Modula-2 R10

Revision 2010

MODULA-2 R10 (Revision_2010) Concise Language Description with Grammar and Syntax Diagrams (* by b.Kowarsch and r.Sutcliffe *)

Status: June 22, 2010

Synopsis

This document is a concise description of the R10 dialect of Modula-2, jointly developed in 2009 and 2010 by B.Kowarsch and R.Sutcliffe, as a modern revision of classic Modula-2. Its primary design goals are type safety, utmost readability and consistency, and suitability as a core language for domain specific supersets. The targeted areas of application are systems implementation, engineering and mathematics. The moniker R10 is a shorthand for "Revision 2010".

At the time of writing, the authoritative language report is still work in progress and has not yet been published. The present document is intended strictly as a non-authoritative informational source for general peer-review, and for contributors who wish to participate in work on the reference compiler, the standard library implementation and associated compliance test tools.

Details are subject to change without prior notice. The latest version of the text and grammar in this document are available in plain text format from the public repository at

- http://bitbucket.org/trijezdci/m2r10stdlib/src/tip/_LANGUAGE/Language.txt
- http://bitbucket.org/trijezdci/m2r10stdlib/src/tip/_GRAMMAR/Modula2.g

Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
EBNF	Extended Backus-Naur Formalism
SXF	Scalar Exchange Format
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:

- 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

Items To Do

The following items still need to be done before the language report can be finalised:

- describe module initialisation order in more detail.
- describe semantics of `Exceptions` library in more detail.
- definition and description of the `COROUTINES` pseudo-module.

Outstanding Issues To Resolve

The following issue still needs to be resolved before the language report can be finalised:

- whether to remove the `FOR TO BY` statement in favour of an enhanced `FOR IN` statement.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Table of Contents

1 Lexical Entities ... 9
1.1 Special Symbols ... 9
1.2 Literals ... 10
1.2.1 Numeric literals ... 10
1.2.1.1 Character Code Literals ... 10
1.2.1.2 Whole Number Literals ... 10
1.2.1.3 Real Number Literals ... 11
1.2.2 String Literals ... 11
1.2.3 Structured Literals ... 11
1.3 Identifiers ... 12
1.3.1 User-definable Identifiers ... 12
1.3.2 Reserved Words ... 12
1.4 Non-Semantic Symbols ... 12
1.4.1 Pragmas ... 12
1.4.1.1 Language Defined Pragmas ... 13
1.4.1.2 Implementation Defined Pragmas ... 13
1.4.2 Comments ... 13
1.4.3 Lexical Separators ... 13
1.5 Symbols Reserved for Language Extensions and External Utilities ... 13
1.5.1 Symbols Reserved for Use by Language Extensions ... 13
1.5.1.1 Symbols Reserved for Parallel Modula-2 ... 14
1.5.1.2 Symbols Reserved for Objective Modula-2 ... 14
1.5.1.3 Symbols Reserved for Single-Pass Compilers ... 14
1.5.2 Symbols Reserved for Use by External Utilities ... 14
2 Compilation Units ... 15
2.1 Prototype Definitions ... 15
2.2 Program Modules ... 15
2.3 Definition Part of Library Modules ... 15
2.4 Implementation Part of Library Modules ... 16
2.5 Module Initialisation ... 16
2.6 Module Termination ... 16
3 Import of Identifiers ... 17
3.1 Qualified Import ... 17
3.1.1 Import Aggregation ... 17
3.1.2 Importing Modules as Types ... 17
3.2 Unqualified Import ... 18
3.2.1 Wildcard Import ... 18
4 Data Types ... 19
4.1 Type Compatibility ... 19
4.1.1 Alias Type Compatibility ... 19
4.1.2 Literal Compatibility ... 19
4.1.3 Assignment Compatibility ... 20
4.1.4 Parameter Passing Compatibility ... 20
4.1.4.1 Named Type Parameters ... 20
4.1.4.2 Open Array Parameters ... 20
4.1.4.3 Auto-Casting Open Array Parameters ... 20

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

4.1.4.4 Variadic Parameters......20
4.2 Type Conversions......21
4.2.1 Convertibility of Ordinal Types......21
4.2.2 Convertibility of Pervasive Numeric Types......21
4.2.3 Convertibility of Set Types......21
4.2.4 Convertibility of Array Types......21
4.2.5 Convertibility of Record Types......21
4.2.6 Convertibility of Pointer Types......21
4.2.7 Convertibility of Procedure Types......21
4.2.8 Convertibility of Opaque Types......21
4.2.9 Convertibility of Scalar Types......22
4.2.10 Non-Convertibility of SYSTEM Types......22
4.3 Semantics of Types......22
4.3.1 The Semantics of Ordinal Types......22
4.3.2 The Semantics of the Boolean Type......23
4.3.3 The Semantics of Set Types......23
4.3.4 The Semantics of Whole Number Types......23
4.3.5 The Semantics of Real Number Types......24
4.3.6 The Semantics of Array Types......24
4.3.7 The Semantics of Character String Types......25
4.3.8 The Semantics of Collection Types......25
4.3.9 The Semantics of Record Types......25
4.3.10 The Semantics of Pointer Types......26
4.3.11 The Semantics of Procedure Types......26
4.3.12 The Semantics of Opaque Types......26
4.3.13 The Semantics of SYSTEM Types......27
4.4 Library Defined Prototypes......27
4.4.1 Prototype ZTYPE......27
4.4.2 Prototype RTYPE......27
4.4.3 Prototype CTYPE......27
4.4.4 Prototype VTYPE......27
4.4.5 Prototype DateType......27
4.4.6 Prototype SetType......28
4.4.7 Prototype ArrayType......28
4.4.8 Prototype CollectionType......28
4.4.9 Prototype StringType......28
4.4.10 CAUTION - Do Not Modify Standard Library Prototypes......28
4.4.11 User Defined Prototypes......28
4.5 Abstract Data Types......28
4.6 Library Defined ADTs Using Prototypes and Bindings......29
4.6.1 Library Defined Bitset ADTs......29
4.6.2 Library Defined Unsigned Integer ADTs......30
4.6.3 Library Defined Signed Integer ADTs......30
4.6.4 Library Defined BCD Real Number ADTs......30
4.6.5 Library Defined Complex Number ADTs......31
4.6.6 Library Defined Character Set ADTs......31
4.6.7 Library Defined Character String ADTs......31
4.6.8 Library Defined DateTime ADTs......31

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

5 Definitions and Declarations...... 33
5.1 Constant Definitions and Declarations...... 33
5.2 Variable Definitions and Declarations...... 33
5.2.1 Global Variables...... 33
5.2.2 Local Variables...... 34
5.3 Type Definitions and Declarations...... 34
5.3.1 Strict Name Equivalence...... 34
5.3.2 Alias Types...... 34
5.3.3 Opaque Types...... 35
5.3.3.1 Opaque Pointers...... 35
5.3.3.2 Opaque Records...... 35
5.3.4 Anonymous Types...... 36
5.3.5 Enumeration Types...... 36
5.3.6 Array Types...... 37
5.3.6.1 Indexed Array Types...... 37
5.3.6.2 Associative Array Types...... 37
5.3.7 Record Types...... 37
5.3.8 Indeterminate Record Types...... 38
5.3.8.1 Declaration of Indeterminate Record Types...... 38
5.3.8.2 Allocating Indeterminate Records...... 38
5.3.8.3 Immutability of the Determinant Field...... 39
5.3.8.4 Run-time Bounds Checking...... 39
5.3.8.5 Assignment Compatibility...... 39
5.3.8.6 Parameter Passing...... 39
5.3.8.7 Deallocating Indeterminate Records...... 40
5.3.9 Set Types...... 40
5.3.10 Pointer Types...... 40
5.3.11 Procedure Types...... 40
5.4 Procedure Definitions and Declarations...... 41
5.4.1 The Procedure Header...... 41
5.4.2 The Procedure Body...... 42
5.4.3 Formal Parameters...... 42
5.4.3.1 Simple Formal Parameters...... 42
5.4.3.2 Pass By Value...... 43
5.4.3.3 Pass By Reference - Mutable...... 43
5.4.3.4 Pass By Reference - Immutable...... 43
5.4.3.5 Variadic Formal Parameters...... 43
5.4.3.6 Automatic Variadic Counter...... 44
5.4.3.7 Explicit Variadic Counter...... 44
5.4.3.8 Variadic List Terminator...... 45
5.4.3.9 Variadic List With Multiple Components...... 45
5.4.3.10 Variadic List Followed By Further Parameters...... 46
5.4.3.11 Open Array Parameters...... 46
5.4.3.12 Auto-Casting Open Array Parameters...... 46
5.4.4 Procedure Type Compatibility...... 47
5.4.5 Operator Bound Procedures...... 47
6 Statements...... 49
6.1 Assignments...... 49

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

6.2 Procedure Calls 49
6.3 Increment and Decrement Statements 49
6.4 IF Statements 50
6.5 CASE Statements 50
6.6 WHILE Statements 51
6.7 REPEAT Statements 51
6.8 LOOP and EXIT Statements 51
6.9 FOR statements 52
6.9.1 FOR IN statements 52
6.9.2 FOR TO BY statements 53
6.10 RETURN Statements 53
6.11 Statement Sequences 53
7 Expressions 55
7.1 Operands 55
7.2 Operators 56
7.3 Structured Values 56
8 Pervasive Identifiers 59
8.1 Predefined Constants 59
8.2 Predefined Types 59
8.3 Predefined Procedures 59
8.3.1 Procedure NEW 59
8.3.2 Procedure DISPOSE 60
8.3.3 Procedure READ 60
8.3.4 Procedure WRITE 61
8.3.5 Procedure WRITEF 61
8.4 Predefined Functions 62
8.4.1 Function ABS 62
8.4.2 Function NEG 62
8.4.3 Function ODD 63
8.4.4 Function PRED 63
8.4.5 Function SUCC 63
8.4.6 Function ORD 63
8.4.7 Function CHR 63
8.4.8 Function COUNT 63
8.4.9 Function SIZE 63
8.4.10 Function HIGH 64
8.4.11 Function LENGTH 64
8.4.12 Function NEXTV 64
8.4.13 Function TMIN 64
8.4.14 Function TMAX 64
8.4.15 Function TSIZE 64
8.5 Built-in Lexical Macros 65
8.5.1 Macro MIN 65
8.5.2 Macro MAX 65
9 Scalar Conversion Primitives 67
9.1 Primitive SXF 67
9.2 Primitive VAL 67

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10 Low-Level Facilities ... 69
10.1 Pseudo-Module SYSTEM ... 69
10.1.1 SYSTEM Constants ... 69
10.1.2 SYSTEM Types ... 69
10.1.3 SYSTEM Intrinsics ... 69
10.1.3.1 Intrinsic ADR ... 70
10.1.3.2 Intrinsic CAST ... 70
10.1.3.3 Intrinsic INC ... 70
10.1.3.4 Intrinsic DEC ... 70
10.1.3.5 Intrinsic ADDC ... 70
10.1.3.6 Intrinsic SUBC ... 70
10.1.3.7 Intrinsic SHL ... 70
10.1.3.8 Intrinsic SHR ... 70
10.1.3.9 Intrinsic ASHR ... 71
10.1.3.10 Intrinsic ROTL ... 71
10.1.3.11 Intrinsic ROTR ... 71
10.1.3.12 Intrinsic ROTLC ... 71
10.1.3.13 Intrinsic ROTRC ... 71
10.1.3.14 Intrinsic BWNOT ... 71
10.1.3.15 Intrinsic BWAND ... 71
10.1.3.16 Intrinsic BWOR ... 72
10.1.3.17 Intrinsic BWXOR ... 72
10.1.3.18 Intrinsic BWNAND ... 72
10.1.3.19 Intrinsic BWNOR ... 72
10.1.3.20 Intrinsic SETBIT ... 72
10.1.3.21 Intrinsic TESTBIT ... 72
10.1.3.22 Intrinsic LSBIT ... 72
10.1.3.23 Intrinsic MSBIT ... 73
10.1.3.24 Intrinsic CSBITS ... 73
10.1.3.25 Intrinsic BAIL ... 73
10.1.3.26 Intrinsic HALT ... 73
10.2 Pseudo-Module ATOMIC ... 73
10.2.1 Testing The Availability Of Atomic Intrinsics ... 73
10.2.2 ATOMIC Intrinsics ... 73
10.2.2.1 Intrinsic SWAP ... 74
10.2.2.2 Intrinsic CAS ... 74
10.2.2.3 Intrinsic INC ... 74
10.2.2.4 Intrinsic DEC ... 74
10.2.2.5 Intrinsic BWAND ... 74
10.2.2.5 Intrinsic BWNAND ... 74
10.2.2.6 Intrinsic BWOR ... 74
10.2.2.7 Intrinsic BWXOR ... 75
10.3 Pseudo-Module COROUTINES ... 75
10.4 Pseudo-Module ASSEMBLER ... 75
11 Pragmas ... 77
11.1 Language Defined Pragmas ... 77
11.1.1 Conditional Compilation Pragmas ... 77
11.1.2 Compile Time Console Message Pragmas ... 77

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

11.1.3 Code Generation Pragmas ... 78
11.2 Implementation Defined Pragmas ... 78
12 Generics ... 79
14 Standard Library ... 81
14.1 Pseudo Modules and Documentation Modules ... 81
14.2 Prototype Library ... 81
14.3 Memory Management Modules ... 81
14.4 Modules for Exception Handling and Termination ... 81
14.5 File System Modules ... 81
14.6 File IO Modules ... 82
14.7 IO Modules for SYSTEM Types ... 82
14.8 IO Modules for Pervasive Types ... 82
14.9 Library Modules Implementing Basic Types ... 82
14.10 Modules Defining Alias Types ... 83
14.11 Modules Providing Math for Basic Types ... 83
14.12 Modules Providing Primitives for Text Handling ... 83
14.13 Modules for Date and Time Handling ... 83
14.14 Modules with Legacy Interfaces ... 83
14.15 Template Library ... 83
Appendix A: Grammar in EBNF ... 85
A.1 Non-Terminal Symbols ... 85
A.2 Pragmas ... 92
A.3 Terminal Symbols ... 93
A.4 Ignore Symbols ... 94
Appendix B: Syntax Diagrams ... 95
B.1 Non-Terminal Symbols ... 95
B.2 Pragmas ... 106
B.3 Terminal Symbols ... 107
B.4 Ignore Symbols ... 109

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1 Lexical Entities

1.1 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, obsoletes `<>`
`*` 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 operator
`/` 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 a quoted string

`[ ]` 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 literals

`!` 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 delimiter for pragmas
`*>` closing delimiter for pragmas

`//` prefix for single-line comment
`(*` opening delimiter for multi-line comment
`*)` closing delimiter for multi-line comment

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1.2 Literals

There are three types of literals:

- numeric literals
- string literals
- structured literals

1.2.1 Numeric literals

There are three types of numeric literals

- character code literals
- whole number literals
- real number literals

1.2.1.1 Character Code Literals

Character code literals represent Unicode code points. They are encoded as base-16 whole numbers using lowercase digits with prefix `0u` or uppercase digits with suffix `U`.

