Zonnon Tutorial

Jürg Gutknecht, Roman Mitin, Nina Gonova

Document version: 0.1 (draft)

Last update: January 13, 2010

# Introduction

It is important to say that Zonnon (although superficially similar to

Pascal) represents some radically new concepts such as

\* Inclusion of both modular and OO programming model

\* Compositional OO model (in opposite to single inheritance class tree)

\* Advanced abstract data types including mathematical structure types

\* Matlab-like notation in combination with efficiency of compiled code

\* Clean integration of concurrency and OOP in the form of active objects

\* Uniform and generalized communication model based on dialogs (subsuming both synchronous and asynchronous method calls)

# Language alphabet

Alphabet is a set of symbols allowed to use in the language. In Zonnon programming language a subset of ASCII character set is used. Alphabet includes letters, digits, hexadecimal digits, special symbols, spaces and reserved words.

*Letters* – are letters of the Latin alphabet from a to z and form A to Z, and underscore symbol.

*Digits* – Arabic digits 0 – 9.

*Hexadecimal digits* have values from 0 to 15. First 10 values denoted with Arabic digits, other with Latin letters A...F or a...f.

*Special symbols* – signs of an operation (+ - \* / = #), delimiters (, ; .), string delimiters (" ‘). Also to special symbols belong pair of: <= >= := etc. In the code these pairs cannot be separated with spaces.

## Identifiers

Identifier is a name of any program entity. It can consist of letters and digits. First symbol should be a letter. In identifiers capital and small letters are distinguishable.

Reserved keywords cannot be used as identifiers (see Keywords).

## Delimiters

Delimiters are used to separate identifiers, numbers, keywords one from another. As a delimeter:

* space;
* any control symbol (codes from 0 to 31);
* comments.

In any place in a program where one delimiter can be placed any number of delimiters allowed. This allows shape the program to improve readability. Refer to coding guidelines on for recommended practices.

## Punctuation symbols

|  |  |
| --- | --- |
| *Symbol* | *Use* |
| (\* \*) | Comments |
| [] | Indexing arrays, sets, sub-arrays |
| () | Sub-expressions, list of arguments |
| " | Character or string constant |
| := | Assignment |
| ; | Statement separator |
| , | List separator |
| . | End of module, unit of compilation, separation of integer part in a real number, separation of a filed in a record |
| .. | Separation of range boundaries |

## Keywords

Following keywords are reserved in Zonnon language:

|  |  |  |  |
| --- | --- | --- | --- |
| 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  launch | loop  mod  module  new  nil  object  of  on  operator  or  procedure  receive  record | refines  repeat  return  self  send  then  to  true  type  until  var  while  unused |

Keywords cannot be used as identifiers.

Program structure

Zonnon is a modular programming language. Programming in Zonnon is based on four main units:

* **object** – type template to describe objects.
* **definition** – abstract term used to define interfaces.
* **implementation** – container for reusable partial object implementations.
* **module** – textual container and object of program structure.

These terms create basis for program composition as a whole and also for textual segmentation and separate compilation during development.

Text of a program includes modules, objects, definitions and implementations. Running program includes one or more dynamically created modules. Runtime system provides mechanisms for dynamic loading and unloading of modules.

A **module** has dual nature: it defines syntactic container for logical grouping of program declarations and simultaneously describes an object which life cycle is controlled by the system (in contrast to instances of objects controlled by the user). Hence a module provides mechanisms for textual separation of source codes and for dynamic loading of parts of an application at runtime.

Each module has a unique name and contains text which can be separately compiled as a module. Module can implement one or more definitions, can import elements from one or more implementations.

Module can contain:

* Definitions, implementations and objects.
* Simple declarations of constants, types, variables, procedures.
* Declarations for user-defined operators.
* Declarations of activities.

The role of “program” in terms of Pascal programming language in Zonnon is overtaken by **module**.

Consider for an example a program to compute sum of two numbers:

module Example21;

var

x, y, sum: integer;

begin

write("X: "); readln(x);

write("Y: "); readln(y);

sum := x + y; (\* computing the sum \*)

writeln("X + Y = ", sum);

end Example21.

Line module Example21; denotes the beginning of module named Example21.

Then keyword **var** begins a section with variable declarations

**var**

x, y, sum: **integer**;

in which three integer variables with names x, y, sum are declared.

After **begin** keyword the body of the program is located. The program sequentially asks user to enter two numbers and then outputs their sum. It is required to repeat the name of the module after **end** keyword.

# Data types

A data type is a set of values and the operations on those values. In Zonnon programming language there are following groups of data types:

* Simple types (integer, real, etc.);
* Arrays (static, dynamically created, open, mathematical);
* Objects (value objects, reference objects).

Among types used in the language distinguish standard (predefined) and user defined. Standard types are integer, Boolean and character types, sets and strings.

All other data types should be declared in a type declaration section or defined directly at variable declarations.

A type declaration section starts with reserved keyword **type**, followed by type declarations. Format for new type declarations is as follows:

**type**

<type name 1> **=** <type declaration 1>;

<type name 2> **=** <type declaration 2>;

. . .

<type name N> **=** <type declaration N>;

Standard types denoted by predefined identifiers, which are not considered as keywords. Following standard types provided in Zonnon:

* **boolean**
* **char**
* **integer**
* **real**
* **set**
* **string**

For types **char**, **integer**, **real** and **set** number of bits used to store values can be defined by a constant expression in curve brackets **{ }** after the type name. For each type there are own limit values and default values (see section for concrete types). Inside these types a larger type includes all values of a smaller type. For example **integer{16}** includes **integer{8}**, and **set{32}** includes **set{16}**, however **real{80}** does not include **integer{16}**, as types are incompatible. Conversions between different types must be explicit (See Compatibility and type conversion).

## Integer types

In contrast to Turbo Pascal, where for representation of a signed integer five different types are defined, in Zonnon there is single **integer** type with a capability to define its size. For a set of unsigned integer numbers in Zonnon exists type **cardinal**. To get minimal and maximal possible values predefined functions **min()** and **max()**can be used (see Arithmetic functions). Default size for these types is 32 bits. Consider an example that outputs limits for these types:

**module** Example31;

**type**

word = **integer**{16};

**var** i: **integer**;

j: word;

**begin**

i := *max*(**integer**); *(\* Maximal value \*)*

*writeln*(i, " ", *min*(**integer**));

j := *max*(**word**); *(\* Maximal value \*)*

*writeln*(j, " ", *min*(**word**));

**end** Example31.

Executing this will output:

2147483647 -2147483648  
32767 -32768

## Boolean type

A variable of standard logical **boolean** can have one of two values, represented by keywoeds **true** and **false**. At the same time the following is true:

**integer**(**false**) = 0

**integer**(**true**) = 1

Example:

**module** Example32;

**var** b: **boolean**;

**begin**

b := 2 \* 2 = 4; *(\* Assign result of a Boolean expression \*)*

*writeln*(" 2 \* 2 = 4 is ", b);

**end** Example32.

In this example variable b is assigned a value of an expression «is 2 \* 2 equals to 4». If executed this program will output:

2 \* 2 = 4 is true

## Sets

An element of a standard type **set** as its value can have any subset of integer numbers from interval **min(set)** . . **max(set)**. Operations on this type will be discussed in Functions to work with sets.

Example:

**module** Example33;

**var** s : **set** ;

s33 : **set** { 16 };

s128 : **set** { 64 };

**begin**

s := { 1 };

s := { *min*(**set**) .. *max*(**set**) };

**end** Example33;

## Character type

Type **char** allows working with symbols from a set used in the underlying platform (e.g. .NET). By default this is UTF-16. On practice in .NET also can be used UTF-8 и UTF-32. UTF-16 – is a 16-bit type **char**, used to store Unicode-symbols. Format Unicode is used to represent symbols of most known written languages. In UTF-16 codes for the most used symbols are located in the range from 0 to FFFF16 and encoded via one 16-bit constant code. Codes from 10000016 to 10FFFF16 are encoded via two words that represent one symbol. In this case the value of the first word should be in the range from D80016 to DBFF16, and the second from DС0016 to DBFF16. Codes in the interval from D80016 to DBFF16 are udes only for this mechanism and will never be used to represent symbols directly.

Example:

**module** Example34;

**var** ch:**char**;

**begin**

ch := 100X;

*writeln*(ch);

**end** Example34.

As a result of execution of this program symbol A will be displayed.

## Enumeration

Enumeration is a type which values are defined by a set of identifiers. When used in a program as constants these identifiers should be qualified by the type name. Values are ordered and their order is defined by the order of identifiers in the declaration. Neither any other value, except from the listed, belongs to the type. Number of the first value is zero.

Example of enumeration declarations:

**type**

NumberKind **= (**Bin, Oct, Dec, Hex**)**;

Month **= (**Jan, Feb, Mar, Apr, May,   
 Jun, July, Sep, Oct, Nov, Dec**)**;

Use of a declaration:

**module** Example35;

**type**

NumberKind **= (**Bin, Oct, Dec, Hex**)**;

**var**

b: NumberKind;

**begin**

b := NumberKind.Bin;

**end** Example35;

Names used in different enumerations do not have to be distinct as they always qualified by the type name. So, for example, NumberKind.Oct is distinct from Month.Oct.

Predefined function *pred* returns a value in the enumeration of a predecessor to the element given as an argument. If the argument is a first element in the enumeration an exception is triggered. Similarly predefined *succ* returns a value of the next element in the enumeration.

## String type

Type **string** in Zonnon, similarly to **char**, is a wrapper for System.String class from .NET standard library.

In .NET a string is a sequence of Unicode characters, encoded via UTF-16, typically used to represent a text. Strings in .NET are constructed as sequential collections and their values cannot be changed. A result of any string operation is a new string.

# Expressions

An expression is a syntactical unit of a language. Expressions in Zonnon consist of constants, variables, object fields, functions, operations and parenthesis. At first we discuss elements used to construct expressions and then in Evaluation order of expressions.

## Variables

Variables are symbolic names associated with a value and whose associated value may be changed. All variables used in the program must be declared with specifying their type.

A section for variable declarations begins with a keyword **var** followed by variable declarations. Each such declaration consists of a variable name or a list of names, a colon and a type. Each declaration is terminated via a semicolon:

**var**

<list of variables 1>**:** <type 1>;

<list of variables 2>**:** <type 2>;

. . .

<list of variables N>**:** <type N>;

In the section of variable declarations predefined and declared in a program types can be used. For user defined types declaration can be made both earlier in the code (in current or enclosing scope) and later in the code. A width modifier can be applied to predefined types (see section Data types).

Example of variable declarations:

**module** Example41;

**type**

Figure: **(**Star, Rectangle, Circle, Ellipse**)**;

**var**

a, b: **real**; *(\* Predefined type \*)*

x, y: **integer**; *(\* Predefined type \*)*

myfigure: Figure; *(\* Earlier defined type \*)*

align: ( Top, Bottom, Left, Right ); *(\* declaration \*)*

**begin**

align := align.Right;

*writeln*( **integer**( align ) );

**end** Example41.

Variables a and b have real type, x and y are integer, myfigure is an enumeration whose type is declared earlier, align is an enumeration defined right at the variable declaration. The program assigns to the variable of enumeration type one of its possible values. If executed it displays number 3.

