


[RUSSIAN\(Русский\)](#)

[ENGLISH](#)
[Change color scheme](#)

This function requires cookies.
If the browser does not accept cookies,
the light theme is used always.

[Content](#)

Introduction

A programming language AL/IV is a
 minimal
 imperative
 object oriented
 platform independent
 high-level programming language
 with static types,
claiming to a very high level of safety and stability
 of a controllable level of code protection.

Important semantic things of the language are:

NONE-objects instead of NULLs

- fields of NONE objects are always equal to NONE.
- Writing to fields of **NONE**-objects is ignored.
- Reading of fields gives always zeroes and **NONE**-objects of correspondent types.
- **NONE**-objects are returned in a case when arrays are accessed out of the array bounds,
- and when the object referenced was lost.
- All object variables are always initialized with **NONE**
- and never have value **NULL**.

Separating references to objects onto strong (owning) and weak_ (using) references

- Strong references can not be used to create closed link chains between objects.
- Garbage collection is not necessary at all.
- All objects are automatically destroyed when its usage (by strong references) counters becomes zero.
- when the object is destroyed, all weak references onto it immediately become referencing to **NONE** objects of corresponding class.
- New objects must be addressed either a strong pointer reference or an existing object should be specified becoming an owner of the creating object.

Total thread data isolation from other simultaneously executing threads

- Special mechanisms to provide exclusive access to shared data are not necessary.
- A thread can access only its own objects and even does not see objects of other threads.
- There are not possible deadlocks due to data access conflicts.
- There are no global variables so there are no problems with simultaneous access them from different threads.

Detection of infinitive loops and endless recursions

- Loops are possible only in form **FOR var IN arr[] : ...** (there are no loops like **WHILE / REPEAT-UNTIL**). There are loops **FOR, INFINITE**, but its usage is restricted hardly. Any loop **FOR** is finished obligatory, so infinitive looping become impossible.
- Recursive functions always are marked with a modifier **RECURSIVE**, and a recursion deep is controlled: in case of danger of too deep recursion an immediate return to the first level of the recursion in the calls chain is performed and it returns **NONE** value.

PUSH statement for variables and objects

- In a **PUSH** statement for a variable its value is stored and new value can be assigned.
- On a block end, stored value of a variable is restored always.

Parameters passed by value, can not be changed in a function

- Reusing input parameters to store intermediate values is not allowed.
- Value of a scalar parameter can not be changed until the function ends.
- For an array parameter it is allowed to change items, and for dynamic arrays - adding and deleting items.

"Read only" fields and module variables

- Simplifies creating of "properties" which can be modified only by the owner (class and its descendants).

Enumerations and collections

- These are very similar to enumerations and sets in Pascal-like languages.
- These allow to work with named constants and flags.
- It is possible to use Boolean arrays with such enumerations as indexes to implement collections of flags.
- Enumeration items are included into apostrophes.
- The **CASE** statement with an enumeration as a condition requires to list all possible values in its branches. If an enumeration used in **CASES** was extended with new values, the compiler is complaining on all the **CASE** statements where such new values are not yet listed.

There are no handling exception instruments

- Passing control to a top level recursive function when a maximal allowed level of a recursion is achieved is performed only to speed up malfunction detection and to prevent stuck of execution in case of a big amount of nested loops.
- Most of errors are ignored by replacing a supposed result with the **NONE** value.
- In case of errors (working with files, exceptions in native code, etc.) these are collecting (in a system array of errors) and later can be handled in the code.

Testing engine embedded to the language

- **TEST** functions allows to test code.
- Covering of the code by tests is measured by the compiler.
- A unit insufficiently covered by tests must be marked by the modifier **UNTESTED**.

It is possible to redefine arithmetic operations

- Only arithmetic operations can be redefined (+, -, *, /).
- Any function using redefined operators, must declare in its header a list of classes where such operators are from.

Important syntax things of the language are:**One class - one module**

- Each source text file contains a single class definition.
- All public names and function parameters have names starting from uppercase letter, all the hidden names and local variables are starting from a lowercase letter.
- Underlining at the end of a name is used only for object fields of class which are weak references to the object.

Double naming of types, variables, functions, constants

- Most of objects (variables, functions, data types, enumeration items) have several names.
- In declarations, full name is separated onto parts via a '|'. The first part before '|' is representing a short name of an item.
- Additionally, other variants of the name can be specified after the separator symbol '|'. The first symbol in chains should be the same (though the letter case can be different). E.g.,
Name|_first|_variant || name|
- All named things (excluding **FOR** loop variables) should have long names (at least 8 characters).
- Language keywords are written in the uppercase and can not be used to name functions or variables.