EBNF:

```
CharCodeLiteral : "0u" Base16DigitLowercase+ | Digit Base16Digit* "U" ;
Base16DigitLowercase : Digit | "a" | "b" | "c" | "d" | "e" | "f" ;
Base16Digit : Digit | "A" | "B" | "C" | "D" | "E" | "F" ;
Digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Examples:

```
0u7f, 7FU (* DEL *); 0ua9, 0A9U (* copyright *); 0u20ac, 20ACU (* Euro sign *)
```

1.2.1.2 Whole Number Literals

There are three types of whole number literals

- base-10, without prefix or suffix
- base-2, with prefix `0b` or suffix `B`
- base-16, lowercase digits with prefix `0x` or uppercase digits with suffix `H`

EBNF:

```
DecimalLiteral : Digit+ ;
BinaryLiteral : "0b" BinaryDigit+ | BinaryDigit+ "B" ;
Base16Literal : "0x" Base16DigitLowercase+ | Digit Base16Digit* "H" ;
BinaryDigit : "0" | "1" ;
```

Examples:

```
42, 32767 (* decimal *)
0b1101010, 01101010B (* base-2 *)
0xdeadbeef, 0DEADBEEFH (* base-16 *)
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1.2.1.3 Real Number Literals

Real number literals are base-10 numbers, always with an integral part, a decimal point and a fractional part, optionally followed by an exponent.

EBNF:

```
RealLiteral : Digit+ "." Digit+ ( ( "e" | "E" ) ( "+" | "-" )? Digit+ )? ;
```

Examples:

```
0.0, 123.45, 3.1415926
7.523012e+12, 7.523012E+12
```

1.2.2 String Literals

String literals are sequences of printable characters and optional escape sequences, enclosed in single quotes or double quotes. String literals must not span multiple lines.

EBNF:

```
String : "'" ( Character | EscapeSequence | '"' )* "'" |
         '"' ( Character | EscapeSequence | "'" )* '"' ;

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

Digit : "0" .. "9" ;

Letter : "A" .. "Z" | "a" .. "z" ;

EscapeSequence : "\" ( "0" | "n" | "r" | "t" | "\" | "'" | '"' ) ;
```

Examples:

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

1.2.3 Structured Literals

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

EBNF:

```
structuredLiteral : "{" otherSymbols? "}" ;
```

Examples:

```
{ 1, 2, 3 }
{ "a", "b", "c" }
{ 42, 3.1415926, "abc" }
{ { 1, 2, 3 }, { "a", "b", "c" }, 42 }
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1.3 Identifiers

Identifiers are names for syntactic entities in a program. There are two types of identifiers:

- user-definable identifiers
- language defined identifiers, aka reserved words

1.3.1 User-definable Identifiers

Identifiers are names that start with a letter, lowline or dollar sign, followed by any number and combination of letters, lowlines, dollar signs and digits. User definable identifiers must not coincide with reserved words. The use of the dollar sign in identifiers should be limited to referencing entities defined by an operating system API for public use (e.g. OpenVMS).

EBNF:

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

1.3.2 Reserved Words

Reserved words are identifiers that have special meaning in the language.

List of Reserved Words

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

1.4 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 of non-semantic symbols:

- pragmas
- comments
- lexical separators

1.4.1 Pragmas

Pragmas are directives used to control the translation process but do not change the meaning of the program text. Pragmas consist of a pragma name optionally followed by other symbols, delimited by `<*` and `*>`. Pragma names never conflict with identifiers or reserved words because they can only occur within pragma delimiters. There are two types of pragmas:

- language defined pragmas
- implementation defined pragmas

EBNF:

```
pragma : "<*" pragmaName otherSymbols? "*>" ;
pragmaName : Ident ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1.4.1.1 Language Defined Pragmas

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

List of Language Defined Pragma Names

```
IF          ENDIF       ERROR       FOREIGN     NOINLINE
ELSE        INFO        FATAL       MAKE        VOLATILE
ELSIF       WARN        ALIGN       INLINE
```

1.4.1.2 Implementation Defined Pragmas

Any implementation may define its own set of pragmas, specific to the compiler. Implementation defined pragma names must not be all-uppercase words.

1.4.2 Comments

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

- single-line comments
- multi-line comments

Multi-line comments may be nested up to a nesting level of ten, including the outermost comment. A compile time error occurs if this limit is exceeded.

EBNF:

```
Comment : SingleLineComment | MultiLineComment ;
SingleLineComment : "//" ~( EndOfLine )* EndOfLine ;
MultiLineComment : "(*" ~( "*)" )* MultiLineComment* "*)" ;
```

Examples:

```
// comment until end-of-line
(* This is a comment (* and a comment within *) *)
(* Level L1 (* L2 (* L3 (* L4 (* L5 (* L6 (* L7 (* L8 (* L9 (* L10 (* Error
```

1.4.3 Lexical Separators

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

EBNF:

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

1.5 Symbols Reserved for Language Extensions and External Utilities

1.5.1 Symbols Reserved for Use by Language Extensions

Although not part of the language specification, certain symbols are reserved specifically for use by language extensions. Three such language extensions are explicitly supported:

- Parallel Modula-2
- Objective Modula-2
- single-pass compilers

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

1.5.1.1 Symbols Reserved for Parallel Modula-2

The following identifiers and pragma names are reserved for exclusive use by the Parallel Modula-2 language superset:

Reserved words:

```
ALL PARALLEL SYNC
```

Pragma names:

```
LOCAL SPREAD CYCLE SBLOCK CBLOCK
```

1.5.1.2 Symbols Reserved for Objective Modula-2

The following identifiers and pragma names are reserved for exclusive use by the Objective Modula-2 language superset:

Reserved words:

```
BYCOPY BYREF CLASS CONTINUE CRITICAL INOUT METHOD ON OPTIONAL OUT PRIVATE PRO-
TECTED PROTOCOL PUBLIC SUPER TRY
```

Pragma names:

```
FRAMEWORK IBACTION IBOUTLET QUALIFIED
```

1.5.1.3 Symbols Reserved for Single-Pass Compilers

The following pragma name is reserved for use by single-pass compilers:

Pragma name:

```
FORWARD
```

1.5.2 Symbols Reserved for Use by External Utilities

To assist external source code processing prior to compilation, the following symbols are reserved for exclusive use by external source code processors:

`!!` and `??` reserved for use by editors or di-/tri-graph converters
`@@` and `%%` reserved for use by the Modula-2 Template Engine or make utilities

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 prototype
- a program module
- the interface part of a library module
- the implementation part of a library module

EBNF:

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

2.1 Prototype Definitions

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

EBNF:

```
prototype :
    PROTOTYPE prototypeId ";"
    TYPE "="
    ( RECORD | OPAQUE RECORD? ( ":=" ( literalType | "{" ".." "}" ) )? ) ";"
    ( ASSOCIATIVE ";" )?
    requiredBinding*
    END prototypeId "." ;

prototypeId : Ident ;
```

2.2 Program Modules

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

EBNF:

```
programModule :
    MODULE moduleId ( "[" priority "]" )? ";"
    importList* block moduleId "." ;

moduleId : Ident ;
```

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 "." ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 of 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.

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.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 moduleId IMPORT ( identList | "*" ) |
    IMPORT Ident "+"? ( "," Ident "+"? )* ) ";"
```

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 to allow clients to import an entire library with a single import statement.

Example:

```
DEFINITION MODULE FooBarBaz;
IMPORT Foo+, Bar+, Baz+; (* import Foo, Bar and Baz into client module *)
END FooBarBaz.
IMPORT FooBarBaz; (* imports all modules imported by FooBarBaz and marked + *)
```

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;
VAR colour : Colour;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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, a name conflict will occur and cause a compile time error. In most cases qualified import should therefore be used instead.

Example:

```
FROM FileIO IMPORT Status; (* unqualified import of Status *)
VAR status : Status; (* unqualified identifier of Status *)
```

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 caution because it increases the likelihood of name conflicts.

Example:

```
FROM FileIO IMPORT *; (* import all identifiers from module FileIO *)
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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.

There are twelve language predefined (pervasive) data types:

```
BOOLEAN, BITSET, LONGBITSET, CHAR, UNICHAR, OCTET, CARDINAL, LONGCARD,
INTEGER, LONGINT, REAL and 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 one type is defined as an alias type of the other, or unless they are both defined as alias types of the same type.

Type compatibility is transitive. If types `T1` and `T2` are compatible and types `T2` and `T3` are compatible, then types `T1` and `T3` are also compatible.

4.1.1 Alias Type Compatibility

If type `A` is an alias type of type `T`, then `A` is compatible with `T`.

4.1.2 Literal Compatibility

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 ADT whose prototype 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 whose prototype 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 whose prototype 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

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

- any ADT whose prototype permits the use of structured literals
  provided the literal is structurally equivalent to the variadic parameter of the procedure that the ADT defines to bind to the assignment operator

4.1.3 Assignment Compatibility

An expression `e` may be assigned to a mutable variable `v` if:

- both `e` and v are of the same type
- the type of `e` is an alias type of `v` or vice versa
- the type of `e` and the type of `v` are both alias types of the same type
- `e` is a literal that is compatible under literal compatibility rules

Immutable variables may not be assigned to, regardless of type compatibility.

4.1.4 Parameter Passing Compatibility

4.1.4.1 Named Type Parameters

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`

4.1.4.2 Open Array Parameters

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`

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`

4.1.4.3 Auto-Casting Open Array Parameters

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`

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`

4.1.4.4 Variadic Parameters

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

- the formal variadic parameter is the last parameter of the procedure
  and the list is structurally equivalent to the formal variadic parameter

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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

- 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`.

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 they 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` that is bound to the conversion operator, and `v` is a legal value of `T2`

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

- `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.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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`. 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 structured literals and expressions (`:=`)
- type conversion (`::`)
- number of actual elements (`COUNT`)
- ordinal value of highest formal element (`HIGH`)
- 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 whole number 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`)

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

- 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 subranges of 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 real number 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`)

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

- retrieve component (`value := array[index]`)
- ordinal value of highest index (`HIGH`)
- 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 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]`)
- ordinal value of highest index (`HIGH`)
- 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 objects 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`)
- 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 (#)

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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.

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 of the implementation part the following operations are defined:

for opaque pointers:

- allocation (`NEW`)
- deallocation (`DISPOSE`)
- assignment (`:=`) of `NIL` and objects of the same type only
- equal (=)
- not-equal (#)

for opaque records:

- assignment (`:=`) of objects 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 a the `OPAQUE RECORD` type constructor are opaque record types.

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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`)
- 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 in order 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 library defined prototype, the library that defines the ADT must specify the prototype in the module header of its definition part.

4.4.1 Prototype ZTYPE

The standard library provides the `ZTYPE` as a library defined prototype for library defined whole number types. For semantic compatibility, its definition matches the semantics of the built-in prototype for whole number types.

4.4.2 Prototype RTYPE

The standard library provides the `RTYPE` as a library defined prototype for library defined real number types. For semantic compatibility, its definition matches the semantics of the built-in prototype for real number types.

4.4.3 Prototype CTYPE

The standard library provides the `CTYPE` as a library defined prototype for library defined complex number types. Its definition matches the mathematic definition of complex numbers. `CTYPE` has no built-in equivalent.

4.4.4 Prototype VTYPE

The standard library provides the `VTYPE` as a library defined prototype for library defined numeric vector types. Its definition matches the mathematic definition of numeric vectors.
`VTYPE` has no built-in equivalent.

4.4.5 Prototype DateType

The standard library provides the `DateType` as a library defined prototype for library defined datetime types. `DateType` has no built-in equivalent.

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

4.4.6 Prototype SetType

The standard library provides the `SetType` as a library defined prototype for library defined static set types. For semantic compatibility, its definition matches the semantics of the built-in prototype for set types.

4.4.7 Prototype ArrayType

The standard library provides the `ArrayType` as a library defined prototype for library defined dynamic array types. For semantic compatibility, its definition matches the semantics of the built-in prototype for array types.

4.4.8 Prototype CollectionType

The standard library provides the `CollectionType` as a library defined prototype for library defined dynamic collection types. For semantic compatibility, its definition matches the semantics of the built-in prototype for collection types.

4.4.9 Prototype StringType

The standard library provides the `StringType` as a library defined prototype for library defined dynamic character string types. For semantic compatibility, its definition matches the semantics of the built-in prototype for character string types.

4.4.10 CAUTION - Do Not Modify Standard Library Prototypes

For semantic compatibility, the prototype definitions provided by the standard library exactly match the semantics of their corresponding built-in counterparts. Modification of these prototype definitions will cause library defined types to behave differently from their built-in counterparts. For this reason, standard library prototypes should not be changed.

4.4.11 User Defined Prototypes

User libraries may provide their own prototype definitions for their own custom designed abstract data types.

4.5 Abstract Data Types

Opaque pointer types and opaque record types are predominantly used 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 are superimposed by a new set of 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.

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.

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:

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

- 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:

```
DEFINITION MODULE BCD [RTYPE];
FROM FileIO IMPORT File;
TYPE BCD = OPAQUE RECORD value : ARRAY 8 OF OCTET END;
PROCEDURE [:=] assign ( VAR assignTo : BCD; CONST literal : ARRAY OF CHAR );
PROCEDURE [+] add ( a, b : BCD ) : BCD;
...
PROCEDURE Write ( f : File; b : BCD );
...
END BCD.