## Constants

Constants are values that never change. Constants can be used directly or be given a symbolic name. Constants can be of integer, real, character, Boolean or string types.

Syntax for giving a constant a symbolic name is as follows:

**const**

<name of constant 1> **=** <value 1>;

<name of constant 2> **=** <value 2>;

. . .

<name of constant N> **=** <value N>;

Values of constants can be given by concrete values or by constant expressions. Examples:

**const**

N **=** 10;

LIMIT **=** 2\*N – 1;

FULLSET **=** {min(set)..max(set)};

UNIVERSITY **=** "ETH Zurich";

COMPANY **=** ‘Intel‘;

SEPARATOR **=** ‘\*‘;

### Integer constants

In the representation of integer constants only digits and possibly a sign are used. By default constant are specified in decimal notation. To use hexadecimal notation suffix H is used. To constants as well as variables a width modifier can be applied.

Examples of integer constants:

|  |  |  |
| --- | --- | --- |
| *Constant* | *Type* | *Value* |
| 1991 | **integer** / **cardinal** | 1991 |
| 0DH**{**8**}** | **integer{**8**}** | 13 |
| 10H | **integer** / **cardinal** | 16 |

### Real constants

Representations of real constants always contain a decimal dot. Symbol E can be used as power indicator.

Examples of real constants:

|  |  |  |
| --- | --- | --- |
| *Constant* | *Type* | *Value* |
| 12.3 | **real** | 12.3 |
| 4.567E8 | **real** | 456700000 |
| 0.57712566E-6**{**64**}** | **real{**64**}** | 0.00000057712566 |

### String and symbol constants

Symbol string is a sequence of any (including zero) number of ASCII symbols located on one line and limited with apostrophes or quotation marks. Symbol constants can be also defined by a symbol code in hexadecimal notation terminated by symbol X. Hexadecimal notation can be used to represent symbols that cannot be displayed or for symbols form extended set of Unicode.

Examples of symbol constants:

"a", ‘n‘, "‘", ‘"‘, 20X

Examples of string constants:

"Hello world",

‘Language guide‘,

"Don‘t worry",

‘"‘

String constants can be assigned to a string variable. If there is a need to use within the constant a limiting symbol (a quotation mark or an apostrophe) it should be doubled. Symbol constant can be used anywhere where string constants are allowed.

### Constant expressions

Constant expressions are expressions which can be evaluated at compile time without executing the program. They are a special case of expressions and can contain constants, operations, parenthesis and standard functions.

## Standard functions

Power of a programming language in many ways depends on the set of base objects to construct programs. In Zonnon there is a predefined set of subprograms and on top of that it provides access to standard .NET libraries.

### Predefined functions

Predefined functions are implemented in a standard Zonnon library and their implementation in general case depends on the compiler.

#### Arithmetic functions

| *Name* | *Type of arguments* | *Type of result* | *Description* |
| --- | --- | --- | --- |
| *abs*(x) | integer, cardinal *or* real | type of x | Module *x* |
| *dec*(v) | v: integer, cardinal *or* enumeration | No | v := v - 1 |
| *dec*(v, n) | v: integer, cardinal *or* enumeration  n: integer *or* cardinal | No | v := v - n |
| *inc*(v) | v: integer, cardinal *or* enumeration | No | v := v + 1 |
| *inc*(v, n) | v: integer, cardinal *or* enumeration  n: integer or cardinal | No | v := v + n |
| *max*(T) | integer{w} | Integer | Maximal value of integer{w} |
| *max*(T) | cardinal{w} | Cardinal | Maximal value of cardinal{w} |
| *max*(T) | Enumeration | Enumeration | Maximal value of enumeration |
| *max*(T) | char{w} | Integer | Maximal symbol |
| *max*(T) | real{w} | Real | Maximal value of real{w} |
| *min*(T) | integer{w} | Integer | Minimal value of integer{w} |
| *min*(T) | Enumeration | Enumeration | Minimal value of |
| *min*(T) | char{w} | Integer | Minimal symbol |
| *min*(T) | real{w} | Real | Minimal value of real{w} |
| *odd*(x) | x: integer | Boolean | x **mod** 2 = 1 |

#### String functions and arrays

| *Name* | *Type of arguments* | *Type of result* | *Description* |
| --- | --- | --- | --- |
| *cap*(x) | x: char | char | Corresponding capital symbol |
| *copy*(x, v) | x: string;  v: symbol array | No | v := x |
| *copy*(v, x) | x: string;  v: symbol array | No | x := v |
| *len*(v, n) | v: array;  n: integer *or* cardinal const | integer | Number of elements in dimenstion n (first dimension is 0) |
| *len*(v) | v: array | integer | The same as len(v, 0) |
| *low*(x) | x: char | char | Corresponding small symbol |
| *size*(T) | Any type | integer | Number of bytes requred for a variable of type T |

#### Functions to work with sets

| *Name* | *Type of arguments* | *Type of result* | *Description* |
| --- | --- | --- | --- |
| *excl*(v, x) | v: set;  x: integer *или* cardinal type | No | v := v - {x} |
| *incl*(v, x) | v: set;  x: integer *или* cardinal type | No | v := v + {x} |
| *max*(T) | set{w} | integer | Maximal element of set{w} |
| *min*(T) | set{w} | integer | 0 |

Consider an example that used functions to work with sets. A user inputs numbers from the range 1..63. Zero means exit. Using *incl* numbers are included into the set s. Then in the loop all numbers from the range 1..63 that present in the set are printed to the screen:

**module** Example331;

**var** s: **set{**64**}**;

n: **integer**;

**begin** s := {}; *(\* Empty set \*)*

*writeln*(" Enter numbers < 64. 0 - exit.");

**repeat** *write*("> "); *readln*(n); *incl*(s,n);

**until** n = 0;

*writeln*;

**for** n:= 1 **to** 63 **do**

**if** n **in** s **then** *write*(n:3) **end**

**end**;

*writeln*;  
**end** Example331.

#### Functions for types with order

| *Name* | *Type of arguments* | *Type of result* | *Description* |
| --- | --- | --- | --- |
| *pred*(x) | x: integer | integer | x – 1 |
| *pred*(x) | x: enumeration | type of x | Previous element in the enumeration |
| *pred*(x) | x: char | char | Previous symbol |
| *succ*(x) | x: integer *or* cardinal | integer | x + 1 |
| *succ*(x) | x: enumeration | type of x | Next element in the enumeration |
| *succ*(x) | x: char | char | Next symbol |

#### Debug functions and functions to control execution

|  |  |  |  |
| --- | --- | --- | --- |
| *Name* | *Type of arguments* | *Type of result* | *Description* |
| *assert*(b) | b: boolean | No | If not b then terminate |
| *assert*(b, n) | b: boolean;  n: integer *or* cardinal | No | If not b then terminate with code n |
| *halt*(n) | n: integer *or* cardinal const | No | Terminate the program with code n |

In *assert*(x, n) and *halt*(n) interpretation of n may depend on the implementation.

#### References to objects

| *Name* | *Type of arguments* | *Type of result* | *Description* |
| --- | --- | --- | --- |
| *box*(v) | v: value object | reference of v | Transorms record into reference |
| *unbox*(v) | v: ref object | type of v | Transorms reference into record |

### Functions of standard library

Zonnon language provides access to instruments of the operational environment. Implementaion of Zonnon for .NET provides access to the standard library of .NET which posses a rich set of useful objects and functions.

To access standard library of .NET a Zonnon module must explicitly import a namespace from .NET. A namespace is a named aggregation of logically connected program resources. Example:

**module** Example432;

**import** System; *(\* System is a namespace from standard .NET library \*)*

**begin**

*writeln*(System.Math.Abs(-10));

**end** Example432.

It is possible give shortcut synonyms for imported namespaces. This allows for more compact notation. For example it makes sense in the case of access to a concrete section of the standard library:

**module** Example432b;

**import** System.Math **as** Math;

**begin**

*writeln*(Math.Abs(-10));

**end** Example432b.

Namespace *System* contains many useful classes. One of them is class *Console*, which is used for input/output in console applications. Functions *read* and *write* in Zonnon in the case of .NET are no more than a convenient wrappers to access methods of this class.

## Operations

All operations in Zonnon can be grouped into predefined and user defined operations. Simultaneous use of the user defined and predefined operators will not be discussed in this section of the tutorial. For additional in-depth information refer to the Zonnon Language Report.

### Arithmetic operators

Operators **+**, **-**, **\*** and **/** can be applied to numeric operands. The result is of the type of the operand that includes the type of the other operand. Exception is the division which result is the smallest real type that includes types of both operands.

Arithmetic operators can be applied to operands of integer and real types. There are unary and binary operators.

Unary “plus” (**+**), placed before an integer or real value do not have any impact on it.

Unary “minus” (**-**) changes the sign of the value.

Following operators can be applied to integer and real values:

**+** sum;

**-** difference;

**\*** product;

**/** real quotient.

Operators **div** and **mod** can be applied only to integer operands:

**div** integer quotient;

**mod** reminder.

It is mathematically true that a **= (**a **div** b**)\***b **+** a **mod** b.

Operators **+**, **-**, **\*** can be also used with other types, but then they have different meaning (e.g. for strings as discussed in String operators). These operators can also be redefined by user.

### Boolean operators

These operators can be applied to operands of **boolean** type and provide the result of the same type.

**or** logical disjunction (logical “or”)

p or q means “if p, then **true**, else q”

**&** logical conjunction (logical “and”)

p & q denotes “if p then q, else **false**”

**~** negation **~** p “not p” (logical “not”)

### Operators on set

Operators on sets can be applied to operands of type **set** and provide result of the same type. Declared width in bets of operands must be the same. Unary “minus” denotes complement to x, i.e. –x denotes a set of numbers in the range from 0 to *max*(**set**)which do not belong x.

|  |  |  |
| --- | --- | --- |
| **+** | Union | Bitwise OR |
| **-** | Difference *(x* **–** *y* **=** *x* **\*** *(* **-***y ))* | Bitwise subtraction |
| **\*** | Intersection | Bitwise AND |
| **/** | Symmetric difference:  *x* **/** *y* **=** *((x* **-** *y)* **+** *( y* **-** *x ))* | Bitwise XOR |

Constructor of a set defines the value by enumerating all values in curve brackets. Elements must be integers from range 0.. *max*(**set**). Range m..n denotes all integer numbers starting element m and finishing with element n, including m and n.

### String operators

Sign **+** can be used for concatenation of strings. All other methods from .NET System.String are available. Refer to MSDN or language report for details.

