Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Zonnon Language Report

Jürg Gutknecht and Eugene Zueff

Editors: Brian Kirk and David Lightfoot

December 2005

Abstract

Zonnon is a general-purpose programming language in the Pascal, Modula-2 and Oberon family. It retains an emphasis on simplicity, clear syntax and separation of concerns whilst focusing on concurrency and ease of composition and expression. Unification of abstractions is at the heart of its design and this is reflected in its conceptual model based on modules, objects, definitions and implementations. Zonnon offers a new computing model based on active objects with their interaction defined by syntax controlled dialogs. It also introduces new features including operator overloading, indexers and exception handling, and is specifically designed to be platform independent.

Document Details

Title: Zonnon Language Report
Version: 03
Revision: 01
Issued: 14 December 2005

Language Designer: Prof. Jürg Gutknecht
Language Implementers: Eugene Zueff, Roman Mitin
Test Suite Implementer: Vladimir Romanov
Report Editors: Brian Kirk and David Lightfoot

Published by:

Institute of Computer Systems
ETH Zentrum, RZ H 24
CH-8092 Zürich
Switzerland

The latest version of this report is available on-line at www.zonnon.ethz.ch
Please send details of any errors and omissions in this document to zonnon@inf.ethz.ch

The typographic conventions used in the report are:

New concepts are indicated in italics
Programming language keywords in the text are in italics.
Main headings are in 12-point Arial
Subheadings are in 11-point Arial
Sub-subheadings are in 10-point Arial
Sub-sub-subheadings are in 9-point Arial
Main text is in 10-point Times New Roman
Syntax is in 8-point Verdana
References appear in square brackets e.g. [Compiler]
In general spelling is in ‘US English’

Contents

Zonnon Language Report

With a new computer language one not only learns a new vocabulary and grammar but opens oneself to a new world of thought …

– Niklaus Wirth

1 Introduction

Zonnon is a new programming language in the Pascal, Modula-2 and Oberon family. It retains an emphasis on simplicity, clear syntax and separation of concerns. Although more compact than languages such as C#, Java and Ada, it is a general-purpose language suited to a wide range of applications. Typically this includes component-oriented composition, concurrent systems, algorithms and data structures, object-oriented and structured programming, graphics, mathematical programming and low-level systems programming. Zonnon provides a rich object model with encapsulated behavior and syntax controlled dialogs which encapsulate state. It may be used to write programs in procedural, object-oriented and concurrent styles [see Zonnon]. Zonnon is also well suited for teaching purposes, from basic principles right through to advanced concepts.

Unification of abstractions is at the heart of Zonnon's design. This is reflected in its four pillars

- the *module*—both a textual container and program composition object
- the *object*—a type template for defining objects
- the *definition*—a concept of abstraction and composition for defining interfaces
- the *implementation*—a container for reusable fragments of object implementations

These entities provide the basis for program composition in the large and also for textual partitioning and separate compilation during program development—they are 'first-class citizens' in the language.

The object model in Zonnon is based on the notion that 'everything is an object'. It supports three views of them, firstly as entities with an intrinsic type, used by abstract operators in a type-safe way, secondly as providers of services accessed via defined interfaces and thirdly as autonomous agents interoperating via formal dialogs. *Activities* are used both for adding behavior to objects and for implementing dialogs. They integrate concurrency seamlessly into the language.

Many of the concepts in Zonnon have been drawn from its heritage. The intention has been to offer expressive and cohesive features which have proved their worth. Zonnon also introduces some new features such as operator overloading for representing mathematical and other expressions in a natural way and exception handling for improving reliability. Some features have been reintroduced from earlier members of the Pascal language family, for example the *definition*, *implementation* pairs and enumeration types from Modula-2 and, for pragmatic reasons, the *read* and *write* statements from Pascal.

When choosing a language for building modern systems achieving interoperability between programs written in different languages within the same system is an important consideration. The Zonnon language is specifically designed to be platform-independent whilst supporting interoperability.

A companion document, Zonnon Programmers' Manual, contains implementation specific details for a particular compiler and runtime support package for a computing platform, see [Compiler].

2 Program Composition

Zonnon programs are composed from four constructs: module, object, definition and implementation. More precisely:

A *module* has a dual nature: it declares a syntactic container for logically cohesive program declarations and it simultaneously declares an object whose lifecycle is controlled by the system. So the module provides the mechanism for the textual partitioning of a source program and also the dynamic loading at execution time of a part of a program, in the form of an instantiated object.

Any number of dynamically created objects may have their lifecycles managed by a program, however only a single instance of each module's object may be instantiated by the system at any given time.

Because the module forms a unit of encapsulation and data hiding, it is also ideal as a container for implementing abstract data types.

An *object* is a type template comprising fields, methods and activities. The fields represent the object's state, the methods its functionality and the activities its concurrent activities. It can expose its interface to its system environment in two ways. Firstly by its *intrinsic interface*, that is, the set of all the elements which the programmer chooses to make public rather than keep private, and secondly by a number of *definitions*, each of which exposes a distinct *facet* representing an aspect of the object's services to its clients.

A *definition* defines a distinct *facet* of an object in terms of an abstract interface comprising field declarations and method signatures. Definitions can form *a network of related types*, not just a hierarchy.

An *implementation* defines an aggregate of field and method implementations intended for re-use when incorporated into a program via one or more object templates. An object implementing a definition is required to implement all of its fields and methods. However, if an object imports an implementation of a definition with the same name as the definition then this is implicitly presumed to be its (possibly partial) implementation.

A *program text* comprises modules, objects, definitions and implementations. The program's *intrinsic interface* is the set of declarations made public by all of its parts. A *run-time program* comprises one or more modules and any objects that are created dynamically. The *system* provides mechanisms for dynamic program loading and unloading of modules and dynamic management of object resources at execution time, when a program runs.

These constructs are used to form the overall structure of a program as *module*, *object*, *definition* and *implementation* program *units*. Each construct may exist as a *separately compiled unit* or may be textually embedded within certain of the other constructs. A number of relations hold between these constructs which define how they may be used together; they are as follows, where *x* and *y* each represent a construct:

x contains y
Construct *x* may have one or more of construct *y* textually nested within it.

x imports y
Construct *x* may import declarations from one or more construct *y*.

x aggregates from y
Construct *x* may import implementation fragments from a construct *y*.

x implements y
If the names of a *definition* and an *implementation* are identical then the *implementation* provides at least part of the implementation of the *definition*, or it may provide implementations for one or more *definitions*.

x refines y
definition x refines *definition y*, omitting, adding to, or modifying its services.

The rules for valid use of the constructs (program units) are illustrated in Figure 1. They are:

A *module* unit can have *definition*, *implementation* and *object* constructs textually nested in it

module, *definition*, *implementation* and *object* units can import declarations from other *module*, *definition* and *object* units

module, *implementation* and *object* units can aggregate from other *implementation* units

module, *implementation* and *object* units can implement *definition* constructs

definition constructs can refine other *definition* constructs

x/y	D	I	O	M
D				
I				
O				
M	+	+	+	

x contains y

x/y	D	I	O	M
D	+		+	+
I	+		+	+
O	+		+	+
M	+		+	+

x imports y

x/y	D	I	O	M
D				
I		+		
O		+		
M		+		

x aggregates from y

x/y	D	I	O	M
D				
I	+			
O	+			
M	+			

x implements y

x/y	D	I	O	M
D	+			
I				
O				
M				

x refines y

Key:
D= definition
I = implementation
O= object
M= module
+= Relation is valid

Figure 1 Valid relations between Constructs (Program Units)

3 Syntax Notation

The syntax of the language is defined in an *Extended Backus-Naur Formalism* (EBNF). The complete syntax of Zonnon is defined in section 17. Relevant fragments of the syntax are also provided in the text as each feature of the language is defined.

3.1 Definition of Extended Backus-Naur Formalism

The EBNF notation used in this report has the following features:

- Alternatives are separated by |.
- Brackets [and] denote that the enclosed expression is optional.
- Braces { and } denote its repetition (possibly 0 times).
- Parentheses (and) are used to form groups of items.
- Non-terminal symbols start with an upper-case letter (e.g. *Statement*).
- Terminal symbols either start with a lower-case letter (e.g. *letter)*, or are written in bold letters (e.g. **begin**), or are denoted by strings (e.g. ":=").
- Comments start with // and continue to the end of the line.

3.2 EBNF defined in EBNF

It is possible to define the EBNF syntax using EBNF as an example

```
Syntax= {Production}.
Production= NonTerminalSymbol "=" Expression ".".
Expression= Term {"|" Term}.
Term = Factor {Factor}.
Factor = terminalSymbol | NonTerminalSymbol |
        "(" Expression ")" | "[" Expression "]" | "{" Expression "}" .
```

3.3 Description of EBNF

The EBNF constructs are described below:

3.3.1 Sequence

```
A = BC.
```

An *A* consists of a *B* followed by a *C*

Examples:

```
Sentence = Subject Predicate.
FileName = Name '.' Extension.
Name = FirstName Surname.
```

3.3.2 Repetition

```
A = {B}.
```

An *A* consists of zero or more *B*'s.

Examples:

```
File = {Record}.
Bill = {Item Price}.
```

3.3.3 Selection

```
A = B | C.
```

An *A* consists of a *B* or a *C*.

Examples:

```
Fork = Resource | Data.
Meal = Breakfast | Lunch | Dinner.
```

3.3.4 Option

```
A = [B].
```

An *A* consists of a *B* or nothing.

Example:

```
SelectedDrink = [ Tea | Coffee | Chocolate ]. // Possibly none!
```

3.3.5 Quotes and **bold** font

Text in quotes or in a **bold** font stands for itself.

Examples:

```
ImportDeclaration = import Import {"," Import}.
OwnSymbol = "me" | self.
```

4 Language Symbols and Identifiers

4.1 Vocabulary and Representation

Symbols are identifiers, numbers, strings, operators, and delimiters. There are some lexical rules:

- Blanks and line breaks must not occur within symbols and are ignored unless they are essential to separate two consecutive symbols (except in comments, and within strings).
- Capital and lower-case letters are considered as distinct.

4.2 Identifiers

Identifiers are sequences of letters and digits and underscores '_'. The first character must be a letter or an underscore.

```
ident = ( letter | "_" ) { letter | digit | "_" }.
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
```

The case of letters is significant in identifiers, except in predefined identifiers which may be written *either* entirely in lower-case letters *or* entirely in upper-case letters (see 5.3.1 and section 17)

Examples:

```
X     Scan    ZonnonGetSymbol firstLetter

_external_package27  (* underscore typically used for interoperability with other languages *)
```

4.3 Modifiers and Specifiers

A *modifier* is used to indicate alternative semantics, where the same syntax is used for more than one purpose. It is a list of identifiers contained in braces and separated by commas. The valid identifiers depend on the context in which the modifier is applied.

```
Modifiers = "{" IdentList "}".
IdentList = ident { "," ident }.
```

```
ident = ( letter | "_" ) { letter | digit | "_" }.
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
```

Examples:

```
{ value }        { public }        { public, value}
```

A *specifier* is used to provide additional information such as the type of an expected object, or a width. It comprises a list of words or numbers contained in braces { } or an EBNF protocol specification (See also 5.3.4)

Examples:

```
var r: real{32}; (* real in 32-bit format *)
i := integer(t); (* the value of t expressed as an integer *)
{ bodypart = LEG | NECK| ARM} (* EBNF protocol *)
```

4.4 Numeric Constants

Numbers are (unsigned) integer, cardinal or real constants. If the constant is specified with the suffix *H*, the representation is hexadecimal, otherwise the representation is decimal. A real number always contains a decimal point and optionally it may also contain a decimal scale factor. The letter *E* means 'times ten to the power of'. A numeric constant may optionally be followed by a width modifier which is the number of bits to be used for its representation (surrounded by braces). If no width is specified then the default value defined in the *Zonnon Programmers' Manual* is used [Compiler]. For further information on types see 12.1.

```
number = (whole | real) [ "{" Width "}" ].
whole = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+" | "–"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
Width = ConstExpression.
```

A *whole* constant is compatible with both integer (signed) types and cardinal (unsigned) types.

Examples:

constant	*type*	*value*
1991	integer or cardinal	1991
0DH{8}	integer{8} or cardinal{8}	13
12.3	real	12.3
4.567E8	real	456700000
0.57712566E–6{64}	real{64}	0.00000057712566

4.5 Character Constants

A character constant is a character enclosed in single (') or double (") quote marks. The opening quote must be the same as the closing quote and must not be the character itself. Character constants may also be denoted by the ordinal number of the character in hexadecimal notation followed by the letter X.

```
CharConstant = '"' character '"' | "'" character "'" | digit { HexDigit } "X".
character = // Any character from the alphabet except the current delimiter character
```

This is useful for expressing special characters that are either non-printable or that are part of an extended character set.

Examples:

```
"a"   'n'   '"'   '"'   20X
```

4.6 String Constants

String constants are sequences of characters enclosed in single (') or double (") quote marks; the opening quote must be the same as the closing quote. A string constant may not contain its delimiting quote character or a line break. The number of characters in a string is called its length. A single character string constant (of length 1) can be used wherever a character constant is allowed and vice versa. String constants can be assigned to variables of type *string* (see 5.3.1).

```
StringConstant = '"' { character } '"' | "'" { character } "'".
character = // Any character from the alphabet except the current delimiter character
```

Examples:

```
"Zonnon" "Don't worry!"    "x"        'hello world'
```

4.7 Reserved Words, Delimiters and Operators

Operators and delimiters are the special characters, character pairs, or reserved words listed below.

4.7.1 Reserved Words

The following reserved words (shown in **bold** in this report) may not be used as identifiers and are written *either* entirely in lower-case letters:

accept activity array as await begin by case const definition div do else elsif end exception exit false for if implementation implements import in is loop mod module new nil object of on operator or procedure protocol record refines repeat return self termination then to true type until var while

or entirely in upper-case letters:

ACCEPT ACTIVITY ARRAY AS AWAIT BEGIN BY CASE CONST DEFINITION DIV DO ELSE ELSIF END EXCEPTION EXIT FALSE FOR IF IMPLEMENTATION IMPLEMENTS IMPORT IN IS LOOP MOD MODULE NEW NIL OBJECT OF ON OPERATOR OR PROCEDURE PROTOCOL RECORD REFINES REPEAT RETURN SELF TERMINATION THEN TO TRUE TYPE UNTIL VAR WHILE

4.7.2 Delimiters

The delimiter characters are:

( ) [ ] { } . (dot) , (comma) ; (semicolon) : (colon) .. (range)

| (case separator) ' (single quote) " (double quote)

4.7.3 Predefined Operators

The predefined operators are:

– (unary minus) + (unary plus) ~ (negation)

** (exponentiation)

^ (unary dereference)

+ – * / **div** **mod** & **or**

:= (assignment) = (equality) # (not equal) < <= > >= **in** **implements**

4.7.4 User-Defined Operators

Zonnon introduces the concept of user-defined operators. They are declared like procedures. (See 6.3).

4.8 Comments

Comments may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (* and closed by *). Comments may be nested. They do not affect the meaning of a program. They are shown in italics in this report.

```
(* this is a comment *)
(* This comment continues
  on to the next line *)
(* Nested comments (* look like *) this *)
```

5 Declarations

Declarations are used to introduce identifiers and to indicate their type.

```
Declarations = { SimpleDeclaration } { ProcedureDeclaration }.

SimpleDeclaration = ( const [ Modifiers ] { ConstantDeclaration ";" }
                    | type  [ Modifiers ] { TypeDeclaration ";" }
                    | var   [ Modifiers ] { VariableDeclaration ";" } ).
```

5.1 Identifier Declarations and Scope Rules

Every identifier occurring in a program must be introduced by a declaration, unless it is predefined. Declarations also specify certain permanent properties of an item, such as whether it is a constant, a type, a variable (see 5.4), or a procedure (see section 1). The identifier is then used to refer to the associated item.