MODULE UseBCD;
IMPORT BCD;
VAR a, b, sum : BCD;
BEGIN
    a := 1.5; (* replaced by BCD.assign(a, "1.5"); *)
    b := 2.75; (* replaced by BCD.assign(b, "2.75"); *)
    sum := a + b; (* replaced by sum := BCD.add(a, b); *)
    WRITE(stdOut, sum); (* replaced by BCD.Write(stdOut, sum); *)
END UseBCD.
```

4.6 Library Defined ADTs Using Prototypes and Bindings

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

4.6.1 Library Defined Bitset ADTs

The standard library provides bitset ADTs whose semantics match those of the pervasive bitset types `BITSET` and `LONGBITSET`. The library defined bitset ADTs are `BS16`, `BS32`, `BS64`, `BS128` and their respective number of bits is indicated in the identifier. However, client modules should not use these types using these identifiers directly.

The alias type identifiers `BITSET16`, `BITSET32`, `BITSET64` and `BITSET128` should be used instead. The aliases of whichever bitset types match the number of bits of the pervasive types `BITSET` and `LONGBITSET` are automatically defined as aliases of the pervasive types.

Two additional alias types are defined as `SHORTBITSET` and `LONGLONGBITSET`.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Example:

```
IMPORT Bitsets;
VAR set, union : BITSET16;
BEGIN set := { 0, 7, 15 }; union := set + { 1, 2, 4 }; set[1] := FALSE END;
```

4.6.2 Library Defined Unsigned Integer ADTs

The standard library provides unsigned integer ADTs whose semantics match those of the pervasive types `CARDINAL` and `LONGCARD`. The library defined unsigned integer ADTs are `CARD16`, `CARD32`, `CARD64`, `CARD128` and their respective bit width is indicated in the identifier. However, client modules should not use these types using these identifiers directly.

The alias type identifiers `CARDINAL16`, `CARDINAL32`, `CARDINAL64` and `CARDINAL128` should be used instead. The aliases of whichever types match the size of the pervasive types `CARDINAL` and `LONGCARD` are automatically defined as aliases of the pervasive types.

Two additional alias types are defined as `SHORTCARD` and `LONGLONGCARD`.

Example:

```
IMPORT Cardinals;
VAR a, sum : CARDINAL16;
BEGIN a := 123; sum := a + 456; WRITE(stdOut, sum) END;
```

4.6.3 Library Defined Signed Integer ADTs

The standard library provides signed integer ADTs whose semantics match those of the pervasive types `INTEGER` and `LONGINT`. The library defined signed integer ADTs are `INT16`, `INT32`, `INT64`, `INT128` and their respective bit width is indicated in the identifier. However, client modules should not use these types using these identifiers directly.

The alias type identifiers `INTEGER16`, `INTEGER32`, `INTEGER64` and `INTEGER128` should be used instead. The aliases of whichever types match the size of the pervasive types `INTEGER` and `LONGINT` are automatically defined as aliases of the pervasive types.

Two additional alias types are defined as `SHORTINT` and `LONGLONGINT`.

Example:

```
IMPORT Integers;
VAR a, diff : INTEGER16;
BEGIN a := 123; diff := a - 456; WRITE(stdOut, diff) END;
```

4.6.4 Library Defined BCD Real Number ADTs

The standard library provides Binary Coded Decimal (BCD) real number ADTs whose semantics match those of the pervasive types `REAL` and `LONGREAL`. The library defined BCD ADTs are `BCD` and `LONGBCD`.

Example:

```
IMPORT BCD;
VAR a, amount : BCD;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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

4.6.5 Library Defined Complex Number ADTs

The standard library provides complex number ADTs whose semantics conform to prototype `CTYPE`. The library defined complex number ADTs are `COMPLEX` and `LONGCOMPLEX`.

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 Library Defined Character Set ADTs

The standard library provides a character set ADT whose semantics conform to prototype `SetType`. The library defined character set ADT is `CHARSET`.

Example:

```
IMPORT CHARSET;
VAR delimiters : CHARSET; found : BOOLEAN;
BEGIN found := FALSE; delimiters := { ":", ",", "." };
FOR char IN "foo:bar.baz,bam" DO IF char IN delimiters THEN EXIT END END END;
```

4.6.7 Library Defined Character String ADTs

The standard library provides dynamic string ADTs whose semantics conform to prototype `StringType`. The library defined dynamic string ADTs are `STRING` and `UNISTRING`.

Example:

```
IMPORT STRING;
VAR s : STRING;
BEGIN NEW(s, 20); s := "the quick brown fox"; WRITE(stdOut, s); DISPOSE(s) END;
```

4.6.8 Library Defined DateTime ADTs

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

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;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 a variant 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 ( Ident ( "[" address "]" | "," identList )?
          :  ( ARRAY constComponentCount OF )? namedType ";" )* ;

varDeclaration : varDefinition ;
```

Examples:

```
VAR x, y : REAL;

VAR portA [mappedIOAddr] : OCTET;
```

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 it is defined or declared in and within modules that import it.

A variable that is defined in the top level of a library module is always exported immutable. It may be modified within the library module's implementation part but it cannot be modified within modules that import it.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 it is declared in and within procedures local to the procedure it is declared in.

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.

Example:

```
TYPE INT = ALIAS OF INTEGER;

VAR i : INT; j : INTEGER;

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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. Objects 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. Objects of the abstract data type can be allocated either statically as global or local variables or dynamically.

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 where it is defined but it is semantically treated as if it was hidden in the corresponding implementation part. Only the identifier of an opaque record is visible to modules that import it. Its internal structure is not available to them.

EBNF:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Example:

```
DEFINITION MODULE BigInteger;
TYPE BigInteger = OPAQUE RECORD highDigits, lowDigits : INTEGER END;
(* public interface *)
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. However, 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 allow the construction of VLAs, Modula-2 R10 supports the use of anonymous one-dimensional arrays in record fields and in variable declarations.

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 ) )* ")" ;
```

Example:

```
TYPE Colour = ( red, green, blue );
```

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: ( red, green, blue, orange, magenta, cyan ) *)
```

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:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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.

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 may not be an opaque record. The names of the fields of the base type may not be used as field names in the type extension.

Record types are defined using the RECORD type constructor.

EBNF:

```
recordType :
    RECORD ( "(" baseType ")" )? fieldListSequence? END ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
fieldListSequence :
    fieldList ( ";" fieldList )* ;

fieldList :
    ident
    ( ( "," ident )+ ":" namedType |
      ":" ( ARRAY determinantField OF )? namedType )

baseType : Ident ;

determinantField : Ident ;
```

Examples:

```
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 with an indeterminate field and a determinant 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 determinant field is a field that determines the size of an indeterminate field.

5.3.8.1 Declaration of Indeterminate Record Types

Indeterminate record types may be declared provided that the type declaration meets all of the following constraints:

- it contains only one indeterminate field
- the indeterminate field is the last field
- the type of its indeterminate field is an array
- it also contains a determinant field of a whole number type
- the determinant field is referenced as the indeterminate field's array size

Indeterminate record types are the type-safe equivalent of structs with variable length array (VLA) members in C99, sometimes called flex-arrays.

Example:

```
TYPE VLA = POINTER TO VLADescriptor;

TYPE VLADescriptor = RECORD

    a, b, c : Foo; (* other fields *)

    size    : CARDINAL; (* determinant field *)

    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 determinant value must be passed to `NEW` as an additional parameter. Any attempt to allocate a record of indeterminate type without passing the determinant value results in a compile time error.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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) + determinant * TSIZE(baseType(T.indeterminateField))
```

where `T` is the indeterminate record type, `determinant` is the determinant 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));
```

5.3.8.3 Immutability of the Determinant Field

The determinant field of a record of indeterminate type is automatically initialised when it is allocated. After initialisation the determinant field becomes immutable and the compiler enforces its immutability as follows:

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

Example:

```
INC(vla^.size); (* determinant field may not be passed as VAR parameter *)
vla^.size := 42; (* determinant field may not be assigned to *)
vla^.size++; (* determinant 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 run-time in the same manner as access to a determinate array is checked. The compiler automatically inserts the code to check array indices against the determinant field. Any attempt to access the array with a subscript that is out of bounds results 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 only be passed to an auto-casing formal open array parameter `CAST ARRAY OF OCTET`, `CAST ARRAY OF SYSTEM.BYTE` or `CAST ARRAY OF SYSTEM.WORD`.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 static collection type for a fixed number of key/value pairs. Its keys are of ordinal type and its values are of type `BOOLEAN`. A key whose value is `TRUE` is said to be an element of the set. Set types are defined using the `SET OF` type constructor.

EBNF:

```
setType :
    SET OF ( namedEnumType | "(" identList ")" ) ;
namedEnumType : namedType;
```

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? namedType ;
```

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 *)
immPtr^ := 0; (* compile time error: attempt to modify an immutable object *)
```

5.3.11 Procedure Types

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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 actions to perform a computation 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 proper 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 a procedure's identifier. 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 definition of the prototype. The signature of a procedure header that binds to an operator or pervasive procedure must conform to the language defined signature for the respective operator. A list of procedure signatures for bindable operators and pervasive procedures is provided in documentation module `BINDINGS.def`.

EBNF:

```
procedureHeader :
    PROCEDURE
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
    ( "[" ( "::" | bindableOperator | bindableIdent ) "]" )?
    ident ( "(" formalParamList ")" )? ( ":" returnedType )? ;
procedureIdent : Ident ;
```

Example:

```
PROCEDURE IsNegative ( x : INTEGER ) : BOOLEAN;
```

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, 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 passing conventions:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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.

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 passed in variable's value.

Immutable objects 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); (* number is now 2 *)
CONST zero = 0; Increment(zero); (* compile time error: immutable object *)
```

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. Mutable and immutable objects 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

Modula-2 R10 supports type-safe variadic procedures. For this purpose, a formal parameter list may contain one or more variadic formal parameter lists. A variadic formal parameter is denoted by reserved word `VARIADIC`.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

EBNF:

```
variadicFormalParams :
    VARIADIC ( variadicCounter | "[" variadicTerminator "]" )? OF
    ( simpleFormalType |
      "(" simpleFormalParams ( ";" simpleFormalParams )* ")" ) ;

variadicCounter : Ident ;

variadicTerminator : constExpression ;
```

There are three variadic parameter passing conventions:

- automatic variadic counter
- explicit variadic counter
- variadic list terminator

5.4.3.6 Automatic Variadic Counter

When the automatic-variadic-counter convention is used, the compiler determines the number of actual parameters passed in the procedure call and automatically inserts the count as an additional parameter immediately before the variadic list. Within the procedure the variadic parameter is presented as an array and pervasive function `HIGH` is used to obtain its highest index.

Example:

```
PROCEDURE Variadic ( v : VARIADIC OF INTEGER );
BEGIN
    FOR item IN v DO WRITE(f, item) END;
    RETURN
END Variadic;
```

invoked as:

```
Variadic(0, 1, 2, 3, 4);
```

compiled as:

```
Variadic(5, 0, 1, 2, 3, 4); (* compiler inserts counter *)
```

5.4.3.7 Explicit Variadic Counter

When the explicit-variadic-counter convention is used, the compiler determines the number of actual parameters passed in the procedure call and automatically inserts the count at the position indicated by the formal count parameter in the procedure's formal parameter list. This facility is useful to map to C APIs with variadic functions that use a counter that is not placed immediately before the variadic list. Within the procedure the variadic parameter is presented as an array and the counter contains the number of items.

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Example:

```
PROCEDURE Variadic ( count, x : CARDINAL; v : VARIADIC count OF INTEGER );
BEGIN
    FOR item IN v DO WRITE(f, v[item]) END;
    RETURN
END Variadic;
```

invoked as:

```
Variadic(42, 0, 1, 2, 3, 4);
```

compiled as:

```
Variadic(5, 42, 0, 1, 2, 3, 4); (* compiler inserts counter *)
```

5.4.3.8 Variadic List Terminator

When the variadic-list-terminator convention is used, the compiler determines the number of actual parameters passed in the procedure call and automatically inserts the terminator value indicated by the procedure's formal parameter after the variadic list. This facility is useful to map to C APIs with variadic functions that use a terminator value to indicate the end of a variadic list.

Within the procedure, pervasive function `NEXTV` is used to traverse the list of arguments in the variadic list. Each time `NEXTV` is called it returns a pointer to the next variadic argument or tuple.

Example:

```
PROCEDURE Variadic ( v : VARIADIC [-1] OF INTEGER );
VAR item : INTEGER;
BEGIN
    item := NEXTV(v); WHILE item # -1 DO WRITE(f, item); item := NEXTV(v) END;
    RETURN
END Variadic;
```

invoked as:

```
Variadic(0, 1, 2, 3, 4);
```

compiled as:

```
Variadic(0, 1, 2, 3, 4, -1); (* compiler inserts terminator *)
```

5.4.3.9 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;
                   keysAndValues : VARIADIC OF ( key : Key; value : Value ) );
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

invoked as:

```
Insert(tree, "foo", 123, "bar", 456, "baz", 789);
```

compiled as:

```
Insert(tree, 3, "foo", 123, "bar", 456, "baz", 789);
```

5.4.3.10 Variadic List Followed By Further Parameters

A variadic parameter list may always be passed as a structured value, regardless of the formal parameter list of the variadic procedure. However, when 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 where the variadic list ends.

Example:

```
PROCEDURE Insert ( tree : Tree;
                   keysAndValues : VARIADIC OF ( key : Key; value : Value );
                   VAR status : Status );
```

invoked as:

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

compiled as:

```
Insert(tree, 3, "foo", 123, "bar", 456, "baz", 789, status);
```

5.4.3.11 Open Array Parameters

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

For a procedure with an open array parameter to determine the highest index of the array passed to it, the value of the highest index is automatically passed as a hidden parameter. Within the procedure, pervasive function `HIGH` may be used to obtain the value of the highest index of the passed in array.

Example:

```
TYPE String10 = ARRAY 10 OF CHAR;
VAR str : String10;
PROCEDURE Write ( s : ARRAY OF CHAR );
str := "hello"; Write(str); (* any CHAR array is compatible *)
```

5.4.3.12 Auto-Casting Open Array Parameters

A formal parameter may be declared as an auto-casting open array parameter with component type `OCTET`, `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. To declare an auto-casting formal parameter, `CAST` must be explicitly imported from pseudo-module `SYSTEM`.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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. 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. Only bindings required by the prototype the ADT conforms to may be defined except for bindings to the conversion operator which are always permitted.

Example:

```
DEFINITION MODULE BCD [RTYPE];

TYPE BCD = OPAQUE RECORD value : ARRAY 8 OF OCTET END;

PROCEDURE [+] add ( a, b : BCD ) : BCD;
...

END BCD.
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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
- procedure calls
- increment/decrement statements
- 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 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 Procedure Calls

A procedure call statement is used to invoke a procedure. A procedure call may include a list of parameters to be passed to the called procedure. Parameters passed are called actual parameters. Parameters defined in the procedure's definition and declaration are called formal parameters. In a procedure call with parameters, the types of actual and formal parameters must match. Procedure calls may be recursive, that is, a procedure may call itself.

EBNF:

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

Examples:

```
Write(stdOut, ch); WriteLn;
```

6.3 Increment and Decrement Statements

Increment and decrement statements are used to increment or decrement numeric variables by one.

EBNF:

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


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Example;

```
VAR counter : CARDINAL;
counter--;
```

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`.

