Zonnon
A Quick Reference Guide

Alan D. Freed
Clifford H. Spicer Chair in Engineering
College of Science, Engineering & Technology
Saginaw Valley State University
202 Pioneer Hall, 7400 Bay Road
University Center, MI 48710
E-Mail: adfreed@svsu.edu

draft copy
January 23, 2013

Abstract

Write abstract

1 Introduction

Zonnon is the most recent language in the Euler/Algol/Pascal/Modula/Oberon family of programming languages developed by Profs. Niklaus Wirth and Jürg Gutknecht from the Computer Systems Institute at ETH Zurich (Eidgenössische Technische Hochschule, Zürich, i.e., the Swiss Federal Institute of Technology, Zurich). Eugene Zueff, Roman Mitin and Nina Gonova, students and research affiliates at ETH Zurich, were the primary compiler developers. The Zonnon compiler, limited documentation, and example programs are available for download from their website `http://www.zonnon.ethz.ch`.

To date, no textbook has been written in English that describes how to program in Zonnon. A tutorial, originally written in Russian, has been translated into English by its author, Roman Mitin, but, like this document, it is not in a finished state at this time. This document is not a substitute for a textbook on Zonnon, but it might be useful as a quick reference guide to refer to when a programming question arises to those learning the language. This guide does not cover all of the capabilities of the language, as its intent is for the novice to use as a resouce next to them when they are writing code on the computer.

2 Comments

It is good parctice to begin each program file with a few comment statements that identify who you are, the programmer, what the date is, and any necessary affiliations. This header should also contain a brief statement as to what this particular code is suppose to do, or why it was written, if it is copyrighted, or under a license of some kind or another, e.g.,

```
(* Joe Student, XYZ University *)
(* today's date                *)
(* Programming Assignment #1   *)
```

A comment statement in Zonnon begins with `(*` and ends with `*)`, as seen above. It is not necessary to place each line in a comment. Comments may span over how many ever lines of code you choose, and may begin at any position within a line of code. They may even be nested, but each opening `(*` must be paired with a closing `*)`.

3 Words

There are two kinds of words that the programmer has at their disposal. The first set includes the *reserved words* of the language, while the second set includes the words the programmer creates to describe an entity to be represented. An alphabetical listing of the reserved words that make up the Zonnon language includes: **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**, **return**, **self**, **termination**, **then**, **to**, **true**, **type**, **until**, **var** and **while**. As the compiler currently stands, these reserved words must be used in lower case. Although the language definition permits them to be used either in all lower-case or in all upper-case, that feature has not been implemented yet.

An admissible word that the programmer can create contains an ensemble of any English letter “a”|. . . |“z”|“A”|. . . |“Z”, the underscore “_”, and any digit “0”|. . . |“9”. It can be of arbitrary length, but it cannot begin with a digit. Good word choice goes a very long way to readability of a code. Mixing the case allows the programmer to use an otherwise reserved word, e.g., `Object` could be used by the programmer as a programmer defined entity. In the code fragments throughout this guide, user defined words are expressed in angled brackets like <`userWord`>.

A *program* is a collection of words arranged in such a manner that, collectively, they become understandable to the Zonnon compiler. A program is a text file ending in a ‘.znn’ extension. It can be written in any text editor of your choice. Several have highlighting features that recognize the reserved words, thereby aiding in the overall coding process, e.g., Kate or the editor in Zonnon Builder that comes with the Windows™ version of the compiler.

4 Basic Types

There are a number of base types in the Zonnon language that provide a foundation upon which the programmer can build applications, and even other types of user specification. These core types are:

boolean	A truth value of either **true** or **false**.
cardinal	A counting number beginning at 0 and going to **max(cardinal)**.
char	Any character from the character set of your system, e.g., ASCII or UTF8.
integer	A counting number between **min(integer)** and **max(integer)**, where |**min(integer)**| = 1 + **max(integer)**.
object	The base type from which all user-defined Zonnon objects are derived.
real	A floating point number with values between **min(real)** and **max(real)**.
string	A concatenation of characters from the base character set.

