 Factor



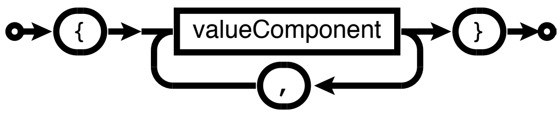
#54 Designator Or Function Call



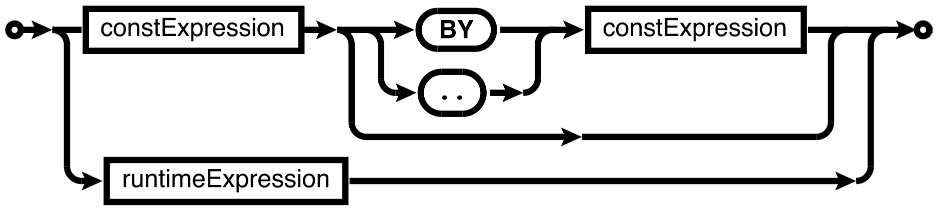
#55 Actual Parameters



#56 Structured Value

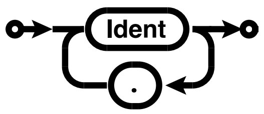


#57 Value Component

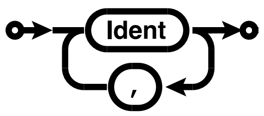


#57.1 Runtime Expression

#58 Qualified Identifier

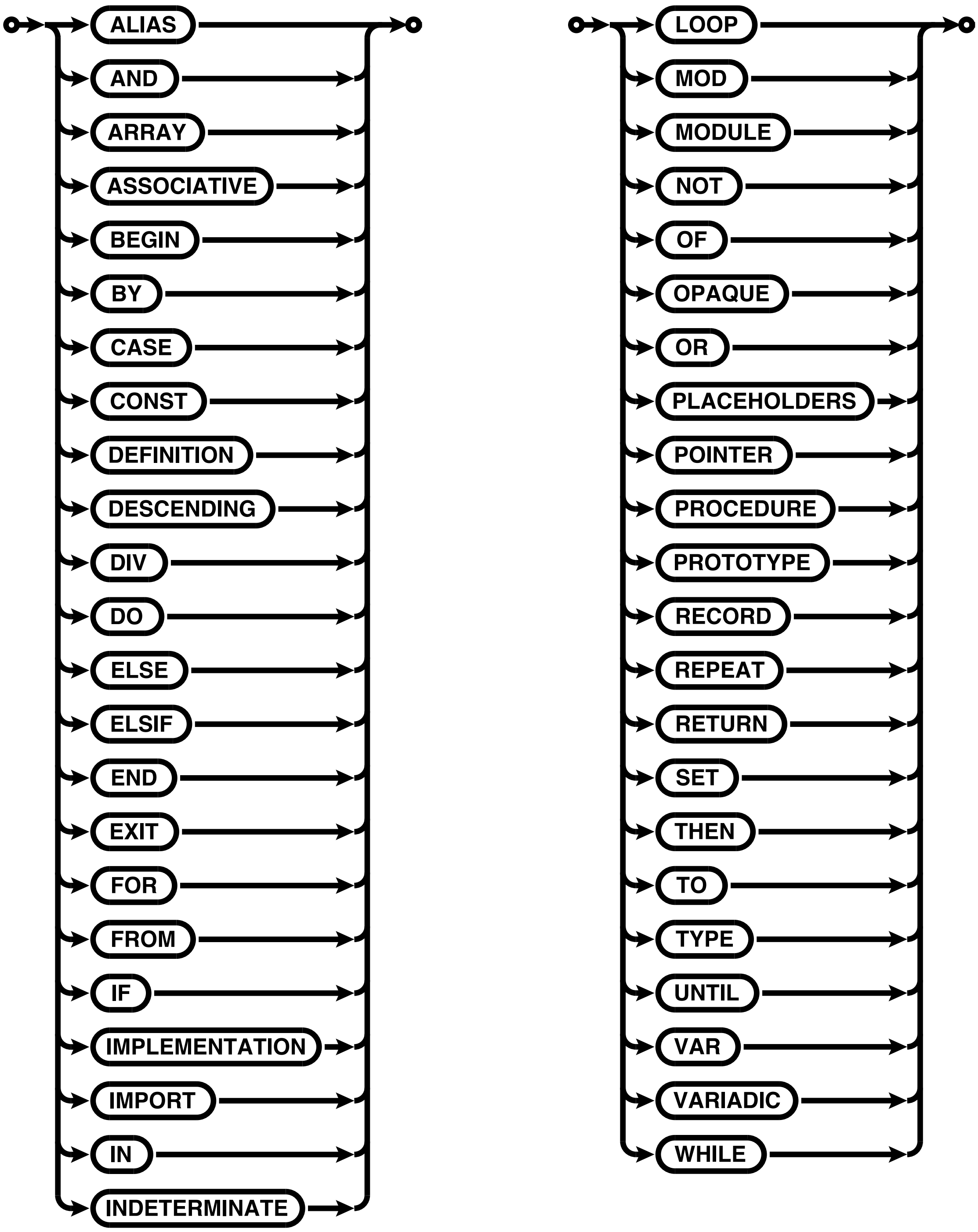


#59 Identifier List

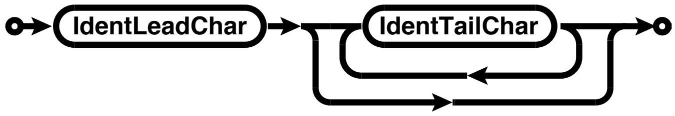


B.2 Terminal Symbols

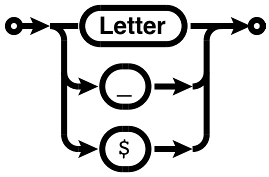