The scope of an identifier is the scope to which its declaration belongs and hence to which it is local. It excludes the scopes of identically named identifiers which are declared in nested blocks. The scope rules are:

- No identifier may denote more than one item within a given scope (i.e. no identifier may be declared more than once in a block).
- An identifier may only be referenced within its scope.
- Identifiers denoting object fields or methods/procedures are valid only in object designators, where they must be qualified by the name of the object.
- Related declarations within a scope may be declared in any order.

```
IdentList = ident { "," ident }.
```

Examples:

```
Month.Oct (* see 5.3.3 *)
NameSpace.Program
```

5.1.1 Declaration Modifiers

Declarations may have optional modifiers which are defined as follows:

- *private*: the identifiers are visible only in the scope of their declarations.
- *public*: the identifiers are visible in the scope in which they are declared and in any constructs that explicitly import the program construct that contains its declaration.
- *immutable*: is used in conjunction with *public* and indicates that values are read-only from outside the scope in which of declaration.

Declaration modifiers may be used with declarations in the inner scope of a procedure/method.

```
SimpleDeclaration = ( const [ Modifiers ] { ConstantDeclaration ";" }
                    | type  [ Modifiers ] { TypeDeclaration ";" }
                    | var   [ Modifiers ] { VariableDeclaration ";" } ).
```

Example:

```
var {private} flag, statusWord: boolean; (* flag and statusWord are both private *)
var {public, immutable} refCount: integer; (* read only access *)
```

5.2 Constant Declarations

A constant declaration associates an identifier with a constant value.

```
ConstantDeclaration = ident "=" ConstExpression.
ConstExpression  = Expression.
```

Examples:

```
const N = 10;
      limit = 2 * N – 1;  (* see 6.2.2 *)
      fullSet = { min(set) .. max(set) };  (* see 5.3.1 *)
```

A constant expression is an expression that can be evaluated solely by a textual scan without actually executing the program. Its operands must be constants or calls of predefined functions.

5.3 Type Declarations

A data type determines the set of values variables of that type may assume and the operators that are applicable to them. A type declaration associates an identifier with a type. In the case of the structured types (arrays and objects) it also defines the structure of variables of this type. Object types are defined in 5.3.6 and 10.1

```
TypeDeclaration = ident "=" Type.
```

```
Type = (TypeName [ "{" Width "}" ] | EnumType | ArrayType | ProcedureType
        | InterfaceType | ObjectType | RecordType | ProtocolType ).
```

5.3.1 Basic Types

The basic types are denoted by predefined identifiers. The associated operators are defined in 6.2 and the predefined function procedures in section 13. The values of the basic types are the following:

- **object** the generic type from which object types are derived (**object** is a reserved word)
- boolean the truth values *true* and *false*
- char the underlying character set of the environment
- integer the integers between *min*(*integer*) and *max*(*integer*)
- cardinal positive whole numbers between *min*(*cardinal*) and *max*(*cardinal*)
- fixed large numbers with fixed precision between *min*(*fixed*) and *max*(*fixed*)
- real the real numbers between *min*(*real*) and *max*(*real*)
- set the set of whole numbers (integer or cardinal) between 0 and *max*(*set*)
- string character strings

5.3.2 Type widths

For types *char*, *integer*, *cardinal*, *real* and *set* the number of bits required to contain the value can be specified by a modifier stating a whole number of bits as a constant value in braces { } after the type name. The available widths for a given implementation are defined in the *Zonnon Programmers' Manual.* The default type widths are:

```
char{16}, set{32}, integer{32}, real {80}, cardinal {32}, fixed{128}
```

For conversion between different types see section 5.3.12.

5.3.3 Enumeration Types

An enumeration is a type that comprises a named list of identifiers denoting the values which constitute the type. These identifiers are qualified by the type name when used as named constants in the program. The values are ordered and their ordering relation is defined by their textual sequence in the enumeration list. No other values belong to the type. The ordinal number of the first value is zero and increases by one for each subsequent identifier.

```
EnumType = "(" IdentList ")".
IdentList = ident { "," ident }.
```

Examples:

```
type NumberKind = (Bin, Oct, Dec, Hex);
     Month = (Jan, Feb, Mar, Apr, May, Jun, July, Sep, Oct, Nov, Dec);
```

Names in separate enumerations need not be different as their use is always qualified. So for example *NumberKind.Oct* is distinct from *Month.Oct*.

Values of expressions can be converted to a different type. (See section 5.3.12).

The predefined function *pred* returns the value of the predecessor of the enumeration value given as its parameter, for all except the first value of the enumeration. The predefined function *succ* returns the value of the successor of the enumeration value given as its parameter, for all except the last value of the enumeration.

5.3.4 Protocol types

A *protocol* is a special form of enumeration that also includes the syntax of the controlled interaction of activities. In this case, the elements of the enumeration directly correspond to the terminal symbols in the protocol EBNF syntax which defines the ordering of communication between activities. The EBNF productions of the protocol are separated by commas, The last production must be the main (root) production. The "?" character indicates a response from the partner activity. The list of enumerated identifiers (of the EBNF tokens) may be null; this indicates that there is no communication between the two activities involved. See section 9 for more details.

```
ProtocolDeclaration = protocol ProtocolName "=" "(" ProtocolSpecification ")" ";".
ProtocolSpecification = [ Alphabet "," ] Grammar
                      | Alphabet [ "," Grammar ].
Alphabet = TerminalSymbol { "," TerminalSymbol }.
Grammar = Production { "," Production }.
Production = ProductionName "=" Alternative.
Alternative = ItemSequence { "|" ItemSequence }.
ItemSequence = Item { Item }.
Item = ( ["?"] TerminalSymbol | ProductionName | TypeName |
         Alternative | Group | Optional | Repetition).
Group = "(" ItemSequence ")".
Optional = "[" ItemSequence "]".
Repetition = "{" ItemSequence "}".
TerminalSymbol = number | ident | charConstant.
ProductionName = ident.
```

Example:

```
protocol P = ( ); (* a 'null' protocol *)
protocol  UART = (* the tokens are enumerated as terminal symbols of  the EBNF syntax *)
  ( dataOUT = char, dataIN = char,
    readyToSend,  clearToSend, dataOUT, clearToReceive, readyToReceive, dataIN,
    readyToSend  ?clearToSend dataOUT} | { ?clearToReceive readyToReceive {?dataIN} }
  )
```

See also the extended example in Section 16.

5.3.5 Array Types

An array is a structure consisting of a number of elements that are all of the same type, called the element type. Arrays can be indexed either by a positive whole number or by a value of an enumeration type. In the first case, the number of elements in the array's declaration determines its length. The array's elements are designated by indices, which are whole-number values between 0 and the array length minus 1. In the second case the name of the enumeration type is used in the declaration and the array's elements are designated by values of the enumeration type.

The syntax rules for the array type are:

```
ArrayType = array Length {"," Length} of Type.
Length = ConstExpression | "*".
```

Arrays can be multidimensional; that is, the array elements may themselves be arrays, and mixing the different length specification forms is acceptable in principle. An example and a counter example are:

```
type  Acceptable = array * of array 42 of T; (* array *, 42 of T *)
            Jagged = array 42 of array * of T; (* 'jagged' array *)
```

The declaration *array m, n of T* is textually equivalent to *array m of array n of T*.

For example **array** * **of array** 42 **of** T can be written **array** *, 42 **of** T

The expression *len(a, n)* returns the number of elements in dimension *n* of the array *a*. The expression *len(a)* is a shorthand for *len(a, 0)*.

In an array the number of elements in any dimension may be variable and is then denoted by an asterisk. It is the programmer's responsibility to allocate storage space on the heap for an array by using the reserved word *new* for each instance of the array:

```
arrayVariable := new ArrayType(length0, length1, ... );
```

The length values must be expressed by positive expressions of integer or cardinal type and the number of such expressions must correspond to the number of dimensions of the variable. An array must be declared as either fully static or fully dynamic.

Examples of the use of arrays are:

```
type Vector = array * of integer;

procedure CreateAndReadVector(var a: Vector);
      var i, n: integer;
begin
      read(n);
```

```
        a := new Vector(n);
        for i := 0 to len(a) – 1 do
            read(a[i])
        end
end CreateAndReadVector;

procedure InitializeMatrix(var mat: array *, * of real);
        var i, j: integer;
begin
        for i := 0 to len(mat, 0) – 1 do
            for j := 0 to len(mat, 1) – 1 do
                mat[i, j] := 0.0
            end
        end
end InitializeMatrix;
…
var m: array 10, 10 of real;
…
InitializeMatrix(m);
```

5.3.6 The *string* Type

Variables of type *string* represent immutable sequences of characters.
Strings can be compared for equality and inequality by using the '=' and '#' operators. The operator '+' signifies concatenation of strings and ':=' signifies assignment. The predefined procedure *copy* converts between *string* type and *array of char* representation and vice versa and the predefined function *len* delivers the length of a string (see Section 13). The properties of the string type are not defined as part of the Zonnon language; see *Zonnon Programmers' Manual* [Compiler]. An example of a library module *Strings* is shown in Section 15. String syntax and constants are defined in section 4.6.

5.3.7 Object Types

An object is a data type template comprising fields, methods and activities. The fields represent the object's state, the methods its functionality and the activities its concurrent activities. It can expose its interface to its system environment in two ways. Firstly by the interface of its intrinsic type (referred to as its intrinsic interface), that is the set of all the elements which the programmer chooses to make public rather than keep private, and secondly by a number of definitions, each of which exposes a distinct facet representing an aspect of the object's services to its clients.

```
ObjectType = object ObjectDefinition ident.

ObjectDefinition = [ FormalParameters ] [ ImplementationClause ] ";"
                    [ ImportDeclaration ]
                    { SimpleDeclaration | ProcedureDeclaration |
                      ProtocolDeclaration | ActivityDeclaration }
                    ( UnitBody | end ).

Object = object [ Modifiers ] ObjectName ObjectDefinition SimpleName.  // when declared as a unit

ImplementationClause = implements ImplementedDefinitionName { ","
ImplementedDefinitionName }.

ImplementedDefinitionName = DefinitionName | "[" "]".

ImportDeclaration = import Import { "," Import } ";".

Import = ImportedName [ as ident ].

ImportedName = ( ModuleName |  DefinitionName|  ImplementationName
               |  NamespaceName |  ObjectName ).

UnitBody = begin [ StatementSequence ] end.
```

An object is composed of declarations including constants, types, variables (referred to as fields), and procedures (referred to as methods). Variables which are reference objects provide references to objects which are created dynamically during program execution within the program using *new*. An object may optionally have parameters which are used in the body of the object to initialize fields when the object is instantiated using *new*.

The modifiers *public* and *private* can be used to declare the visibility of the contents of an object. If no modifier is present then the default is *private*. Individual items may be made public by explicit use of the modifier *public* following their declaration. The object itself can also have a modifier which

denotes it as either a value object or a reference object using the modifier values *value* and *ref* respectively. The default modifier is *value*.

The modifier *protected* can be used to declare that the object is a monitor [Monitor]. This is a construct which at runtime prevents more than one thread of execution being executed within an object instance at any time. It is used to manage mutually exclusive access to the fields within an object instance, particularly in concurrent programs, see also section 9.4 .

Examples:

```
object {ref} Box(w, h: integer);
        var width, height: integer;

        procedure Area( ): integer;
        begin
             return width * height
        end Area;

begin
        self.width := w; self.height := h (* self is optional in both cases here *)
end Box.
…
var box: Box;
…
box := new Box(3, 7);  (* makes new Box object with width 3 and height 7 *)
```

See 10.2 on *objects* as separately compiled program units; object declarations cannot be nested.

5.3.8 Record Types

A *record* is a value object type. It can be used to encapsulate variable declarations. The keyword *record* is equivalent to *object* {*value*}. Record declarations cannot be nested.

```
RecordType = record { VariableDeclaration ";" } end ident.
```

Examples:

```
record Position; (* declares the record-type Position *)
        x, y: integer
end Position;
```

which is equivalent to:

```
object {value} Position; (* declares the record-type Position *)
        var x, y: integer
end Position;

record Date; (* declares the record-type Date *)
        year: integer{8};
        month: Month;
        day: integer{8}
end Date;
```

5.3.9 Postulated Interface Types

An interface is a type for a postulated object composed from one or more definitions; see 5.3.10 and 10.2 for further details.

```
InterfaceType = object [ PostulatedInterface ].

PostulatedInterface = "{" DefinitionName { "," DefinitionName } "}".
```

For example: the construct

```
var x: object {D,E};
```

declares a variable of a generic object type with a specifier that defines that it implements the definitions *D* and *E*. When an object instance is assigned to x at runtime it must at least implement definitions *D* and *E*.

5.3.10 Procedure Types

A procedure type is a template for a procedure which can be referenced via a procedure variable. A variable of a procedure type *T* has a procedure or method *P* or *nil* as its value. If *P* is assigned to a

variable of type *T*, the formal parameter lists of *P* and *T* must match according to a set of rules. (See 12.4). *P* must not be a predefined procedure nor may it be local to another procedure. However, the sole exception is a global module procedure, which may be used with or without qualification within the module in which it is declared.

When a method is assigned to a variable of type procedure it must be prefixed by (the designator of) an object instance that contains it. This may also be referred to as a 'delegate'. When any procedure is called it shares the thread of execution with its caller, on termination of the procedure the caller resumes execution from the statement immediately following the call.

```
ProcedureType = procedure [ ProcedureTypeFormals ].
ProcedureTypeFormals = "(" [ PTFSection { ";" PTFSection } ] ")" [ ":" FormalType ].
PTFSection = [ var ] FormalType { "," FormalType }.
FormalType = { array "*" of } ( TypeName | InterfaceType ).
```

Example:

```
type  Delegate = procedure;
      Action = procedure (n: integer);
      Function = procedure (n: integer): integer;
```

5.3.11 Activity Types

An activity type is a template for an activity which can be instantiated and referenced via a variable. A variable of an activity type *T* has an activity *A* or *nil* as its value. The declaration of an activity is very similar to that of a procedure, however there are significant differences in their runtime behaviour. When a procedure is called it shares the thread of execution with its caller, on termination of the procedure the caller resumes execution from the statement immediately following the call. When an activity is instantiated it then has its own thread of execution which lasts until the activity terminates. Parameters are passed to and from an activity via a protocol using a procedure call –like notation. See section 9.1 for more details and examples.

```
ActivityDeclaration = activity ActivityName [ FormalParameters ] [ProcImplementationClause]";"
                               Declarations
                               ( UnitBody | end ) SimpleName.
```

5.3.12 Conversion between Types

In Zonnon, type conversions within a 'family' (such as *integer*) are implicit when guaranteed to be safe. However, conversions between families must be explicit (because a change of internal representation is involved). Inverse conversions (for example, *integer*{32} to *integer*{16}) must always be explicit. The exception mechanism detects conversion anomalies (see 7.10.1).

The interoperability between types is summarized in the table below and is based on the ECMA Common Type System model [CLI], as used in .NET:

Type family	*width in bits*								
	8		16		32		64		128
fixed									M
								↗	
real					M	→	M		
				↗		↗			
integer	M	→	M	→	M	→	M		
		↗		↗		↗			
cardinal	M	→	M	→	M	→	M		
		↗	↑						
char	M	→	M						

M mandatory type for conforming implementation
→ implicit conversion always allowed (within same family)
↗, ↑ explicit conversion always allowed (change of representation)

Note that implicit conversions are transitive. Inverse conversion (in the opposite direction of the arrows) requires an explicit conversion and may result in truncation or an exception.

5.3.12.1 Type name used as conversion function (predefined types)

To achieve a type conversion, the name of the destination type is regarded as a built-in function which takes an expression of the source type as a parameter and returns the converted value. An optional second parameter indicates the desired width of the result.

Syntax:

```
TypeName "(" Expression [ "," Size ] ")"
```

Examples:

```
integer(x + e/f, 16)
```

is the value of the expression *x + e/f* represented as a 16-bit integer (exception may be raised if conversion not possible).

```
integer(x + e/f)
```

is the value of the expression *x + e/f* represented as a 32-bit integer (assuming that 32 is the implementation's default width for integer).