Types **cardinal**, **char**, **integer**and **real** can be used in different sizes by specifying their length in bits inside a pair of braces { and } immediately following the type name, e.g., **real**{32} associates with a 32-bit real. The default sizes are: **cardinal**{32}, **char**{16}, **integer**{32} and **real**{64}. Type **char** also comes in an 8-bit size; types **cardinal** and **integer** also come in 8-, 16- and 64-bit sizes; and type **real** also comes in a 32-bit size.

Zonnon base types that are not listed above include: **fixed**, **range** and **set**, as they are not discussed in this document.

4.1 Operators

Instances of types can be combined to create new instances through a variety of operators. Instances of type **string** can be concatenated with the `+` operator. The numeric types of **cardinal**, **integer** and **real** have a unary operator `-` used for negation (except **cardinal** numbers which do not admit negative numbers), and binary operators for addition +, subtraction -, multiplication * and exponentiation **. **real** division has the operator /, while **cardinal** and **integer** division have operators div and mod returning the divisor and remainder, respectively. Binary operators for comparing two numeric values are equal =, not equal #, less than <, less than or equal <=, greater than > or greater then or equal >=, each returning a **boolean** result.

For **boolean** instances, there are three operators: the uniary operator ~ for negation, and the binary operators & for logical and, and or for logical or. Two **boolean** instances can also be compared with operators = and #.

When a line of code is parsed by the compiler, execution is left to right unless terms are grouped with parentheses (and). Operator precedence from highest to lowest is:

1. unary operations - and ~
2. exponentiation operator **
3. multiplication operators *, /, div and mod
4. addition operators + and -
5. relations =, #, <, <=, > and >=

5 Statements

Statements are used for two purposes: assignment and comparison. An *assignment statement* has three parts to it, as shown below.

```
<receiver> := <sender>;
```

A statement may span several lines of code. It ends with a semicolon. In the statement above, entity <sender> on the right-hand side is *assigned* by the operator := to entity <receiver> on the left-hand side. Zonnon is a strongly typed language. This means that the type which instance <receiver> belongs to must be compatible with the type which instance <sender> belongs to. The compiler will produce an error message if their declared types are not compatible with one another.

Within a scope, a sequence of statements might look like:

```
begin
   <receiver1> := <sender1>;
   <receiver2> := <sender2>;
   ...
   <receiverN> := <senderN>
end;
```

Notice that the last assignment statement in a scope does not need to close with a semicolon. It can, it just doesn't have to.

A *comparison statement* also has three parts to it, as shown below.

```
<left> =  <right>;
<left> #  <right>;
<left> <  <right>;
<left> <= <right>;
<left> >  <right>;
<left> >= <right>;
```

These statements compare the value held by <right> with the value held by <left>, and report on the outcome of the associated truth test. A comparison statement takes on the result of its associated truth test, i.e., it can be either **true** or **false**. Once again, the strong typing of Zonnon requires that the type belonging to instance <right> must be compatible with the type belonging to instance <left>, and such operators must be defined for said type, otherwise the compiler will output an error message.

6 Declarations

Two different types of declaration blocks are provided for: constants and variables. A *constant declaration* associates an identifier with its value.

```
const
   <name1> = <value1>;
   <name2> = <value2>;
```

Two things are worthy of notice here. The list of constants are semicolon separated, and the equality operator =, not the assignment operator `:=`, associates the identifier with its value.

A *variable declaration* has the following structure.

```
var {private|public|public, immutable}
   <name1T1>, ..., <nameNT1> : <type1>;
   <name1T2>, ..., <nameNT2> : <type2>;
   ...
   <name1TM>, ..., <nameNTM> : <typeM>;
```

The instances of a type are placed to the left of the colon, while the type that they associate with is to the right of the colon. Instances are comma separated, while type clauses are semicolon separated. All variables declared in a **var** block are either `private` (visible only inside the scope of their definition), or `public` (visible and writable outside the scope of their definition), or `public, immutable` (read-only outside the scope of their definition). Multiple **var** blocks are permitted so that all necessary instances of each visibility case can be dealt with.

7 Types

Two kinds of user-defined types are now discussed; they are the **array** and **enumeration** types. Later, the **procedure**, **record** and **object** types will also be discussed. Not addressed in this introductory guide are the **activity**, **interface** and **protocol** types, which are very powerful, but whose use lies beyond the capabilities of an introductory course.