#1 Reserved Words



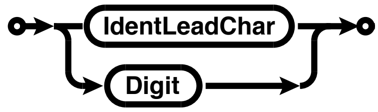
#2 Identifier



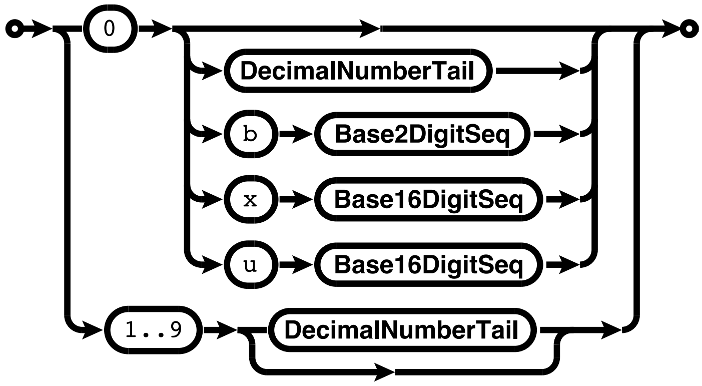
#2.1 Identifier Leading Character



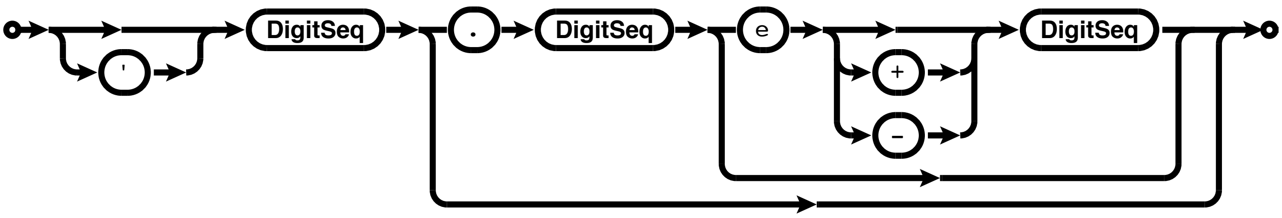
#2.2 Identifier Tail Character



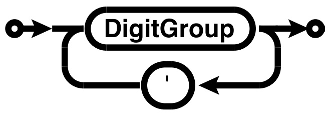
#3 Numeric Literal



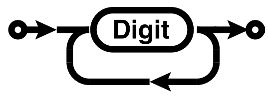
#3.1 Decimal Number Tail



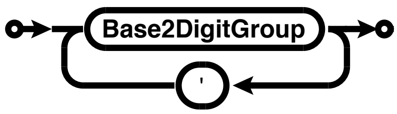
#3.2 Digit Sequence



#3.2b Digit Group

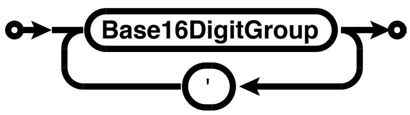


#3.3 Base-2 Digit Sequence