Note that integers cannot be implicitly conversion to real and so:

```
var count, sum: integer; mean: real;
...
mean := sum / count
```

is *not* syntactically allowed and requires explicit conversions:

```
mean := real(sum) / real(count)
```

5.3.12.2 Type name used as conversion function (object types)

Zonnon supports both object-oriented programming and operator-style programming:

In object-oriented programming, the desired definition (interface) of a servant object needs to be known but not its full type. If an object type *X* implements definitions *D* and *E*, instances of *X* can be regarded as being of types *D* or *E* respectively, depending on the client's perspective. So, if *D* exports a method *f* and *E* exports a method *g* and *x* is a variable of type *X*, we can write *D(x).f* and *E(x).g*, for example.
In operator-style programming, we apply operators to operands of a certain statically known type (strong typing). For example, we might want to apply the operator procedure:

```
operator "*" (x: X; y: Y): Z; …
```

to generic objects:

```
var s, t: object;
```

We need to a type cast which takes the function-like *type-name(expression)* in this case:

```
if (s is X) & (t is Y) then z := f (X(s), Y(t )) else (* type error *) end
```

5.3.12.3 Implicit type of constant

The type of a simple numeric constant is determined by the declaration of the variable to which it is assigned. So for instance, given the declaration:

```
var i: integer {16};
```

then the assignment

```
i := 1
```

is actually treated by the compiler as being

```
i := 1{16}
```

If no width is specified, then the default width for that type is assumed (see 5.3.2)

Other type conversions are achieved by means of predefined procedures (see Section 13).

5.4 Variable declarations

A variable holds a value that can be assigned to it from an expression in an assignment operation (see 7.1). A variable is defined to have a type, which may not change, and which defines the set of values that it may hold. Variable declarations introduce variables by defining an identifier and a data type for each one.

```
VariableDeclaration = IdentList ":" Type.
```

Examples:

```
var i, j, k: integer;
    x, y: real;
    p, q: boolean;
    s: set {32};
    a: array 100 of real;
    name: array 32 of char;
    size, count: integer;
    mousePosition: Position;
    dateOfBirth, today: Date;
```

6 Expressions

An expression is a construct which specifies a computation. In an expression constants and current values of variables are combined to compute other values by the application of operators and function procedures. An expression consists of operands and operators; parentheses may be used to express specific associations of operators and operands. The types of intermediate values used during expression evaluation are the responsibility of the implementation (see [Compiler]). The type of the result of an expression is defined in the section on expression compatibility (see 12.6).

```
Expression = SimpleExpression
                    [ ( "=" | "#" | "<" | "<=" | ">" | ">=" | in ) SimpleExpression ]
           | Designator implements DefinitionName
           | Designator is TypeName.

SimpleExpression = [ "+"|"-" ] Term { ( "+" | "-" | or ) Term }.

Term = Factor { ( "*" | "/" | div | mod | "&" ) Factor }.

Factor =  number
        | CharConstant
        | string
        | nil
        | Set
        | Designator
        | new TypeName [ "(" ActualParameters ")" ]
        | new ActivityName [ "(" ActualParameters ")" ]
        | "(" Expression ")"
        | "~" Factor
        | Factor "**" Factor.
```

6.1 Operands and Designators

With the exception of set constructors (see 6.2.3) and literal constants (numbers, character constants, or strings constants), operands are denoted by designators. A designator consists of an identifier referring to a constant, variable, or procedure. This identifier may possibly be qualified by an identifier denoting a module, definition, implementation or object and may be followed by selectors if the designated object is an element of a structure.

```
Designator = Instance
           | TypeName "(" Expression [ "," Size ] ")"          // Conversion
           | Designator "^"                                    // Dereference
           | Designator "[" Expression { "," Expression } "]"  // Array element
           | Designator "(" [ ActualParameters ] ")"           // Function call
           | Designator "." MemberName                         // Member selector

Instance = ( self | InstanceName | DefinitionName "(" InstanceName ")" ).

Size = ConstantExpression.

ActualParameters = Actual { "," Actual }.

Actual = Expression [ "{" [ var ] FormalType "}" ].  // Argument with type signature
```

The ^ symbol is used so that a reference can optionally be made explicit in a program text.

Examples:

designator	*type*	*meaning*
size	integer	value of the variable called *size*
a[i]	real	the element of the array *a* at position *i*
dateOfBirth.day	integer{8}	the *day* field of the object called *dateOfBirth*
w[3].name[i]	char	the element at position *i* in the field called *name* of the element at position 3 of the array called *w*

If *a* designates an array, then *a[e]* denotes that element of *a* whose index is the current value of the expression *e*. The expression *e* must be of an enumeration, a cardinal or an integer type. A designator of the form $a[e_0, e_1, \ldots, e_n]$ stands for $a[e_0][e_1]\ldots[e_n]$.

If *obj* designates an object, then *obj.f* denotes the field *f* of *obj* or the method *f* of the object *obj*, (see 10.1).

If the designated object is a constant or a variable, then the designator refers to its current value. If it is a procedure without any parameter list, the designator refers to the procedure itself. However, if it is a function procedure and is followed by a (possibly empty) parameter list it causes an activation of that procedure and stands for its resulting value. The actual parameters must correspond to the formal parameters as in proper (non-function) procedure calls. (See 7.2).

6.2 Predefined Operators

Predefined operators are fixed and built into the language.

6.2.1 Logical operators

These operators apply to *boolean* operands and yield a *boolean* result.

or	logical disjunction	p **or** q	'if *p* then *true*, else *q*'
&	logical conjunction	p & q	'if *p* then *q*, else *false*'
~	negation	~ p	'not *p*'

6.2.2 Arithmetic operators

The operators +, –, and * apply to operands of numeric types in an expression. (See 6.3.1). The division operator / applies only to operands of type *real* and produces a result of type *real*. When used as monadic operators, – denotes sign inversion and + denotes the identity operation.

+	sum
–	difference
*	product
/	real quotient (of reals)
**	power (x**y signifies x^y)

Examples:

```
i := j + k;
x := real(i) / real(j); (* see section 5.3.12 *)
```

The operators *div* and *mod* apply to integer and cardinal operands only.

div	integer quotient
mod	modulus

They are related by the following formulas defined for any *x* and positive divisors *y*:

```
x = (x div y) * y + (x mod y)
0 <= (x mod y) < y
```

If the value of the divisor *y* is negative then the meanings of the operators *div* and *mod* are mathematically ambiguous and so are left undefined; their effect is implementation specific. (See [Compiler]). It is recommended that programmers test for this condition and employ mathematics to ensure that only positive divisors values are used. For example:

x	*y*	*x* ***div*** *y*	*x* ***mod*** *y*
5	3	1	2
–5	3	–2	1

6.2.3 Set Operators

Set operators apply to operands of type *set* and yield a result of type *set*. The declared bit widths of the operand *SET*s must be identical. The monadic minus sign denotes the complement of *x*, that is, *–x* denotes the set of integers between 0 and *max(set)* which are *not* elements of *x*.

+	union	bitwise or
–	difference (x – y = x * (–y))	bitwise subtraction
*	intersection	bitwise and
/	symmetric set difference (x / y = (x – y) + (y – x))	bitwise exclusive or

A set constructor defines the value of a set by listing its elements, if any, between braces. The elements must be integers in the range 0 .. *max*(*set*). A range *m* .. *n* denotes all integers in the interval starting with element *m* and ending with element *n*, inclusive of *m* and *n*. If *m* > *n* then *m* .. *n* denotes an empty set.

```
Set = "{" [ SetElement { "," SetElement } ] "}".
SetElement = Expression [ ".." Expression ].
```

Examples of the use of sets:

```
const left = 0; right = 1; top = 2; bottom = 3;
var edges: set; x, y: integer;
begin
        edges := { }; (* the empty set *)
        if x < xMin then edges := edges + {left}
        …
        if left in edges then … (* clip at left *)  …

const opCodemask = {0..3};
var opCode, word: set;
        …
        opCode := word * opCodeMask; (* extract the op-code *)
```

6.2.4 Relations

Relations yield a *boolean* result. The relations =, #, <, <=, >, and >= apply to the numeric types and *char*. The relations = and # also apply to *boolean, set* and *string*, as well as to procedure types (including the value *nil*). *x in s* stands for '*x* is an element of *s*'. *x* must be of an integer type, and *s* of type *set*.

=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
in	set membership
implements	*x* ***implements*** *D* is true if object *x* implements definition *D*
is	*x* ***is*** *T* is true if and only if the intrinsic type of *x* is *T*, for any type *T*

Examples of expressions

expression	*type*	*meaning*
1991	integer	simple constant value
i **div** 3	integer	integer division of *i* by 3
~wellFormed **or** outOfRange	boolean	(not well-formed) or out-of-range
(i + j) * (i – j)	integer	arithmetic expression
s – {8, 9, 13}	set{8}	*s* with 8, 9, 13 removed
keys **in** {left, right}	boolean	*keys* is *left* or *right* or *both*
('0' <= ch) & (ch <= '9')	boolean	*ch* is a digit

6.3 User-Defined Operators and Operator Declarations

Operator overloading introduces the notion of user-defined operators and the opportunity to use familiar syntax in expressions involving them. Operators are defined only in a module implementing an abstract data type i.e. one that defines a new user-defined type and that implements a set of operations on it. Typically this can be used when introducing new data types such as complex numbers or matrices.

6.3.1 Operators overloading

The set of predefined operators that can be overloaded is as follows:

```
– (unary minus)  + (unary plus)  ~
** (exponentiation)
^ (unary dereference)
+ – * / div mod & or
= # < <= > >= in
:= (assignment is a special case, see 6.3.3)
```

The precedence of operators is defined in 6.4.

Note that the *implements* and *is* operators cannot be overloaded, see 10.1.

6.3.2 New Operator Declarations

Overloaded and new operators are introduced as operator declarations. The syntax of the declaration is as follows:

```
OperatorDeclaration = operator [ Modifiers ] OpSymbol [ FormalParameters ] ";" OperatorBody ";".
OperatorBody = Declarations UnitBody OpSymbol.
OpSymbol = string.   // A 1, 2 or 3-character string; the set of possible symbols is restricted
```

Example:

```
record Complex;
        re, im: real;
end Complex;

operator '+' (x1, x2: Complex): Complex;
var res: Complex;
begin
        res.re := x1.re + x2.re;
        res.im := x1.im + x2.im;
        return res
end '+';
```

For all overloaded operators parameters are passed by value (as for all predefined operators) and the operator must produce a result. The sole exception is for the assignment operator where the first parameter must be passed by reference and the operator must *not* produce a result.

For overloaded operators the number of parameters in an operator declaration must be the same as that of the predefined operator with the same symbol. For a new operator declaration the number of parameters in an operator declaration must be one or two, depending on whether it is a unary or binary operation.

It is only possible to declare overloaded and new operators in a *module*, but not in an *object* or *definition*. The reason is to enable complete overloading resolution statically at compile time to support strong type checking. This is also intended to clearly separate two concepts: objects implementing interfaces (definitions) and abstract data types with associated operators. If necessary type conversion must be used at runtime, for example: to apply the operator procedure *"*"(x: X; y: Y): Z* to the generic objects ***var*** *s, t:* ***object*** would require conversion between types, so if s is of type *X* and t is of type *Y* then *z := f (X(s), Y(t))*.

Operator declaration can be made available outside the module where it is declared. In that case, it is legal to use those operators in units importing the module in normal expressions, together with the predefined operators. The compiler is responsible for selecting the right version of the operator in each case.

It is possible to define operators in a module to extend an abstract data type. These operators must be defined in terms of the operations already defined in the module where the abstract data type is declared.

Normally, all imported entities should be qualified by the name of the imported unit. This is also possible, but not required, for operators. For example, there are two legal ways to use 'new addition' for operands of some type *T*.

```
module M;
        type {public} T = …;
        operator {public} "+" ( a, b: T): T;
        begin
        …
        end "+";
end M.
```

```
object Obj;
        import M;
        var x, y : T;
begin
        x := x + y;          (* like a normal expression *)
        x := x M."+" y;      (* fully qualified, but less conventional *)
end Obj.
```

An operator procedure cannot be called as a normal function:

```
x := M."+"(x, y);  (* not legal; must use expression notation *)
```

6.3.3 Rules governing overloading

The following set of rules applies to overloaded operators:

1) Operators can only be introduced to define previously undefined operations, but *not* to refine previously defined operations

2) The type of at least one operand of an overloaded operator must be a user-defined type (an array type, an object type, a procedure type, an enumeration type). It is illegal to introduce user-defined operator versions for 'basic' types such as *integer*, *real*, and *boolean*.

3) Specifying an object type with a specified interface (such as *object { D })* as the operator's parameter is not allowed. The reason is that it must be possible to resolve operator overloading completely at compile time (i.e. statically).

4) The number of arguments, the precedence of an overloaded operator and the form (prefix or postfix) of unary operators, must be the same as those features for predefined operators with the same symbols.

5) The dereference construct with '^' symbol (see *Designator* production in the syntax) is considered here as postfix unary operator. Therefore, any overloaded ^ operator keeps the form of unary postfix operator; similarly, unary + and – operators are always unary prefix operators.

6) All operators except assignment must produce a result, which may be of any type.

7) It is also possible to overload assignment. In this case, the assignment symbol is considered as a special operator with the symbol ':=' performing a certain side effect and producing no value. Note that the assignment operator can be used to copy the value of an object of the same type. If it is overloaded with parameters of the same object type then it will be used instead of the predefined := operator for that type. In any case the value is copied by default by the predefined assignment operator semantics.

8) In the overloaded operator for assignment there must be two parameters, and the first one must be passed by reference.

9) The number of operands of a new operator is determined by the number of parameters from the operator declaration. (See section 9).

10) It is legal to specify more than one version of the overloaded and new operators with the same symbol; in that case, the types of the parameters of the corresponding operator declarations must differ from any other operator declaration for the same symbol. (See section 6.3.1)

11) .Overloaded operators can only be defined in a module where at least one of the operands is declared.

6.4 Operator Precedence

Four classes of operators with different levels of precedence (binding strengths) are syntactically distinguished when used in expressions. Operators of the same precedence associate from left to right. For example, *x – y – z* stands for *(x – y) – z*. Operator precedence from highest to lowest is:

1. unary negation operator ~
2. exponentiation operator **
3. multiplication operators * **div mod** /
4. addition operators + -
5. relations < <= = # >= > **implements in is**

The available operators are listed in the following tables. Some operators are applicable to operands of various types, denoting different operations. In these cases, the actual operation is effectively 'overloaded' and the appropriate one to use is identified by the type of the operands. The operands must be expression compatible with respect to the operator, see 12.6.

6.5 Numeric resolution within expressions

An expression consists of a series of evaluations of operators on their operands. For each operator the relationship between the resolution (width) of each of its operands and the result of the operation is defined as follows:

operator	*first operand*	*second operand*	*result*
+	integer{s}	integer{t}	integer{max(s, t)}
–	integer{s}	integer{t}	integer{max(s, t)}
*	integer{s}	integer{t}	integer{s + t}
div	integer{s}	integer{t}	integer{s}
mod	integer{s}	integer{t}	integer{t}
**	integer{s}	integer{t}	integer{s + t}
+	cardinal{s}	cardinal{t}	cardinal{max(s, t)}
–	cardinal{s}	cardinal{t}	cardinal{max(s, t)}
*	cardinal{s}	cardinal{t}	cardinal{s + t}
div	cardinal{s}	cardinal{t}	cardinal{s}
mod	cardinal{s}	cardinal{t}	cardinal{t}
+	real{s}	real{t}	real{max(s, t)}
–	real{s}	real{t}	real{max(s, t)}
*	real{s}	real{t}	real{s + t}
/	real{s}	real{t}	real{s + t}
**	real{s}	real{t}	real{s + t}
+	fixed	fixed	fixed
–	fixed	fixed	fixed
*	fixed	fixed	fixed
/	fixed	fixed	fixed
**	fixed	fixed	fixed

Note: max(s, t) = s, if s > t else t

The compiler implementation is responsible for conserving the integrity of intermediate values during the evaluation of an expression [Compiler].