Names of classes, enumerations and records are always written in figure brackets

- There are 5 simple data types: **BOOL**, **BYTE**, **INT|EGER**, **REAL**, **STR|ING**.
- Classes and enumerations are named using figure brackets, e.g. **{My|_class}**.
- Classes have names in figure brackets starting from uppercase. Enumeration and record names are always starting from a lowercase letter.
- Type like "char" is not present (**STR** is used).
- Implicit conversion is allowed only for converting **BYTE** to **INT**, and **INT** to **REAL**.

Source code string length restriction

- Source code should not contain lines with length greater than 80 characters (not taking into account comments and ending spaces, and accounting tabulating symbols as spaces).
- The rule is not affecting lines after the final directive **END** and to lines located in long comments ***/ ... */**.

A rule to continue line of code

- There is no a special symbol to end a simple statement. The semicolon symbol ';' is used to finish block statements **CASE**, **FOR** and so on.
- A statement usually occupies a single line of code.

- A line of code is continuing in the next line if it is ending with one of character ',', '(', '['
- or if the next line is not empty and do not start new statement, i.e. it is not start from '{', '<', '[' or a letter,
- and it is not ending a block or a function, i.e. it is not starting from symbols ';', '.'

Static function with a single parameter can be called always in a postfix form

- Calling in form **sin(x)** can be always replaced with **x.sin()**

Arrays are always used together with its square brackets: A[]

- Only single dimension arrays are present.
- An array can be either dynamic or a fixed.
- For fixed arrays enumerations can be used as indexes (as data types).

A line of code contains only a single statement

- Symbol ';' is used to end a block statement (**CASE**, **FOR**, **PUSH**, **DEBUG**, **SILENT**).
- Symbol '.' is ending a function, a enumeration, a constant block or an import list.
- Ending symbol ';' or '.' must be the last such symbol in a line (so, two semicolons are not possible but ';' '.' is allowed at the end of the last line in a function).
- Any simple statement (except **BREAK**, **CONTINUE** or **STOP**) can have a statement '==>' attached (return from a function statement).

Restrictions on a code quality:

- It is allowed not more then 3 nested levels of block statements **CASE** / **FOR** (though this is not concerning statements **PUSH**, **DEBUG**, **SILENT**).
- It is allowed the fourth level **FOR** / **CASE**, but having a single nested statement only.
- It is allowed not more then 7 simple and 7 block statements in a single block, after that a block comment is required
----- 'comment'
- In case of abandoning any of rules listed above, a class must be marked with a bad code modifier (**BAD**).

Overloading operators

- It is possible to redefine basic arithmetic operators for classes and record. Using redefined operators is declaring in a function header.

The language has lack of:

Preprocessor

- There are no non-typed macros
- **EXCLUSION:** there is a **LIKE** statement, but it is restricted and can not be used nested or out of class bounds.
- There are no conditional compilations.
- There are no file includes.

Function pointers

- There are no function pointers. Use instead virtual methods.

Array as a result of a function

- Function can return only a scalar value.
- If it is necessary to return an array, the result is passed is a parameter.

Multiple inheritance

- Class can be inherited only from one ancestor class.

go to statement

- It is possible to continue or break a loop from a deeply nested block (**CONTINUE x**, **BREAK y**).

Character data type

- To represent characters, string variables and constants of length 1 are used.

Precision specification for simple data type variables

- Precision is not specified.
- It is always same for embedded data types **INT** and **REAL**, and depends on a target platform and project settings (for integer variables except loop counters: 64 bits, but with key /int32 - 32 bits).

Implicit type conversions for distinct data types

- There are no implicit data type conversions except converting **INT** to **REAL** (or converting to strings in the out to console operators <<).
- There are no type casts.

Handling exceptions

- Exceptions due to an incorrect addressing or corrupting data in memory are practically not possible (because of using **NONE**

objects, instant initializations of variables, checking arrays bounds).

- In most cases when in other environments an exception is generated, in AL/IV no actions are fired, and an empty result is returned (**NONE**).
- So, exception handling is not required, so exception handling is not provided.