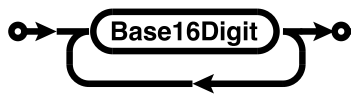


#3.3b Base-2 Digit Group

#3.4 Base-16 Digit Sequence

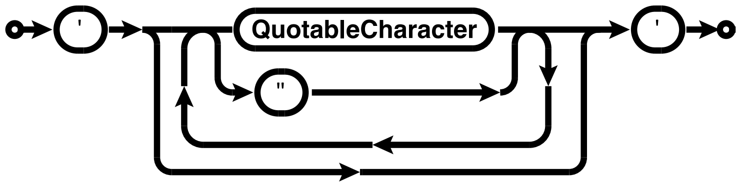


#3.4b Base-16 Digit Group

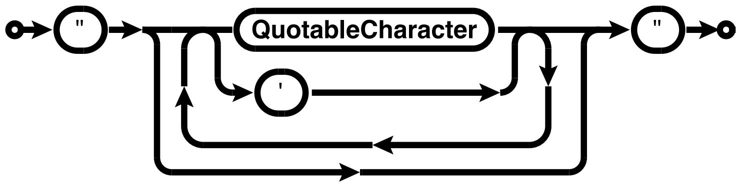


#4 String Literal

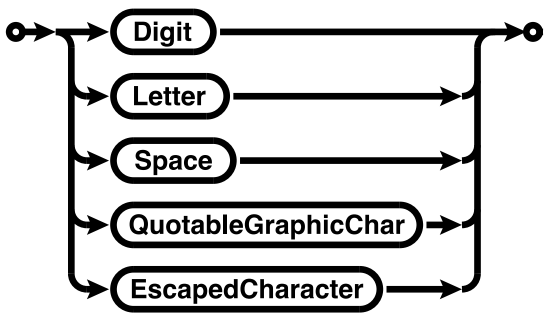
#4.1 Single Quoted String



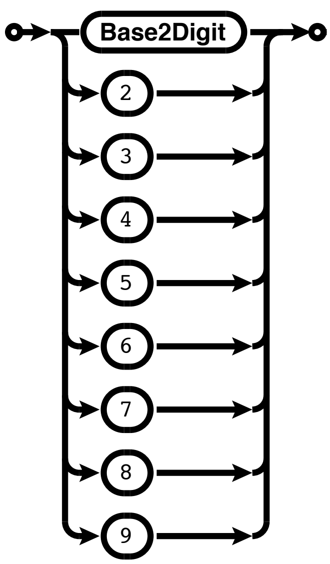
#4.2 Double Quoted String



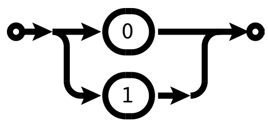
#4.3 Quotable Character



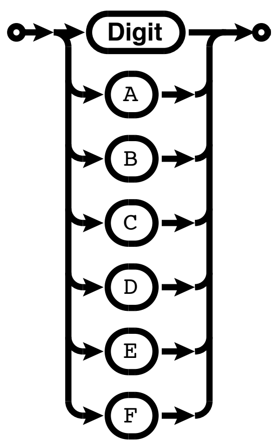
#4.4 Digit