7 Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, *await*, *return* and *exit* statements. Structured statements are composed of parts that are themselves statements. They are used to express sequencing and conditional, selective, and repetitive execution. A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

```
Statement = [ Assignment
            | ProcedureCall
            | IfStatement
            | CaseStatement
            | WhileStatement
            | RepeatStatement
            | LoopStatement
            | ForStatement
            | await Expression
            | exit
            | return [ Expression { "," Expression } ]
            | BlockStatement
            | Send
            | Receive
            | SendReceive
            | Accept
            | LaunchActivity
            | AnonymousActivity
            ].
```

Statement sequences denote the sequence of actions specified by the component statements which are separated by semicolons.

```
StatementSequence = Statement { ";" Statement }.
```

Example:

```
temp := a; a := b; b := temp (* swap values in a and b*)
```

7.1 The Assignment Statement

An assignment statement replaces the current value of a variable by a new value specified by an expression. The expression must be assignment compatible with the variable. (See 12.4). The assignment operator is written as ‘:=’ and pronounced as ‘becomes’.

```
Assignment = Designator { "," Designator } ":=" Expression { "," Expression }.
```

Note that multiple assignments may be made using a single statement. The effect when such statements are evaluated is as follows:

1. Each expression on the right-hand side is evaluated to produce a value.
2. Then the values are assigned in any order to their corresponding designated variables on the left-hand side.

The semantics are the same as for guarded commands [Dijkstra] and support the possibility of execution.

Examples:

```
i := 0;
p := i = j;
x := i + 1;
k := log2(i + j);
F := log2;
s := {2, 3, 5, 7, 11, 13};
a[i] := (x + y) * (x – y);
t.key := I;
w[i + 1].name := "John";
t := c;
x, y, z := a ,b, c;
```

7.1.1 Indexer Assignments

It is convenient to access the fields within an object as if it were an array using an *indexer*, otherwise such variables are usually indirectly accessed via specific method calls. Indexer access is achieved using a built-in generic definition called “[ ]” with actual methods *Get* and *Set* which enable object types to implement this definition. The Get and Set methods implement the indexed mapping from the callers index to the objects fields. A template for this is:

```
object x implements … , [ ] ;
    …
    procedure Get (i, j: integer): real implements [ ].Get ;
    begin … (* implements self [i, j] *)
    end Get;

    procedure Set (i, j: integer; x: real): implements [ ].Set ;
    begin … (* implements self [i, j] := x *)
    end Set;
    …
end x;
```

For indexed access of an object the following abbreviated syntax is allowed:

```
ImplementationClause = implements
    ImplementedDefinitionName { ","ImplementedDefinitionName }.

ImplementedDefinitionName = DefinitionName | "[" "]".
```

```
var x: X;
    x[i, j] (* instead of [ ](x).Get(i, j) *)
    x[i, j] := 3.14; (* instead of [ ](x).Set(i, j, 3.14) *)
```

For an indexer the number of index dimensions allowed depends on the compiler implementation [Compiler].

7.1.2 Abstract Assignments

The notion of an indexer is also used to achieve a so-called abstract assignment for direct access to a field of an object. Here the assignment operator ":=" is dropped and the abstract assignment is re-interpreted as the *Set* method of a zero dimensional indexer.

A template for defining an abstract assignment is:

```
object A implements [ ] ;
    …
    procedure Set (b: B): implements [ ].Set ;
    begin … (* implements self [ ] := b *)
    end Set;
    …
end A;
```

This can be then used in a full or an abbreviated form as follows:

```
var a: A; b: B;
    …
    a[ ] := b;     (* full form *)
    a := b;        (* abbreviated form *)
```

7.1.3 Properties

A *property* is a variable or object field for which access procedures are provided by the programmer and automatically called whenever its variable is accessed. Whenever the value of the variable is accessed in an expression the function that implements *Get* is called. Also whenever the value of the variable is set by an assignment, the procedure that implements *Set*, is called. A variable for which only a *Get* property is provided is 'read only' and a variable for which only a *Set* property is provided is 'write only'.
The behavior of a property is defined by its implicit methods *Get* and *Set*. So, for a property *p*: *P* defined in a definition *D* the objects implementation has the form:

```
definition D;
    var p: P;
end D.

object Obj implements D;

    procedure f ( …): P implements D.p.Get;
          (* 'getter': called automatically whenever p is accessed *)
    begin …
        return (…)
    end f;

    procedure g ( …) implements D.p.Set;
          (* 'setter': called automatically whenever p is assigned the value of the expression *)
    begin …
    end g;
end Obj.
```

The invocation of the property's *Get* and *Set* methods is transparent in the original program text at the access; in other words their behaviour is effectively a side effect, but an intentional one.

7.2 The Procedure Call

Within a *module* a procedure call invokes a procedure. When it is declared within an *object* a procedure is referred to as a method. In either case it may contain a list of actual parameters which replace the corresponding formal parameters defined in the procedure declaration. (See section 1). The correspondence is established by the relative ordering of the parameters in the actual and formal parameter lists. There are two kinds of parameters: variable and value parameters.

If a formal parameter is a variable parameter, the corresponding actual parameter must be a designator denoting a variable. If it denotes an element of a structured variable, the component

selectors are evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If a formal parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated before the procedure activation, and the resulting value is assigned to the formal parameter.

```
ProcedureCall = Designator.
```

Examples:

```
WriteInt(i * 2 + 1)
inc(w[k].count)
t.Insert("John")
```

A method call consists of the name of an object, followed by a period and then the name of a procedure declared within the object type declaration of the object. Within the method the reserved word *self* refers to the object on which the method was called.

A specific procedure call may also be 'safeguarded', by prefixing the object with a definition. For example:

```
object T implements I, D; … end T;
var t: T;
```

A client who wants to make specific use of *t*'s interpretation of the services specified by *D* (e.g. as a *supercall*) would then simply call *D*'s methods and fields safeguarded by *t*:

```
D(t).f(..); ..; .. := D(t).x;
```

The order in which the parameters is evaluated during procedure/method invocation is defined in the *Zonnon Programmers' Manual* [Compiler].

7.3 The *if* Statement

```
IfStatement =
    if Expression then StatementSequence
    {elsif Expression then StatementSequence}
    [else StatementSequence]
    end.
```

Example:

```
if ("A" <= ch) & (ch <= "Z") then ReadIdentifier
elsif ("0" <= ch) & (ch <= "9") then ReadNumber
elsif (ch = """) or (ch = ' " ') then ReadString
else SpecialCharacter
end
```

An *if* statement specifies the conditional execution of guarded statement sequences. The expression preceding a statement sequence is called its guard and its type must be *boolean*. The guards are evaluated in sequence of occurrence; if one evaluates to *true*, its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol *else* is executed, if there is one.

7.4 The *case* Statement

The *case* statement specifies the selection and execution of a statement sequence according to the value of an expression. First the *case* expression is evaluated then the statement sequence whose *case* label list contains the obtained value is executed. The *case* expression must either be of an integer or cardinal type that is expression compatible (see 12.6) with the types of all *case* labels, or both the *case* expression and the *case* labels must be of type *char* or an enumeration. *case* labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any *case*, the statement sequence following the symbol *else* is selected, if there is one, otherwise the *UnmatchedCase* exception is raised (see 7.10.1).

```
CaseStatement =  case Expression of
                   Case { "|" Case }
                 [ else StatementSequence ]
                 end.
Case = [ CaseLabel { "," CaseLabel } ":" StatementSequence ].
CaseLabel = ConstExpression [ ".." ConstExpression ].
```

Example:

```
case ch of
        "A" .. "Z": ReadIdentifier (* assumes contiguous encoding of letters*)
        | "0" .. "9": ReadNumber
        | "'", '"': ReadString
        else SpecialCharacter
end

case month of
        Month.Apr, Month.Jun, Month.Sep, Month.Nov: days := 30
|       Month.Feb:
            if Leap(year) then
                days := 29
            else
                days := 28
            end
        else days := 31
end
```

7.5 The *while* Statement

The *while* statement specifies the repeated execution of a statement sequence while the expression of type *boolean* (its guard) yields *true*. The guard is checked before every execution of the statement sequence and so the statement sequence will be executed zero or more times.

```
WhileStatement = while Expression do StatementSequence end.
```

Examples

```
var i, k, idNumber: integer;
…
while i # 3 do writeln('Hello'); i := i + 1 end

read(idNumber);
while ~Valid(idNumber) do
        write('Type ID number again ');
        read(idNumber)
end;
(* Valid(idNumber) *)

while i > 0 do i := i div 2; k := k + 1 end

while (t # nil) & (t.key # i) do t := t.left end
```

7.6 The *repeat* Statement

A *repeat* statement specifies the repeated execution of a statement sequence until a condition specified by an expression of type *boolean* is satisfied. The statement sequence is executed at least once.

```
RepeatStatement = repeat StatementSequence until Expression.
```

Examples:

```
var idNumber: integer;

repeat
        write ('Type ID number '); read(idNumber)
until Valid(idNumber);
…

var i, x: integer; buffer: array 10 of integer;
…
(* convert non-negative value of x to decimal representation *)
i := 0;
repeat buffer[i] := x mod 10; x := x div 10; inc(i) until x = 0;

(* write out digit characters in correct order *)
repeat dec(i); write(char(buffer[i] + integer("0"))) until i = 0
```

7.7 The *for* Statement

A *for* statement specifies the repeated execution of a statement sequence for a fixed number of times while a progression of values is assigned to a variable of integer or cardinal type called the control variable of the *for* statement.

```
ForStatement = for ident ":=" Expression to Expression [by ConstExpression] do
    StatementSequence
end.
```

The statement

```
for v := low to high by step do statements end
```

is equivalent to

```
v := low; temp := high;
if step > 0 then
        while v <= temp do statements; v := v + step end
else
        while v >= temp do statements; v := v + step end
end
```

The value of the expression *low* must be assignment compatible with *v* and that of *high* must be expression compatible with *v*. The value of *step* must be a non-zero constant expression of an integer or cardinal type. If *by step* is omitted, then *step* defaults to the value 1.

Example:

```
var i : integer;
…
for i := 0 to 79 do k := k + a[i] end
for i := 79 to 1 by –1 do a[i] := a[i–1] end
```

7.8 The *loop* Statement

A *loop* statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an *exit* statement within that sequence.

```
LoopStatement = loop StatementSequence end.
```

Example:

```
loop (* copy integers from input to output until 0 is typed *)
        read(i);
        if i = 0 then exit end;
        write(i)
end
```

loop statements are useful for expressing repetitions with several exit points or cases where the exit condition occurs naturally in the middle of the repeated statement sequence.

An exit statement is denoted by the symbol *exit*. It specifies termination of the enclosing *loop* statement and continuation with the statement following that *loop* statement. An *exit* statement is contextually, although not syntactically, associated with the *loop* statement which contains it.

7.9 The *return* Statement

A *return* statement is used within procedures and activities. In a procedure it is used to return a value from back its caller. It is denoted by the symbol *return*, followed by an expression if the procedure is a function procedure. The type of the expression must be assignment compatible (see 12.4) with the result type specified in the procedure. Function procedures require the presence of a *return* statement indicating the result value. In proper procedures, a *return* statement is implied by the end of the procedure body. Any explicit *return* statement therefore appears as an additional (probably exceptional) termination point.

In an activity the *return* statement must have a parameter and it is used to return the parameter value back to the client activity according to a protocol. However, in contrast to procedures statement execution continues with the next statement in the activity. See also section 9.2.2.

7.10 The Block Statement

The block statement allows the grouping together of logically related statements and the introduction of exception handlers. Block statements can be nested.

```
BlockStatement =    do [ Modifiers ]
                    [ StatementSequence ]
                    { ExceptionHandler }
                    [ CommonExceptionHandler ]
```

```
        [ TerminationHandler ]
    end.
```

The statement sequence within the block is carried out.

7.10.1 Exception Handling

If an exception occurs then the exception handlers are tried in the order in which they appear textually until one that matches the exception is found or the general exception is reached. The statement sequence corresponding to the exception name is then carried out.

```
ExceptionHandler = on ExceptionName { "," ExceptionName } do StatementSequence.
CommonExceptionHandler = on exception do StatementSequence.
TerminationHandler = on termination do StatementSequence.
```

Exception names take the form of predefined identifiers and include:

- ZeroDivision: division by zero
- Overflow: value does not lie within min(*type*) .. max(*type*)
- OutOfRange: array index out of bounds
- NilReference: uninitialized array/object/activity/protocol instance
- UnmatchedCase: control flow reached missing ***else*** in ***case*** statement
- Conversion: invalid type conversion (not guarded by '*t* ***is*** *type*')
- Read: wrongly formatted input value for *read* or *readln*
- ProtocolMismatch: the sequence of use of tokens does not match the EBNF protocol syntax

Extra information about the exception can be accessed by calling the predefined function *reason*. This causes the runtime system to return a string which explains the reason for the exception, and possibly the system context, to aid program development. For example:

```
do
    Statements
on T1, T2 do
    (* reason returns a string containing system defined info and the name T1 or T2 *)
    s := reason
on exception do
    (* reason returns a string with the name of the exception thrown and possible system-defined information *)
    s := reason
end
```

If *reason* is called outside the scope of an exception it returns the current error or warning status information from the runtime system. See also [Compiler].

The following form acts as a 'catch all':

```
do
    Statements
…
on exception do
    CatchAll
end
```

means that *CatchAll* is only executed if an exception has occurred but no textually earlier exception clause in the block matched the exception.

Example:

```
var idNumber: integer; idValid: boolean;
begin
    do
        read(idNumber);
        if Valid(idNumber) then
            idValid := true; Process(idNumber)
        else
            idValid := false (* wrong number *)
        end
        on exception do
        idValid := false (* wrong sort of characters typed *)
    end
end
```

7.11 The *await* Statement

The *await* statement is used for conditional scheduling within an activity in an object or module [AOS].

```
await Expression
```

It must occur within a block statement which has a *locked* modifier. The expression defines the precondition of continuation of execution.

When it is executed the Boolean expression is evaluated and if it is *true* then execution continues at the next statement. However, if it is *false* then execution is suspended until the system scheduler subsequently re-evaluates the condition (possibly on more than one occasion) and finds that it has become *true*. When this occurs execution continues at the next statement.

Example: object *Buffer*

This example shows how a first-in-first-out buffer can be implemented using an object. The producer, which 'puts' the data, is assumed to belong to a different activity to the consumer, which 'gets' it. The *await* statements regulate the content of the buffer. The *locked* modifiers ensure mutual exclusion of access to the shared buffers whenever they are being altered, to conserve their integrity.

```
object Buffer;
(* First-in first-out buffer ('thread safe') *)
        const bufLen = 1000;
        var data: array bufLen of integer;
            in, out: integer;

        procedure {public} Put (i: integer); (* put element into the buffer *)
        begin {locked}
            await (in + 1) mod bufLen # out;  (* wait until not full *)
            data[in] := i;
            in := (in + 1) mod bufLen
        end Put;

        procedure {public} Get (var i: integer); (* get element from the buffer *)
        begin {locked}
            await in # out;  (* wait until not empty *)
            i := data[out];
            out := (out + 1) mod bufLen
        end Get;

begin
        in := 0; out := 0;
end Buffer;
```

7.12 Protocol Send, Receive, SendReceive, Accept and Return Statements

There are several statements associated with activities and protocols, they are the *send*, *receive*, *sendReceive*, *accept* and *return* statements. They are described in sections 9.2.1 and 9.2.2.

7.13 Activity Launch Statement

An activity is launched using new to instantiate it, and possibly to provide initializing parameters, see section 9.1 for details.

```
LaunchActivity = new ActivityName [ "(" ActualParameters ")" ].
```

8 Procedure and Method Declarations and Formal Parameters

A procedure declaration consists of a procedure heading and a procedure body. The heading specifies the procedure's identifier and its formal parameters, if any. The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration. A procedure declared within an object is called a method.

There are two kinds of procedures: proper procedures and function procedures. The latter are activated by a function designator as a constituent of an expression and yield a result that is an operand of the expression. Proper procedures are activated by a procedure call. A procedure is a function procedure if its formal parameters specify a result type. The body of a function procedure must contain a return statement that defines its result.

All constants, variables, types, and procedures declared within a procedure body are local to the procedure. Since procedures may be declared as local items too, procedure declarations may be nested

(subject to implementation restrictions). The call of a procedure within its declaration implies recursive activation.