### Relations

In Zonnon following relations are defined:

**=** equality  
**#** inequality  
**<** smaller  
**<=** smaller or equal   
**>** greater  
**>=** greater or equal  
**in** membership of a set

Relational operators always return a result of type **boolean**.

Relations **=**, **#**, **<**, **<=**, **>** and **>=** can be applied to numeric types, **char**, (open) character arrays and strings. Relations **=** and **#** can be applied to operands of **boolean** type, **set** type, and procedure types including value **nil**. A construct x **in** s denotes “x is an element in s”, where x should be of integer type, and s of type **set**.

## Evaluation order of expressions

Operators can be grouped into several classes with different priorities which syntactically differ when used in expressions. Operators with the same priority are executed from left to right.

For example x **–** y **–** z is equivalent to (x **–** y) **–** z.

Parentheses are used to change the order of execution of operators. There can be any number of parentheses in an expression.

Priority of operators is changing as follows:

|  |  |
| --- | --- |
| Highest priority | 1. Expression in parenthesis 2. Function evaluation 3. Unary operators (**~**, unary **-**, unary **+**) 4. Multiplication operators 5. Addition operators 6. Relational operators |
|  |
| Lowest priority |

For example expression  in Zonnon can be written as follows:

((0.5\*b+a)\*(c+d)/(a\*x\*x+b\*x+c))/a.

# Statements

Statements describe certain algorithmic actions. A body of procedures can be represented as sequence of such statements.

## Assignment statement

An assignment replaces current value of a variable with a new value determined by an expression. The expression should be assignment compatible with the variable. The assignment statements contains an operator **:=**, but does not have any return value. Examples:

x **:=** y;

z **:=** a + b;

Res **:=** (i **>** 0) **and** (i **<** 10);

D **:=** Sqr(b) **–** 4 **\*** a **\*** c;

## Structured statements

### Processing block

Processing block allowed grouping together logically connected statements and introduce exception handlers. Refer to language report for more details.

**module** Example521;

**var** a: **boolean**;

**begin**

*read*(a);

**begin** *(\* Begin of processing block \*)*

a := ~ a

**end**; *(\* End of processing block \*)*

*write*(a);

**end** Example521.

### Conditional statement IF

Statement **if** defines conditional execution of a certain sequence of statements. Boolean expression preceding the sequence of statements is called a condition. Condition is evaluated and if its value equals to **true** then corresponding sequence of statements is executed. If the condition is not satisfied then the sequence of statements after keyword **else** is executed if present.

General scheme of if statement:

**if** <condition> **then**

<statements to be executed if condition is true>

**else**

<statements to be executed if condition is false>

**end**

Shortened form:

**if** <condition> **then**

<statements to be executed if condition is true>

**end**

Example:

**module** example522;

**var** a,b,m: **integer**;

**begin**

*write*("A: "); *readln*(a);

*write*("B: "); *readln*(b);

**if** a > b **then**

m := a

**else**

m := b

**end**;

*writeln*("Maximum form a and b is ", m);

**end** example522.

### Conditional statement CASE

Statement **case** defines selection and execution of a sequence of statements. Selection of a concrete sequence for execution is made based on the value (or set of values) corresponding to each sequence of statements. At first the expression given after the keyword **case** is evaluated. Then the sequence is executed whose corresponding set of values includes the result of evaluation of **case**-expression. A **case**–expression can have integer, symbol or enumeration type. Sets of values that correspond to sequences must have the same type as the **case**–expression. These values are called **case**-labels. These labels should be constant values and none of the values may encounter more than once. If the value of case-expression does not match any of the labels then sequence of statements that **else** keyword is selected.

General scheme of case statement:

**case** <expression> **of**

<set of values 1>: <statement sequence>

<set of values 2>: <statement sequence>

<set of values 3>: <statement sequence>

**else**

<statement sequence>

**end**

The following example displays whether the symbol is a digit or not:

**module** example523;

**var** a: **char**;

**begin**

*write*("Enter one symbol: "); *readln*(a);

**case** a **of**

"0".."9":

*writeln*(“It is a digit”)

**else**

*writeln*(“It is not a digit”)

**end**

**end** example523.

### Statement WHILE

Statement **while** defines a repeated execution of a sequence of statements while a condition is evaluated to **true**. The condition is checked before each execution of the sequence of statements. Hence the sequence can be executed zero or more times.

General scheme of while statement:

**while** <condition> **do**

<statement 1>;

…

<statement N>

**end**

The following example computes number of digits in a natural number:

**module** example524;

**var** num, dig**:** **integer**;

**begin**

*write*("Enter a natural number: ");

*readln*(num);

dig **:=** 0;

**while** num # 0 **do**

*inc*(dig); num **:=** num **div** 10

**end**;

*writeln*("Number of digits: ", dig);

**end** example524.

### Statement REPEAT

Statement **repeat** defines a repeated execution of a sequence of statements until an exit condition is evaluated to true. The exit condition is defined by a Boolean expression. The sequence of statements is executed at least once.

General scheme of repeat statement:

**repeat**   
 <statement 1>;  
 …  
 <statement N>  
**until** <condition>;

Example of a protected input:

**module** example525;

**var** length**:** **integer**;

**begin**

**repeat**

*write*("Enter length( > 0): "); *readln*(length)

**until** length **>** 0;

*writeln*("Accepted: ", length);

**end** example525.

### Statement FOR

Statement **for** defines a repeated execution of a sequence of statements fixed number of times. A special variable is used called a loop counter. Value of this variable is changing in the range from a given starting value to a given ending value with a given step. In the case when an ending value is lower than a starting value the step should be negative.

General scheme of for statement:

**for** <variable> := <starting value> **to** <ending value> **by** <step> **do**

<statement 1>;

…

<statement N>

**end**;

The variable used as a loop counter must be visible in the scope where loop statement is placed.

The following example checks if the given number equals to itself written backwards:

**module** example526;

**var** m,n,a,i**:** **integer**;

**begin**

*write*("Enter 4-digit number: ");

*readln*(n);

m **:=** n;

a **:=** 0;

**for** i **:=** 1 **to** 4 **do**

a **:=** a **\*** 10 **+** m **mod** 10;

m **:=** m **div** 10

**end**;

**if** a **=** n **then**

*writeln*("It is symmetric")

**else**

*writeln*("It is not symmetric")

**end**

**end** example526.

### Statement LOOP

Statement **loop** declares a repetition of a sequence of statements. The repetition can be terminated by statement **exit** executed from within the sequence. Statement **exit** defines a termination of enclosing **loop** statement and continuation from the next (following **loop**) statement. Therefore statement **exit** is context dependent and related to **loop** in which it is nested. Statement **loop** is useful for loops with several exit points, were exit conditions is naturally in the middle of the sequence that is repeated.

General scheme of loop statement:

**loop**

<statement 1>

…

<statement k containing **exit**;>

…

<statement N>

**end;**

An example:

**module** example527;

**import** System;

**var** ch**: char**; h, r**:** **real**;

**begin**

**loop**

*write*("Enter height h:"); *readln*(h);

*write*("Enter radius r:");

*readln*(r);

*writeln*("Square of isosceles cone:", System.Math.PI**\***h**\***r**\***r**/**3);

*write*("Terminate the application?(Y/N):");

*readln*(ch);

**if** (ch **=** 'Y') **or** (ch **=** 'y') **then** **exit**; **end**;

*writeln*("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

*writeln*

**end**

**end** example527.

In this example to obtain value of π number module imports System namespace form .NET standard library and uses the constant from its subspace System.Math.

# Structured data types

Structured data types are composed out of simple or structured types.

## Array

Array is a structure composed out of elements of the same type. Number of elements in the array determines its length. Elements designated by indices which are integer numbers in the range from 0 to length of the array minus 1. Syntactical rules for arrays allow constructing three kinds of arrays that differ in how the length is specified. If length is constant then a *static array* is defined. If length is defined with symbol \* then it is a *dynamic array*.

To access elements of an array indexing operation is used **[**<index>**]**. An expression of integer type resulting within the range from 0 to the value by one less than the length of the array can be used as an index.

### Static arrays

In a static array number of elements in any dimension is fixed and is defined by a constant expression.

**module** example610;

**const** N **=** 100;

**var** i**:** **integer**;

a**:** **array** N **of** **integer**;

**begin**

**for** i **:=** 0 **to** N **-** 1 **do** a[i] **:=** i **end**;

**for** i **:=** 0 **to** N **-** 1 **do** *write*(a[i]:4) **end**;

**end** example610.

### Open arrays

In open arrays number of elements is not defined. This type is used only as a type for a formal argument in a procedure declaration.

Actual size of the open array can be accessed using predefined function *len*.

Example of a declaration of open array:

a**: array \* of real;**

Example of a function that returns the length of the array:

**procedure** MyLen(**var** arr: **array** \* **of** **integer**):**integer**;

**begin**

**return** *len*(arr);

**end** MyLen;

### Dynamically created arrays

In a dynamically created array number of elements in any dimension is not defined at declaration and denoted with symbol \*. Dynamic arrays should be explicitly created. Responsibility for memory allocation for such array is entrusted to the programmer. To create a dynamic array a **new** operator is used:

arrayVariable **:=** **new** ArrayType(len0, len1, … );

In more detail operator **new** will be discussed in 0.

Lengths in operator **new** should be set by arbitrary expressions of integer type. At runtime evaluation of each of these expressions should lead to a positive integer number. Otherwise a runtime exception will be generated. Lengths of all dimensions are initialized with 0’s to prevent access to nonexistent elements of the array.

In the next example memory is allocated after a user inputs size of the data:

**module** example610a;

**type** Vector = **array** \* **of** **integer**;

**var** i, n: **integer**;

**var** a: Vector;

**begin**

*write*("Number of elements: "); *readln*(n);

a := **new** Vector(n); *(\* Creating a vector that contains n elements \*)*

*(\* Input of vector components \*)*

**for** i := 0 **to** *len*(a) - 1 **do**

*write*("a[",i:2,"]: "); *read*(a[i])

**end**;

*writeln*;

*(\* Displaying the vector on a screen \*)*

**for** i := 0 **to** *len*(a) - 1 **do**

*write*(a[i]:3);

**end**;

*writeln*;

**end** example610a.

Next example illustrates work with a 2 dimensional array and use of arrays in procedures. Procedures in Zonnon will be discussed in section 0.

**module** example610b;

**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] **:=** i **\*** 10 **+** j

**end**

**end**

**end** InitializeMatrix;

**procedure** PrintMatrix( **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**

*write*(mat[i][j]**:**3**:**0)

**end**;

*writeln*;

**end**

**end** PrintMatrix;

**var** m: **array** 10**,** 10 **of** **real**;

**begin**

InitializeMatrix(m);

*writeln*;

PrintMatrix(m)

**end** example610b.

## Record

Historically in the Pascal family of programming languages record is a type that includes several components called *fields* which can be of different types. Number of fields can be arbitrary. Record declaration looks as follows:

**type** <name of a new type> = **record**

<field>: **<type>**;

<field>: **<type>**;

**end**;

Example:

**type** <name of a new type> = **record**

Re: **real**;

Im: **real**;

**end**;

If the declaration needs to be used outside module then modifier {*public*} should be used.