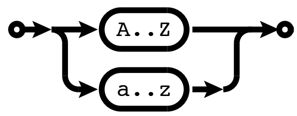
#4.5 Base-2 Digit



#4.6 Base-16 Digit



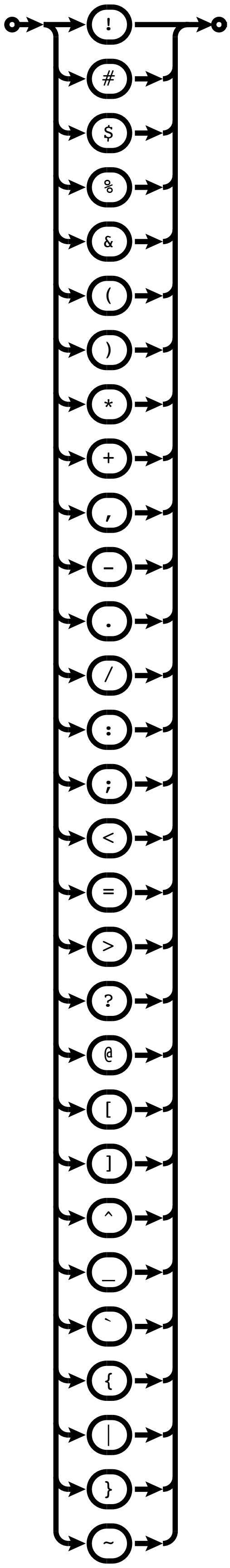
#4.7 Letter



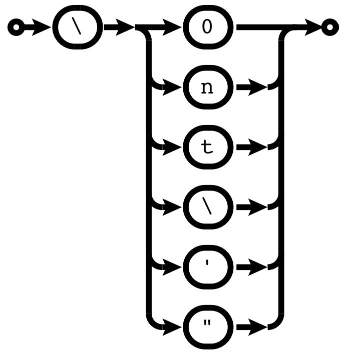
#4.8 Space

```
CONST Space = CHR(32)
```

#4.9 Quotable Graphic Character

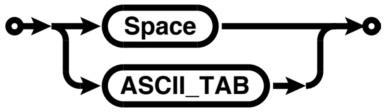


#4.10 Escaped Character



B.3 Ignore Symbols

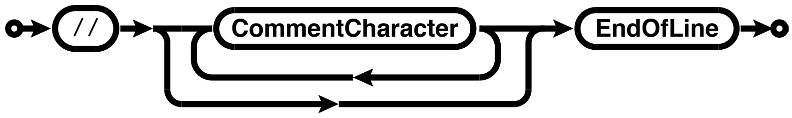
#1 Whitespace



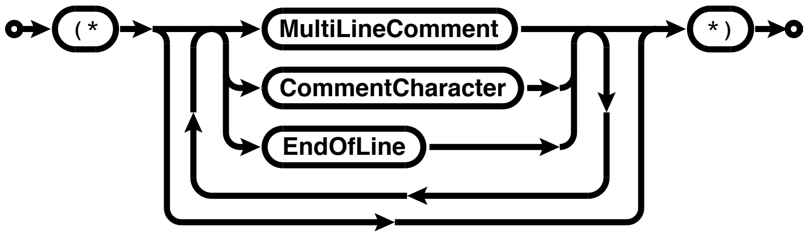
#1.1 ASCII Tabulator

```
CONST ASCII_TAB = CHR(8)
```