In addition to its formal parameters and locally declared items, the items declared in the environment of the procedure are also visible in the procedure (with the exception of those items that have the same name as an item declared locally).

```
ProcedureDeclaration = ProcedureHeading [ ProcImplementationClause ] ";" [ ProcedureBody ";" ].
ProcImplementationClause = implements ImplementedMemberName { ","
    ImplementedMemberName }.
ImplementedMemberName = ( DefinitionName | "[" "]" ) "." MemberName.
ProcedureHeading = procedure [ Modifiers ] ProcedureName [ FormalParameters ].
ProcedureBody = Declarations UnitBody SimpleName.
FormalParameters = "(" [ FPSection { ";" FPSection } ] ")" [ ":" FormalType ].
FPSection = [ var ] ident { "," ident } ":" FormalType.
```

Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, value and variable parameters, indicated in the formal parameter list by the absence or presence of the keyword ***var***. Value parameters are local variables to which the value of the corresponding actual parameter is assigned as an initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. The scope of a formal parameter extends from its declaration to the end of the procedure block in which it is declared. A function procedure without parameters must have an empty parameter list. It must be called by a function designator whose actual parameter list is empty too.

The rules for the correspondence between formal and actual parameters are as follows. Let T_f be the type of a formal parameter *f* (not an open array) and T_a the type of the corresponding actual parameter *a*. For variable parameters, T_a must be the same as T_f, or T_f must be an *object* type and T_a must be derived from T_f. For value parameters, *a* must be assignment compatible with *f*. (See 12.4).

If *Tf* is an open array, then *a* must be array compatible with *f*. (See 12.5). The lengths of *f* are taken from *a*.

8.1 Procedure Modifiers

A modifier may optionally occur after the reserved word *procedure* to denote its nature. The following modifiers are defined:

- *private*: the procedure is only visible in the scope in which it is declared; this is the default.
- *public*: the procedure is visible in the scope in which it is declared and within any construct that imports the construct in which it is declared.
- *sealed*: the procedure may not be further redefined (overridden),
  The inverse of being sealed is referred to as being *open*

Examples:

```
procedure ReadInt(var x: integer);
        var i: integer; ch: char;
begin
        i := 0; read(ch);
        while ("0" <= ch) & (ch <= "9") do
            i := 10 * i + (integer(ch) – integer("0")); read(ch)
        end;
        x := i
end ReadInt;

procedure {private} WriteHex(x: integer);
(* precondition: 0 <= x < 100000H *)
        var i: integer; buf: array 5 of integer;
begin
        i := 0;
        repeat buf[i] := x mod 10H; x := x div 10H; inc(i) until x = 0;
        repeat dec(i);
            if buf[i] < 10 then write(char(buf[i] + integer("0")))
            else write(char(buf[i] – 10 + integer("A")))
            end
        until i = 0
end WriteHex;
```

```
procedure log2(x: integer): integer;
(* precondition: x > 0 *)
        var y: integer;
begin
        y := 0;
        while x > 1 do x := x div 2; inc(y) end;
        return y
end log2;
```

9 Concurrency, Activities and Protocols

Zonnon can be used to write either sequential or concurrent programs. They can be designed in a modular- or an object oriented style, or a mix of both. It is possible to express the concurrency structure of the program's solution using dynamically instantiated activities, each of which has its own thread of statement execution. Pairs of activities can be bound together by a *protocol type* and interact via *links* according to a dialog specified in EBNF. The topology of the network is created dynamically at run time by the program i.e. the links are logical rather than physical and the activities are not pre-allocated to particular processors. There may be one or more processors. The execution of the activities is dynamically scheduled by the runtime system.

Zonnon is designed for writing high-level concurrent programs in a clear, well structured and type-safe way. The important point is that it should be possible to express concurrent problem solutions in a natural way i.e. in terms of high-level abstractions rather than direct manipulations of low-level abstractions such as *processes*, *threads* or *fibers* via library calls.

9.1 Activities, Active Objects and Active Modules

An *activity* encapsulates its own state space and a thread of execution of statements separate from that of the *new* statement which instantiates and then initiates it. This enables modules and objects to contain one or more dynamically created separate activities which can then be allocated by the runtime system to one or more processors for execution. Activities run in addition to, and concurrently with, the program's thread of execution shared by all the module and object bodies. Objects and modules can develop structures of cooperating activities dynamically and facilities are available for managing simultaneous access to object and module state spaces, see 10.

Activities can be declared wherever procedures can be declared and they resemble them syntactically, however they differ from procedures in some respects:

1. Activities are declared and then instantiated using the *new* operator rather than being called.
2. An activity declaration may be instantiated many times to create many clone activities.
3. There is one exception to this when an anonymous activity is declared in-line at the statement level. In this case it is instantiated when its statement is executed.
4. Any instantiated activity executes independently from its creator and has its own procedure activation stack.
5. Activities only terminate when all statements within their body have been executed.
6. An activity may optionally implement a 'protocol' which syntactically defines an exchange of tokens with its parent activity.

Activities use a procedure-like model of parameter passing to achieve the passing of protocol tokens. Notice that a simple form of 'multiple assignment' is provided for an activity to conveniently receive a list of tokens, see.7.1.2 . From this viewpoint a procedure call can be considered an equivalent of an activity so:

```
procedure P (s: S; t: T): R; ….; r := P(s, t)
        is equivalent to

do
        create activity of type P; send tokens s and t; receive tokens r
end
```

The reserved word **activity** is used to differentiate an activity declaration from that of a procedure. Once an activity has been declared then instances of it can be created in any active object or module. Activities provide a means of encapsulating concurrent behavior added to an object or module (in its role as a singleton object). An object may contain an arbitrary number of activities, or none at all, in

which case it is a passive object. Typically activities are private to the object (or module) that contains them and are instantiated by the constructor *new*.

Example:

```
object Cell (* of a pipeline *);        (* declaration of an object *)
        type Job = …;
        var in, out, n: integer;        (* the object's state space when instantiated *)
            buf: array N of Job;

        procedure Get (j: Job);         (* the object's methods *)
        begin …
        end Get;

        procedure { public } Put (j: Job);
        begin …
        end Put;

        activity Process;               (* activity declaration within the object *)
            var …                       (* declare state space of the activity *)
        begin
            …
        end Process;

        var p: Process; (* reference variable for an activity instance *)

begin
        n := 0; in := 0; out := 0;
        p := new Process                (* create an activity instance in the instance of  the Cell object *)
end Cell;
```

9.1.1 Initialising Activity Variables

It is possible to send a sequence of arguments to an activity when it is instantiated in order to initialise its state space (variables). The arguments are passed as parameters to the instantiator **new**. The types of the arguments are checked in the same way as for a procedure. For example:

```
activity A( a, b, c: integer );
begin
  …
end A;

var a: A;
  …
  a := new A( 10, 10, 10 );
  …
```

The result of the example above is equivalent to:

```
activity A;
  var a, b, c: integer;
begin
  accept a, b, c;
end A;
```

9.1.2 Activity Termination

Once an activity has been spawned there is no facility within the language to pre-emptively terminate it, i.e. there is no concept of ‘assassination’. It is possible that some Zonnon implementations might offer such a facility in the form of an additional ‘kill’ procedure in the runtime system [Compiler].

Note that an object containing activities may only terminate when there are no longer any references to it, *and* when all of its activities have terminated. An activity terminates after the execution of the statement immediately preceding the **end** of its body.

9.2 Protocol Controlled Activities

An activity can spawn another activity in the same, or a different, object or module. When this occurs the two activities can share a formal *protocol* which governs the dialog of interaction between them defined in the form of an EBNF syntax. Of course it is possible for a protocol to be null, in which case there is no interaction specified. See also 5.3.4 Protocol types. A complete example of the use of activities is presented in section 16 .

Such activities are created and interact in the following way:

1. A protocol type is declared which defines the valid sequence of interaction between the activities in the form of an EBNF syntax.
2. The first activity is instantiated within a module or object, it is referred to as the *client*.
3. The client activity then instantiates the second activity in a module or object, it is referred to as the *server*. The client and the server both implement (share) the same protocol. The client activity is always anonymous from the server's viewpoint. The client references its server using an variable of the activity's type.
4. The two activities can now interact according to the EBNF syntax defined in the protocol they share. This is achieved by using the client *sending* and *receiving* parameters to and from its server; in turn the server accepts parameters from its client and returns parameters back to it.
5. The *is* operator can be used to discriminate between different types of syntactic tokens (see 6.2.4). The allowed types are integer, real, string, keywords (i.e. the enumerated tokens of the protocol) and special characters.
6. The dialog is only ended when *either* of the two activities has terminated; either by reaching the **end** of its **begin ... end** body, or terminating due to trap to the system (catastrophic error).

A protocol controlled activity supports explicit synchronisation and communication with its client activity; it may be declared in the scope of a different object or module. An activity may spawn another activity in either the same or another object or module hence gaining access to its scope via parameters passed according to the protocol. Here is an example of a simple protocol and a server activity:

```
protocol P = (a, b, c, P = a | b | c);
activity A implements P;
  var p: P;
  begin
        accept p; (* receives the token sent by the client activity *)
        if p = P.a then
            (* token a *)
        elsif p = P.b then
            (* token b *)
        else
            (* token c *)
        end
  end A;
```

A complete example of the use of activities is presented in section 16 .

9.2.1 Sending and Receiving Tokens with an Activity

An activity can be created instantiated from a module, object, procedure or activity body. It is possible to send tokens (parameters) to it and receive tokens back from it. Note that the compiler does not check the types of the parameters. The format for sending tokens resembles a procedure call. The send operation is non-blocking, that is, execution continues immediately with the next statement.

Example:

```
activity A; (* declaration of the activity type *)
begin
        …
end A;
…
(* Now in the module, object or activity that is going to create the instance for the client … *)
var a: A; s: string;
begin
   a := new A;    (* Create the instance of the activity and store its reference in a variable *)
   a ( 1024 );    (* Send an integer constant via the activity reference *)
   a ( "Hello");  (* Send a string constant *)
   a ( "My name is ", s, 17 ); (* Send a string constant, a reference to the string object and an integer constant *)
end;
```

Tokens that are then returned by the activity are received by the original sender. The received token values are implicitly converted to destination variable type if necessary. If a type conversion error occurs the standard system exception *conversion* will be thrown, see 7.10.1. The format for receiving

tokens resembles a function procedure call. The receive operation is blocking, that is, execution only continues with the next statement when all the tokens involved in the statement have been received.

Example:

```
(* Now following on from the previous example and looking from the original sender's viewpoint … *)
var a: A; s, b: string; i, d: integer; c: object;
  …
  (* receive a stream of tokens from the activity … *)
  i := a ( );          (* receive token and convert to integer *)
  s := a ( );          (* receive token and convert to a string *)
  b, c, d := a ( );    (* receive three tokens: the literal string, a reference to a string and an integer *)
  ...
```

It is also possible to combine the sending and receiving of tokens into a single statement. Note that in this case the statement effectively blocks the sender until all the expected returned tokens have arrived and have been assigned to their variables.

Example:

```
activity A; (* declaration of the activity type *)
begin
        …
end A;

(* Now in the module, object or activity that is going to create the instance for the client … *)
var a: A; s, b: string; i, d: integer; c: object;

begin
  a := new A;   (* Create the instance of the activity and store its reference in a variable *)
  …
  (* now send and receive tokens *)
  i := a ( "Hello");          (* send "Hello" and receive integer token in response *)
  s := a ( );                 (* receive token and convert to a string *)
  b, c, d := a ( 1, "a" );    (* send constants 1  and "a" then receive three tokens:
                                 the literal string, a reference to a string and an integer *)
```

9.2.2 Sending and Receiving Tokens in Server Activities

When a client activity creates a server activity it can then interact with it using tokens according to the EBNF dialog defined in its protocol (type). So within the server activity the *accept* statement is used to receive tokens and the *return* statement is used to send tokens back to the client activity. Accepted tokens undergo any necessary type conversion implicitly. The accept statement is blocking and the return statement is non-blocking.

Examples:

```
activity A; (* instantiated in the role of a server *)
  var s: string; b,c,d: integer;
begin
                    (* an activity receieving tokens from its client *)
  accept s;         (* accept a token and if necessary convert to string *)
  accept b, c, d;   (* accept three tokens and if necessary convert to integer *)
  …
end A;
```

and for a server activity returning tokens to its client:

```
activity A;
  var s: string; b,c,d: integer;
begin
  return s;         (* Send a string *)
  return b, c, d;   (* Send three integers *)
end A;
```

9.2.3 Using **is** operator to check token type

When receiving or accepting tokens the type of the token can be checked at run time by using the *is* operator. This can help to differentiate between types and also to avoid exceptions due to type mismatches. For example:

```
protocol P = (START_TEXT, MODIFIER1, MODIFIER2, END_TEXT,
                    P = START_TEXT { string | MODIFIER1 | MODIFIER2 } END_TEXT);

activity A implements P; (* the client and server activities both are associated with this type P *)
    var request: object;

    procedure processCmd( cmd: P );
    end processCmd;

    procedure addTextLine( s: string );
    end addTextLine;

begin (* transfer text made up of strings and modifiers, see the protocol type for its structure (syntax)  *)
    repeat
        accept request;             (* receive the token *)
        if request is P then        (* check if it is a protocol token from the enumeration in type P *)
            processCmd( P( request ) )
        else                        (* the token must of type string *)
            addTextLine( string( request ) )
        end
    until (request is P) & (request = P.END_TEXT);
end A;
```

9.3 Barrier Controlled Activities

An arbitrary number of child ‘activities’ may be created from within the ‘parent’ scope of an object, module, procedure, activity or *do…end* body in combination with the modifier {*barrier*}. At the statement level this also applies to any statement sequence marked with a leading *barrier* modifier. Object (or module) bodies always act as the root of a hierarchy. The end of the ‘parent’ scope by definition takes the role an execution ‘barrier’ that must not be crossed until all ‘child’ activities have terminated.

An independent activity needs to be declared as an (arbitrarily nested) procedure with no result that has the advantage of enabling access to data within the static link of the procedure-activation stack. An activity of this type may then be instantiated by any parent within its scope of visibility by using the *new* operator followed by the procedure name and a matching list of actual parameters.

Example 1: Declaration and instantiation of independent activities within object scope

```
object X;
    var x: array N of X; s0, s1: S;
    activity Q (s: S);
        var t: T;
    begin ... (* has access to x *)
    end Q;
begin { barrier } ...
    new Q(s0); new Q(s1);
  ...
end X; (* acts as barrier *)
```

Example 2: Independent activities within procedural scope

```
procedure P (...);
    var x: array N of X; s0, s1: S;
    activity Q (s: S);          (* local declaration *)
        var ...
    begin ...                   (* activity body will have access to x *)
    end Q;
begin { barrier }  ...
    new Q(s0); new Q(s1); (* instantiate two activities in procedure body *)
    ...
end P; (* acts as barrier: both activies must terminate before it is crossed *)
```

Example 3: Independent activities within wrapping barrier

```
do { barrier }
    get (s0, s1);           (* Snapshot values from enclosing scope *)
    while ... do            (* instantiate pairs of activities *)
        new Q(s0); new Q(s1);
    get (s0, s1)
    end
end (* acts as barrier: all activies must terminate before it is crossed *)
```

Example 4: Independent activities within inner barrier

```
get (s0, s1);                (* Snapshot values from enclosing scope *)
repeat
    do { barrier }
        new Q(s0); new Q(s1);
        get (s0, s1)
    end (* acts as barrier, all activies must terminate before it is crossed *)
until …
```

9.4 Protected Objects and Modules

In concurrent systems there is the possibility for multiple activities to have read and write access to an object or module's state space. This access sometime needs to be managed to avoid data corruption e.g. an activity overwriting data written by another activity before the consumer of the data has had chance to read it. Zonnon provides two levels of access protection: the object level and at the method level.

9.4.1 Object-level Protection

Objects (and modules) may optionally specify a {*protected*} modifier. This declares a hidden shareable lock associated with the instance of each object. For an activity to access the object, i.e. call a method, as a precondition it must 'own' the object-level lock. This can only be achieved by entering a protected object when it is unoccupied by any other activities. On acquiring the lock the activity can use the **await** statement to wait for any condition local to the object, in which case the following sequence happens:

1. The system 'suspends' the execution of the activity and *releases the lock*. This allows other activities a chance to establish their own **await** conditions.
2. The system monitors the awaited condition or our activity.

When the awaited condition is established and the lock is releasable the system resumes the execution of the waiting activity.