**type** {*public*} Complex = **record**

Re: **real**;

Im: **real**;

**end** Complex;

Access to a field of a record can be done via “.”. Example:

**var** a: Complex;

**begin**

a.Re := 10;

a.Im := 7;

*writeln*(a.Re," + ",a.Im,"i");

**end**.

# Dynamic data structures

Dynamic data structures and dynamic memory allocation are used for those cases when amount of memory needed is unknown in advance. Zonnon allows programmer to allocate memory when needed. Freeing of unused memory is done by .NET platform automatically.

## Reference

Reference is a variable that stores internal object identifier or special value **nil** as an indication of absence of any object. References are used to organize dynamic data structures. Reference type is declared using modifier *{ref}*. Example:

|  |  |
| --- | --- |
| *record* | *reference* |
| **type** Complex= **record**  Re: **real**;  Im: **real**;  **end** Complex; | **type** {*ref*}PComplex= **record**  Re: **real**;  Im: **real**;  **end** PComplex; |
|  |  |

For a reference type declaration **var** a: PComplex; will not create an instance of an object. The variable will store only internal for the system identifier. After creation of an object using operator **new** this identifier will denote a location of the object in the memory.

## Operator NEW

To create an object declared by a reference type **new**.

For the previous example with PComplex type a statement that actually creates an object looks as follows:

a := **new** PComplex;

There is no need to manually free the memory.

## Example on reference and value types

The following example illustrates difference in use and behavior of value and reference objects.

module Main;

type {ref} RefNumber = record

val: integer;

end RefNumber;

type Number = record

val: integer;

end Number;

var a,b: RefNumber;

var c,d: Number;

begin

a := new RefNumber;

a.val := 10;

b := a; (\* reference to the same object \*)

b.val := 11; (\* affects both a and b \*)

writeln(a.val, " = ", b.val); (\* 11 = 11 \*)

c.val := 15;

d := c; (\* a copy of the object is created \*)

d.val := 16; (\* only one copy is affected \*)

writeln(c.val, " # ", d.val); (\* 15 # 16 \*)

end Main.

# Compatibility and type conversion

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.

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)

# Subprograms in Zonnon

Procedure is a way to declare a subprogram in Zonnon. All variables that can be used by a procedure can be grouped into local and global. Global variables are declared in the module and accessible from a procedure. Local variables are declared in the procedure and visible only to it. If a procedure returns some value then this procedure called a function.

## Procedure declaration

Subprogram procedure is used to execute a sequence of actions. Procedure declaration consists of a signature and a body. A signature has a name and a list of formal parameters. A body contains local declarations of variables and statements. Name of the procedure should be repeated at the end of declaration.

**procedure** {*<access modifier>*} <Name>(<list of arguments>);

<local declarations>

**begin**

<body>

**end** <Name>;

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 **import** (see section 0).

Example of a procedure computing maximum of two numbers:

**module** Example91;

**procedure** Maximum(a,b:**integer**; **var** m:**integer**);

**begin**

**if** a > b **then**

m := a

**else**

m := b

**end**

**end** Maximum;

**var** a,b,m: **integer**;

**begin**

*write*("A = "); *readln*(a);

*write*("B = "); *readln*(b);

Maximum(a, b, m);

*writeln*("max( A, B ) = ", m);

**end** Example91.

## Function declarations

Subprogram function is used to compute a value. In Zonnon a function declaration in contrast to a procedure declaration should have a type of the return value. The body of such procedure should contain **return** statement returning the computed value (see Return statement in procedures).

The same example implemented as a function:

**module** Example92;

**procedure** Maximum(a,b:**integer**):**integer**;

**begin**

**if** a > b **then**

**return** a

**else**

**return** b

**end**

**end** Maximum;

**var** a,b:**integer**;

**begin**

*write*("A = "); *readln*(a);

*write*("B = "); *readln*(b);

*writeln*("max( A, B ) = ", Maximum(a,b));

**end** Example92.

## Formal and actual parameters

Formal parameters declare with which actual arguments this subprogram should be called: their number and types.

There are two main ways to pass arguments to a procedure:

* by value (subprogram will receive a copy of the value and will not be able to change the parameter itself);
* by reference (subprogram will receive a reference to an object and will be able to modify it).

## Value parameter

Value parameter is declared by its name and its type. Example of a function accepting one value parameter:

**procedure** Sqr(a: **real**): **real**;

**begin**

**return** a \* a;

**end** Sqr;

## Reference parameter

Reference parameter is declared similar to value parameter but with leading keyword **var**. Example of a function accepting one reference parameter:

**procedure** Null(**var** a: **real**);

**begin**

a :=0;

**end** Null;

When a variable passed by a reference it can be assigned new values. This mechanism is typically used when there is a need to modify the variable or when the variable has a very large size and creating a copy is not efficient.

## Return statement in procedures

Statement **return** terminates the execution of a procedure and returns control to its caller. For procedures that return a value **return** should be followed by an expression which defines a rerning value. Type of this expression must be compatible with the return type of the procedure.

# Input and output

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.

## Input procedures

### Procedure read

Format: *read* (v1, …, vn)

It can have one or more arguments each of which is a variable of a certain type.

If v is a variable of type **char** then *read*(v) will read a character from input into v.

If v is of types **integer** or **real** then *read*(v) will read a sequence of characters from input text, parse it into a number and then assign to v. Separators and new line symbols are skipped and thrown away.

*It depends on operating system setting which symbol is used as a delimiter in real numbers.*

### Procedure readln

Format: *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.

## Output procedures

### Procedure write

Format: *write* (p1, …, pn)

Write procedure can have one or more arguments each of which has the format e , e:m or e:m:n where e is the value to be printed on the and m with n specify the width of the output. If the value of e needs less than m symbols for its representation then there will be spaces added to ensure that m symbols are printed. If m is omitted then the default value is used. Format e:m:n can be applied only for values of type **real**.

Arguments of *write* can be of **char**, **string**, **boolean**, **byte**, **integer** and **real** types:

* If e is **char** then *write* (e:m) will output m-1 spaces, and then the symbol in e. If m is omitted then default width is 1.
* If e is **string** then *write* (e:m) will print a string of symbols preceded by spaces such as the total length of the output is m. Default width is 4.
* If e is **boolean** then one of two words **true** or **false** will be printed preceded by spaces such as the total length of the output is m. Default width is 6.
* If e is **integer** then the decimal representation of the value will be printed. Default width is 20.
* If e is **real** then the decimal representation of the value will be printed. If n is omitted then a scientific representation will be used. If n is specified then a representation with fixed point is used with n digits after the point. Default width is 20.

### Procedure writeln

Format: *writeln* (v1, …, vn)

The procedure has the same functionality as *write*, but after the output of vn the end of line is sent to the output.

## File input and output

Standard IO functions read and write in contrast to Pascal allow working only with console. To work with files standard .NET library should be used. Example:

**module** Example102;

**import** System, System.IO;

**type**

SW = System.IO.StreamWriter;

**var** sw : SW;

fn : System.String;

**begin**

fn := "myfile.txt";

sw := **new** System.IO.StreamWriter(fn, **false**);

sw.Write("Text to be written to file");

sw.Close();

readln();

**end** Example102.

For more information about classes in the System namespace refer to MSDN.

# Examples on Zonnon

## Sorting an array

**module** Example91;

**const** MAX\_SIZE = 20; *(\* Size of the array \*)*

**type** ElementOfArray = **integer**;

**type** DefaultArray = **array** MAX\_SIZE   
 **of** ElementOfArray;

*(\* Declaration and creation of the array \*)*

**var** MyArray: DefaultArray;

*(\* Recursive implementation of a quick sort \*)*

**procedure** QuickSort(**var** a: DefaultArray; L, R: **integer**);

**var** i,j: **integer**; w,x: ElementOfArray;

**begin**

i := L;

j := R;

x := a[(L + R) **div** 2];

**repeat**

**while** a[i] < x **do** i := i + 1 **end**;

**while** x < a[j] **do** j := j - 1 **end**;

**if** i <= j **then**

w := a[i];

a[i] := a[j];

a[j] := w;

i := i + 1;

j := j - 1

**end**

**until** i > j;

**if** L < j **then** QuickSort(a, L, j) **end**;

**if** i < R **then** QuickSort(a, i, R) **end**;

**end** QuickSort;

**procedure** FillTheArray;

**var** i:**integer**;

**begin**

**for** i := 0 **to** MAX\_SIZE - 1 **do**

MyArray[i] := *abs*( 10 - i );

**end**;

**end** FillTheArray;

**procedure** PrintTheArray;

**var** i:**integer**;

**begin**

*writeln*("Array:");

**for** i := 0 **to** MAX\_SIZE - 1 **do**

*write*( MyArray[i]:2, ', ' );

**end**;

*writeln*;

**end** PrintTheArray;

**procedure** Execute;

**begin**

QuickSort(MyArray, 0, MAX\_SIZE - 1);

**end** Execute;

**begin**

*writeln*("Example 2.10 (Quick sort)");

FillTheArray;

PrintTheArray;

Execute;

PrintTheArray;

**end** Example91.

## Linked list example

Taking into consideration specialties of Zonnon programming language and .NET platform consider as an example a simplest implementation of a tack using linked list.

val

next

*Node*

root

val

next

*Node*

**nil**

Stack it a list where adding and removing of elements is performed from one end of the list only. Idea of the implementation using one directional linked list is depicted above.

**module** Stack;

**type** {*ref, public*} Node **= record**

val: **integer**;

next: Node;

**end** Node;

**var** root: Node;

**procedure** push(n: **integer**);

**var**

newnode: Node;

**begin**

newnode **:=** new Node;

newnode.val **:=** n;

newnode.next **:=** root;

root **:=** newnode

**end** push;

**procedure** pop():**integer**;

**var** t: **integer**;

**begin**

**if** root **=** **nil** **then** **return** 0 **end**;

t **:=** root.val;

root **:=** root.next;

**return** t

**end** pop;

**begin**

push(3);

push(7);

push(10);

*writeln*(pop());

*writeln*(pop());

*writeln*(pop())

**end** Stack.

# Module programming in Zonnon

Modular development and debugging in Zonnon is supported on the level of modules. To use within one module types and procedures of another module they should be imported using **import** section.

Import declaration:

**import** <declaration name> **as** <new synonym>;

Synonym can be omitted, and then the import looks like:

**import** <declaration>;

Example:

**module** Messages;

**procedure** {*public*} PrintHello;

**begin**

*writeln*("Zonnon Language");

**end** PrintHello;

**end** Messages.

**module** MainModule;

**import** Messages;

**begin**

Messages.PrintHello;

**end** MainModule.

Instruction **import** allows importing modules, objects and definitions. To get access to a function a scope where this function is declared should be imported.