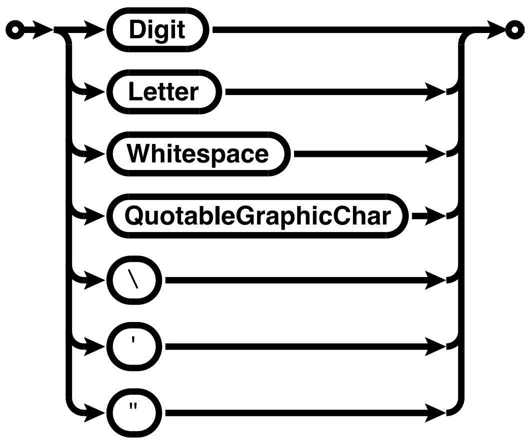
#2 Single-line Comment



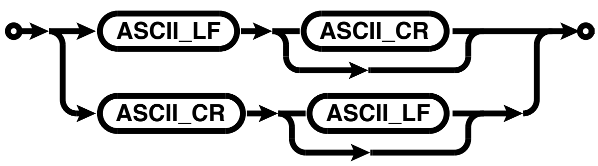
#3 Multi-line Comment



#3.1 Comment Character



#4 End Of Line Marker



#4.1 ASCII Line Feed

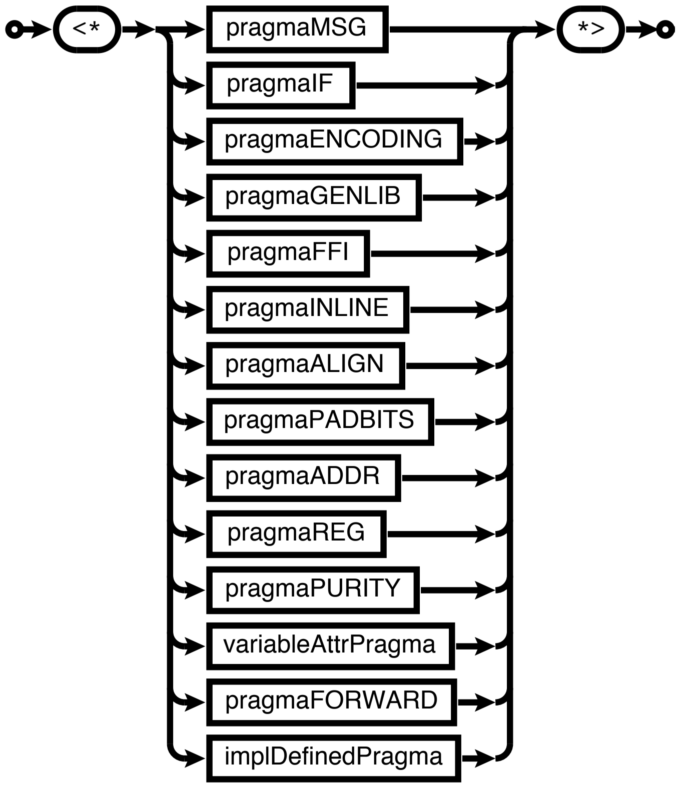
```
CONST ASCII_LF = CHR(10)
```

#4.2 ASCII Carriage Return

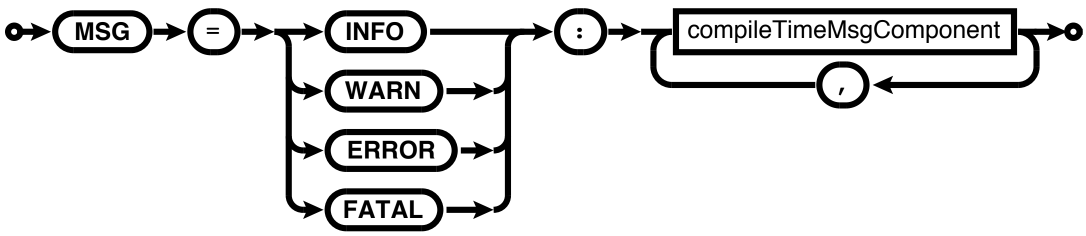
```
CONST ASCII_CR = CHR(13)
```


B.4 Pragma Grammar

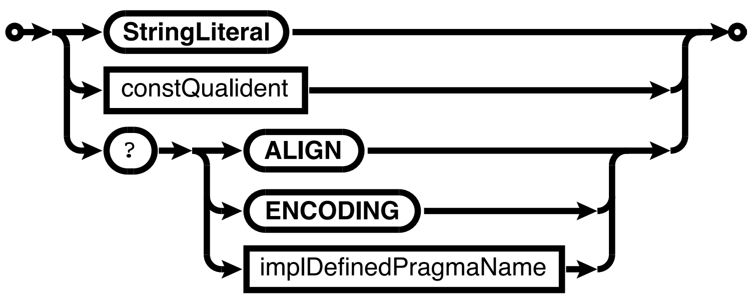
#1 Pragma



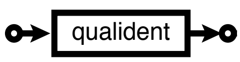
#2 Body Of Compile Time Message Pragma



#3 Compile Time Message Component

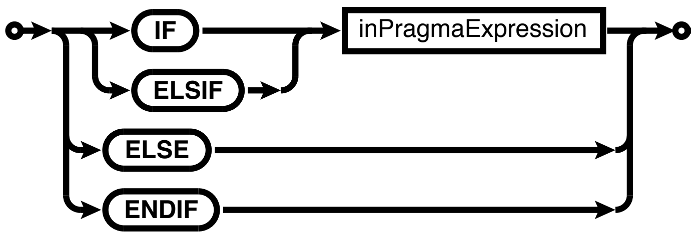


#3.1 Constant Qualified Identifier