9.4.2 Method-level Protection

A finer granularity of protection is also available at the method level. Any method within a protected object can be 'shared' by marking its declaration with the {*shared*} modifier at the beginning of the method body. When running a block of shared statements within an object only a 'share' of the total lock can be owned by the accessing activity, and any number of activities may get their own share. However, statements within such a shared statement sequence must not modify (write to) the object's data i.e. method-level shared accesses must be 'read-only'. This provides for efficient access to read object data by many activities concurrently without the overhead of each on in turn having to own the object-level lock.

10 Program Units

A Zonnon program may be textually partitioned into units, each of which can be compiled separately. The units are: the module, the definition, the implementation, and the object. It is possible to textually nest some of these units; the rules governing this are in section 2; note that a unit declaration cannot be nested within itself.

```
CompilationUnit = { ProgramUnit "." }.
ProgramUnit = ( Module | Definition | Implementation | Object ).
```

Within a program the module and object program units are implementers of functionality, whereas the definition and implementation units are implementees.

10.1 The Module

A *module* has a dual nature, it declares a syntactic container for logically cohesive program declarations and it simultaneously declares an object which is managed by the system. So the module provides the mechanism for the textual partitioning of a source program and also the dynamic loading at execution time of a part of a program, in the form of an instantiated object.

Any number of dynamically created objects may have their lifecycles managed by a program, however only a single instance of each module's object may be instantiated by the system at any given

time. For this reason the module is also ideal for implementing abstract data types. Nesting of module declarations is not allowed.

```
Module = module [ Modifiers ] ModuleName [ ImplementationClause ] ";"
            [ ImportDeclaration ]
            ModuleDeclarations
            ( UnitBody | end ) SimpleName.

Modifiers = "{" IdentList "}".

ModuleDeclarations =    { SimpleDeclaration | NestedUnit ";" |
                          ProcedureDeclaration | OperatorDeclaration
                          ProtocolDeclaration | ActivityDeclaration }.

NestedUnit = ( Definition | Implementation).

ImplementationClause = implements ImplementedDefinitionName { ","
   ImplementedDefinitionName }.

ImplementedDefinitionName = DefinitionName | "[" "]".

ImportDeclaration = import Import { "," Import } ";".

Import = ImportedName [ as ident ].

ImportedName = ( ModuleName
             |  DefinitionName
             |  ImplementationName
             |  NamespaceName
             |  ObjectName ).

UnitBody = begin [ StatementSequence ] end.
```

Each *module* has a unique name and constitutes a text that may be separately compiled as a unit. Optionally a *module* may *implement* one or more *definitions*. (See section 2). In this case the distinct facets of the object are defined separately in *definition* units which provide an abstract interface. A *module* may optionally *import* elements from one or more other *implementations*, that is, gain access to their scope and make possible the aggregation of their content. By using the *as* clause it is also possible to rename all entities as they are imported. This can be used to avoid name clashes and/or to simplify long external names to promote program readability.

Example:

```
import System.Console as S;
…
S.WriteLine('Hello'); (* equivalent to System.Console.WriteLine('Hello') *)
```

A module may optionally contain

- Other textual units i.e. *definitions*, *implementations* and objects
- Simple declarations of constants, types, variables, and procedures
- Operator declarations, for defining user defined operators
- Activity declarations, for defining activities within the *module* on instantiation

Examples:

```
module Small;
begin
      write ('Hello World')
end Small.

module BodyMassIndex; (* calculate body mass index *)
      var height, weight, bmi: real;
begin
      write('weight in kg? '); read(weight);
      write('height in m? '); read(height);
      bmi := weight / (height * height);
      write(' body mass index is', bmi : 6: 2);
      if bmi < 19.0 then
          write('too thin')
      elsif bmi < 27.0 then
          write('OK')
      else
          write('too fat')
      end
end BodyMassIndex.
```

```
definition D; …end D.

definition E; …end E.

module M;
import D, E;
      var a: object{D, E};  (* object is one that implements both D and E *)
      …
end M.
```

10.2 The Object

An object type can be separately compiled and is composed of its local declarations (const, type, var, procedures and activities) and its body. Nesting of object declarations is not allowed. An object is an instance of such a type. When the object is instantiated using **new** the statements in the object body are run on the objects' new thread of execution. The functionality of the object can be accessed by other modules, objects and activities by calling its method procedures. It can also be accessed by activities external to it instantiating activities within it and then interacting with them; in this way they gain controlled access to the objects' scope. Access can also be achieved by the use of indexers; see 7.1.1. A record is a special form of object which has no body.

```
Object =  object [ Modifiers ] ObjectName ObjectDefinition SimpleName.

ObjectDefinition = [ FormalParameters ] [ ImplementationClause ] ";"
                      [ ImportDeclaration ]
                     { SimpleDeclaration | ProcedureDeclaration |
                        ProtocolDeclaration | ActivityDeclaration }
                     ( UnitBody | end ).

Record = record ObjectName { VariableDeclaration ";" } end SimpleName.
```

In order to avoid the notion of free-floating types the standalone object declaration is equivalent to the type being declared in an anonymous module. So the declaration of the form:

```
object T;
…
end T.
```

is shorthand for:

```
module ;
  type T = object … end;
end .
```

The use of this form is restricted to the 'topmost' level of declarations.

Optionally an object may *implement* one or more *definitions*. (See section 2). In this case the distinct facets of the object are defined separately in *definition* units which provide an abstract interface. Also an object may *import* elements from a *module* or *implementation*; that is, gain access to their scope. By using the *as* clause it is also possible to rename all entities as they are imported. This can be used to avoid name clashes and/or to simplify long external names to promote readability of the programming within the object.

Note that an *object* importing a *definition D* to make use of the *implementation D* must explicitly aggregate it by importing *D*, see sections 10.2.3 and 10.4.

10.2.1 Inheritance and Multiple Inheritance

There are two kinds of inheritance supported in Zonnon: refinement and aggregation. Refinement is the inheritance of an interface definition whilst aggregation is the inheritance (reuse) of (fragments of) an existing implementation. All object declarations that do not explicitly refine some other object are deemed to refine *object*. Thus all objects (directly or indirectly) refine *object*. If an object *B* refines an object *A*, then *B* is said to be 'derived from' *A*; for further details see section 10.2.3.

Multiple inheritance is characterized by the possibility to refine from multiple definitions and/or to aggregate from multiple implementations. In Zonnon there is no ambiguity associated with multiple inheritance, due to the use of qualified identifiers for naming (see 5.1).

10.2.2 Polymorphism

Polymorphism involves the selection of the appropriate method to invoke at execution time, depending on the type of the variable that it is to be acted upon. There are two concepts:

1) an object of type *T* is required here, and

2) an object is required here that implements an interface definition *D*

Zonnon emphasizes the second more general concept (2 above) and goes further by allowing the specification of multiple definitions (so called 'facets' of the object's overall interface) and so in this context polymorphism means 'an object is required here that implements *D1* and *D2* and …'.

10.2.3 Activities

Activities may be declared as types within objects. They may also be instantiated within the object itself or by another activity either within the same object or by an activity in another object. See section 9 for details of activities and objects.

10.3 The Definition

A *definition* defines a distinct facet of an object in terms of an abstract interface comprising field declarations and method signatures (but not method bodies). Definitions can form a network of related types, not just a hierarchy. The dependencies between definitions may not be cyclic.

```
Definition = definition [ Modifiers ] DefinitionName [ RefinementClause ] ";"
                [ ImportDeclaration ]
                DefinitionDeclarations
             end SimpleName.

RefinementClause = refines DefinitionName.

DefinitionDeclarations   = { SimpleDeclaration
                           | { ProcedureHeading ";" }
                           | ProtocolDeclaration }.
```

A *definition* has a unique name and optionally *refines* another *definition*, presenting a new facet of an object, possibly adding new fields and behavior and thus forming a specialized form of the original definition.

It may also optionally *import* elements from one or more *implementations*, that is gain access to their scope and make possible the literal aggregation of their content. By using the *as* clause it is also possible to rename all entities as they are imported. This can be used to avoid name clashes and/or to simplify long external names to promote readability of the programming within the object. The modifiers *public* and *private* can be used to declare the visibility of the contents of a definition. If no modifier is present then the default is *public*. The *definition* can contain a set of declarations of constant, types and variables and also method procedure headings (signatures), but not the bodies of procedures.

Examples:

```
definition Graphical; (* features of all graphical objects *)
    var x, y: integer; (* object's position *)

    procedure MoveTo (newX, newY: integer);
    (* post: (x = newX) & (y = newY) *)

    procedure MoveBy (dx, dy: integer);

    procedure Draw;
end Graphical.

definition Rectangle refines Graphical; (* features specific to rectangle objects *)
    var width, height: integer;

    procedure Area ( ): integer;
end Rectangle.

implementation Graphical; (* see example in section 10.4 *)

    …
end Graphical.
```

```
object {ref} Box implements Rectangle;
    procedure Area ( ): integer;
    begin
        return width * height
    end Area;
end Box.
```

10.4 The Implementation

An *implementation* defines an aggregate of field and method implementation fragments intended for re-use when incorporated into a program via one or more object templates. An *implementation* has a unique name unless it has the same name as its corresponding *definition*. It may optionally *import* elements from one or more other *implementation*s, that is, gain access to their scope and make possible the aggregation of their content. By using the *as* clause it is also possible to rename all entities as they are imported. This can be used to avoid name clashes and/or to simplify long external names to promote readability of the programming within the object. Nesting of implementation declarations is not allowed.

The modifiers *public* and *private* can be used to declare the visibility of the contents of an implementation. If no modifier is present then the default is *public*.

An *object* implementing a *definition* is required to implement *all* of its methods unless the *definition* has a corresponding implementation which is imported to the object.

```
Implementation =    implementation [ Modifiers ] ImplementationName ";"
                    [ ImportDeclaration ]
                    Declarations
                    ( UnitBody | end ) SimpleName.

UnitBody = begin [ StatementSequence ] end.
```

The *implementation* can contain a set of declarations of constants, types and variables and also method procedure headings and bodies. These bodies ultimately form the concrete implementations of the methods of objects.

Type extension, as introduced in Oberon [Oberon] can easily be emulated. If *Tf* is a type consisting only of fields then a new type *T* can “inherit” the fields of *Tf* by implementing it. Examples:

```
implementation Graphical; (* an implementation of the definition Graphical *)
    (* X and Y are declared in the definition, see section 10.3 above *)
    procedure MoveTo (newX, newY: integer);
    begin
        x := newX; y := newY
    end MoveTo;

    procedure MoveBy (dx, dy: integer);
    begin
        x := x + dx; y := y + dy
    end MoveBy;

end Graphical.
```

Before an object is instantiated each implementation that it implements is itself aggregated as a whole and initialized. Furthermore it is possible for one implementation unit to implement part of another implementation unit. This makes it possible to encapsulate program fragments a hierarchy of implementations, for instance to enable their reuse.

11 Reflection

It is sometimes desirable to access information about the constructs and their attributes (e.g. modifiers) of a Zonnon source program. To make this possible the compiler can produce an *XML* definition of the salient features of each separately compiled item of source text. This can later be accessed by a run-time program using the predefined procedure *getAttribute*. The *construct* parameter is the name of any Zonnon entity, including program units, types, constants, variables, objects, procedures, parameters, blocks and operators.

The attribute values may be accessed using two forms of *getAttribute*:

```
getAttribute(construct, var string);
```

or

```
string := getAttribute (construct);
```

The information is returned in a single string, possibly containing several attribute values.

11.1 XML Schema

The following list defines the XML schema used to describe the information reflected from the program with typical examples provided in each case:

11.1.1 Access Rights

```
<access>public</access>
<access>private</access>
```

11.1.2 Objects

```
<object>ref</object>
<object>value</object>
```

11.1.3 Procedure Parameters (parameter passing mode):

```
<parameter>var</parameter>
<parameter>value</parameter>
```

11.1.4 Procedure and Variable Immutability:

```
<immutable>open</immutable>
<immutable>sealed</immutable>
```

11.1.5 Operator Priority

```
<priority>3</priority>
```

11.1.6 Blocks and Procedure Bodies

```
<behavior>passive</behavior> //neither locked nor concurrent
<behavior>locked</behavior>
<behavior>concurrent</behavior>
```

11.1.7 Type, Variable and Constant Widths

```
<width>64</width>
```

11.1.8 Enumeration Cardinality

```
<ordinal>7</ordinal>
```

11.2 Example: program reflection and information

When the following program runs

```
definition d;
    procedure p1 (var x: integer {32});
    procedure p2 { sealed };
    var v: integer {64};
    type T = ( one, two, three );
end d.

object o implements d;
    procedure p1 (var x: integer {32}) implements d.p1;
    var attrs1, attrs2, attrs3, attrs4, attrs5, attrs6: string;
    begin { locked }
        attrs1 := getAttribute(d);
        attrs2 := getAttribute(d.v);
        attrs3 := getAttribute(p1.x);
        attrs4 := getAttribute(d.T);
        attrs5 := getAttribute(p1);
        attrs6 := getAttribute(d.p2);
```

```
    end p1;
begin
end o.
```

it reveals its form via the reflection information as follows:

```
attrs1(d) contains:
        "<attributes> <access>public</access> </attributes>"

attrs2(d.v) contains:
"<attributes> <access>public</access> <implement>open</implement>
<width>64</width> </attributes>"

attrs3(p1.x) contains:
        "<attributes> <parameter>var</parameter> <width>32</width> </attributes>"

attrs4(d.T) contains:

        "<attributes> <access>public</access> <width>32</width> <ordinal>3</ordinal>
        </attributes>"

attrs5(p1) contains:
        "<attributes> <access>public</access> <implement>sealed</implement>
        <behavior>locked</behavior> </attributes>"

attrs6(d.p2) contains:
        "<attributes> <access>public</access> <implement>sealed</implement>
        <behavior>passive</behavior> </attributes>"
```

12 Definition of Terminology

12.1 Numeric Types

The numeric types are:

- Integer types integer or integer{width}
- Cardinal types cardinal or cardinal{width}
- Real types real or real{width}

12.2 Same Types

Two variables *a* and *b* with types *Ta* and *Tb* are of the *same* type if

- *Ta* and *Tb* are both denoted by the same type identifier, or
- *Ta* is declared to equal *Tb* in a type declaration of the form *Ta = Tb*, or
- *a* and *b* appear in the same identifier list in a variable, object field, or formal parameter declaration and are not open arrays.

12.3 Equal Types

Two types *Ta* and *Tb* are *equal* if

- *Ta* and *Tb* are the same type, or
- *Ta* and *Tb* are open array types with equal element types, or
- *Ta* and *Tb* are procedure types whose formal parameter lists match.

12.4 Assignment Compatible

An expression e of type *Te* is assignment compatible with a variable *v* of type *Tv* if one of the following conditions hold:

- *Te* and *Tv* are the same type;
- Within each of the type families
  integer, cardinal, real, set, char
  an expression of type *Te* may be assigned to a variable *v* whose type *Tv* is large enough (defined by its width) to hold the set of values of type *Te;*
- *Tv* is a procedure type and *e* is *nil;*

- *Tv* is a procedure type and *e* is the name of a procedure whose formal parameters match the signature of *Tv*

12.5 Array Compatible

An actual parameter a of type *Ta* is array compatible with a formal parameter *f* of type *Tf* if

- *Tf* and *Ta* are the same type, or
- *Tf* is an open array, *Ta* is any array, and their element types are array compatible

12.6 Compatible for Expressions and Operator Overloading

For a given operator, the types of its operands are expression compatible if they conform to the following table (which shows also the result type of the expression), for example: *op1* > *op2*. The table also implicitly defines the sets of operand combinations that are supported for operator overloading.