7.1 Arrays

An **array** is a collection of elements that are instances of the same type. They can either be created as `value` (i.e., static) or `ref` (i.e., dynamic, by pointer) entities, e.g.,

```
type {private|public}
   <staticArrayType>
      = array <length> of <dataType>;
   <dynamicArrayType>
      = array * of <dataType>;
```

where `<length>` is any cardinal value, while * acts as a wild card, implying that the size of the array will be established by the user at runtime. Declaring these arrays takes on the form

```
var
   <staticArray>  : <staticArrayType>;
   <dynamicArray> : <dynamicArrayType>;
begin
   <dynamicArray>
      := new <dynamicArrayType>(<rows>);
```

so a `<staticArray>` only requires its **var** declaration by the programmer, while a `<dynamicArray>` also needs to be allocated with a call to **new**. The advantage here is that the number of `<rows>` that are to be assigned to `<dynamicArray>` need not be known a priori, only at runtime.

Arrays can be of multiple dimensions, and are created by replacing `<length>` with `<length1>`, `<length2>`, ..., `<lengthN>` in a static allocation, or by replacing * with *, *, ..., * for a dynamically allocated array. There needs to be one designator, either `<length>` or *, for each dimension desired.

The length of an array can be retrieved by calling the reserved word **len** in one of two ways, either as `<length> := len(<array>)` for a one-dimensional array, or as `<rows> := len(<matrix>,0)` or as `<columns> := len(<matrix>,1)` for, in this case, a two-dimensional array, etc.

Elements of an array can be either assigned to or retrieved from an indexer, which takes the form of '`<array>[<index>]`' for a one-dimensional array, '`<matrix>[<index>,<index>]`' for a two-dimensional array or matrix, etc. Admissible values for an `<index>` range from `0` to `len(<array>) - 1`, and similarly for multi-dimensioned arrays. An example of assigning and retrieving values is

```
<matrix>[<row>,<column>] := <setValue>;
<getValue> := <matrix>[<row>,<column>];
```

There is a `{math}` modifier for arrays that is not discussed here. This feature turns on a number of performance enhancements, and an extra syntax for arrays, making their handling more mathematical like. The interested reader is refered to to Zonnon Language Report.[1]

7.2 Enumeration

An enumerated type is one that can assume only a select and known number of values, e.g., the suits in a deck of cards, or the pieces in the game of chess, etc. Use of these types enhances the readability of code, which is the main strength in its usage. They are created as

```
type {private|public}
   <enumeratedType>
      = (<value1>, ..., <valueN>);
```

When used in coding, an instance of an enumerated type would be identified as, e.g., `<enumerated-Type>.<valueN>`.

8 Control Structures

Contol structures in a computer language are used to direct the flow of an algorithm. Two such structures are discussed below: the **if** and **case** statements.

8.1 If Statement

The **if** statement can take on one of three forms. The first control structure executes an extra section of code whenever a comparison statement returns a **true**, as shown below.

```
if <truthTestIsTrue> then
   <statements>
end;
```

In the second control structure, there are different sequences of statements to be executed for both outcomes of a comparison statement.

```
if <truthTestIsTrue> then
   <statementsWhenTrue>
else
   <statementsWhenFalse>
end;
```

And the third control structure applies whenever there are multiple comparison statements to test against, in which case it looks like the following.

```
if <truthTest1IsTrue> then
   <statements1>
elsif <truthTest2IsTrue> then
   <statements2>
...
elsif <truthTestNIsTrue> then
   <statementsN>
else
   <statementsWhenAllAreFalse>
end;
```

There is no restriction on the number of **elsif** repeat clauses that one uses. Control ratchets down the structure until a truth test returns **true**, or the **else** clause is reached. After executing the ensuing statements, programmatic flow continues with the line of code immediately following the **if** structure.

It is good programming practice to always use an **else** segment whenever an **elsif** clause is present so that the structure can handle the all-false scenario. If an all-false scenario is encountered during a runtime, and no **else** leg is available for escape from the logic structure of the if-then-else statement, than a runtime error will be thrown.