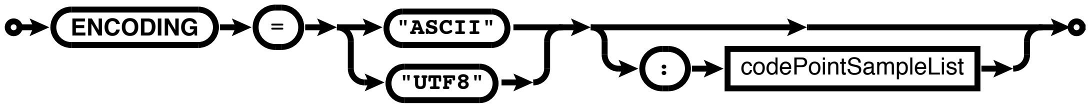


#3.2 Implementation Defined Pragma Name

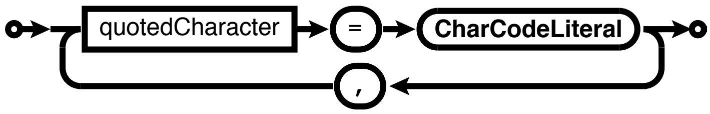
#4 Body Of Conditional Compilation Pragma



#5 Body Of Character Encoding Pragma



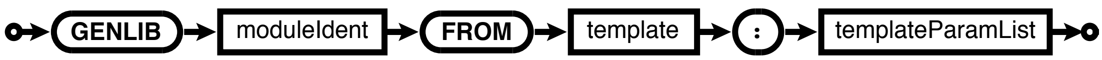
#6 Code Point Sample List



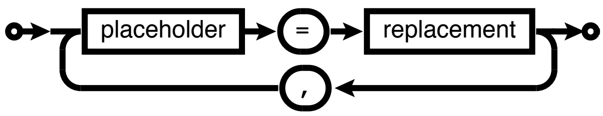
#6.1 Quoted Character

#6.2 Character Code Literal

#7 Body Of Library Template Expansion Pragma



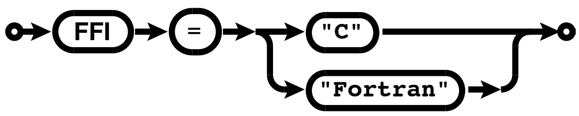
#8 Template Parameter List



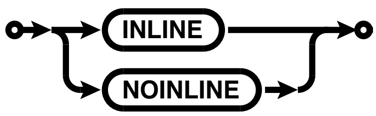
#8.1 Placeholder

#8.2 Replacement

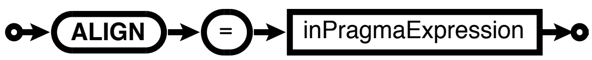
#9 Body Of Foreign Function Interface Pragma