Operator	*First operand (op1)*	*Second operand (op2)*	*Result type*
+ – * **	integer{m}	integer{n}	max of integer{m} and integer{n}
+ – * **	cardinal{m}	cardinal{n}	max of cardinal{m} and cardinal{n}
+ – * **	real{m}	real{n}	max of real{m} and real{n}
/	real{m}	real{n} pre: op2 # 0	max of real{m} and real{n}
+ – *	set{m}	set{n}	max of set{m} and set{n}
div mod	integer{m}	integer{n} pre: op2 # 0	max of integer{m} and integer{n}
or & ~	boolean	boolean	boolean
= # < <= > >=	integer{m}	integer{n}	boolean
= # < <= > >=	cardinal{m}	cardinal{n}	boolean
= # < <= > >=	real{m}	real{n}	boolean
= # < <= > >=	enumeration T	enumeration T	boolean
= # < <= > >=	char	char	boolean
= # < <= > >=	character array,	character array	boolean
= # < <= > >=	string	string	boolean
= #	boolean	boolean	boolean
= #	set	set	boolean
= #	procedure type T	procedure type T	boolean
= #	**nil**	**nil**	boolean
in	integer	set	boolean
implements	object	definition	boolean
is	object	object type	boolean

12.7 Matching Formal Parameter Lists

Two formal parameter lists match if

- they have the same number of parameters, and
- they have either the same function result type or none, and
- parameters at corresponding positions have equal types, and
- parameters at corresponding positions are both either value or variable parameters.

13 Predefined Procedures

The following table lists the predefined procedures. Some are generic procedures, i.e. they apply to several types of operands. Within the specifications *v* stands for a variable, *x* and *n* for expressions, and *T* for a type. The names of the predefined procedures can also be written entirely in upper-case letters.

Name	**Argument(s) type(s)**	**Result type**	**Purpose**
abs(x)	integer, cardinal or real	type of x	absolute value of *x*
assert(b)	b: boolean	none	if ~b terminate
assert(b, n)	b: boolean; n: integer or cardinal	none	if ~b terminate, report n to environment
cap(x)	x: char	char	corresponding capital letter precondition: x is a letter
copy(x, v)	x: string; v: character array	none	v := x
copy(v, x)	x: string; v: character array	none	x := v
copyvalue(v)	v: ref object	value object	dereference an object
dec(v)	v: integer, cardinal or enumeration type	none	v := v – 1
dec(v, n)	v: integer, cardinal or enumeration type n: integer or cardinal type	none	v := v – n
excl(v, x)	v: set; x: integer or cardinal type	none	v := v – {x}
halt(n)	n: integer or cardinal const	none	terminate program execution
inc(v)	v: integer, cardinal or enumeration	none	v := v + 1
inc(v, n)	v: integer, cardinal or enumeration n: integer or cardinal type	none	v := v + n
incl(v, x)	v: set; x: integer or cardinal type	none	v := v + {x}
len(v, n)	v:array; n: integer or cardinal const	integer	length of v in dimension n (first dimension = 0)
len(v)	v: array	integer	equivalent to len(v, 0)
len(v)	v: string	integer	number of characters in string v (see 5.3.6)
low(x)	x: char	char	corresponding lower-case letter precondition: x is a letter
max(T)	integer	integer	maximum value of type integer{w}
max(T)	cardinal	cardinal	maximum value of type cardinal{w}
max(T)	enumeration	enumeration	maximum value of the enumeration
max(T)	char{w}	integer	maximum character
max(T)	real{w}	real	maximum value of type real{w}
max(T)	set{w}	integer	maximum element of a set{w}
min(T)	integer	integer	mininmum value of type integer{w}
min(T)	enumeration	enumeration	minimum value of the enumeration
min(T)	char{w}	integer	minimum character
min(T)	real{w}	real	minimum value of type real{w}
min(T)	set{w}	integer	0
odd(x)	x: integer	boolean	x **mod** 2 = 1
pred(x)	x: integer	integer	x – 1, pre: x # min(integer)
pred(x)	x: enumeration	type of x	predecessor enumeration value, pre: x # min(enumeration)
pred(x)	x: char	char	predecessor char, pre: x # min(char)
reason	none	string	returns system information on current exception
size(T)	any type	integer	number of bytes required by T
succ(x)	x: integer or cardinal	integer	x + 1, pre: x # max(integer)
succ(x)	x: enumeration	type of x	successor enumeration value, pre: x # max(enumeration)
succ(x)	x: char	char	successor char, pre: x not max(char)

In *assert(x, n)* and *halt(n)*, the interpretation of *n* is implementation specific. (See [Compiler]).

14 Input and Output Procedures

The language includes built-in features for simple textual input and output. Conceptually, reading and writing corresponds to receiving and sending tokens from and to the predefined activities *standard input* and *standard output* respectively.

For convenience, predefined procedures in a similar style to Pascal are provided for reading and writing text. The procedures for inputting text are *read* and *readln* and for outputting are *write* and

writeln. All input and output is to texts which are implicitly assumed to be represented as lines of characters delimited by end of line markers.

14.1 Parameters and special syntax

The procedures are used with a non-standard syntax for their parameter lists. This allows for a variable number of parameters which may be of various data types. Parameters of type *char* require no data type conversion, however for other types such as integer, real, etc the data transfer includes an implicit data type conversion.

14.2 Input Procedures

14.2.1 The *read* procedure

The form of the *read* procedure is

```
read (v1, …, vn)
```

It may have one or more parameters, each of which is a value of some basic data type. If *v* is a value of type *char* then *read(v)* transfers the next character from the input text to *v*. If *v* is a value of type *integer*, *cardinal* or *real* then *read(v)* implies the reading of a sequence of characters from the input text and assignment of that number to *v*. Preceding blanks and line markers are skipped and discarded.

14.2.2 The *readln* procedure

The form of the *readln* procedure is

```
readln(v1, …, vn)
```

readln has the same functionality as *read* except that after reading *vn* all remaining characters on the line are skipped up to and including the next end of line marker.

14.3 Output Procedures

14.3.1 The write procedure

The form of the *write* procedure is

```
write (p1, …, pn)
```

It may have one or more parameters, each of which has the form

```
e : e:m  or  e:m:n
```

Where *e* represents the value to be output and *m* and *n* are field-width specifiers. If the value of *e* requires less than *m* characters for its representation then blanks (spaces) are output to ensure that a total of exactly *m* characters are written. If *m* is omitted an implementation-defined default value will be assumed. The form *e:m:n* is only applicable to numbers of type *real*. (See below).

The *write* procedure parameters can be of type *char*, *string*, *boolean*, *integer*, *cardinal* and *real*.

- If *e* is of type *char* then *write (e : m)* writes out *m* – 1 spaces followed by the character contained in *e*. If *m* is omitted then only the character is written.
- If *e* is of type string then *write (e : m)* writes the characters of the string, preceded by blanks to ensure a total field width of *m*.
- If *e* is of type *boolean* then either the word *true* or *false* is written, preceded by blanks to ensure a total field width of *m*.
- If *e* is of type *integer* or *cardinal* then the decimal representation of the number *e* will be written, preceded by blanks to ensure a total width of *m*.
- If *e* is of type *real* then the decimal representation of the number *e* will be written, preceded by blanks to ensure a total width of *m*. If the parameter *n* is missing a floating point representation consisting of a coefficient and a scale factor will be written. If *n* is present then a fixed-point representation with *n* digits after the decimal point is provided.

14.3.2 The *writeln* procedure

The form of the *writeln* procedure is:

```
writeln (v1, …, vn)
```

writeln has the same functionality as *write* except that after writing *vn* an end of line marker is written.

14.3.3 Default values of widths in *write* and *writeln*

The default field width for *write* and *writeln* procedure parameters depends on the type of the parameter, the default widths are:

- *char* default field width 1
- *string* default field width is the length of the string
- *boolean* default field width is 6
- *integer* default field width is 20
- *cardinal* default field width is 20
- *real* default field width is 20

15 Example Module Strings

This is an example of a library module which builds on the minimal string features built into Zonnon. See section 5.3.6. In this example 'zonnon' is the namespace, the standard implementation of Strings is specified by the *definition* zonnon.Strings and its new implementation is implemented in the *module* zonnon.NativeStrings .

15.1 Zonnon Strings definition

```
definition {public} zonnon.Strings;
 (*The definition of standard Zonnon strings *)

    (* length: return the length of the given string this *)
    procedure length  (this : string) : integer;

    (* substring: returns the substring of  the given string this from position start and of length count  *)
    procedure substring (this : string; start, count : integer) : string;

    (* insert: returns the string with the whole string this inserted into the string s starting at position start  *)
    procedure insert  (this : string; start : integer; s : string) : string;

    (* remove:  returns the string with count characters removed from the string this  at position start *)
    procedure remove  (this : string; start, count : integer) : string;

    (* replace: returns the string with with all fromc characters replaced by toc characters in string this *)
    procedure replace  (this : string; fromc, toc : char) : string;

    (* startsWith: returns true is the string this is the initial substring of string s , otherwise false *)
    procedure startsWith (this : string; s : string) : boolean;

    (* endsWith: returns true is the string this is the final substring of string s , otherwise false *)
    procedure endsWith (this : string; s : string) : boolean;

    (* indexOf: returns the first located position of character c in string this *)
    procedure indexOf  (this : string; c : char) : integer;

    (* indexOf: returns the last located position of character c in string this *)
    procedure lastIndexOf (this : string; c : char) : integer;

    (* toUpper: returns the string of string this with all characters converted to upper case *)
    procedure toUpper  (this : string) : string;

    (* toLower: returns the string of string this with all characters converted to lower case *)
    procedure toLower  (this : string) : string;

end Strings.
```

15.2 Zonnon Strings implementation by native Zonnon character arrays

```
(*The implementation of standard Zonnon strings definition. Module NativeStrings *)
(* implements standard Zonnon strings definition by character arrays of Zonnon language *)
(* VR October 2005 *)

module {public} zonnon.NativeStrings implements zonnon.Strings;

    type {private} CharArray = array * of char;

    (* length: return the length of the given string this *)
    procedure {public} length(this : string) : integer implements zonnon.Strings.length;
    begin
        return len(this)
    end len;

    (* substring: returns the substring of  the given string this from position start and of length count  *)
    procedure {public} substring(this : string; start, count : integer) : string implements zonnon.Strings.substring;
    var k : integer;
        result : string;
        source, target : CharArray;
    begin
        source := new CharArray( len(this) );
        target := new CharArray(count);
        copy(this, source);
        for k := 0 to count-1 do
            target[k] := source[start + k]
        end;
        copy(target, result);
        return result
    end substring;

(* insert: returns the string with the whole string this inserted into the string s starting at position start *)
procedure {public} insert(this : string; start : integer; inserted : string) : string implements zonnon.Strings.insert;
    var k, i : integer;
        result : string;
        source, target : CharArray;
        sourceL, insertL : integer;
begin
    sourceL := len(this);
    insertL := len(inserted);
    source := new CharArray(sourceL);
    target := new CharArray(sourceL + insertL);
    copy(this, source);
    i := 0;
    for k := 0  to start-1 do target[i] := source[k]; inc(i) end;
    for k := 0  to insertL-1 do target[i] := inserted[k]; inc(i) end;
    for k := start to sourceL-1 do target[i] :=  source[k]; inc(i) end;
    copy(target, result);
    return result
end insert;

(* remove:  returns the string with count characters removed from the string this  at position start *)
procedure {public} remove(this : string; start, count : integer) : string implements zonnon.Strings.remove;
var result : string;
    source, target : CharArray;
    k, si, ti, N, N1, N2 : integer;
begin
    N := len(this);
    source := new CharArray(N);
    target := new CharArray(N - count);
    copy(this, source);
    N1 := start;
    N2 := N - N1 - count;
    si := 0; ti := 0;
    for k := 1 to N1 do target[ti] := source[si]; inc(si); inc(ti)
    end;
    inc(si, count);
    for k := 1 to N2 do target[ti] := source[si]; inc(si); inc(ti)
    end;
    copy(target, result);
    return result
end remove;
```

```
(* replace: returns the string with with all fromc characters replaced by toc characters in string this *)
procedure {public} replace(this : string; froms, tos : char) : string implements zonnon.Strings.replace;
	var k, n : integer; result : string;
		source, target : CharArray;
begin
	n := len(this);
	source := new CharArray(n);
	target := new CharArray(n);
	copy(this, source);
	for k := 0 to n-1 do
		if source[k] = froms
		then target[k] := tos
		else target[k] := source[k]
		end;
	end;
	copy(target, result);
	return result
end replace;

(* toUpper: returns the string of string this with all characters converted to upper case *)
procedure {public} toUpper(this : string) : string implements zonnon.Strings.toUpper;
	var c : char;
		ia, i, iz, k, n : integer;
		result : string;
		source, target : CharArray;
begin
	n := len(this);
	source := new CharArray(n);
	target := new CharArray(n);
	ia := integer('a');
	iz := integer('z');
	copy(this, source);
	for k := 0 to n-1 do
		c := source[k];
		i := integer(c);
		if (i < ia) or (iz < i)
		then target[k] := c
		else target[k] := char(i-32)
		end
	end;
	copy(target, result);
	return result
end toUpper;

(* toLower: returns the string of string this with all characters converted to lower case *)
procedure {public} toLower(this : string) : string implements zonnon.Strings.toLower;
	var c : char;
	iA, i, iZ, k, n : integer;
	result : string;
	source, target : CharArray;
begin
	n := len(this);
	source := new CharArray(n);
	target := new CharArray(n);
	iA := integer('A');
	iZ := integer('Z');
	copy(this, source);
	for k := 0 to n-1 do
		c := source[k];
		i := integer(c);
		if (i < iA) or (iZ < i)
		then target[k] := c
		else target[k] := char(i+32)
		end
	end;
	copy(target, result);
	return result
end toLower;

(* startsWith: returns true is the string this is the initial substring of string s , otherwise false *)
procedure {public} startsWith(this : string; s : string) : boolean implements zonnon.Strings.startsWith;
	var k, n : integer;
begin
	n := len(s);
```

```
        if len(this) < n then return false
        end;
        for k := 0 to n-1 do
            if s[k] # this[k] then return false
            end
        end;

        return true
    end startsWith;

    (* endsWith: returns true is the string this is the final substring of string s , otherwise false *)
    procedure {public} endsWith(this : string; s : string) : boolean implements zonnon.Strings.endsWith;
        var     k, n, start : integer;
    begin
        n := len(s);
        start := len(this) - n;
        if start < 0 then return false end;
        for k := 0 to n-1 do
            if s[k] # this[start + k] then return false
            end
        end;
        return true
    end endsWith;

    (* indexOf: returns the first located position of character c in string this *)
    procedure {public} indexOf(this : string; c : char) : integer implements zonnon.Strings.indexOf;
        var     k, n : integer;
    begin
        n := len(this)-1;
        for k := 0 to n do
            if this[k] = c then return k
            end
        end;
        return -1
    end indexOf;

    (* indexOf: returns the last located position of character c in string this *)
    procedure {public} lastIndexOf(this : string; c : char) : integer implements zonnon.Strings.lastIndexOf;
        var k, n : integer;
    begin
        n := len(this)-1;
        for k := 0 to n do
            if this[n-k] = c then return n-k
          end
        end;
        return -1
    end indexOf;

end (* of Module *) NativeStrings .
```