# Mathematical extensions

Mathematical extensions of Zonnon programming language are constructions which help to develop mathematical applications. They include operations and functions on multidimensional matrices, indexing by means of ranges and vectors. Such structures lead to a more readable, natural and compact code in applications implementing linear algebra algorithms.

## Mathematical arrays

Array is an ordered set of elements that are all of the same type called base type. In order to use arrays in mathematical extensions, they must be declared with the *{math}* modifier. One can operate such arrays as multidimensional matrices using all available operations and functions for it. Moreover, assignment of two arrays is implemented as an element-wise copy if at least one array was declared as a mathematical one; otherwise assignment is a reference copy.

The following notations are used in this chapter:

*Vector* – one-dimensional array.

*Matrix* – two-dimensional array.

*Rank* – number of dimensions in an array.

*Subarray* – subset of elements of an array set by indices.

*Expression array* – expression used for creating or providing a set of values for an array. An expression array is written as a comma-separated list of expressions enclosed by braces *"["* and *"]"*. Rank of the constructed array will be equal the depth of nesting.

module ExampleM1;

var

a: array {math} 2, 3 of integer;

begin

a := [[1, 2, 3], [4, 5, 6]];

*(\* it is equal to*

*a[0, 0] := 1;*

*a[0, 1] := 2;*

*a[0, 2] := 3;*

*a[1, 0] := 4;*

*a[1, 1] := 5;*

*a[1, 2] := 6; \*)*

end ExampleM1.

## Indices

In Zonnon it is possible to define a subset of a mathematical array using complex indexing. If *a* is a *n*-dimensional mathematical array, a subset of its elements can be specified as

a[index0 , … , indexn-1].

Here *indexi* (which is responsible for a subset in *i*-th dimension) can be either a simple index, a range, a numerical vector, or a boolean vector. Arrays subsets defined in this way are treated as mathematical arrays and can be used in appropriate expressions.

Let us consider different type of indices in Zonnon mathematical extension.

*Simple index* is an expression of *integer* or *cardinal* type which value lies between *0* and *len(a, i) - 1*. Simple indices are the only indices which decrease rank of the resulting subset.

*Range* is a special type for constants and variables used in indices for mathematical arrays.

A constant range looks like

c..d by f

and stands for an ordered set {*c + i \* f : i ; 0 ≤ i \* f ≤ d – c*}. Here *c* {*0, … , len(a, i) – 1*}, *d* {*0, … , len(a, i) – 1*}, *c ≤ d* and *f* . Note that *f* can be omitted if it is equal to *1*. In this context *c* and *d* should be of *integer* or *cardinal* type and can be not specified when corresponding range is used as an index. In this case *c* will be equal to *0*, *d* to the length of the array's corresponding dimension reduced by one (or equal to l*en(a, i) – 1*).

The only allowed operation with variables of range type is assignment of another range variable or of a constant range.

module ExampleM2;

type

cardinal3d = array {math} \*,\*,\* of cardinal;

cardinal2d = array {math} \*,\* of cardinal;

var

a: cardinal3d;

b: cardinal2d;

r: range;

begin

a := new cardinal3d(4, 7, 5);

r := 1..3;

b := a[2, r, .. by 3];

*(\* it is equal to*

*b := a[2, 1..3 by 1, 0..4 by 3];*

*or*

*b := new cardinal2d(3, 2);*

*b[0, 0] := a[2, 1, 0];*

*b[0, 1] := a[2, 1, 3];*

*b[1, 0] := a[2, 2, 0];*

*b[1, 1] := a[2, 2, 3];*

*b[2, 0] := a[2, 3, 0];*

*b[2, 1] := a[2, 3, 3]; \*)*

end ExampleM2.

*Numerical vector index* is a vector of *integer* or *cardinal* type which elements should lie in the set {*0, … , len(a, i) – 1*}. The resulting subset in proper dimension will be of the same length as the index vector. Elements of an initial array are selected and concatenated in the same order, as they stand in the index vector. Hence, when using a numerical vector index, it is possible to choose any collection of the elements in any order, and one element can be chosen several times.

*Boolean vector index* is a vector of *boolean* type which must be of the length of *i*-th dimension of the array from which elements are selected (or equal to *len(a, i)*). The length of the result vector will be the same as the number of *true* elements in the boolean vector. Values corresponding to *true* in the index boolean vector are selected and those corresponding to *false* are omitted.

module ExampleM3;

type

matrix = array {math} \*,\* of real;

iVector = array {math} \* of integer;

bVector = array {math} \* of boolean;

var

a, b: matrix;

iva, ivb: iVector;

bv: bVector;

begin

a := new matrix(5, 3);

b := new matrix(7, 2);

iva := [4, 1];

ivb := [3, 5];

bv := [true, false, true];

b[ivb, ..] := a[iva, bv];

*(\* it is equal to*

*b := new matrix(2, 2);*

*b[3, 0] := a[4, 0];*

*b[3, 1] := a[4, 2];*

*b[5, 0] := a[1, 0];*

*b[5, 1] := a[1, 2]; \*)*

end ExampleM3.

## Operators

Base type of the resulting array will be equal to the type of the corresponding operation with scalars.

|  |  |  |  |
| --- | --- | --- | --- |
| Unary operators | | | |
| *Operator* | *Operand* | *Result* | *Meaning* |
| +, - | array of number | array of number | element-wise operation |
| ~ | array of boolean | array of boolean | element-wise negation |
| <A>, where <A> denotes a type | array of number | array of number of type <A> | element-wise type conversion |

|  |  |  |  |
| --- | --- | --- | --- |
| Binary operators | | | |
| *Operator* | *Operands* | *Result* | *Meaning* |
| := | array of number, array of number | - | element-wise assignment |
| := | array of number, expression array | - | element-wise assignment |
| +, -, \*, /, div, mod | array of number, scalar | array of number | element-wise operation with a scalar |
| +, -,.\*,./, div, mod | array of number, array of number | array of number | element-wise operation |
| +\*[[1]](#footnote-1) | array of number, array of number | scalar | (pseudo)scalar product |
| .>, .<, .>=, .<=, .=, .# | array of number, scalar | array of boolean | element-wise comparison with a scalar |
| .>, .<, .>=, .<=, .=, .# | array of number, array of number | array of boolean | element-wise comparison |
| >, <, >=, <=, =, # | array of number, scalar | boolean | generalized comparison with a scalar |
| >, <, >=, <=, =, # | array of number, array of number | boolean | generalized comparison |

|  |  |  |  |
| --- | --- | --- | --- |
| Unary matrix operators | | | |
| *Operator* | *Operand* | *Result* | *Meaning* |
| ! | 2d array of number | 2d array of number | matrix transposition |
| Binary matrix operators | | | |
| *Operator* | *Operands* | *Result* | *Meaning* |
| \* | 1d array of number, 1d array of number | scalar | scalar product |
| \* | 1d array of number, 2d array of number | 1d array of number | row-matrix or matrix-column product |
| \* | 2d array of number, 2d array of number | 2d array of number | matrix product |
| / | 1st operand: 1d or 2d array of number  2nd operand: 2d array of matrix (square matrix) | 1d or 2d array of number | solving linear equations (or matrix right division) |
| \ | 1st operand: 2d array of matrix (square matrix)  2nd operand: 1d or 2d array of number | 1d or 2d array of number | solving linear equations (or matrix left division) |

module ExampleM4;

type

matrix = array {math} \*,\* of real;

var

a, b, c, d: matrix;

k: integer;

begin

a := new matrix(4, 3);

b := new matrix(3, 4);

c := new matrix(4, 7);

d := (!a .\* (b + k)) \* c;

end ExampleM4.

Base type of a mathematical array can be any type. However, when using arrays in mathematical operations, corresponding operators should be defined or overloaded for the base types. For example, to sum two mathematical arrays (or apply operator “+”), their base type should be either integer, cardinal, real, or other types for which operator “+” is overloaded. Consider *ExampleM5* with complex numbers.

module ExampleM5;

*(\*defining new complex numbers type\*)*

type {public, ref} complex = object(r, i: real)

var {public}

re, im: real;

procedure {public} print;

begin

writeln(re, " + ", im, "i");

end print;

begin

re := r; im := i;

end complex;

*(\*overloading operator '+' for complex numbers\*)*

operator {public} '+' (z1, z2: complex): complex;

var res: complex;

begin

res := new complex(z1.re + z2.re, z1.im + z2.im);

return res;

end '+';

*(\*overloading operator '\*' for complex numbers\*)*

operator {public} '\*' (z1, z2 : complex) : complex;

var res : complex;

begin

res := new complex(z1.re \* z2.re - z1.im \* z2.im,

z1.re \* z2.im + z1.im \* z2.re);

return res;

end '\*';

*(\*overloading operator '<' as squared modulus comparison\*)*

operator {public} '<' (z1, z2: complex): boolean;

var res: boolean;

begin

res := z1.re \* z1.re + z1.im \* z1.im < z2.re \* z2.re + z2.im \* z2.im;

return res;

end '<';

var

c1, c2, c3: array {math} 3 of complex;

b: array {math} 3 of boolean;

z: complex;

i: integer;

begin

*(\*initialization\*)*

c1[0] := new complex(1.2, 4.5); c2[0] := new complex(2.3, 3.1);

c1[1] := new complex(0.3, 2.1); c2[1] := new complex(-1.5, -3.2);

c1[2] := new complex(-5.3, 4.5); c2[2] := new complex(4.4, 0.1);

c3 := c1 + c2;

writeln("c1 + c2 = ");

for i := 0 to len(c3) - 1 do

c3[i].print end;

*(\* c1 + c2 =*

*3,500000E+000 + 7,600000E+000 i*

*-1,200000E+000 + -1,100000E+000 i*

*-9,000000E-001 + 4,600000E+000 i\*)*

z := c1 +\* c2;

write("c1 +\* c2 = ");

z.print;

*(\* c1 +\* c2 = -2,869000E+001 + 2,923000E+001 i \*)*

b := c1 .< c2;

write("c1 < c2 = ");

for i := 0 to len(c3) - 1 do

write(b[i]) end;

*(\* c1 < c2 = false true false \*)*

end ExampleM5.

Note that in this example we overload necessary operators. For example, when using operator “+\*” for arrays of *complex* type, it is essential to overload operators “+” and “\*” for complex numbers.

## Functions

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| *Function* | *Parameter(s)* | *Result* | *Meaning* |
| abs | array of number | array of number | element-wise absolute value |
| min | array of number | scalar | minimal element in the array |
| max | array of number | scalar | mximal element in the array |
| sum | array of number | scalar | maximal element in the array |
| all | array of boolean | boolean | check whether all elements are *true* |
| any | array of boolean | boolean | check whether any element is *true* |
| find | scalar, 1d array | 1d array of number | indices where elements are equal to scalar |
| find | 1d array of boolean | 1d array of number | indices where elements are *true* |

module ExampleM6;

type

vector = array {math} \* of integer;

var

a: vector;

k: integer;

begin

a := new vector(5);

writeln("max = ", max(a));

writeln("there are ", len(find(k, a)), k, " in array a");

end ExampleM6.