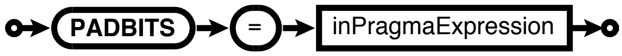
#10 Body Of Procedure Inlining Pragma



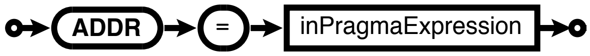
#11 Body Of Memory Alignment Pragma



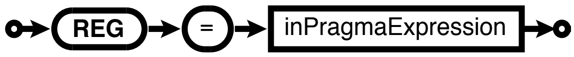
#12 Body Of Bit Padding Pragma



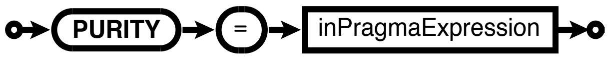
#13 Body Of Memory Mapping Pragma



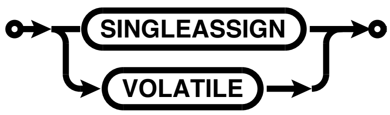
#14 Body Of Register Mapping Pragma



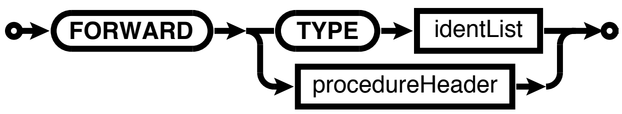
#15 Body Of Purity Attribute Pragma



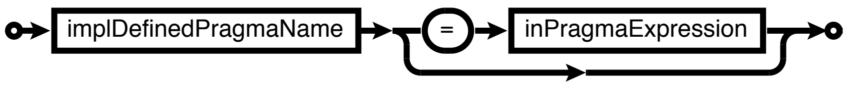
#16 Body Of Variable Attribute Pragma



#17 Body Of Forward Declaration Pragma

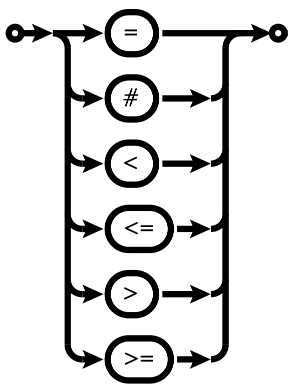


#18 Body Of Implementation Defined Pragma

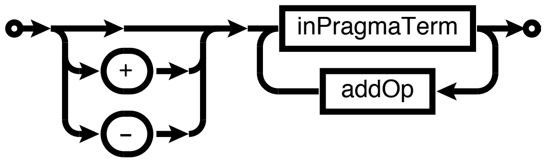


#19 In-Pragma Expression

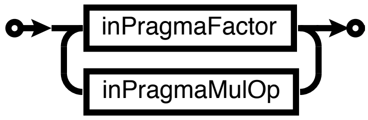
#19.1 In-Pragma Relational Operator



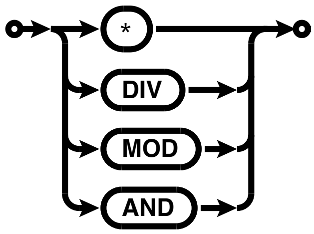
#20 In-Pragma Simple Expression



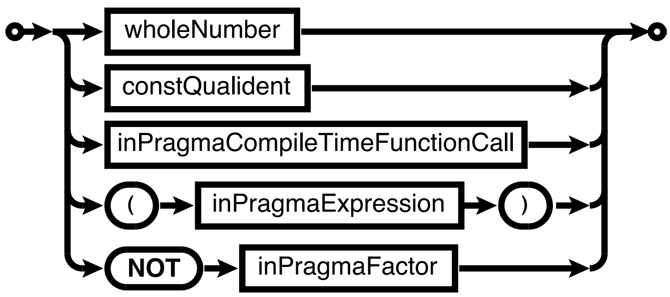
#21 In-Pragma Term



#21.1 In-Pragma Multiply Operator

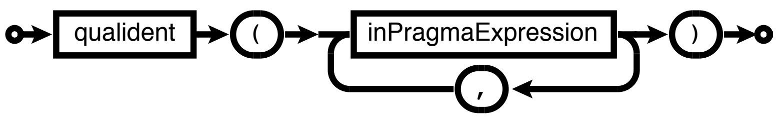


#22 In-Pragma Factor



#22.1 Whole Number

#23 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	UTF8 BOM	optional	
1.3	Compilation Units	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 ~~other than FOR TO BY loop~~	mandatory	
1.10	Pervasive Constants	mandatory	
1.11	Pervasive Types	mandatory	
1.12	Pervasive Procedures and Functions	mandatory	
1.13	Binding to Operators and Pervasives	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 SYSTEM	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	Pragma ENCODING	optional	
3.4	Pragma GENLIB	mandatory	
3.5	Pragma FFI	optional	
3.5.1	Foreign Function Interface to C	mandatory if FFI is provided	
3.5.2	Foreign Function Interface to Fortran	optional	
3.6	Inlining Pragmas	mandatory	
3.7	Pragma ALIGN	optional	
3.8	Pragma PADBITS	optional	
3.9	Pragma ADDR	optional	
3.10	Pragma REG	optional	
3.11	Pragma PURITY	optional	
3.12	Variable Attribute Pragmas	optional	
3.13	Pragma FORWARD	single-pass compilers only	
4	**Generics**		
4.1	Modula-2 Template Engine	mandatory	
5	**Standard Library**		
5.1	Standard Library Prototypes	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]

D.5 Document Version Log

http://modula2.net/resources/M2R10~~m2r10/spec~~.versionlog.txt

D.6 Document Review Issue Tracker

http://modula2.net/resources/M2R10.IssueTracker.pdf

Appendix E: Statistics

E.1 Specification

- the language specification document has 148 pages, 34 800 words, 194 500 characters

E.2 Base Language

- the core grammar has 59 non-terminals, 4 terminals, 4 ignore symbols
- the pragma grammar has 22 non-terminals and re-uses the terminals of the core grammar
- the language has 45 reserved words, 15 reserved pragma symbols, 37~~5~~ pervasive identifiers, thereof 3 built-in constants, 12 built-in types, 22~~0~~ built-in procedures and functions; 5 operator precedence levels and 16 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 2 primitives
- module `SYSTEM` 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.