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 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 *)
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
END;
```

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 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 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 and EXIT Statements

The `LOOP` statement is used to repeat a statement or sequence of statements indefinitely unless explicitly terminated by an `EXIT` statement. An `EXIT` statement terminates the loop body in which it is invoked and it transfers control to the first statement after the loop body. The loop body may be the body of a `LOOP`, `WHILE`, `REPEAT` or `FOR` statement.

EBNF:

```
loopStatement : LOOP statementSequence END ;
exitStatement : EXIT ;
```

Example:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

6.9 FOR statements

The FOR statement is used to repeat 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 POINTER TO CONST pointers

There are two types of FOR statements:

- FOR IN statements
- FOR TO BY statements

Where possible, the use of FOR IN statements should be given preference.

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 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 ( expression | range OF namedType ) DO statementSequence END ;

controlVariable : Ident;
```

Examples:

```
(* iterating over all values of an ordinal type *)
FOR status IN Status DO WRITE(file, NameOfStatus(status)) END;

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

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

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

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

6.9.2 FOR TO BY statements

FOR TO BY statements iterate over an explicit whole number range denoted by a start and stop value and an optional increment value. The control variable is assigned a value of the whole number range at each iteration and its type must be explicitly declared in the loop header. The type must be a whole number type.

EBNF:

```
forToByStatement :
    FOR controlVariable ":" namedWholeNumberType ":=" expression TO expression
    ( BY constExpression )? DO statementSequence END ;
```

Example:

```
FOR count : CARDINAL := 99 TO 0 BY -1 DO BottlesOfBeer(count) END;
```

NB: FOR TO BY statements are being considered for removal.

6.10 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 Mean ( x, y : REAL ) : REAL ;
BEGIN
    RETURN (x + y) / 2;
END Mean;
```

6.11 Statement Sequences

Statements in a sequence of statements are separated by semicolons.

EBNF:

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

Example:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

7 Expressions

An expression is a computational formula that evaluates to a computational 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. An expression in which only constant operands are permitted is called a constant expression. Constant expressions can always be evaluated at compile time.

Operators are special symbols or reserved words that represent an operation to be carried out using one or more operands to compute a value. An operator that acts on a single operand is called a monadic operator and its operation is called a monadic operation. An operator that acts on a pair of operands is called a dyadic operator and its operation is called a dyadic operation.

Pairs of matching parentheses may be used in an expression to control the order in which its sub-expressions are evaluated.

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 ;

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 ;
```

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 dereference symbol following a pointer, a field of a record or a procedure's actual parameter list.

EBNF:

```
designatorOrProcedureCall :
    qualident designatorTail? actualParameters? ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
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 )? ;
```

Static semantics:

An expression in a structured value that is followed by the range operator `..` must be a constant expression.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Examples:

```
{ x BY 100 }, { 1 .. 31 }
{ "abc", 123, 456.78, { 1, 2, 3 } }
{ 1970, Month.Jan, 1, 0, 0, 0.0, TZ.UTC }
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

8 Pervasive Identifiers

Pervasive Identifiers are predefined identifiers that are available in every scope of a program without having to import them. Unlike reserved words, pervasive identifiers may be redefined by libraries or program modules. There are five groups of pervasive identifiers:

- predefined constants
- predefined types
- predefined procedures
- predefined functions
- built-in lexical macros

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
`LONGREAL`	double precision real number type

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.

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, may not be passed to a procedure as parameters and calls to them are replaced by the compiler with 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> );
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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.

Example:

```
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 determinant value for the type.

Example:

```
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 a library procedure `DEALLOCATE` which must be imported before `DISPOSE` can be used. The standard library provides a `DEALLOCATE` procedure 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 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> );
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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); *)
```

8.3.4 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); *)
```

8.3.5 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:

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
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 SIZE HIGH LENGTH NEXTV TMIN TMAX TSIZE
```

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 numeric type. Its return type is always the same as the operand type. Its pseudo-definition is:

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

8.4.3 Function ODD

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

```
PROCEDURE ODD ( x : <Z-Type> ) : 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. 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 of any set type or collection type. Its return type is `LONGCARD`. Its pseudo-definition is:

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

8.4.9 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 ;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

8.4.10 Function HIGH

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

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

8.4.11 Function LENGTH

Function `LENGTH` returns the number of characters 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 : <CharacterString> ) : LONGCARD ;
```

8.4.12 Function NEXTV

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

```
PROCEDURE NEXTV ( ident : <VariadicList> ) : <PointerToVariadicBaseType> ;
```

8.4.13 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> ;
```

8.4.14 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> ;
```

8.4.15 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 ;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

8.5 Built-in Lexical Macros

User-definable lexical macros are not provided in Modula-2 (on purpose), but there are two built-in lexical macros: `MIN` and `MAX`.

8.5.1 Macro MIN

Macro `MIN` is replaced with the smallest constant from its variadic argument list. All arguments must be scalar constants or scalar constant expressions. Its pseudo-definition is:

```
PROCEDURE MIN ( constant : VARIADIC OF <ScalarType> ) : <OperandType>;
```

Example:

```
MIN( 1, 2, 3 ) => 1
```

8.5.2 Macro MAX

Macro `MAX` is replaced with the largest constant from its variadic argument list. All arguments must be scalar constants or scalar constant expressions. Its pseudo-definition is:

```
PROCEDURE MAX ( constant : VARIADIC OF <ScalarType> ) : <OperandType>;
```

Example:

```
MAX( 1, 2, 3 ) => 3
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

9 Scalar Conversion Primitives

Scalar conversion primitives are bindable built-in pseudo-procedures that are used internally by the compiler to convert between different scalar types when no direct conversion path exists between the types. These primitives do not need to be called directly but they may nevertheless be imported from pseudo-module COMPILER.

9.1 Primitive SXF

The SXF primitive converts a value of a scalar type to scalar exchange format and passes the result back in a string variable. Its pseudo-definition is:

```
PROCEDURE SXF ( value : <ScalarType>; VAR sxfString : ARRAY OF CHAR );
```

9.2 Primitive VAL

The VAL primitive converts a string in scalar exchange format to an equivalent value of a given scalar type. Its pseudo-definition is:

```
PROCEDURE VAL ( CONST sxfString : ARRAY OF CHAR; VAR value : <ScalarType> );
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10 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 `COROUTINES`
- pseudo-module `ASSEMBLER` (optional)

10.1 Pseudo-Module SYSTEM

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

10.1.1 SYSTEM Constants

Module `SYSTEM` provides the following constants:

`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
`OctetsPerMachineWord`	target dependent number of octets per machine word
`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`	byte order of the target architecture
`TargetIsBiEndian`	target dependent boolean value, `TRUE` if target is bi-endian
`MaxWordsPerOperand`	implementation defined value ≥ 4 denoting the size of the largest supported operand in system level operations

10.1.2 SYSTEM Types

Module `SYSTEM` provides the following types:

`BYTE`	implementation defined byte
`WORD`	implementation defined word
`MACHINEBYTE`	target dependent machine byte
`MACHINEWORD`	target dependent machine word
`ADDRESS`	target dependent machine address

Although these types are provided by module `SYSTEM`, their respective IO operations are not. The IO operations corresponding to `READ`, `WRITE` and `WRITEF` for `SYSTEM` types are provided in the standard library and need to be imported to become available.

10.1.3 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.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.1.3.1 Intrinsic ADR

Intrinsic ADR returns the address of its operand which must be a variable. Its pseudo-definition is:

```
PROCEDURE ADR ( var : <AnyType> ) : ADDRESS;
```

10.1.3.2 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>;
```

10.1.3.3 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 );
```

10.1.3.4 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 );
```

10.1.3.5 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 );
```

10.1.3.6 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 ADDC ( VAR x : <AnyType>; y : <TypeOf(x)>; carry : BOOLEAN );
```

10.1.3.7 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)>;
```

10.1.3.8 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)>;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.1.3.9 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)>;
```

10.1.3.10 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)>;
```

10.1.3.11 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 ( VAR x : <AnyType>; n : OCTET ) : <TypeOf(x)>;
```

10.1.3.12 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)>;
```

10.1.3.13 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)>;
```

10.1.3.14 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)>;
```

10.1.3.15 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)>;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.1.3.16 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)>;
```

10.1.3.17 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)>;
```

10.1.3.18 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)>;
```

10.1.3.19 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)>;
```

10.1.3.20 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 );
```

10.1.3.21 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;
```

10.1.3.22 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;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.1.3.23 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;
```

10.1.3.24 Intrinsic CSBITS

Intrinsic `CSBITS` counts and returns the number of set bits of its operand. Its pseudo-definition is:

```
PROCEDURE CSBITS ( x : <AnyType> ) : CARDINAL;
```

10.1.3.25 Intrinsic BAIL

Intrinsic `BAIL` returns program control to the caller of the procedure where `BAIL` was invoked and passes its operand as the return value. The operand type must match the return type of the calling procedure. Its pseudo-definition is:

```
PROCEDURE BAIL ( x : <AnyType> );
```

10.1.3.26 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> );
```

10.2 Pseudo-Module ATOMIC

Pseudo-module `ATOMIC` provides intrinsics for atomic operations.

10.2.1 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;
```

10.2.2 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.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.2.2.1 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> );
```

10.2.2.2 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> );
```

10.2.2.3 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)> );
```

10.2.2.4 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)> );
```

10.2.2.5 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)> );
```

10.2.2.5 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)> );
```

10.2.2.6 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)> );
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

10.2.2.7 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)> );
```

10.3 Pseudo-Module COROUTINES

This section is not yet available.

10.4 Pseudo-Module ASSEMBLER

This section is not yet available.

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

11 Pragmas

Pragmas are directives used to control or influence the translation process but they do not change the meaning of the program text. There are two types of pragmas:

- language defined pragmas
- implementation defined pragmas

11.1 Language Defined Pragmas

Language defined pragmas are portable across implementations. They are easily distinguished because their pragma names are all-uppercase words. There are three groups of language defined pragmas:

- pragmas to control conditional compilation
- pragmas to emit console messages during compilation
- pragmas to control, influence or optimise code-generation

11.1.1 Conditional Compilation Pragmas

Conditional compilation pragmas are used to compile portions of the source text only if a certain condition is met. The condition must be a compile time expression.

EBNF:

```
conditionalCompilationPragma :
    "<*" ( IF | ELSIF ) constExpression | ELSE | ENDIF "*>" ;
```

Example:

```
TYPE Model = ( small, large, custom );
<* IF TSIZE(INTEGER) = 2 *>
CONST model = Model.small;
<* ELSIF TSIZE(INTEGER) = 4 *>
CONST model = Model.large;
<* ELSIF TSIZE(INTEGER) MOD 2 = 0 *>
CONST model = Model.custom;
<* ELSE *>
<* FATAL "unsupported type model." *>
<* ENDIF *>
```

11.1.2 Compile Time Console Message Pragmas

Compile time console message pragmas are used to emit console messages during compilation. There are four types of message pragmas:

- pragmas emitting informational messages
- pragmas emitting compilation warnings
- pragmas emitting compilation errors
- pragmas emitting fatal compilation errors

Informational messages and warnings do not cause the compilation to fail. Error messages cause the compilation to fail but continue. Fatal messages cause the compilation to fail and abort immediately.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

EBNF:

```
compileTimeMessagePragma :
    "<*" ( INFO | WARN | ERROR | FATAL ) quotedStringLiteral "*>" ;
```

Example:

```
<* FATAL "unsupported target architecture." *>
```

11.1.3 Code Generation Pragmas

Code generation pragmas are used to control or influence code generation and optimisation. There are six code generation pragmas:

- pragma to force specified memory alignment
- pragma to force specified calling convention
- pragma to cause the build system to invoke an external utility
- pragma to suggest inlining a procedure
- pragma to suggest not inlining a procedure
- pragma to mark a variable as volatile

EBNF:

```
codeGenerationPragma :
    "<*" ( ALIGN "=" constExpression |
           FOREIGN ( "=" quotedStringLiteral )? |
           MAKE "=" quotedStringLiteral |
           INLINE | NOINLINE | VOLATILE ) "*>" ;
```

Examples:

```
TYPE Point = RECORD <* ALIGN = 8*TSIZE(CARDINAL) *> x, y : OCTET END;
<* FOREIGN = "C" *> DEFINITION MODULE stdio;
<* MAKE = "genhashes foo bar baz > Hashes.def" *>
<* INLINE *> PROCEDURE P;
<* NOINLINE *> PROCEDURE Q;
VAR <* VOLATILE *> signal : Signal;
```

NB: At present, the only parameter defined for pragma `FOREIGN` is "C".

11.2 Implementation Defined Pragmas

Implementation defined pragmas are compiler specific and non-portable. They are easily distinguished because their pragma names are never all-uppercase words.

EBNF:

```
implementationDefinedPragma :
    "<*" pragmaName ( "+" | "-" | "=" constExpression )? "*>" ;

pragmaName : Ident ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

12 Generics

Syntax for generic programming is not provided in Modula-2 R10 (on purpose). Instead, generic programming is supported via the Modula-2 Template Engine, or M2TE, a text templating utility that is external to the compiler.

The M2TE utility is invoked by passing the name of a template file and one or more translations for placeholders in the template file as parameters. 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 `MAKE` 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 `MAKE` pragma may be used to generate a module on demand from within the source text of a client module or program.