# Object oriented programming

To be written

## Compositional Model

Zonnon uses a compositional inheritance model based on aggregation. Typically, an object (or module) is composed of a number of functional components, each of them presenting itself to clients in the form of an abstract *definition*. The set of definitions plus the object’s *intrinsic interface* (that is the set of all public elements of the object) constitutes the interface between the object and its clients.

Fig-composition-m

**Fig.1.** Zonnon Compositional Model Example: object MP3Device.

Figure 1a shows two functional components *MP3Player* and *FlashMemory*, where *MP3Player* consists of a definition and *FlashMemory* consists of a definition and a default implementation, and an object type *MP3Device* that is a composite of *MP3Player* and *FlashMemory*. Object type *MP3Device* exposes two facets *MP3Player* and *FlashMemory* plus its intrinsic (USB) interface which is shown on Figure 1b.

The relationship “object type implements definition” is a fundamental constituent of the Zonnon object model. It represents an obligation for an object type to provide a full implementation of the functionality promised by the definition. However, remembering that a corresponding implementation (if it exists) is automatically aggregated by the compiler, the object type needs to merely implement the missing parts and/ or to customize the default implementation by overriding. For example,

**object** MP3Device **implements** MP3Player, FlashMemory;

(\* the implementation of FlashMemory   
 is aggregated automatically \*)

**procedure** {public} Play **implements** MP3Player.Play;

**var** current: File;

**begin**

current := FirstFile;

**while** current # **nil** **do**

(\* Play song from current file \*);

current := NextFile

**end**

**end** Play;

**procedure** {public} ConnectUSBDevice(speaker: USBDevice);

**begin** … **end** ConnectUSBDevice;

**end** MP3Device.

The implementation of an object above uses following Zonnon definition of an interface to act as a Memory:

**definition** FlashMemory;

**procedure** UploadFile (s: File);

**procedure** FirstFile: File;

**procedure** NextFile: File;

**end** FlashMemory.

This definition could have a default implementation that might look like this:

**implementation** FlashMemory;

**var** ...

**procedure** UploadFile (s: File);

**begin** ...

**end** UploadFile;

**procedure** FirstFile: File;

**begin** ...

**end** FirstFile;

**procedure** NextFile: File;

**begin** ...

**end** NextFile;

**begin** ...

**end** FlashMemory.

Example how this MP3Device can be used from a module:

**module** MusicServer;

**import** MP3Device;

**var** device: MP3Device;

**begin**

device.SetUSBMode(USBMode.Active);

device.UploadFile(**new** File("song1.mp3"));

device.Play;

**end** MusicServer.

Refer to the Zonnon Language Report for more details on objects, definitions and implementations.

# Concurrency

## Activity

Simple concurrent systems can be built using just “active” modules. One or more activities are declared in the modules and then instantiated during module initialization:

**module** Writer;

**activity** A; *(\* declare activity \*)*

**begin**

writeln("A")

**end** A;

**begin**

**new** A; *(\* run the activity \*)*

writeln("B")

**end** Writer.

Note that the order in which the symbols "A" and "B" are displayed is non-deterministic. Similarly, activities can be declared within objects instead of modules.

In its most general form, an object (or module) body becomes active by acting as the root of an entire hierarchy of encapsulated activity. For example the well-known Quicksort algorithm is an activity that splits into locally independent sub-activities in a natural way. Roughly, the algorithm works like this:

Partition the array to be sorted;  
Sort left part; Sort right part

Clearly, the latter two actions are mutually independent and can therefore run concurrently in principle. Making use of child activities, the potential parallelism can easily be expressed in Zonnon. In fact, if

**var** a: array 1 + N of integer; (\* a[1] … a[n] used \*)

is the array to be sorted, a parallel version of Quicksort in Zonnon reads like this:

**activity** Sort (L, R: integer);

**var** i, j, t, x: integer;

**begin** {barrier}  
 i := L; j := R;

x := a[(L + R) **div** 2];

**repeat**

**while** a[i] < x **do** i := i + 1 **end**;

**while** x < a[j] **do** j := j - 1 **end**;

**if** i <= j **then**

t := a[i];

a[i] := a[j]; a[j] := t;

i := i + 1; j := j - 1

**end**

**until** i > j;

**if** L < j **then new** Sort(L, j) **end**;

**if** i < R **then new** Sort(i, R) **end**

**end** Sort;

Figure 2 below depicts a potential scenario of concurrency of the algorithm in the case of an initial state of a like this: a = 10 , 3 , 2 , 5 , 4 , 7 , 8 , 1 ,11 , 6 , 9 ,12.

Qsortpotentialwm

Figure 2. Potential concurrency of the algorithm.

Note that two new instances of the *Sort* activity are created recursively after the partitioning phase of each running instance of the *Sort* activity. Their task consists of sorting one of the two parts (left or right). Obviously, in any real sorting scenario, a large number of short-living activities are generated by the above recursion. In view of the typically limited number of processors available, it is hardly advisable to create a new thread for each activity. A much better strategy is using recycled threads from some existing runtime thread pool. In detail, whenever a new activity needs to be created, the executing thread has the choice of either creating a request that waits for a pool thread to process it at a later time or handling the task of the new activity on its own. The entire sorting machinery is launched by simply calling

**new** Sort (0, N-1)

Figure 3 depicts a possible scenario of parallel execution for the previous example. In practice, it is advisable to limit the overhead in the above code snippet by adding some logic that cuts the recursion when the number of elements still to be sorted reaches a certain lower bound.

Qsortpoolwm

Figure 5. Parallel Quick Sort Running on a Thread Pool.

## Shared objects

Zonnon objects can be made sharable by marking their type with a {protected} modifier, in which case execution within them is limited to only a single activity and thus they essentially become a Hoare monitor, however in a refined version that supports shared locks. Individual methods of a protected object type can be declared as {shared}, providing sharing of the object’s lock and read access, but not write access, to the state within the protected object.

Example:

**object** {protected}SharedObject;

**var** value, hidden: integer;

**procedure** {public} Exchange;

**var** t:integer;

**begin**

value, hidden := hidden, value

**end** Exchange;

**procedure** {public, shared} Get: integer;

**begin**

**return** value;

**end** Get;

**end** SharedObject.

In Zonnon only a single scheduling primitive is used – the await statement:

**await** cond;

*cond* is an object-local Boolean expression that defines the precondition for the continuation of execution.

## Communication of activities

Send and receive commands have different syntax inside and outside activities.

Sending tokens to activities.

**activity** A;

**begin**

...

**end** A;

**var** a: A; s: string;

**begin**

a := new A;

a ( 1024 ); (\* Sending integer constant \*)

a ( “Hello”); (\* Sending string constant \*)

a ( “My name is ”, s, 17 ); (\* Sending string constant,

reference to the string object

and integer constant \*)

**end**;

Compiler doesn’t check sending arguments.

Receiving tokens from activities. Received data is implicitly cast to destination variable type.

**activity** A;

**begin**

...

**end** A;

**var** a: A; s: string; i: integer; a, b: real; c: object;

...

a := new A;

s := a ( ); (\* Receive token and cast to string \*)

i := a ( ); (\* Receive token and cast to integer \*)

a, b, c := a ( ); (\* Receive three tokens and cast

the first two to real and the third to object \*)

**...**

In case of type cast error standard system exception will be thrown.

It is possible to combine sending and receiving tokens in one statement.

**activity** A;

**begin**

**...**

**end** A;

**var** a: A; s: string; i: integer; a, b: real; c: object;

**...**

a := new A;

i := a ( “Hello” );

(\* Send string “Hello” and then receive integer token \*)

a, b, c := a ( 1, “a” ); (\* Send constant 1 and constant “a” then

receive three tokens and cast the first two

to real and the third to object \*)

**...**

Receiving tokens inside activities. Accepted tokens are cast implicitly.

**activity** A;

**var** s: string; a,b,c: integer;

**begin**

**accept** s;

**accept** a, b, c; (\* Accept three tokens and cast to integer \*)

**end** A;

Sending tokens from activity.

**activity** A;

**var** s: string; a,b,c: integer;

**begin**

**return** s; (\* Send string \*)

**return** a, b, c; (\* Send three integer \*)

**end** A;

Activity arguments.

**activity** A( a, b, c: integer );

**begin**

**end** A;

**var** a: A;

...

a := **new** A( 10, 10, 10 );

...

Compiler does type checking for this arguments like for a procedure call during the compilation time. Results of the example above is equal to:

**activity** A;

**var** a, b, c: integer;

**begin**

**accept** a, b, c;

**end** A;

**var** a: A;

...

a := **new** A;

a( 10, 10, 10 );

## Protocols

The definition of an activity can include a dialog that is a formal syntax specification of a communication protocol in EBNF. It is represented as a special enumeration type which defines the alphabet of terminal tokens and a list of EBNF productions that define the syntax. Textually last production defines the root of the syntax.

**protocol** P = (

A, B, C, dialog = A | B | C

);

**activity** A **implements** P;

**var** p: P;

**begin**

**accept** p;

**if** p = P.A **then**

(\* Command A \*)

**elsif** p = P.B **then**

(\* Command B \*)

**else**

(\* Command C \*)

**end**

**end** A;

Example below illustrates using is operator with protocol type:

**protocol** P = (

START\_TEXT, MODIFIER1, MODIFIER2, END\_TEXT,

P = START\_TEXT { string | MODIFIER1 | MODIFIER2 } END\_TEXT

);

**activity** A **implements** P;

**var** request: **object**;

**procedure** processCmd( cmd: P );

**end** processCmd;

**procedure** addTextLine( s: **string** );

**end** addTextLine;

**begin**

**repeat**

**accept** request;

**if** p **is** P **then**

processCmd( P( request ) )

**else** (\* It is a string \*)

addTextLine( **string**( request ) )

**end**

**until** (request **is** P) & (request = P.END\_TEXT);

**end** A;

Protocols simplify coding. An example below illustrates benefits from using a protocol.

Version without using a protocol:

**type** E = (A, B, C, D, E);

**activity** A;

**var** p: E;

**begin**

**accept** p;

**if** p = E.A **then**

(\* Command A \*)

**elsif** p = E.B **then**

(\* Command B \*)

**elsif** p = E.C **then**

(\* Command C \*)

**else**

(\*

For robust code

we have to check this

situation, because E.D and E.E

can come   
 \*)

**end**

**end** A;

With a protocol there is no need in additional checking for robust code.

**protocol** P = (

A, B, C, D, E,

dialog = A | B | C

);

**activity** A **implements** P;

**var** p: P;

**begin**

**accept** p;

**if** p = P.A **then**

(\* Command A \*)

**elsif** p = P.B **then**

(\* Command B \*)

**else**

(\* Command C \*)

**end**

**end** A;

If caller sends neither A nor B nor C the protocol implementation will throw system exception (during the execution time).

During the compilation time it is not possible to check types of arguments of send and receive operators.