I. Syntax

I.1. Formatting statements

I.1.a. Continue statement lines

Statements need not any finalization symbol (like ';').
Symbol ';' is used to finish a block of statements.
E.g.:

```
d|discriminant = B * B - 4 * A * C
CASE d.Sign ?
[-1]: NONE
[0] : R[] << -B / 2 / A
[+1]: R[] << (-B - d.Sqrt) / 2 / A
      R[] << (-B + d.Sqrt) / 2 / A ;
```

A line of a code should not exceed 80 characters (without ending comments and spaces and tabulations).

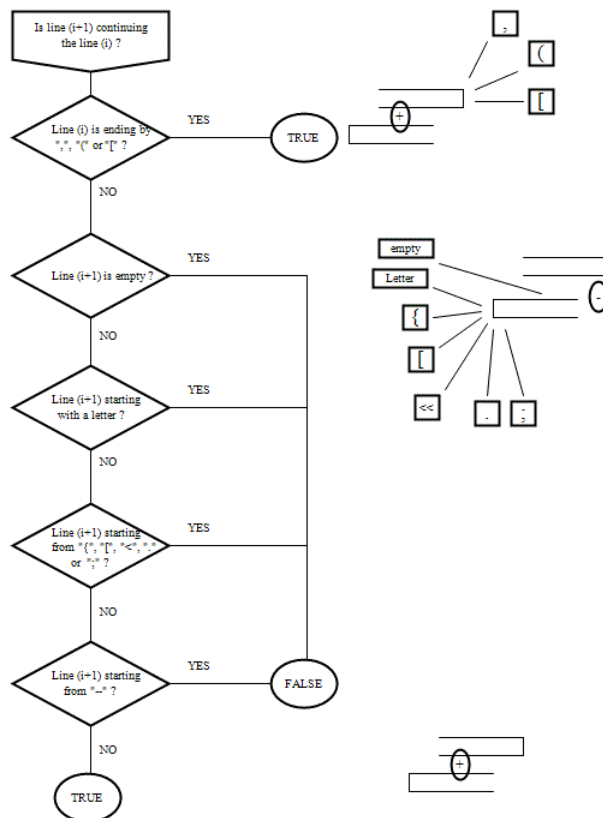
Statement can be continued in several lines only in following cases:

- a line is finished with ',', '(', '[';
- the next line is starting with one of symbols (without quotas):
 - '"' (double quotas - starts string constant)
 - '+'
 - '-' (single only, not '--')
 - '*'
 - '/'
 - '%'
 - '|'
 - '&'
 - '>'
 - '<' (but not '<<')
 - '='
 - '#'

Another way to remember the rule is that a statement is continued only if:

- either the previous line is finished by one of symbols '(', '[', ',',
- or the next line is not empty and it can not a start of a new statement, that is, it is not start from:
 - a letter,
 - a symbol '{' (not a variable declaration),
 - a symbol '[' (not a condition for a CASE statement),
 - a symbol '<<' (not an output to console statement),
 - symbols '.' or ';' (it is not a separate ending of a code block or a function),
 - a symbol '--' (it is not starting a block comment).

Or use the following figure:



To simplify understanding code containing long lines of code with continuations onto another lines, it is recommended to separate all such lines from other lines in blocks with empty lines. (The compiler warns about such possibility).

I. 1. b. Block statements

Block statements always are starting in a new line.

Nesting statements block is ended with the symbol ';'.:

Functions and other blocks of a class level are ending with a symbol '.'. Such blocks are: a class header, an import list, enumeration declaration, constants definition block. (Though it is allowed to use ';' for an import list, too).

The last line in a nested block can not be ended with two same ending symbols ';' or '.'. But it is allowed to end nesting block with its upper leveled function by two symbols ';.' at the end of the last line in the function.

So, if it is required to finish two nesting blocks, then the ending symbol of the external block will be placed in a separate line of code. E.g.:

```
FOR i IN [1 TO 100] :
  CASE i % 3 == 0 ? << "Fizz" ;
  CASE i % 5 == 0 ? << "Buzz" ;
  CASE i % 3 != 0 && i % 5 != 0 ?
    << i.Str ;
;
```

There is only a single keyword for a conditional statement (**CASE**), a single keyword for a basic loop statement (**FOR**), a single save and restore statement (**PUSH**). There is also an infinitive loop statement (**FOR, INFINITE**) using it is restricted. For debugging, special blocks **DEBUG** can be used also. To suppress compiler warnings (e.g. about existing **DEBUG** statements) the block **SILENT** can be used. To restrict time of executing of some code the block **LIMIT** can be used.