Example:

```
<* MAKE = "m2te Stack module:IntegerStack 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.

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

14 Standard Library

The public repository with the complete definition parts of the Modula-2 R10 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 five mandatory pseudo modules, one optional pseudo module and two documentation modules:

`ATOMIC.def`	provides atomic intrinsics
`SYSTEM.def`	access to system dependent resources
`COROUTINES.def`	access to built-in coroutines (TO DO)
`RUNTIME.def`	interface to the Modula-2 runtime system
`COMPILER.def`	interface to the Modula-2 compile-time system
`ASSEMBLER.def`	access to target dependent inline assembler (optional)
`PERVASIVES.def`	documents pervasive constants, types and macros
`BINDINGS.def`	documents procedure headers required for bindings

14.2 Prototype Library

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

`ZTYPE.def`	defines semantic properties for whole number ADTs
`RTYPE.def`	defines semantic properties for real number ADTs
`CTYPE.def`	defines semantic properties for complex number ADTs
`VTYPE.def`	defines semantic properties for numeric vector ADTs
`ArrayType.def`	defines semantic properties for static array ADTs
`SetType.def`	defines semantic properties for static set ADTs
`DateType.def`	defines semantic properties for date-time ADTs
`IntervalType.def`	defines semantic properties for interval ADTs
`StringType.def`	defines semantic properties for dynamic string ADTs
`CollectionType.def`	defines semantic properties for dynamic collection 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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

`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

`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 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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

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

Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Appendix A: Grammar in EBNF

A.1 Non-Terminal Symbols

Compilation Units

#1 Compilation Unit

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

#2 Prototype

```
prototype :
    PROTOTYPE prototypeId ";"
    TYPE '='
    ( RECORD | OPAQUE RECORD? ( ':=' ( literalType | '{' '..' '}' ) )? ) ';'
    ( ASSOCIATIVE ';' )?
    requiredBinding*
    END prototypeId "." ;
prototypeId : Ident ;
literalType : Ident ;
```

#3 Program Module

```
programModule :
    MODULE moduleId ( "[" priority "]" )? ";"
    importList* block moduleId "." ;
moduleId : Ident ;
priority : constExpression ;
```

#4 Definition Of Module

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

#5 Implementation Of Module

```
implementationOfModule :
    IMPLEMENTATION programModule ;
```

Bindings, Import Lists, Blocks, Declarations and Definitions

#6 Required Binding

```
requiredBinding :
    ( CONST "[" bindableIdent "]" |
      PROCEDURE "[" ( bindableOperator | bindableIdent ) "]" ) ";" ;
```

#7 Bindable Operator

```
bindableOperator :
    DIV | MOD | IN | FOR |
    ":=" | "?" | "!" | "~" | "+" | "-" | "*" | "/" | "=" | "<" | ">" ;
bindableIdent : Ident ;
// NEW, DISPOSE, ABS, NEG, ODD, COUNT, LENGTH, TMIN, TMAX, SXF, VAL
```

#8 Import List

```
importList :
    ( FROM moduleId IMPORT ( identList | "*" ) |
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

```
    IMPORT Ident “+”? ( “,” Ident “+”? )* ) “;” ;
```

#9 Block

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

#10 Declaration

```
declaration :
    CONST ( constantDeclaration “;” )* |
    TYPE ( Ident “=” type “;” )* |
    VAR ( variableDeclaration “;” )* |
    procedureDeclaration “;” ;
```

#11 Definition

```
definition :
    CONST ( ( “[” Ident “]” )? constantDeclaration “;” )* |
    TYPE ( Ident “=” ( type | OPAQUE recordType? ) “;” )* |
    VAR ( variableDeclaration “;” )* |
    procedureHeader “;” ;
```

Constant Declarations

#12 Constant Declaration

```
constantDeclaration :
    Ident “=” constExpression ; // No type identifiers
```

Type Declarations

#13 Type

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

#14 Range

```
range :
    “[” constExpression “..” constExpression “]” ;
```

#15 Enumeration Type

```
enumerationType :
    “(” ( ( “+” namedType ) | Ident )
          ( “,” ( ( “+” namedType ) | Ident ) )* “)” ;
```

#16 Array Type

```
arrayType :
    ( ARRAY constComponentCount ( “,” constComponentCount )* |
      ASSOCIATIVE ARRAY ) OF namedType ;
constComponentCount : cardinalConstExpression ;
cardinalConstExpression : constExpression ;
```

#17 Record Type

```
recordType :
    RECORD ( “(”  baseType  “)” )? fieldListSequence? END ;
baseType : Ident ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

#18 Field List Sequence

```
fieldListSequence :
    fieldList ( “;” fieldList )* ;
```

#19 Field List

```
fieldList :
    Ident
    ( ( “,” Ident )+ “:” namedType |
      “:” ( ARRAY determinantField OF )? namedType ) ;
determinantField : Ident ;
```

#20 Set Type

```
setType :
    SET OF ( namedEnumType | “(” identList “)” ) ;
namedEnumType : namedType ;
```

#21 Pointer Type

```
pointerType :
    POINTER TO CONST? namedType ;
```

#22 Procedure Type

```
procedureType :
    PROCEDURE
    ( “(” formalTypeList “)” )?
    ( “:” returnedType )? ;
returnedType : namedType ;
```

#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 :
    Ident ( “[” machineAddress “]” | “,” identList )?
    “:” ( ARRAY constComponentCount OF )? namedType ;
machineAddress : constExpression ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

Procedure Declarations

#29 Procedure Declaration

```
procedureDeclaration :
    procedureHeader ";" block Ident ;
```

#30 Procedure Header

```
procedureHeader :
    PROCEDURE
    ( "[" ( "::" | bindableOperator | bindableIdent ) "]" )?
    Ident ( "(" formalParamList ")" )? ( ":" returnedType )? ;
```

#31 Formal Parameter List

```
formalParamList :
    formalParams ( ";" formalParams )* ;
```

#32 Formal Parameters

```
formalParams :
    simpleFormalParams | variadicFormalParams ;
```

#33 Simple Formal Parameters

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

#34 Variadic Formal Parameters

```
variadicFormalParams :
    VARIADIC ( variadicCounter | "[" variadicTerminator "]" )? OF
    ( simpleFormalType |
      "(" simpleFormalParams ( ";" simpleFormalParams )* ")" ) ;
variadicCounter : Ident ;
variadicTerminator : constExpression ;
```

Statements

#35 Statement

```
statement :
    ( assignmentOrProcedureCall | ifStatement | caseStatement |
      whileStatement | repeatStatement | loopStatement |
      forStatement | RETURN expression? | EXIT )? ;
```

#36 Statement Sequence

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

#37 Assignment Or Procedure Call

```
assignmentOrProcedureCall :
    designator ( ":=" expression | "++" | "--" | actualParameters )? ;
```

#38 IF Statement

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

#39 CASE Statement

```
caseStatement :
    CASE expression OF case ( "|" case )* ( ELSE statementSequence )? END ;
```

#40 Case

```
case :
    caseLabels ( "," caseLabels )* ":" statementSequence ;
```

#41 Case Labels

```
caseLabels :
    constExpression ( ".." constExpression )? ;
```

#42 WHILE Statement

```
whileStatement :
    WHILE expression DO statementSequence END ;
```

#43 REPEAT Statement

```
repeatStatement :
    REPEAT statementSequence UNTIL expression ;
```

#44 LOOP Statement

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

#45 FOR Statement

```
forStatement :
    FOR DESCENDING? controlVariable IN ( expression | range OF namedType )
    DO statementSequence END ;
controlVariable : Ident ;
```

Expressions

#46 Constant Expression

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

#47 Relation

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

#48 Simple Constant Expression

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

#49 Add Operator

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

#50 Constant Term

```
constTerm :
    constFactor ( mulOperator constFactor )* ;
```

#51 Multiply Operator

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

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

#52 Constant Factor

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

#53 Designator

```
designator :
    qualident designatorTail? ;
```

#54 Designator Tail

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

#55 Expression List

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

#56 Expression

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

#57 Simple Expression

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

#58 Term

```
term :
    factor ( mulOperator factor )* ;
```

#59 Factor

```
factor :
    ( Number | String | structuredValue |
      designatorOrProcedureCall | "(" expression ")" |
      CAST "(" namedType "," expression ")" )
    ( "::" namedType )? | NOT factor ;
```

#60 Designator Or Procedure Call

```
designatorOrProcedureCall :
    qualident designatorTail? actualParameters? ;
```

#61 Actual Parameters

```
actualParameters :
    "(" expressionList? ")" ;
```

Value Constructors

#62 Constant Structured Value

```
constStructuredValue :
    "{" ( constValueComponent ( "," constValueComponent )* )? "}" ;
```

#63 Constant Value Component

```
constValueComponent :
    constExpression ( ( BY | ".." ) constExpression  )? ;
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

#64 Structured Value

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

#65 Value Component

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

Identifiers

#66 Qualified Identifier

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

#67 Identifier List

```
identList :
    Ident ( "," Ident )* ;
constQualident : qualident ; // No type and no variable identifiers
```


Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

A.2 Pragmas

#1 Pragma

```
pragma :
    "<*"
    ( conditionalPragma | compileTimeMessagePragma | codeGenerationPragma |
      implementationDefinedPragma )
    "*>" ;
```

#2 Conditional Pragma Body

```
conditionalPragma :
    ( IF | ELSIF ) constExpression | ELSE | ENDIF ;
```

#3 Compile Time Message Pragma Body

```
compileTimeMessagePragma :
    ( INFO | WARN | ERROR | FATAL ) compileTimeMessage ;
compileTimeMessage : String ;
```

#4 Code Generation Pragma Body

```
codeGenerationPragma :
    ALIGN "=" constExpression | FOREIGN ( "=" String )? | MAKE "=" String |
    INLINE | NOINLINE | VOLATILE ;
```

#5 Implementation Defined Pragma Body

```
implementationDefinedPragma :
    pragmaName ( "+" | "-" | "=" constExpression )? ;
pragmaName : Ident ; // lowercase or camelcase only
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

A.3 Terminal Symbols

#1 Identifier

```
Ident :
    ( "_" | "$" | LETTER ) ( "_" | "$" | LETTER | DIGIT )* ;
```

#2 Number Literal

```
Number :
    DIGIT+ ( DIGIT+ "." DIGIT+ ( "E" ( "+" | "-" )? DIGIT+ )? )? |
    ( "0" | "1" )+ "B" | "0b" ( "0" | "1" )+ |
    DIGIT BASE16_DIGIT* ( "H" | "U" ) |
    ( "0x" | "0u" ) BASE16_DIGIT_LOWERCASE+ ;
```

#3 String Literal

```
String :
    "'" ( CHARACTER | EscapeSequence | '"' )* "'" |
    '"' ( CHARACTER | EscapeSequence | "'" )* '"' ;
```

#4 Letter

```
LETTER :
    "A" .. "Z" | "a" .. "z" ;
```

#5 Digit

```
DIGIT :
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

#6 Base-16 Digit

```
BASE16_DIGIT :
    DIGIT | "A" | "B" | "C" | "D" | "E" | "F" ;
```

#6 Lowercase Base-16 Digit

```
BASE16_DIGIT_LOWERCASE :
    DIGIT | "a" | "b" | "c" | "d" | "e" | "f" ;
```

#7 Character

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

#8 Escape Sequence

```
ESCAPE_SEQUENCE :
    "\" ( "0" | "n" | "r" | "t" | "\" | "'" | '"' ) ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

A.4 Ignore Symbols

#1 Whitespace

```
WHITESPACE :
    " " | ASCII_TAB ;
```

#2 Comment

```
COMMENT :
    MULTI_LINE_COMMENT | SINGLE_LINE_COMMENT ;
```

#3 Multi-line Comment

```
MULTI_LINE_COMMENT :
    "(*" ( CHARACTER | ASCII_TAB )* MULTI_LINE_COMMENT* "*)" ;
```

#4 Single-line Comment