8.2 Case Statement

A **case** statement is similar to an **if** statement in its function. The syntax is quite different though. These control structures work well with instances of an enumeration type, and look like

```
var
   <item> : <enumerationType>;
begin
   case <item> of
      <item>.<value1> :
         <statements1>
   |  <item>.<value2> :
         <statements2>
   |  <item>.<value3> :
```

1. The Zonnon Language Report is available from the Zonnon website: `http://www.zonnon.ethz.ch`.

```
        <statements3>
   ...
   else
      <statements>
   end;
```

A colon is used to separate the testing of the condition from the code that is to be executed after a favorable test. Like in an **if** statement, if a condition of test is not favorable, the control of programmatic flow moves to the next block in the **case** structure. These blocks are separated by the delimiter of |, which is the analog of an **elsif** in the if-then-else structure, with the final block being designated with an **else** identifier, as in an **if** structure.

This control structure also works with expressions that are instances of type **char**, i.e., the `<item>` in the above example is a character, where the `<item>.<value>` is a character, e.g., “C”, or a range of characters, e.g., “A”..“F”.

9 Looping Structures

There are four types of looping structures available for the programmer to use in the Zonnon language: the **for** loop, the **while** loop, the **repeat** loop, and a generic **loop** structure. These structures execute a sequence of statements contained within them a multiple number of times, terminating their sequential processing in a manner dictated by the kind of looping structure that is in control. Looping structures can be nested.

9.1 For Loops

This looping structure is often used when managing arrays. Here an indexer is systematically incremented from one looping iteration to the next, terminating when the incrementor reaches or surpasses an assigned limit. A **for** loop has the structure

```
for <integerAssignmentStatement>
   to <limit> by <increment> do
      <statements>
end;
```

where an example of `<integerAssignmentStatement>` might look like `i := 1`. The **to** clause states what the upper `<limit>` of incrementing `i` would be, while the **by** clause specifies how much each increment is to advance by. When not present, the **by** clause is taken to be `1` by default.

For example, `for i := 10 to 0 by -2 do` would set the value for `i` to `10` in the first pass through `<statements>`, `8` in the second pass, etc., and exiting out of the **for** structure after completing the pass where the value of `i` is at `0`.

9.2 While Loops

In a **while** loop, the test for looping occurs at the beginning of the structure, and it is possible that it will fail on its intial test, thereby forgoing the structure altogether. If the test condition has the value **true**, then enterance into the structure is permitted. Programmatic control moves onto the next line of code after exiting the **while** loop, which happens whenever the test condition turns up **false**.

```
while <logicalTestIsTrue> then
   <statements>
end;
```

9.3 Repeat Loops

A **repeat** loop is the anthesis of the **while** loop. A **repeat** loop is executed at least once. Here the test condition for exiting the loop occurs at the end of the structure. The code within the **repeat** block is repeated if the test condition returns **false**, with programmatic control moving beyond the looping structure only when the test condition returns **true**.

```
repeat
   <statements>
until <logicalTestIsTrue>;
```

9.4 A Generic Loop

A common need when writing code is the ability to handle multiple exit strategies in a looping algorithm. This can be dealt with by using the generic **loop** structure. Actually, all of the previous looping structures are special cases of this structure, which takes the form

```
loop
   <statements>;
   if <logicalTest1IsTrue> then exit end;
   <statements>;
   if <logicalTest2IsTrue> then exit end;
   ...
   if <logicalTestNIsTrue> then exit end;
   <statements>
end;
```

Programmatic control leaves the scope of its **loop** structure whenever an **exit** statement is executed.

10 Procedures

There are three types of procedures in programming, and all use the syntax of **procedure** in the Zonnon language: the command, the function, and the procedure.

A command has the simplest form. Commands are used whenever an action is to be taken, and look like

```
procedure {private|public} <name>;
var
   <variableDeclarations>;
begin
   <statements>
end <name>;
```

An example would be something like `Log.Close` to close a log file.

A function returns an instance of a type, and usually has an argument being supplied to it, sometimes several. It has a general structure of

```
procedure {private|public} <name>
   (<values> : <valType) : <returnType>;
var
   <returnValue> : <returnType>;
begin
   <statements>;
   return <returnValue>
end <name>;
```

The function call `y := sin(x)` would be an example. There can be multiple arguments of multiple types. If there is more than one agrument of a given type, then that list is comma separated. Lists between multiple types are semicolon separated. Like in a **var** declaration, a list of instances belonging to a type are separated from the type they are being designated to by a colon. After the argument list closes with a parenthesis `)`, a colon appears followed by the type name belonging to the value being returned.