All other statements are simple: **BREAK**, **CONTINUE**, **STOP**, **LIKE**, **REVERT**, function calls, variable declarations and modifications, input and output (>>, <<), exit form a function (==>). The **BREAK** statement can be used only to exit a **FOR** loop and not for any other purposes.

A compiler requires to mark with a modifier **BAD** classes in which rules of code structuring are not satisfied:

- Number of nested levels of block statements **CASE** and **FOR** should not exceed 4 (blocks **PUSH** / **DEBUG** are not considered). The fourth nesting level can contain only a single simple statement (or a concatenation of a simple statement and a return from function sign ==>).
- Number of simple statement in a section of a block should not exceed 7.
- Number of nested block statements in a section of a block also should not exceed 7.

- A special block comment

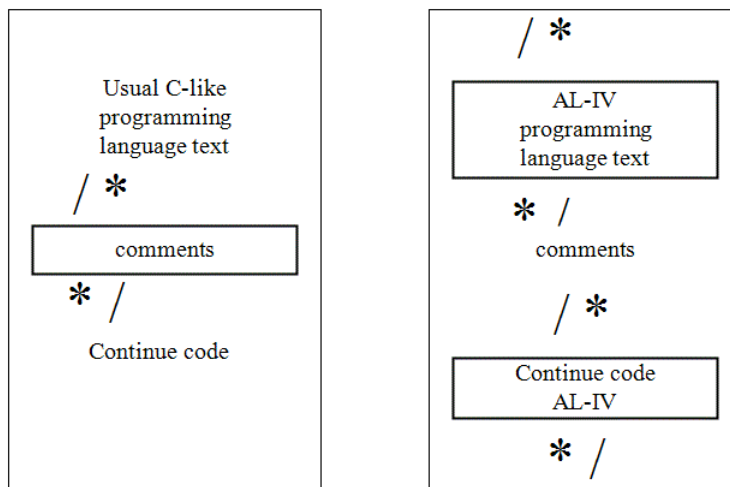
```
----- 'text'
```

can separate groups of statements (number of signs '-' at least 2) treated as sections in a block.

I. 1. c. Comments

- Multi-line comments: starting brackets are always placed at the begin of a line (may be following leading spaces) from the symbol `/*` and finishing with a string started from symbols `*/`
- A text file containing the AL-IV source code is always treating to be started from multi-line comments. A line containing at the end the symbol `/*` becomes the end of such leading comments lines. I.e. the AL-IV code should be located always between lines with symbols `/*` and `*/` (as it would be comments for some C-like programming language around).

See the figure:



- Comment to the end of line (start from `/**`). Such comments do not finish current statement.
- Comments starting with two minuses:

```
----- 'text'
```

are used to separate blocks of code onto sections. These also are used in class declarations and in module headers. A block comment statement can start a reused block of code (in statements **LIKE** / **REVERT**). In such case it should have a modifier **REUSED** (which is written after the label comma separated). E.g.:

```
----- 'start extracting ', REUSED
```

I. 1. d. Case sensitivity for keywords, types and other identifiers

- **KEYWORDS** can be written in the **UPPER** case only. **CASE** if a keyword but **case**, **CASE** and **Case** - not.
- **{Type names}** are enclosed into **figure** brackets.
- **simple type names** are reserved keywords: **BOOL**, **BYTE**, **INT**, **REAL**, **STR** (and there no simple types except these 5 listed here).
- **{Class names}** are starting from uppercase letters.
- **{enumeration_and_structure}** names are starting from lowercase letters.

Type names create a name space not intersecting other names at all.

- **CONSTANT names** are case sensitive.

It is desired to be written **CAPITALIZED**.

- **Names of 'CONSTANTS'** which are items of enumerations are always enclosed into apostrophes.
- **Variable and field names** (of structures and classes) are case sensitive.

So, **A** and **a** are different variables (or constants).

Names of **Public** fields, functions, variables, types are always starting from a capital letter.

All other names are local in a module. (This does not concern local variables which are always local in its functions and can be written from any case).

- The **underscore** at the end of a name_ is an integral part of the name (and used as a marker of a pointer to a structure or of a weak pointer to an object).

I. 1. e. Naming

while declaring a variable, a function, a data type, a table or an enumeration item it's long name can be separated by the symbol '|' onto parts, the first of which is a short name. Also, the symbol '||' can be used to define another name chain, and so on. It is therefore required that all such chains should start from the same letter (though letter case can be different).