```
SINGLE_LINE_COMMENT :
    "//" ( CHARACTER | ASCII_TAB )* END_OF_LINE ;
```

#5 End Of Line Marker

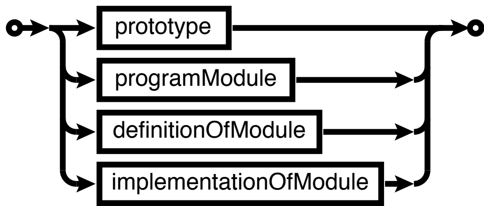
```
END_OF_LINE :
    ASCII_LF ASCII_CR? | ASCII_CR ASCII_LF? ;
```

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

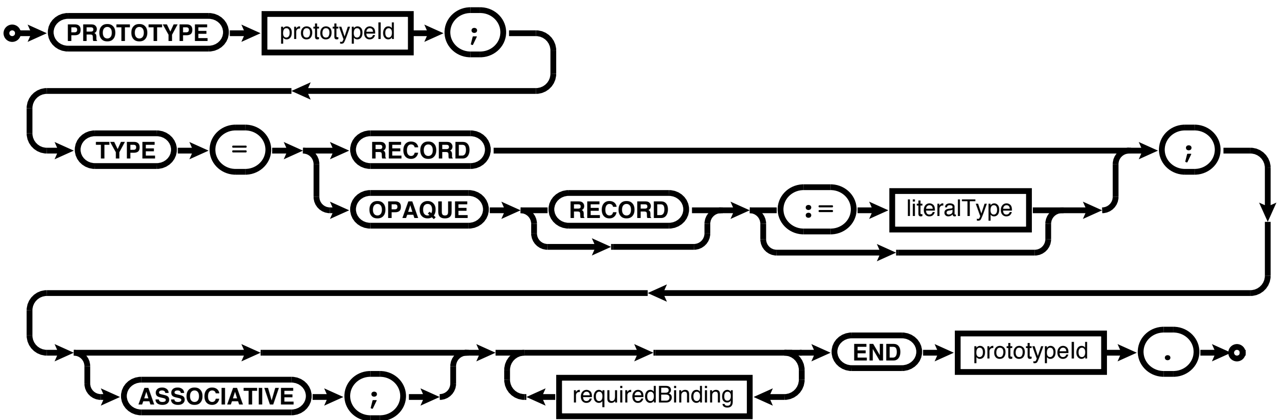
Appendix B: Syntax Diagrams

B.1 Non-Terminal Symbols

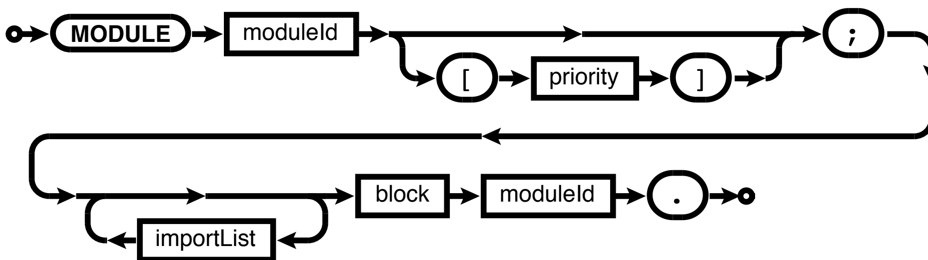
#1 Compilation Unit



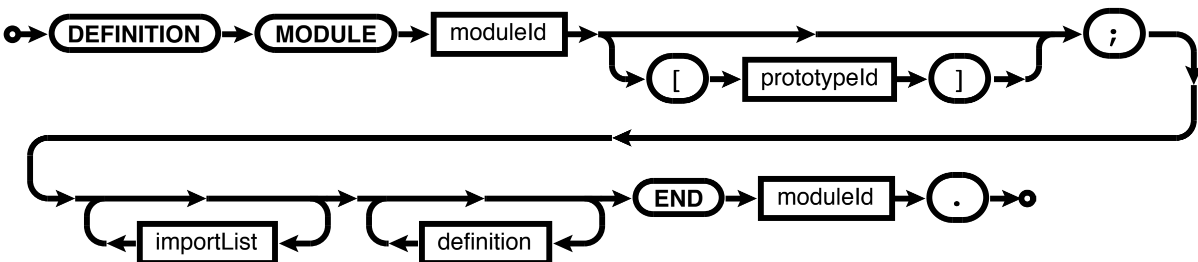
#2 Prototype



#3 Program Module



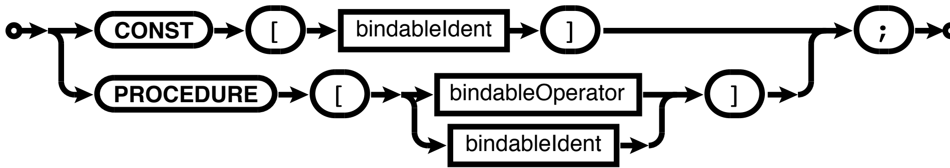
#4 Definition Of Module



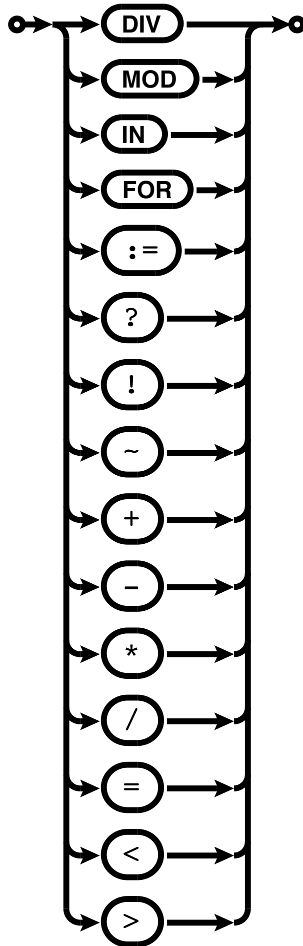
#5 Implementation Of Module

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

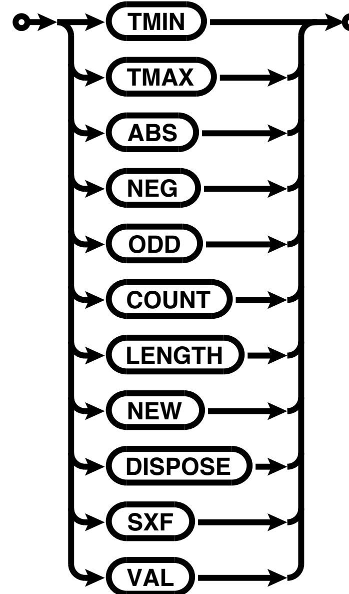
#6 Required Binding



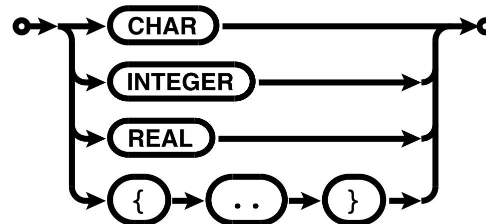
#7 Bindable Operator



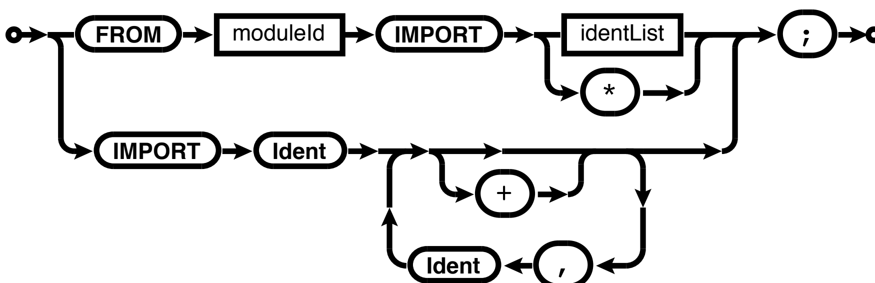
Bindable Identifier



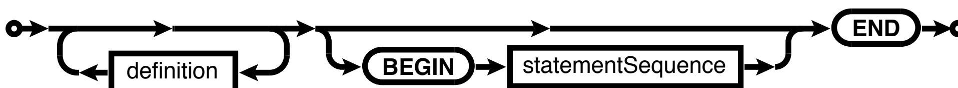
Literal Type



#8 Import List

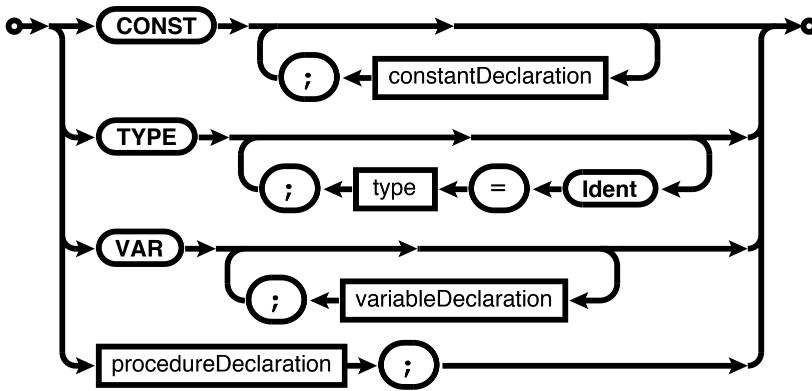


#9 Block

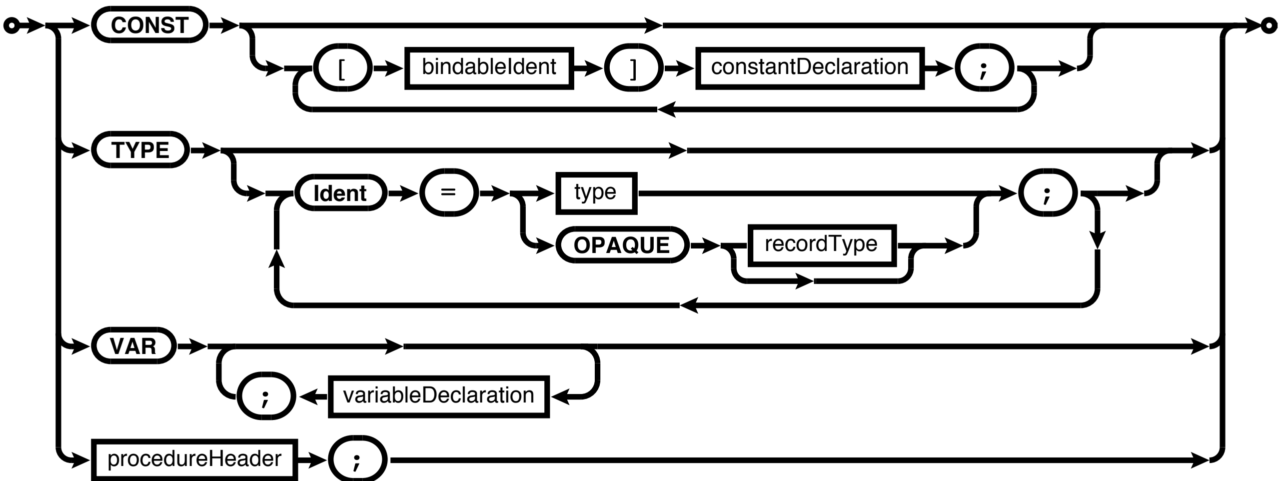


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

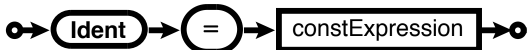
#10 Declaration



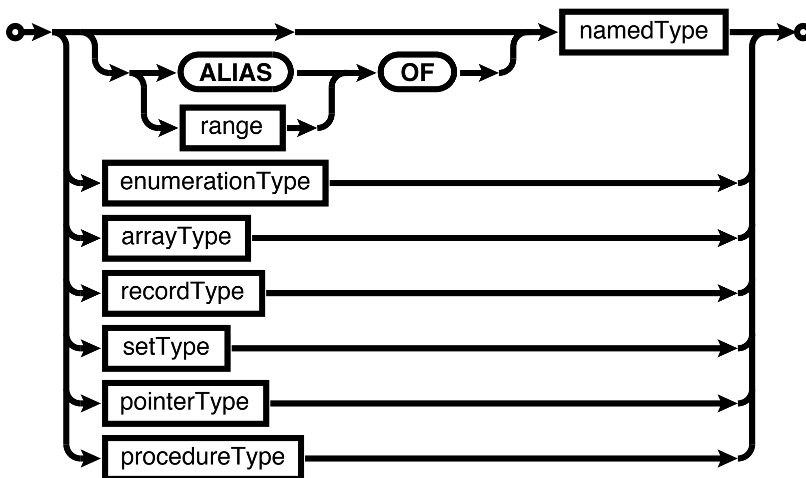
#11 Definition



#12 Constant Declaration



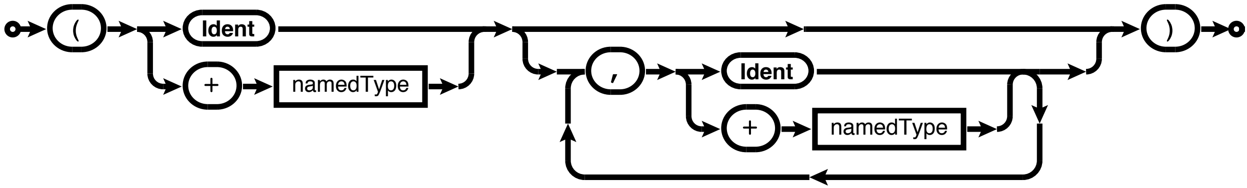
#13 Type



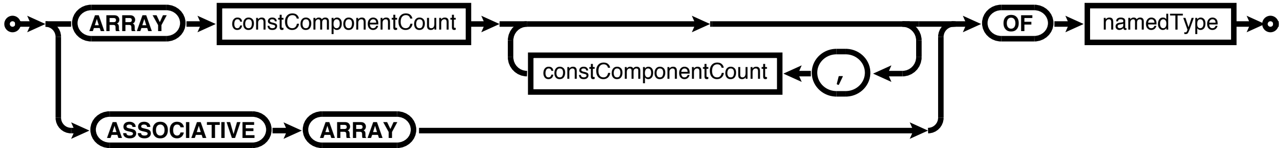
#14 Range

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

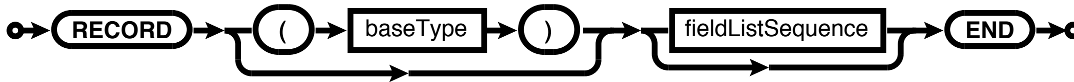
#15 Enumeration Type



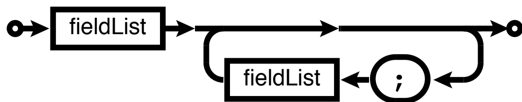
#16 Array Type



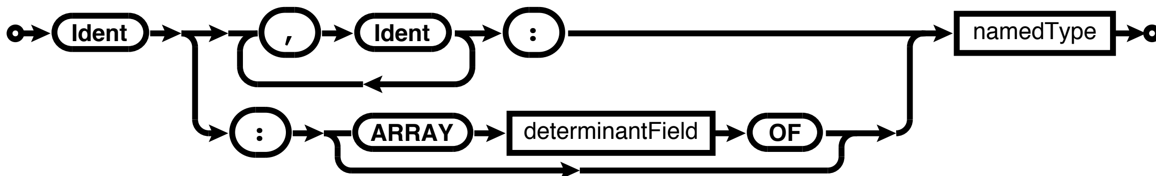
#17 Record Type



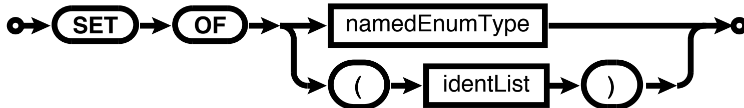
#18 Field List Sequence



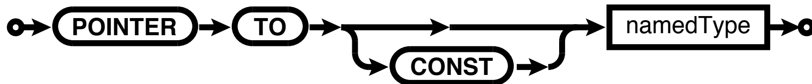
#19 Field List



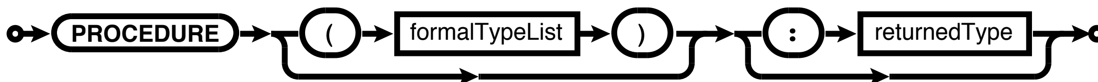
#20 Set Type



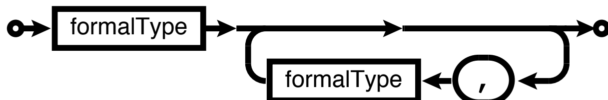
#21 Pointer Type



#22 Procedure Type

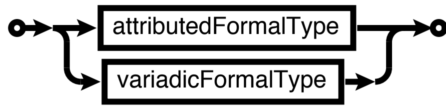


#23 Formal Type List

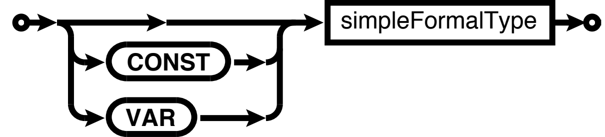


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

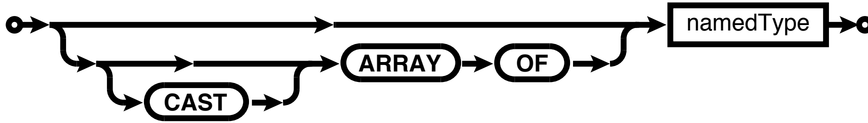
#24 Formal Type



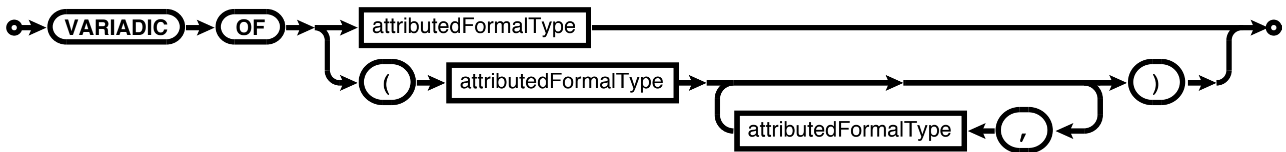
#25 Attributed Formal Type