The final kind of procedure is a **procedure**. Here arguments can be of two kinds: those passed by value (as in the previous example) and those passed by reference, i.e., that follows a **var** in the argument list.

```
procedure {private|public} <name>
   (<values> : <valueType>; ...;
   var <refValues> : <refType>; ...);
var
   <variableDeclarations>;
begin
   <statements>
end <name>;
```

There may be any number of either kind of variable being passed, and there may be multiple variables of any given type being passed. In a call to <`name`>, deep copies of the <`values`> are made so that any changes made to these values inside the procedure will not effect the values held by the variables from the calling statement. However, instances of <`refValues`> are passed as shallow copies so that all changes made to them within the procedure get incorporated into the variables present in the calling statement. This is an alternative way to pass information back to the calling statement.

The choice of modifier, i.e., either `private` or `public`, indicates if the **procedure** is to be visible, or not, outside of the scope in which it is being defined. A **procedure** has a scope, so that the variables defined in <`variableDeclarations`> are visible only inside the **procedure** itself. They can be declared `public`, and thereby be made visible outside the **procedure**, but this is not commonly done, as that is the purpose of passing variables via the argument list.

10.1 Type

Zonnon allows the user to create types that establish classes of procedures. This is useful if one needs to pass a procedure as an argument in a function call, say in a numerical method for differentiating a function. The programmer could pass any function to the differentiation procedure that belonged to the type specified in the argument of the differentiation procedure. A **procedure type** is created in the type declaration as

```
type {private|public}
   <Command>   = procedure <commandName>;
   <Function>  = procedure <functionName>
      (<arguments>:<type>; ...) : <returnType>;
   <Procedure> = procedure <procedureName>
      (<valueArgs>:<valueType>; ...
       var <refArgs>:<refType>; ...);
```

An unassigned procedure type has the **nil** value.

11 Modules

A **module** is one of four compilation units in the Zonnon language, and the most important of the four. A **module** is a container. It holds things in logical groupings. The interface of a module is its contract to the ‘world’. Other modules and applications can load in your **module** to make use of anything that you have made `public`, or variables that are designated as `public, immutable`. A typical **module** has a structure like

```
module <nameSpace>.<name>;
import
   <nameSpace1>.<name1> as <alias1>,
   ...
   <nameSpaceN>.<nameN> as <aliasN>;
const {private|public}
   <constantDeclarations>;
var {private|public|public, immutable}
   <variableDeclarations>;
type {private|public}
   <typeDeclarations>;
procedure {private|public} <procName1>
   (<argDeclarations1>) : <returnType1>;
begin
   <statements>
end <procName1>;
...
procedure {private|public} <procNameM>
   (<argDeclarationsM>) : <returnTypeM>;
begin
   <statements>
end <procNameM>;
begin
   <statements>
end <name>.
```

The full name of a **module** includes its name space, which is akin to a directory structure. A **module** finishes its coding with **end** <name>, closing with a period. The `<nameSpace>` does not appear in the **end** statement. When the module is loaded into memory for the first time, it executes the block of `<statements>` between **begin** and **end** <name>.

Just as a **module** exports an interface that others can call upon and use, it imports a list of modules and objects that others have created and whose procedures, etc. are called upon and used in the module. This is done via an **import** block that contains a comma separated list of all such modules, each followed with an optional alias designated with an **as** clause. The use of an alias greatly improves the readability of one’s code. A **procedure**, **type**, variable or the like that is used within the **module** is expressed as, e.g., `<alias>.<procedureName>(<arguments>)`.

12 Ojbects

Syntactically, an **object** is very similar to a **module** in the Zonnon programming language. Philosophically, they differ in one very important regard. Instances of an **object** have their own data structure that they manage and maintain; whereas, a **module** has only one instance, and therefore has only one data structure to manage and maintain.

Like a **module**, an **ojbect** can **import** other modules and/or objects for use. It can implement a **definition**, or aggrigate several. It can have declaration blocks for constants, variables (or fields), types, and procedures (or methods). The fields that it holds constitutes its data, and the methods it exports provide a vehicle by which they can be managed.