16 Example of Protocol Controlled Activities and Dialog

Here is a longer example illustrating how a protocol can be used to control the dialogue between activities. Within the EBNF protocol specifications the communication of an item from the server to the client is prefixed by a '?'. The example implements the well known Simple Mail Transmission Protocol SMTP used for email delivery.

```
definition {public} MailProtocols;

(*  This provides the definition of  the SMTP protocol to control the interaction between the activities*)
protocol SMTP = ( (* defines the tokens used in the EBNF of the dialog *)
            SMTP_SERVER_READY, SERVICE_NOT_AVAILABLE, HELO,
            OK, MAIL_FROM, RCPT_TO,
            RCPT_REJECTED, DATA, START_MAIL_INPUT,
            END, QUIT, BYE,
            msg_body = DATA ?START_MAIL_INPUT { string } END ?OK,
            session = HELO string ?OK  {mail_from rcpt_to {rcpt_to} msg_body},
            se_end = QUIT ?BYE,
            mail_from = MAIL_FROM string ?OK,
            rcpt_to = RCPT_TO string ( ?OK | ?RCPT_REJECTED
            SMPT = ?SMTP_SEVER_READY [session] se_end | ?SERVICE_NOT_AVAILABLE,);
end MailProtcols.

(* SMTP  Mail Server activity which implements the server side of the protocol *)
object MailServer implements MailProtocols;
      activity SendMail implements MailProtocols.SMTP; (* associates the activity with the protocol type *)
           var request: SMTP; any_request: object;
                host, mfrom, mto, mtextline: string;
       begin
           return SMTP.SMTP_SERVER_READY;
           accept request;
           if request = SMTP.HELO then (* If not HELO then request is QUIT *)
               accept host;
               return OK; (* Confirm that client is accepted *)
               (* Sending a message *)
               loop
                   accept request;
                   if request = SMTP.QUIT then exit end;
                   (* Protocol guarantees that request = SMTP.MAIL_FROM *)
                   accept mfrom; (* Sender e-mail *)
                   return OK; (* Confirm that sender is accepted *)

                   (*** New mail - clearing recipients list ***)
                   accept request;
                   while request = SMTP.RCPT_TO do
                       accept mto; (* One more recipient *)
                       if (* checking the recipient is *) true then
                           return SMTP.OK (* Recipient accepted *)
                           (* Adding mto to the rcpt list *)
                       else
                           return SMTP.RCPT_REJECTED (* Recipient rejected *)
                       end;
                       accept request
                   end; (* Request can be SMTP.DATA or SMTP.QUIT *)
                   if request = SMTP.QUIT then exit end;

                   (* Protocol guarantees that request = SMTP.DATA *)
                   return SMTP.START_MAIL_INPUT;
                   accept any_request;
                   while any_request is string do
                       mtextline := string( request ); (* One more text line  *)
                       accept any_request;
                   end; (* while *) (* any_request is SMTP.END *)
                   (* Accept the lines of message until all received *)
                   return OK
                   (* Putting the message to the queue *)
               end (* loop *)
               return SMTP.BYE
           end (* of activity *) SendMail;

end (* of object *) MailServer.
```

```
(* SMTP  MailClient activity which implements the client side of the protocol *)

object {public} MailClient;
        var {public} server: MailServer;

        procedure {public} Configure(server: MailServer)
        begin
            self.server := server;
        end Configure;

        procedure {public} SendMail;
            var smtp: activity { MailProtocols.SMTP };
                answer: MailProtocols.SMTP;
                i: integer;
        begin
            smtp := new server.SendMail; (* new dialog with the server *)
            answer := smtp();
            if  answer = MailProtocols.SMTP.SMTP_SERVER_READY  then
                answer := smtp( MailProtocols.SMTP.HELO, "www.roman.nnov.ru" );
                if answer = MailProtocols.SMTP.OK then
                    for (* each mail in outbox *) i := 1 to 1 do
                        answer := smtp( MailProtocols.SMTP.MAIL_FROM, "roman.mitin@inf.ethz.ch" );
                        if answer = MailProtocols.SMTP.OK then
                            answer := smtp( MailProtocols.SMTP.RCPT_TO, "zueff@inf.ethz.ch" );
                            if answer = MailProtocols.SMTP.OK then
                                answer := smtp( MailProtocols.SMTP.DATA );
                                if answer = MailProtocols.SMTP.START_MAIL_INPUT then
                                    smtp( "Hi!" );
                                    smtp( "It works!" );
                                    answer := smtp( MailProtocols.SMTP.END );
                                    if answer = MailProtocols.SMTP.OK then
                                        answer := smtp( MailProtocols.SMTP.QUIT )
                                        if answer = MailProtocols.SMTP.BYE then
                                        (* Session closed *)
                                        end
                                    (* the following else parts would provide relevant error recovery *)
                                    else (* SMTP.OK missing, out of sequence *)
                                    end
                                else (* No invitation to input data *)
                                end
                            else (* Recipient has been rejected *)
                            end
                        else (* Sender has been rejected *)
                        end
                    end (* for each mail in mailbox *)
                else (* Host has been rejected *)
                end
            else (* Can't connect, server not ready *)
            end
        end SendMail;

        procedure {public} Synchronise;
        begin
            activity; begin SendMail end; (* Run in a new thread *)
            (* Other synchronisation tasks such as GetMail *)
        end Synchronise;

begin (* object MailClient *)
        smtp := nil; (* initialisation code on instantiation *)
end (* of object *) MailClient.

(* This is the loadable module called User that it the initial root of program execution *)
(* It creates the mail client and server objects and then configures then to communicate with each other *)
module User;
        var     server: MailServer;  (* declare variables for referencing each object *)
                client: MailClient;
begin (* these statements are run when the module is loaded into memory and initialised *)
        server := new MailServer;    (* instantiate the MailServer object *)
        client := new MailClient;    (* instantiate the MailClient object *)
        client.Configure( server );  (* call method in client object to link it with the server *)
        client.Synchronise( );       (* call method in client object to initially synchronise the protocol *)
end User .
```

17 Syntax

```
// Zonnon Syntax in EBNF

// 1. Program and program units
CompilationUnit = { ProgramUnit "." }.
ProgramUnit = ( Module | Definition | Implementation | Object).

// 2. Modules
Module =    module [ Modifiers ] ModuleName [ ImplementationClause ] ";"
              [ ImportDeclaration ]
              ModuleDeclarations
              ( UnitBody | end ) SimpleName.
Modifiers = "{" IdentList "}".
ModuleDeclarations =  { SimpleDeclaration | NestedUnit ";" |
                        ProcedureDeclaration | OperatorDeclaration
                        ProtocolDeclaration | ActivityDeclaration }.
NestedUnit = ( Definition | Implementation ).
ImplementationClause = implements ImplementedDefinitionName { ","
ImplementedDefinitionName }.
ImplementedDefinitionName = DefinitionName | "[" "]".
ImportDeclaration = import Import { "," Import } ";".
Import = ImportedName [ as ident ].
ImportedName =  ( ModuleName
                | DefinitionName
                | ImplementationName
                | NamespaceName
                | ObjectName ).
UnitBody = begin [ StatementSequence ] end.

// 3. Definitions
Definition = definition [ Modifiers ] DefinitionName [ RefinementClause ] ";"
               [ ImportDeclaration ]
               DefinitionDeclarations
             end SimpleName.
RefinementClause = refines DefinitionName.
DefinitionDeclarations = { SimpleDeclaration
                           | { ProcedureHeading ";" }
                           | ProtocolDeclaration }.
ProtocolDeclaration = protocol ProtocolName "=" "(" ProtocolSpecification ")" ";".
ProtocolSpecification = [ Alphabet "," ] Grammar
            | Alphabet [ "," Grammar ].
Alphabet = TerminalSymbol { "," TerminalSymbol }.
Grammar = Production { "," Production }.
Production = ProductionName "=" Alternative.
Alternative = ItemSequence { "|" ItemSequence }.
ItemSequence = Item { Item }.
Item = ( ["?"] TerminalSymbol | ProductionName | TypeName |
      Alternative | Group | Optional | Repetition).
Group = "(" ItemSequence ")".
Optional = "[" ItemSequence "]".
Repetition = "{" ItemSequence "}".
TerminalSymbol = number | ident | charConstant.
ProductionName = ident.

// 4. Implementations
Implementation =  implementation [ Modifiers ] ImplementationName ";"
                    [ ImportDeclaration ]
                    Declarations
                    ( UnitBody | end ) SimpleName.
```

```
// 5. Objects
Object =    object [ Modifiers ] ObjectName ObjectDefinition SimpleName.
ObjectDefinition = [ FormalParameters ] [ ImplementationClause ] ";"
                     [ ImportDeclaration ]
                    { SimpleDeclaration | ProcedureDeclaration |
                     ProtocolDeclaration | ActivityDeclaration }
                     ( UnitBody | end ).
ActivityDeclaration = activity ActivityName [ FormalParameters ] [ProcImplementationClause]";"
                           Declarations
                           ( UnitBody | end ) SimpleName.

// 6. Declarations
Declarations = { SimpleDeclaration | ProcedureDeclaration }.
SimpleDeclaration =   ( const [ Modifiers ] { ConstantDeclaration ";" }
                      | type  [ Modifiers ] { TypeDeclaration ";" }
                      | var   [ Modifiers ] { VariableDeclaration ";" }
                      ).
ConstantDeclaration = ident "=" ConstExpression.
ConstExpression = Expression.
TypeDeclaration = ident "=" Type.
VariableDeclaration = IdentList ":" Type.

// 7. Types
Type = (   TypeName [ "{" Width "}" ] | EnumType | ArrayType | ProcedureType
         | InterfaceType | ObjectType | RecordType | ProtocolType ).
Width = ConstExpression.
ArrayType = array Length { "," Length } of Type.
Length = ( ConstExpression | "*" ).
EnumType = "(" IdentList ")".
ProcedureType = procedure [ ProcedureTypeFormals ].
ProcedureTypeFormals = "(" [ PTFSection { ";" PTFSection } ] ")" [ ":" FormalType ].
PTFSection = [ var ] FormalType { "," FormalType }.
FormalType = { array "*" of } ( TypeName | InterfaceType ).
InterfaceType = object [ PostulatedInterface ].
PostulatedInterface =  "{" DefinitionName { "," DefinitionName } "}".
ObjectType = object ObjectDefinition ident.
RecordType = record { VariableDeclaration ";" } end ident.
ProtocolType = activity [ "{" ProtocolName "}" ].

// 8. Procedures & operators
ProcedureDeclaration = ProcedureHeading [ ProcImplementationClause ] ";" [ ProcedureBody ";" ].
ProcImplementationClause = implements ImplementedMemberName { ","
ImplementedMemberName }.
ImplementedMemberName = ( DefinitionName | "[" "]" ) "." MemberName.
ProcedureHeading = procedure [ Modifiers ] ProcedureName [ FormalParameters ].
ProcedureBody = Declarations UnitBody SimpleName.
FormalParameters = "(" [ FPSection { ";" FPSection } ] ")" [ ":" FormalType ].
FPSection = [ var ] ident { "," ident } ":" FormalType.
OperatorDeclaration = operator [ Modifiers ] OpSymbol [ FormalParameters ] ";" OperatorBody ";".
OperatorBody = Declarations UnitBody OpSymbol.
OpSymbol = string.   // A 1,2,3-character string; the set of possible symbols is restricted
```

```
// 9. Statements
StatementSequence = Statement { ";" Statement }.
Statement =    [ Assignment
               | ProcedureCall
               | IfStatement
               | CaseStatement
               | WhileStatement
               | RepeatStatement
               | LoopStatement
               | ForStatement
               | await Expression
               | exit
               | return [ Expression { "," Expression } ]
               | BlockStatement
               | Send
               | SendReceive
               | LaunchActivity
               | AnonymousActivity
               ].
Assignment = Designator { "," Designator } ":=" Expression { "," Expression }.
ProcedureCall = Designator.
IfStatement =  if Expression then StatementSequence
               { elsif Expression then StatementSequence }
               [ else  StatementSequence ]
                  end.
CaseStatement =  case Expression of
                     Case { "|" Case }
                  [ else StatementSequence ]
                    end.
Case = [ CaseLabel { "," CaseLabel } ":" StatementSequence ].
CaseLabel = ConstExpression [ ".." ConstExpression ].
WhileStatement = while Expression do StatementSequence end.
RepeatStatement = repeat StatementSequence until Expression.
LoopStatement = loop StatementSequence end.
ForStatement = for ident ":=" Expression to Expression [ by ConstExpression ]
                  do StatementSequence end.
BlockStatement =  do [ Modifiers ]
                     [ StatementSequence ]
                     { ExceptionHandler }
                     [ CommonExceptionHandler ]
                     [ TerminationHandler ]
                      end.
ExceptionHandler = on ExceptionName { "," ExceptionName } do StatementSequence.
CommonExceptionHandler = on exception do StatementSequence.
TerminationHandler = on termination do StatementSequence.
Send = ActivityInstanceName [ "(" Expression { "," Expression } ")" ].
SendReceive = [ Designator { "," Designator } ":=" ] Send.

Accept = accept QualIdent {"," QualIdent}.
LaunchActivity = new ActivityName [ "(" ActualParameters ")" ].
AnonymousActivity = activity ";" Declarations UnitBody.


// 10. Expressions
Expression =   SimpleExpression
                     [ ( "=" | "#" | "<" | "<=" | ">" | ">=" | in ) SimpleExpression ]
               | Designator implements DefinitionName
               | Designator is TypeName.
SimpleExpression = [ "+"|"-" ] Term { ( "+" | "" | or ) Term }.
Term = Factor { ( "*" | "/" | div | mod | "&" ) Factor }.
```

```
Factor = number
         | CharConstant
         | string
         | nil
         | Set
         | Designator
         | new TypeName [ "(" ActualParameters ")" ]
         | new ActivityName [ "(" ActualParameters ")" ]
         | "(" Expression ")"
         | "~" Factor
         | Factor "**" Factor.
Set = "{" [ SetElement { "," SetElement } ] "}".
SetElement = Expression [ ".." Expression ].
Designator = Instance
             | TypeName "(" Expression [ "," Size ] ")"          // Conversion
             | Designator "^"                                    // Dereference
             | Designator "[" Expression { "," Expression } "]"  // Array element
             | Designator "(" [ ActualParameters ] ")"           // Function call
             | Designator "." MemberName                         // Member selector
Instance = ( self | InstanceName | DefinitionName "(" InstanceName ")" ).
Size = ConstantExpression.
ActualParameters = Actual { "," Actual }.
Actual = Expression [ "{" [ var ] FormalType "}" ].  // Argument with type signature

// 11. Constants
number = (whole | real) [ "{" Width "}" ].
whole = digit {digit} | digit {hexDigit} "H".
real = digit { digit } "." { digit } [ ScaleFactor ].
ScaleFactor = "E" ["+" | "" ] digit { digit }.
HexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = '"' character '"' | "'" character "'" | digit { HexDigit } "X".
string = '"' { character } '"' | "'" { character } "'".
character = // Any character from the alphabet except the current delimiter character

// 12. Identifiers & names
ident = ( letter | "_" ) { letter | digit | "_" }.
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
IdentList = ident { "," ident }.
QualIdent = { ident "." } ident.
DefinitionName = QualIdent.
ModuleName = QualIdent.
NamespaceName = QualIdent.
ImplementationName = QualIdent.
ObjectName = QualIdent.
TypeName = QualIdent.
ExceptionName = QualIdent.
InstanceName = QualIdent.
ActivityInstanceName = QualIdent.
ProcedureName = ident.
ProtocolName = ident.
ActivityName = ident.
MemberName = ( ident | OpSymbol ).
SimpleName = ident.
```

18 References

The references are ordered alphabetically:

[AOS]
An Active Object System Design and Multiprocessor Implementation
Dr Pieter Muller
PhD Thesis 14755 ETH Zurich

[CLI] Standard ECMA-335:
Common Language Infrastructure (CLI), see section on Common Type System (CTS)
http://www.ecma.ch/ecma1/STAND/ecma-355.htm

[Compiler]
Zonnon Programmers' Manual, ETH Zürich, 2005
This describes the implementation of the compiler for as specific platform e.g. Microsoft .NET
http://zonnon.ethz.ch

[Dijkstra]
E.W. Dijkstra, Guarded Commands, Non-Determinacy and a Calculus for the Derivation of Programs, EWD418, Jun. 1974, 1974
http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD418.pdf

[Mesa]
Mesa Language Manual Version 5.0
J Mitchell, W Maybury, R Sweet
CSL-79-3 April 1979
XEROX Palo Alto Research Center, California, USA

[Modula-2]
Programming in Modula-2
N Wirth
Springer Verlag 1982, 1983, 1985
ISBN 0-540-15078-1, ISBN 0-387-15078-1

[Monitor]
C. A. R. Hoare, Monitors: an operating system structuring concept, Comm. of the ACM, 17 (1974), 549–557.

[Oberon]
Project Oberon: The Design of an Operating System and Compiler
N. Wirth and J. Gutknecht
ACM Press 1992, ISBN 0-201-54428-8

[Pascal]
PASCAL – User Manual and Report, ISO Pascal Standard
Kathleen Jensen and Niklaus Wirth
Springer Verlag 1974, 1985, 1991
ISBN 0-387-97649-3, ISBN 0-540-97649-3

[Zonnon]
Zonnon for .NET: A Language and Compiler Experiment
J. Gutknecht and E. Zueff
LNCS 2789, Springer Verlag 2003, ISBN 3-540-40796-0