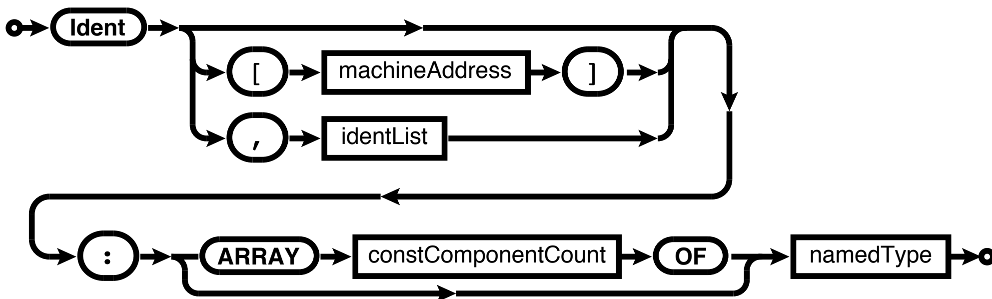
#26 Simple Formal Type



#27 Variadic Formal Type

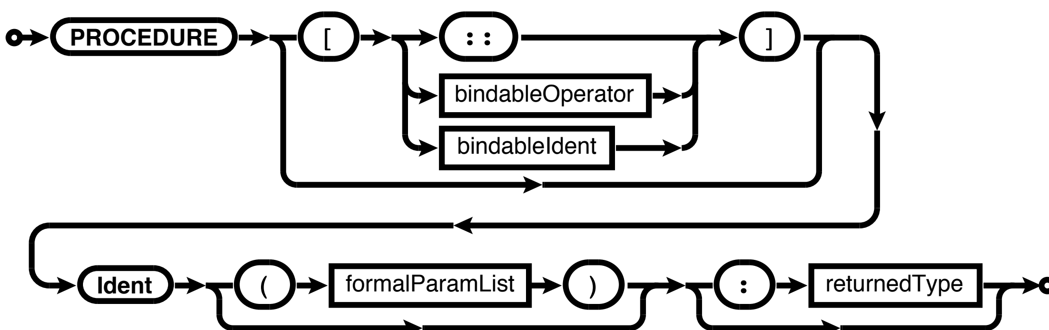


#28 Variable Declaration

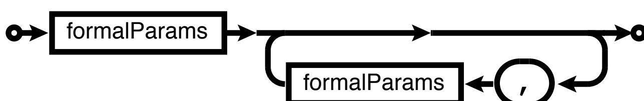


#29 Procedure Declaration

#30 Procedure Header

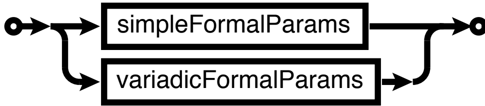


#31 Formal Parameter List

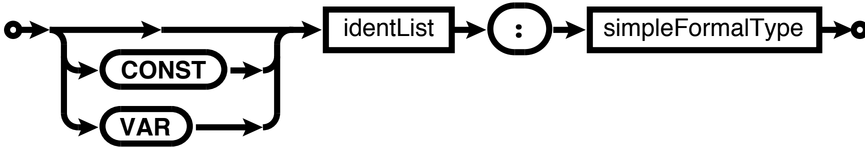


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

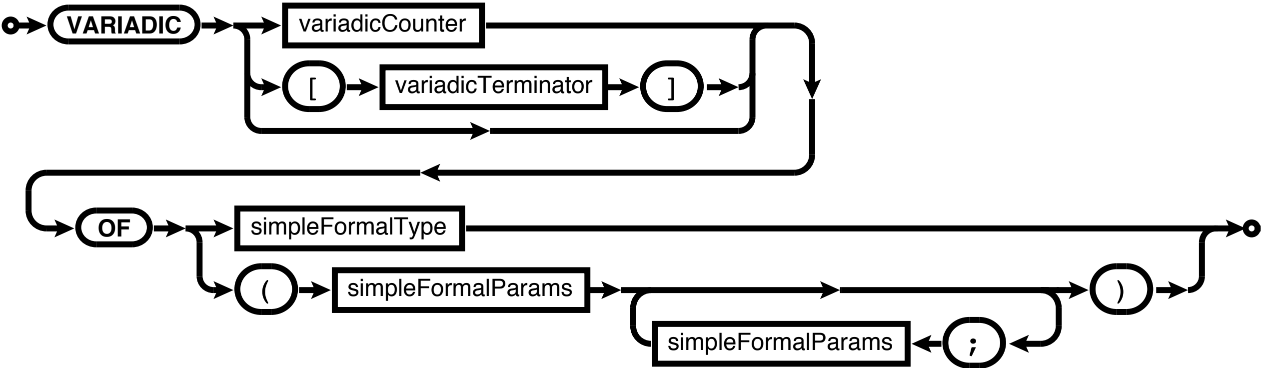
#32 Formal Parameters



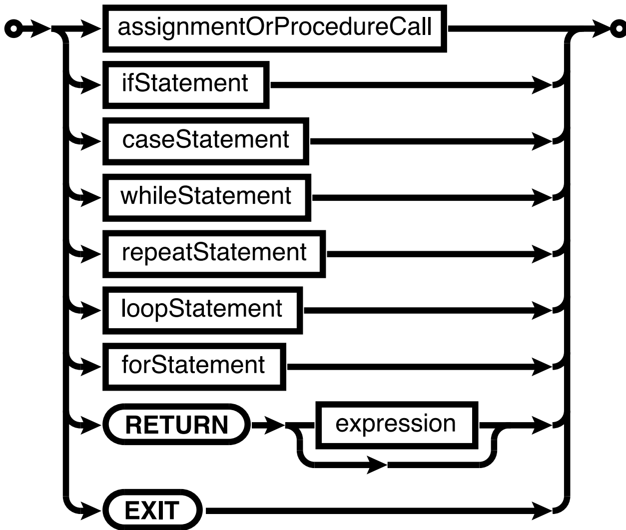
#33 Simple Formal Parameters



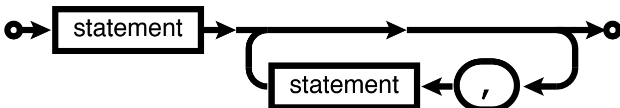
#34 Variadic Formal Parameters



#35 Statement

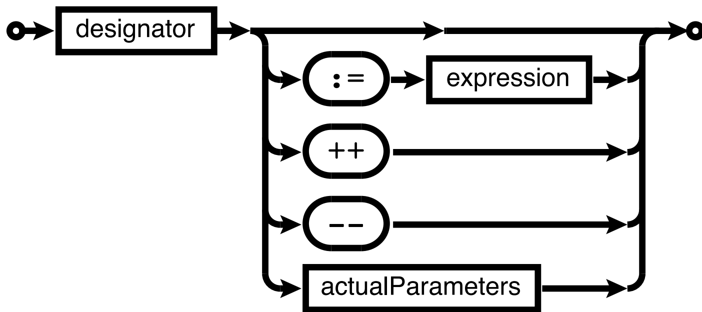


#36 StatementSequence

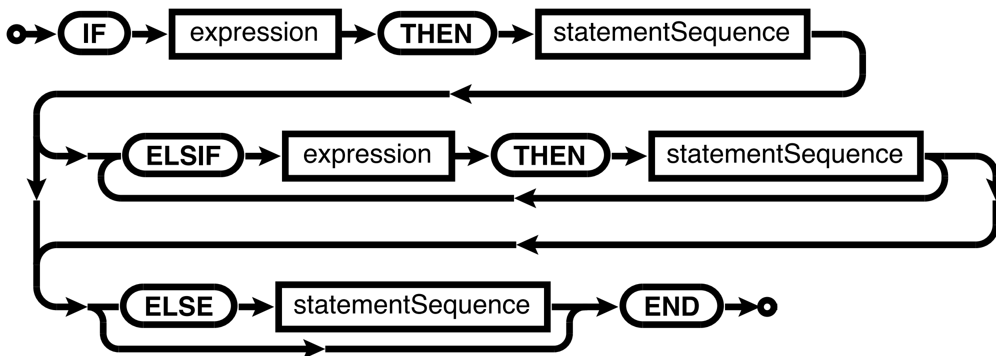


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

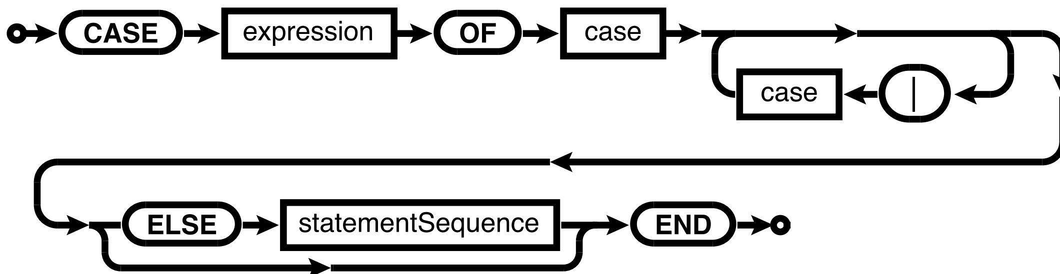
#37 Assignment Or Procedure Call



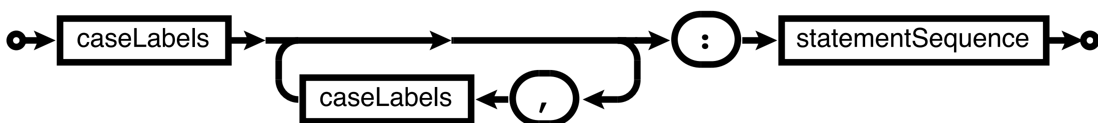
#38 IF Statement



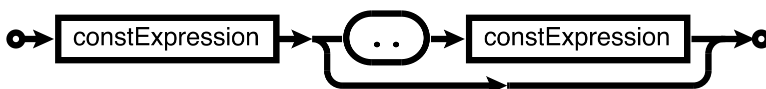
#39 CASE Statement



#40 Case



#41 Case Labels



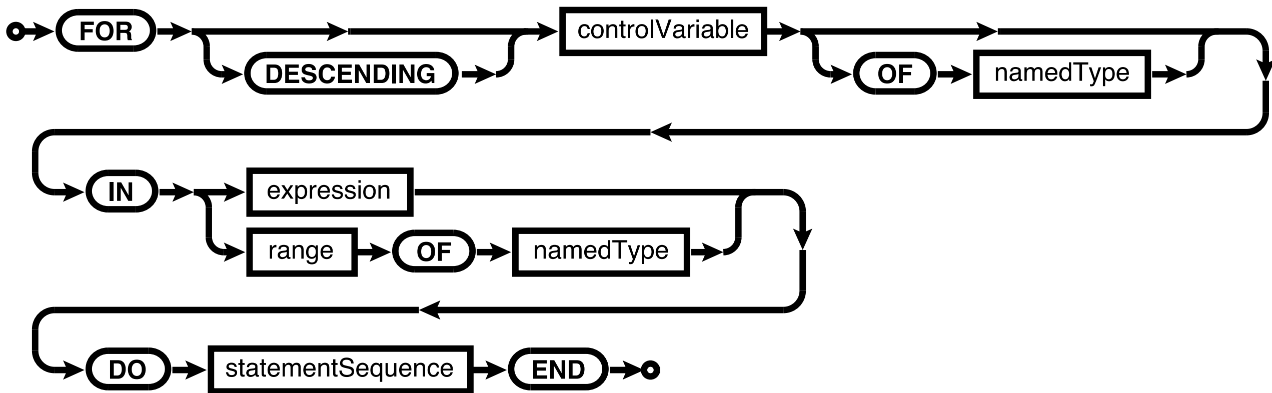
#42 WHILE Statement

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

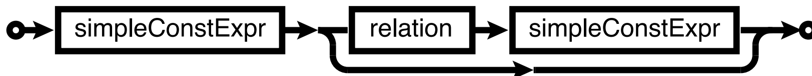
#43 REPEAT Statement

#44 LOOP Statement

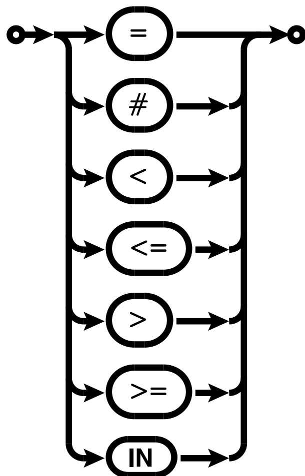
#45 FOR Statement



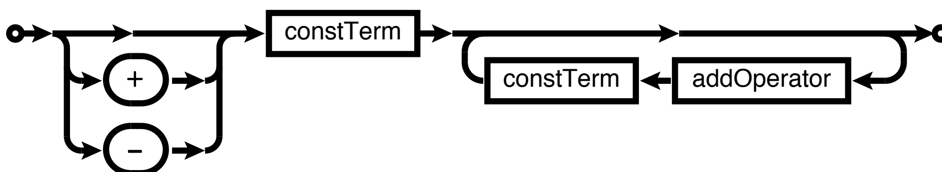
#46 Constant Expression



#47 Relation



#48 Simple Constant Expression



 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

#49 Add Operator

+
−
OR

#50 Constant Term

constFactor
constFactor
mulOperator

#51 Multiply Operator

*
/
DIV
MOD
AND

#52 Constant Factor

Number
String
constQualident
constStructuredValue
(constExpression)
CAST (namedType , constExpression)
NOT constFactor
::
namedType

#53 Designator

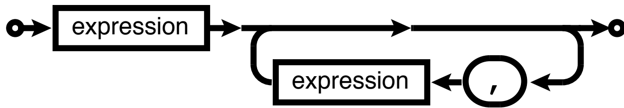
qualident
designatorTail

#54 Designator Tail

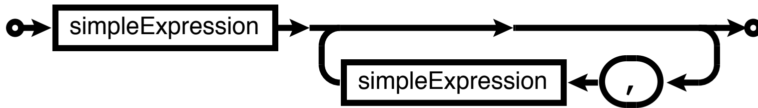
[expressionList]
^
. Ident

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

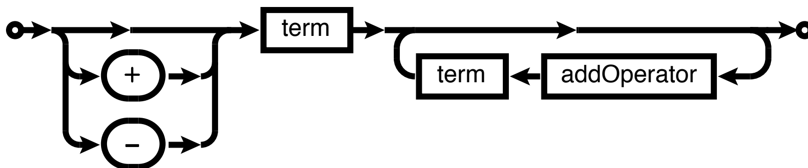
#55 Expression List



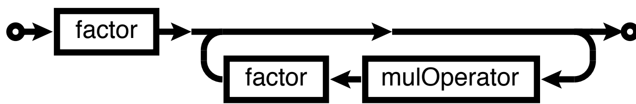
#56 Expression



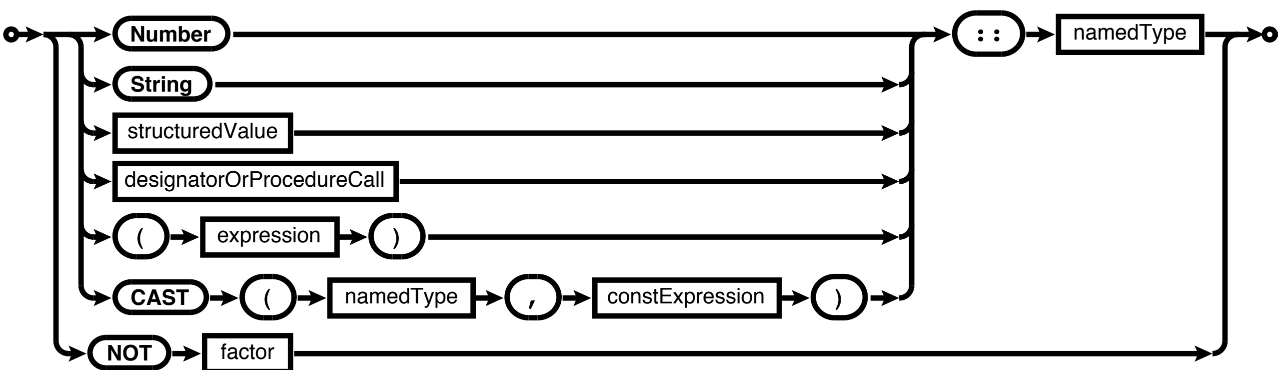
#57 Simple Expression



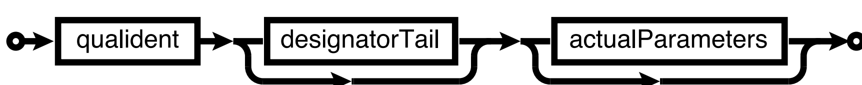