An **object** can be either of `value` or `ref` (or pointer) type, the latter requiring use of the **new** command to create an instance of itself. Typically, an **object** is exported for use outside of its scope by making it public, but this is not necessary.

Objects can be coded in one of two ways: either as a separate compilation unit, or as a **type** declaration within a **module**. They are both first-class citizens. Coding it one way or the other does not affect how it it used or what capabilities it can possess. An important reason for selecting the **module** approach has to do with creating overloaded operators for the newly created **object**. This capability is not addressed in this reference guide.

12.1 Definition

Templates for objects can be created with a **definition** structure. A **definition** is an empty object in the sense that it cannot perform any tasks. What it does do is establish an interface that actual objects can incorporate. This is useful if the same processes are to exist in a collection of objects to be created. A **definition** is a compilation unit, just like a **module**, and looks something like

```
definition {public, value|ref}
   <nameSpace>.<name>
   refines <nameSpaceR>.<nameR>;
import
   <nameSpace1>.<name1> as <alias1>,
   ...
   <nameSpaceN>.<nameN> as <aliasN>;
procedure {public} <procName1>
   (<argDeclarations1>) : <returnType1>;
...
procedure {public} <procNameM>
   (<argDeclarationsM>) : <returnTypeM>;
end <name>.
```

A **definition** creates a programming contract, and therefore only includes `public` features of the interface. It needs to **import** those entities that appear in the interface only. It can **refine** and existing **definition**, adding capabilities from that interface to the current one. If more than one refinement is made, the list is comma separated. A **procedure** has no body in a **definition** statement.

12.2 As a Separate Compilation Unit

As its own compilation unit, an **object** would be coded much like a **module**; for example,

```
object <nameSpace>.<name> implements
   <defNameSpace>.<name>, ...;
import
   <nameSpace1>.<name1> as <alias1>,
   ...
   <nameSpaceN>.<nameN> as <aliasN>;
const {private|public}
   <constantDeclarations>;
var {private|public|public, immutable}
   <variableDeclarations>;
type {private|public}
   <typeDeclarations>;
procedure {public} <proc1>
   (<proc1Interface>) : <proc1ReturnType>
   implements <defNameSpace>.<def>.<proc1>;
begin
   <statements>
end <proc1>;
....
procedure {private|public} <procName1>
   (<argDeclarations1>) : <returnType1>;
begin
   <statements>
end <procName1>;
...
begin
   <statements>
end <name>.
```

Procedures whose interface is inherited must designate what method is being implemented and from which **definition**. This is shown with the **implements** clause. The interface before the designator **implements** must match what it is in the **definition** whose interface is being implemented. Additional methods can be coded just like regular procedures in a module. The syntax is the same, which really simplifies their use in this programming language, and why they can be introduced at such an early stage in one's learning how to program.

12.3 As a Type in a Module

Alternatively, an **object** can be declared in a **type** declaration block within a **module**. Here it would read as

```
type {public, value|ref} <name> = object
   implements <defNameSpace>.<name>, ...
const {private|public}
   <constantDeclarations>;
var {private|public|public, immutable}
   <variableDeclarations>;
type {private|public}
   <typeDeclarations>;
procedure {public} <proc1>
   (<proc1Interface>) : <proc1ReturnType>
   implements <defNameSpace>.<def>.<proc1>;
begin
   <statements>
end <proc1>;
....
procedure {private|public} <procName1>
   (<argDeclarations1>) : <returnType1>;
begin
   <statements>
end <procName1>;
...
begin
   <statements>
end <name>;
```

Imports are now handled by the **module** encapsulating the **object**. A couple of notable syntactical differences include the fact that the first line beginning with **type** does no end with any punctuation. The **object** called <name> inherits the <nameSpace> of the **module**. Also, at the **end**, the line ends in a semicolon, not a period, as this is just one facet of the compilation unit, which is now a **module**. The **object**, so defined, has its own scope which resides within the scope of the **module**.

Like a **module**, when an **object** is first loaded into memory the block of <statements> after **begin** and before **end** <name> is executed. This is a nice way to initialize an **object**, preparing it for use.