Next example will throw type cast exception:

**activity** A;

**var** i: integer;

**begin**

**accept** i;

**end** A;

**procedure** R;

**var** a: A;

**begin**

a := **new** A;

a(“Sending a string”);

**end** R;

Using protocol allows distinguish data transition errors and type cast errors during execution time. Next example will throw protocol exception:

**protocol** P = (

dialog = integer

);

**activity** A **implements** P;

**var** i: integer;

**begin**

**accept** i;

**end** A;

**procedure** R;

**var** a: **activity** {P};

**begin**

a := **new** A;

a(“Sending a string”);

**end** R;

Exception will be thrown both on client and server side. You don’t have to catch it on server side if you don’t need it. If exception is not catched on server side it will just terminate the session. On client side if the exception is not catched the application will be terminated.

This client can be rewritten like this:

**procedure** R;

**var** a: **activity** {P};

**begin**

**do**

a := **new** A;

a(“Sending a string”)

**on** ProtocolMismatch **do**

writeln(“ProtocolError”)

**end**

**end** R;

## Constellation of self-active objects

Objects are called active or self active if their declarations specify one or more activities.

An activity is an asynchronously executed coroutine. Activities can be declared wherever procedures can be declared, and they resemble procedures syntactically. An activity is instantiated, possibly with some initial parameters, and then it executes as a separate thread, where an *active link* with its creator is still preserved. If a callee activity is instantiated within a scope of another object with the intention to communicate, it is called an agent activity. An activity launched in the scope of the same object is called a local activity. Local activities may or may not communicate with its caller and thus can be launched with or without preserving an active link. Both agent and local activities are *child* activities.

aosystem.emf

Figure 4. A system of self-active objects.

On Figure 4 (active) Object 1 contains four encapsulated activities. Object 2 is similar to Object 1 but it has spawned an agent activity in Object 3. Object 4 takes this a step further. It maintains three active links to objects 5, 7 and 8. This constellation is turned into an entire network of links by the caller activity in Object 5 that creates a callee (agent) activities in objects 6 and 8. The structure defined by Objects 4, 8 and 7 illustrates an (active) chain of interoperating objects.

A full hierarchy of child activities may be created from within the parent scope. Each child activity is instantiated by any parent within its scope of visibility by using **new** operator.

A leading {fence} modifier in the parent scope can be used to place an execution *barrier* to prevent continuation before all *child activities* have terminated.

The example in Listing 1 shows two communicating activities: one within program module ParkingLot and the other within an instance of object Car that approaches the ParkingLot. Car has an intrinsic activity that implements the navigation into ParkingLot. This activity instantiates an agent activity within the scope of ParkingLot. The agent activity is defined within the scope of ParkingLot and therefore has access to its fields. These two activities run as separate threads and communicate via message passing using the elements of the enumeration type Service as messages.

Listing 1. Communicating activities.

**module** ParkingLot;

**type** {public}Service=(Request, Allow, Reject, Enter, Leave);

**activity** {public} Serve;

**var** cmd: Service;

**begin**

**accept** cmd; *(\* Request \*)*

**if** bookSpace **then**

**return** Service.Allow;

**accept** cmd; *(\* Enter or Leave \*)*

**if** cmd = Service.Enter **then** *(\* car stays in the lot \*)*

**accept** cmd *(\* Leave \*)*

**end**;

freeSpace

**else**

return Service.Reject

**end**

**end** Serve;

**end** ParkingLot.

**object** Car;

**import** ParkingLot **as** PL;

**activity** {public} Drive;

**var** parking: PL.Serve;

ans: PL.Service;

**begin**

parking := **new** PL.Serve;

ans := parking(PL.Service.Request);

**if** ans = PL.Service.Allow **then**

parking(PL.Service.Enter);

*(\* stay in the parking lot \*)*

parking(PL.Service.Leave)

**end**

**end** Drive;

**end** Car.

Note that caller activities refer to callees via a variable of the callee`s type, while the callers are anonymous from the callee`s point of view. Reference ans to activity PL.Service is used for communication between activities.

Admissible values for communications are values of primitive data types, references, new active links (active links that have been created but not used yet).

Communications in the previous example may be specified using protocols. A protocol type is declared in terms of an enumeration of terminal tokens and a syntax in Extended Backus Naur Form (EBNF). The caller and the callee exchange tokens according to the protocol specified by the callee.

**module** ParkingLot;

**protocol** Service = (

Request, Allow, Reject, Enter, Leave,

park = [Enter] Leave,

dialog = Request (?Allow park | ?Reject)

);

**activity** {public} Serve **implements** Service;

**var** cmd: Service;

**begin**

**accept** cmd; *(\* Request \*)*

**if** bookSpace **then**

**return** Service.Allow;

**accept** cmd; *(\* Enter or Leave \*)*

**if** cmd = Service.Enter **then** *(\* car stays in the lot \*)*

**accept** cmd *(\* Leave \*)*

**end**;

freeSpace

**else**

return Service.Reject

**end**

**end** Serve;

**end** ParkingLot.

**object** Car;

**import** ParkingLot **as** PL;

**activity** {public} Drive;

**var** parking: **activity** {PL.Serveice}; (\* Any activity that implements this protocol\*)

ans: PL.Service;

**begin**

parking := **new** PL.Serve;

ans := parking(PL.Service.Request);

**if** ans = PL.Service.Allow **then**

parking(PL.Service.Enter);

*(\* stay in the parking lot \*)*

parking(PL.Service.Leave)

**end**

**end** Drive;

**end** Car.

In the example above the protocol Service defines an enumeration of tokens Request, Allow, Reject, Enter and Leave which are used for communication. The communication protocol is defined via two EBNF productions park and dialog. The production park defines the optional sending of token Enter and then the mandatory sending of token Leave from caller to callee. The production dialog obliges the caller to send Request token and then the callee to reply with either Allow or Reject. If the callee replies with Allow then the continuation of the dialog is defined by production park.

Grammars that define protocol must be deterministic and context-free. Communications in repetitions must be bidirectional. Since send operations are non-blocking by definition they may lead to unpredictable buffer load unless infinite repetitions of send operations are bound. As an example, constructions like { string } are not allowed. However both { string ?OK } and { string string ?OK } are allowed.

Figure 5 summarizes our previous explanations and Zonnon’s object model. It shows a code template of a “fully-fledged” active Zonnon object.

Activeobjectwm

Figure 5. Active object.

# Lorenz attractor example

This example consists of 5 modules. To compile these modules .NET libraries System.Drawing.dll and System.Windows.Forms.dll should be referenced.

## Dynamics3D module

This module …

module Dynamics3D;

import System;

type {public, ref} Vector = array {math} 3 of real;

definition {public} Particle;

var {public}

dt, t, tMax : real;

x : Vector;

var {private}

x1, k1, k2, k3, k4 : Vector;

procedure {public} f (t0 : real; x0 : Vector): Vector;

procedure {public} Connect (x0, y0 : Vector);

end Particle;

end Dynamics3D.

## Graph3D module

This module …

module Graph3D;

import System.Math as Math, Dynamics3D;

const arc = 0.01745;

type {public, ref} Vector = array {math} 4 of real;

type {public, ref} Matrix = array {math} 4,4 of real;

procedure {public} Hom (x : Dynamics3D.Vector): Vector;

var

VRes : Vector;

begin

VRes[0..2] := x;

VRes[3] := 1.0;

return VRes;

end Hom;

procedure {public} Unhom (x : Vector) : Dynamics3D.Vector;

var

VRes : Dynamics3D.Vector;

begin

VRes := x[0..2] / x[3];

return VRes;

end Unhom;

procedure {public} Trans (v1, v2, v3 : real) : Matrix;

var

MRes : Matrix;

i : integer;

begin

MRes := [[1., 0., 0., v1],

[0., 1., 0., v2],

[0., 0., 1., v3],

[0., 0., 0., 1.]];

return MRes;

end Trans;

procedure {public} RotX (phi : real) : Matrix;

var

c, s : real;

MRes : Matrix;

begin

c := Math.Cos(arc\*phi);

s := Math.Sin(arc\*phi);

MRes := [[1., 0., 0., 0.],

[0., c, -s , 0.],

[0., s, c , 0.],

[0., 0., 0., 1.]];

return MRes;

end RotX;

procedure {public} RotY (phi : real) : Matrix;

var

c, s : real;

MRes : Matrix;

begin

c := Math.Cos(arc\*phi);

s := Math.Sin(arc\*phi);

MRes := [[c , 0., s , 0.],

[0., 1., 0., 0.],

[-s, 0., c , 0.],

[0., 0., 0., 1.]];

return MRes;

end RotY;

procedure {public} Proj (left, right, bottom, top, near, far : real) : Matrix;

var

MRes : Matrix;

begin

MRes := [

[2\*near / (right-left), 0., (right+left) / (right-left), 0.],

[0., 2\*near / (top-bottom), (top+bottom) / (top-bottom), 0.],

[0., 0., -(far+near) / (far-near), -2\*far\*near / (far-near)],

[0., 0., -1., 0.]];

return MRes;

end Proj;

end Graph3D.

## Lorenz3D module

This module …

module Lorenz3D;

import

System.Drawing.Color as Color,

System.Drawing.Pen as Pen,

System.Drawing.Pens as Pens,

System.Drawing.Graphics as Graphics,

System.Math as Math,

XYDrawing, Dynamics3D, Graph3D;

const VW = 600; VH = 600; asp = 800.0/700.0;

var {public}

MainCamera: Camera;

MainGraphics: Graphics;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* Camera \*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

type {public, ref} Camera = object (zm0, d0, azim0, elev0,

left0, right0, bottom0, top0, near0, far0 : real)

var

M : Graph3D.Matrix;

left, right, bottom, top, near, far, zm, d, azim, elev : real;

procedure {public} Move (dazim, delev, dd : real);

begin

azim := azim + dazim;

elev := elev + delev;

d := d + dd;

M := Graph3D.Proj(left, right, bottom, top, near, far)

\* Graph3D.Trans(0.0, 0.0, -d) \* Graph3D.RotX(elev)

\* Graph3D.RotY(-azim) \* Graph3D.Trans(0.0, 0.0,- zm)

end Move;

procedure {public} GetAzim : real;

begin

return azim;

end GetAzim;

procedure {public} GetElev : real;

begin

return elev;

end GetElev;

procedure {public} GetD : real;

begin

return d;

end GetD;

procedure {public} Map (x : Dynamics3D.Vector) : Dynamics3D.Vector;

begin

return Graph3D.Unhom(M \* Graph3D.Hom(x))

end Map;

begin

zm := zm0;

d := d0;

azim := azim0;

elev := elev0;

left := left0;

right := right0;

bottom := bottom0;

top := top0;

near := near0;

far := far0;