#58 Term



#59 Factor

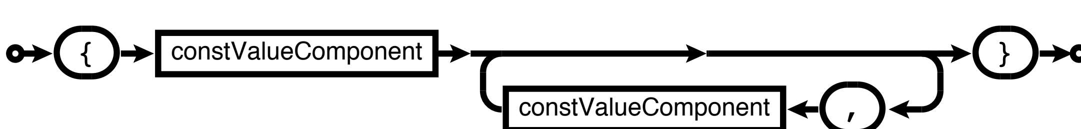


#60 Designator Or Procedure Call



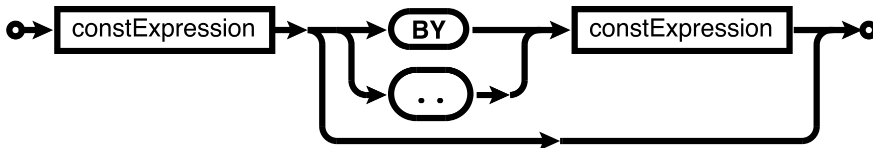
#61 Actual Parameters

#62 Constant Structured Value

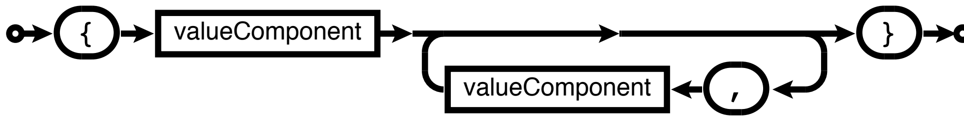


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

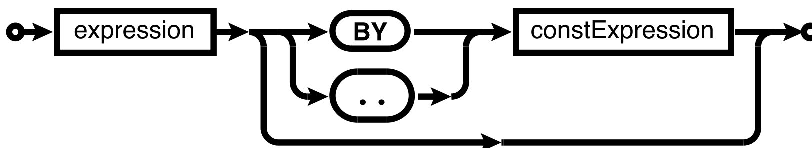
#63 Constant Value Component



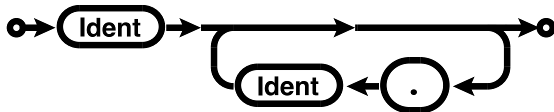
#64 Structured Value



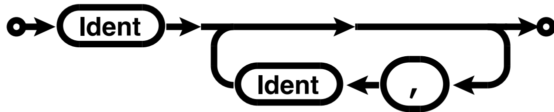
#65 Value Component



#66 Qualified Identifier



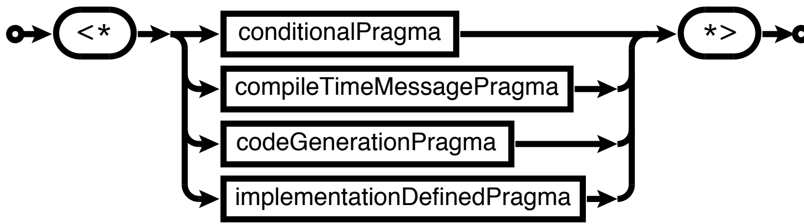
#67 Identifier List



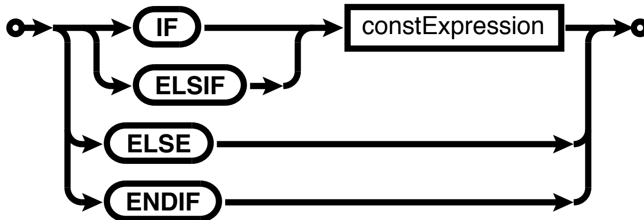
 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

B.2 Pragmas

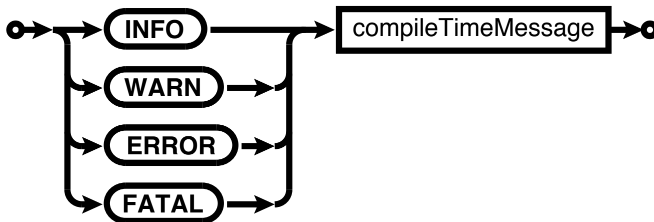
Pragma



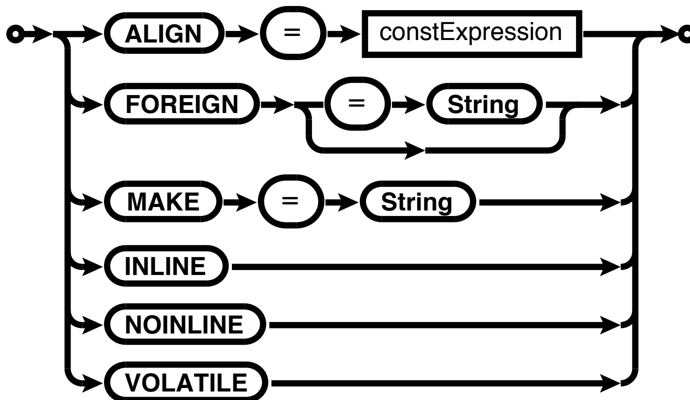
Conditional Pragma Body



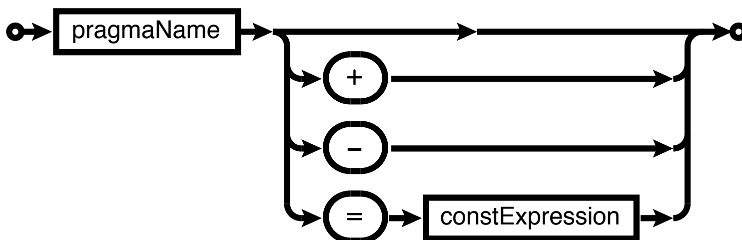
Compile Time Message Pragma Body



Code Generation Pragma Body



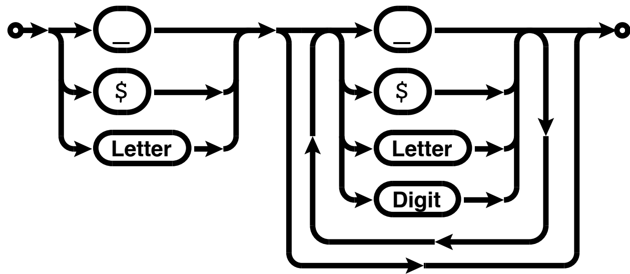
Implementation Defined Pragma Body



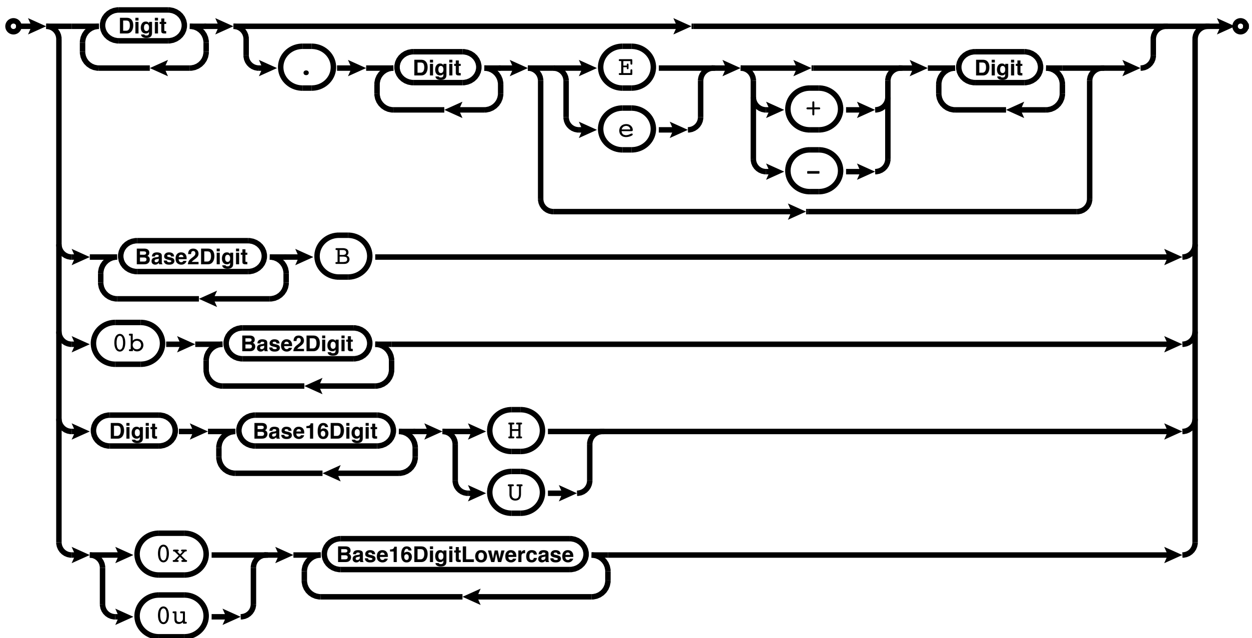
 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

B.3 Terminal Symbols

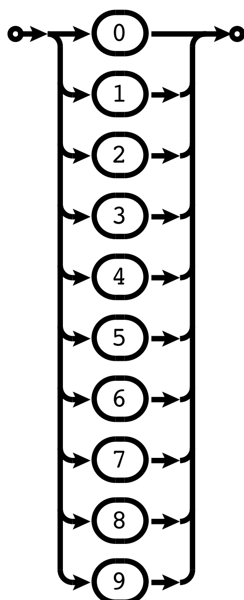
Identifier



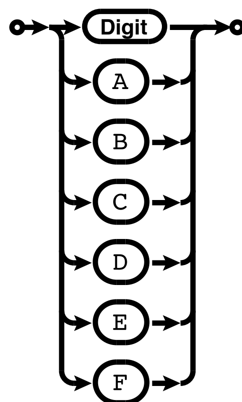
Number Literal



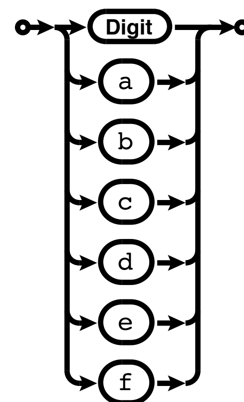
Digit



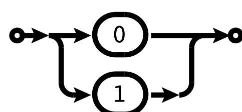
Base 16 Digit



Lowercase Base 16 Digit

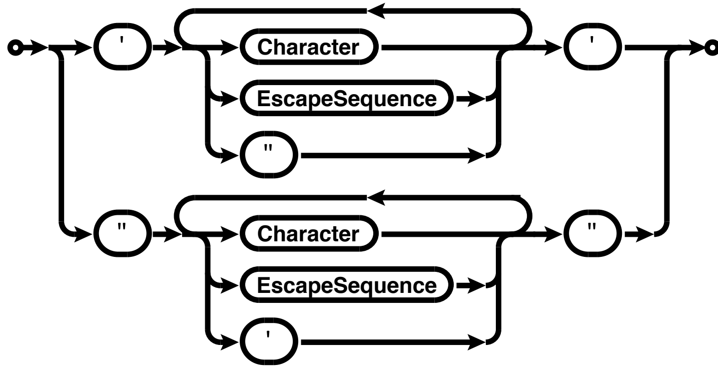


Base 2 Digit

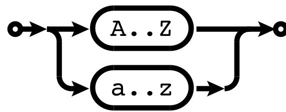


 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

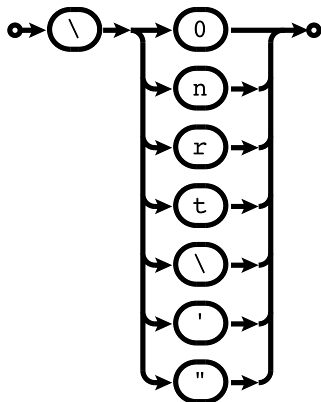
String Literal



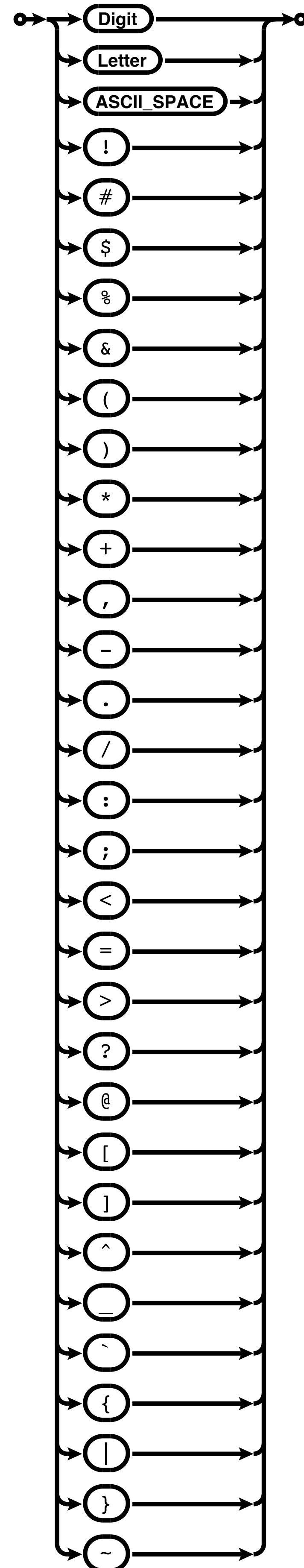
Letter



Escape Sequence



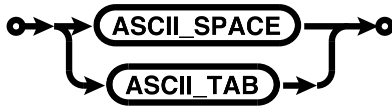
Character



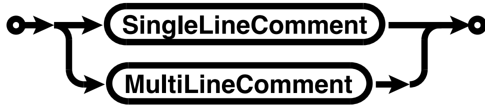
 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

B.4 Ignore Symbols

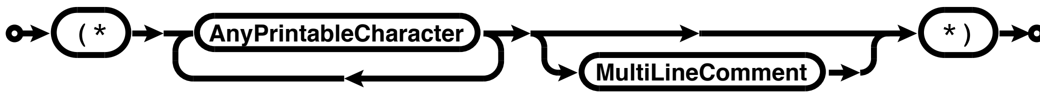
Whitespace



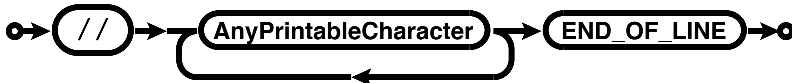
Comment



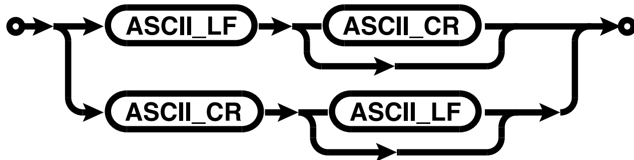
Multi-line Comment



Single-line Comment



End Of Line Marker



 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.

 Copyright © 2010 B.Kowarsch & R.Sutcliffe. All rights reserved.