Move(0.0, 0.0, 0.0);

end Camera;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* LorenzParticle \*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

type {public, ref} LorenzParticle = object (col0 : Color; p0, r0, c0, dt0, tMax0 : real;

x0 : Dynamics3D.Vector) implements Dynamics3D.Particle

var

col: Color;

p, r, c: real;

var {public}

P : Pen;

procedure {public} f (t0 : real; x0 : Dynamics3D.Vector): Dynamics3D.Vector implements Dynamics3D.Particle.f;

var

VRes : Dynamics3D.Vector;

begin

VRes := [-p\*x[0] + p\*x[1],

-x[0]\*x[2] + r\*x[0] - x[1],

x[0]\*x[1] - c\*x[2]];

return VRes;

end f;

procedure {public} Connect (x0, y0 : Dynamics3D.Vector) implements Dynamics3D.Particle.Connect;

begin

MainGraphics.DrawLine(P, integer(Math.Floor((MainCamera.Map(x0)[0] + 1)\*0.5\*VW + 0.5)),

integer(Math.Floor((MainCamera.Map(x0)[1] + 1)\*0.5\*VH + 0.5)),

integer(Math.Floor((MainCamera.Map(y0)[0] + 1)\*0.5\*VW + 0.5)),

integer(Math.Floor((MainCamera.Map(y0)[1] + 1)\*0.5\*VH + 0.5)));

end Connect;

procedure {public} Process;

var

begin

t := 0.0;

while t <= tMax do

k1 := f(t, x);

k2 := f(t + dt/2, x + dt/2 \* k1);

k3 := f(t + dt/2, x + dt/2 \* k2);

k4 := f(t + dt, x + dt \* k3);

x1 := x + dt/3 \* (1/2 \* k1 + k2 + k3 + 1/2 \* k4);

Connect(x, x1);

x := x1;

t := t + dt;

end;

end Process;

begin

P := new Pen(col0);

col := col0;

p := p0;

r := r0;

c := c0;

dt := dt0;

tMax := tMax0;

t := 0.0;

x := x0;

end LorenzParticle;

procedure {public} Initialize;

begin

MainCamera := new Camera(25.0, 120.0, 20.0, 20.0, -0.27, 0.27,

-asp\*0.27, asp\*0.27, 1.0, 1000.0);

XYDrawing.Open(VW, VH, MainGraphics);

end Initialize;

procedure {public} Simulate;

var

lp: LorenzParticle;

xcur: Dynamics3D.Vector;

p, r, c, dt, tMax: real;

begin

MainGraphics.Clear(Color.White);

XYDrawing.DrawAxis;

p := 10.0;

r := 28.0;

c := 8/3.0;

dt := 0.001;

tMax := 100.0;

xcur := [15., -10., 10.];

lp := new LorenzParticle(Color.Red, p, r, c, dt, tMax, xcur);

lp.Process;

xcur := [16., -11., 9.];

lp := new LorenzParticle(Color.Green, p, r, c, dt, tMax, xcur);

lp.Process;

xcur := [17., -12., 8.];

lp := new LorenzParticle(Color.Blue, p, r, c, dt, tMax, xcur);

lp.Process;

XYDrawing.DrawAxis;

end Simulate;

end Lorenz3D.

## XYDrawing module

This module …

module XYDrawing;

import

System,

System.Drawing as Drawing,

System.Drawing.Size as Size,

System.Drawing.Point as Point,

System.Drawing.Rectangle as Rectangle,

System.Drawing.Color as Color,

System.Drawing.Graphics as Graphics,

System.Windows.Forms.Control as Control,

System.Windows.Forms.KeyPressEventHandler as KeyPressEventHandler,

System.Windows.Forms.KeyPressEventArgs as KeyPressEventArgs,

System.Windows.Forms.Application as Application,

System.Windows.Forms.Form as Form,

System.Windows.Forms as Forms,

System.String as String,

System.Math as Math,

Lorenz3D, Graph3D, Dynamics3D;

var {public}

form\_init : boolean;

var {public}

xsize, ysize : integer; (\* Window size. \*)

dbx, dby : integer;

currentStr : String;

var

df : DrawingForm;

startButton : Forms.Button;

downButton : Forms.Button;

upButton : Forms.Button;

leftButton : Forms.Button;

rightButton : Forms.Button;

pageDownButton : Forms.Button;

pageUpButton : Forms.Button;

pictureBox : Forms.PictureBox;

pbGraphics : Drawing.Graphics;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* DrawingForm \*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

type {public, ref} DrawingForm = object implements Form

var {public}

dsize : Size;

rect : Rectangle;

point : Point;

keyPressEventHandler : KeyPressEventHandler;

procedure {private} AddButtonsToMove;

var

width: integer;

begin

width := 40;

leftButton := new Forms.Button;

leftButton.Text := "left";

leftButton.Width := width;

leftButton.Left := width;

leftButton.Top := Height - leftButton.Height - 60;

leftButton.add\_Click(onLeftButtonClick);

Controls.Add(leftButton);

rightButton := new Forms.Button;

rightButton.Text := "right";

rightButton.Width := width;

rightButton.Left := 3 \* width;

rightButton.Top := Height - rightButton.Height - 60;

rightButton.add\_Click(onRightButtonClick);

Controls.Add(rightButton);

downButton := new Forms.Button;

downButton.Text := "down";

downButton.Width := width;

downButton.Left := width + width;

downButton.Top := Height - downButton.Height - 60 + 25;

downButton.add\_Click(onDownButtonClick);

Controls.Add(downButton);

upButton := new Forms.Button;

upButton.Text := "up";

upButton.Width := width;

upButton.Left := width + width;

upButton.Top := Height - upButton.Height - 60 - 25;

upButton.add\_Click(onUpButtonClick);

Controls.Add(upButton);

pageDownButton := new Forms.Button;

pageDownButton.Text := "+";

pageDownButton.Width := width;

pageDownButton.Left := width - 15;

pageDownButton.Top := Height - pageDownButton.Height - 60 - 35;

pageDownButton.add\_Click(onPageDownButtonClick);

Controls.Add(pageDownButton);

pageUpButton := new Forms.Button;

pageUpButton.Text := "-";

pageUpButton.Width := width;

pageUpButton.Left := 3\*width + 15;

pageUpButton.Top := Height - pageUpButton.Height - 60 - 35;

pageUpButton.add\_Click(onPageUpButtonClick);

Controls.Add(pageUpButton);

end AddButtonsToMove;

begin (\* DrawingForm constructor. \*)

rect.X := dbx;

rect.Y := dby;

rect.Width := 0;

rect.Height := 0;

DesktopBounds := rect;

dsize.Width := xsize;

dsize.Height := ysize;

ClientSize := dsize;

Text := "Lorenz Attractor";

BackColor := Color.White;

ShowInTaskbar := true;

(\*AutoScale := false;\*)

AutoScaleMode := Forms.AutoScaleMode.Font;

AllowDrop := false;

DoubleBuffered := true;

form\_init := true;

startButton := new Forms.Button;

startButton.Text := "Simulate";

startButton.Left := Width div 2 - startButton.Width div 2;

startButton.Top := Height - startButton.Height - 60;

startButton.add\_Click(onStartButtonClick);

Controls.Add(startButton);

AddButtonsToMove;

end DrawingForm;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

procedure {public} DrawAxis;

var

p: Drawing.Pen;

vw, vh: integer;

azim, elev, d, temp: integer;

begin

vw := 90;

vh := 90;

pbGraphics.FillRectangle(Drawing.Brushes.White, xsize - 145, ysize - 128, 145, 128);

p := new Drawing.Pen(Color.Black);

azim := integer(Lorenz3D.MainCamera.GetAzim()) mod 90;

if azim < 0 then azim := azim + 90; end;

elev := integer(Lorenz3D.MainCamera.GetElev()) mod 180;

d := integer(Lorenz3D.MainCamera.GetD());

pbGraphics.DrawEllipse(p, xsize - 110,

ysize - 110 + vh div 2 - 45 + azim div 2,

vw, 90 - azim);

temp := integer((vw div 2) \* (Math.Atan(real(elev) / 180. \* (Math.PI \* 2))));

pbGraphics.DrawLine(p, xsize - 110 + vw div 2,

ysize - 110 + vh div 2,

xsize - 110 + vw div 2 + integer(Math.Sqrt(4000 - temp \* temp)),

ysize - 110 + vh div 2 + temp);

pbGraphics.DrawLine(p, xsize - 140, ysize - 110, xsize - 140, ysize - 110 + vh);

if d >= 1000 then

pbGraphics.DrawLine(p, xsize - 138, ysize - 110, xsize - 142, ysize - 110);

else if d <= 0 then

pbGraphics.DrawLine(p, xsize - 138, ysize - 110 + vh, xsize - 142, ysize - 110 + vh);

else

pbGraphics.DrawLine(p, xsize - 138,

ysize - 110 - integer(real(d) / 1000. \* vh) + vh,

xsize - 142,

ysize - 110 - integer(real(d) / 1000. \* vh) + vh);

end;

end;

end DrawAxis;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

procedure {public} onStartButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.Simulate();

end onStartButtonClick;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

procedure {public} onDownButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(0.0, -10.0, 0.0);

DrawAxis;

end onDownButtonClick;

procedure {public} onUpButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(0.0, 10.0, 0.0);

DrawAxis;

end onUpButtonClick;

procedure {public} onLeftButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(-1.0, 0.0, 0.0);

DrawAxis;

end onLeftButtonClick;

procedure {public} onRightButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(1.0, 0.0, 0.0);

DrawAxis;

end onRightButtonClick;

procedure {public} onPageDownButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(0.0, 0.0, -10.0);

DrawAxis;

end onPageDownButtonClick;

procedure {public} onPageUpButtonClick( sender:object; args : System.EventArgs);

begin

Lorenz3D.MainCamera.Move(0.0, 0.0, 10.0);

DrawAxis;

end onPageUpButtonClick;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

procedure {public} Open(wx, wy : integer; var GraphicObj : Graphics);

begin

xsize := wx;

ysize := wy;

df := new DrawingForm();

GraphicObj := df.CreateGraphics();

pbGraphics := GraphicObj;

DrawAxis;

Forms.Application.Run(df);

end Open;

begin

form\_init := false;

dbx := 50; dby := 50;

end XYDrawing.

## Main module

This module …

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Lorenz attractor simulation (using RK4 method).

The Lorenz attractor, named for Edward N. Lorenz, is a fractal structure

corresponding to the long-term behavior of the Lorenz oscillator.

The equations that govern the Lorenz oscillator are:

dx/dt = p(y - x)

dy/dt = x(r - z) - y

dz/dt = xy - cz

The following parameters' values are used in simulation:

p := 10;

r := 28;

c := 8/3;

Method description: http://en.wikipedia.org/wiki/Lorenz\_attractor

Author: Jurg Gutknecht

Implementation in Zonnon: Nina Gonova

Last change: 08.07.2009

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

module Main;

import

Lorenz3D;

begin

Lorenz3D.Initialize();

end Main.

1. (Pseudo)scalar product of two n-dimensional arrays means [↑](#footnote-ref-1)