There are following rules:

- At least one whole name length should not be less than 8 characters:
`S|ample|_val` (the variable can be referenced by names S, Sample and Sample_val);
- The exclusion for variable names: FOR loop variables can have only short name, even short, e.g.:
`FOR x IN M[] : ... ;`
- Another exclusion: if for the variable while declaration the attribute **INDEXING** is provided, it is enough for it to have only short name:
`INT k INDEXING {record}`
- If a name should be finished by an underscore character (which is a mark for a weak reference in the AL-IV), then both variants of the name should be ended with the underscore, e.g.:
`List_of_items_ || L_items_ []`
- For class names, records and enumerations the same rule is active, and figure brackets and split symbols '|' are not accounting in names lengths:
`{My|_class}`, 8 symbols in the name.

Functions names, variables, fields names (of classes and structures), table names and names of enumeration items are identifiers starting from a letter.

Names of {Classes} are identifiers enclosed by figure brackets and starting from a capital letter.

Names of {structures} and {enumerations} also are identifiers in figure brackets but starting from a lowercase letter.

Enumeration 'ITEMS' are written in apostrophes. Still these are constants it is recommended to use CAPITALIZED names.

I. 1. f. Modifiers

Many language declarations (module header, statement, variable of data type declaration) can have modifiers.

- Modifiers are written in form **, NAME** at the end of a declaration (usually).
- If there are several modifiers then these are listed via ','.
- Modifiers are written in the upper case. E.g.:

CLASS {My|_class} , BITWISE, RECURSIVE :

Modifiers do not affect main code semantics.

These just help to compiler to provide more security and sometimes help to optimize resulting code.

If to remove all modifiers in a ready program (and suppose that a language do not require it), this (usually) do not change a result of the program (but can take an effect of changing its size / working speed, either increasing or decreasing it).

In some cases a presence of some identifier in the name of a variable itself is used as a "modifier". E.g., the word "dummy" as a part of a name can be used to prevent the compiler to warn about a variable which is not used in code. And the word temp in a local variable name determines for the compiler that the variable is a temporary and it is not necessary to warn the programmer about assigning a new object to that variable only.

I. 2. Assignment

I. 2. a. Simple assignment statement

`|x = y`

where **x** is a variable or a field of a structure or an object and **y** is an expression.

To compare for equality of two operands a "C"-like operator **==** is used. (In SQL-expressions therefore it is allowed to use '=' sign to compare for equality).

I. 2. b. Assignment in combination with arithmetic, logic or bitwise operation

Additional operations **+=, -=, *=, /=, %=, |=, &=, ^=, ||=, &&=** can be used as an equivalent of the assignment of a result of a correspondent operation **+, -, *, ...** between the variable left to the symbol of the operation and the right-sided expression.

I. 2. c. Data sending operations

Operation `<<` is used to append a string to a string or to append an item to an array, or to output text to a current output console.

- For a string variable **A** and a string expression **B**, a statement **A << B** is equal to **A = A B**
- In case of an array **A[]** and an expression **B**, statement **A[] << B** means adding an item to the end of the array and it is equal to a function call **A[].Add(B)**
- If the left operand is an object of a class having the method **Write** (accepting a string) then it is called passing the result of the right (string) expression as a parameter;
- If a structure is on the left side but the right object is of a class having the method **Read** (returning a structure), then the method is called then as in the previous item;
- If the left operand is not present, then such statement means writing a string to an output stream (to the console for a console application).

A similar operation `>>` is used to add a newly created object to an objects array, see in a section devoted to classes and objects.

Additionally, right to the `>>` operator a declaration of a new variable can be specified.

- Input-output operators `<<` and `>>` can be combined in a single statement in a form:
`<< string [>> variable] [>>variable] ...`
 E.g.:
`<< "Enter number:" >> REAL Number|_entered`

I. 2. d. Repeating assignment and sending data operators

If a statement is starting from a symbol `'..'` (without quotes), then this represents a reference to the left side of the previous assignment statement. So, three dots are meaning such reference with a following "dot operation" used to either name a field or a method of an object or to name a function giving the operand referred by the `'..'` symbol as the first parameter.

For such repeating assignment statement (or data sending statement):

- statements are not counting as simple or nesting (there are no restrictions on such repeating assignments);
- an expression is fixed to be a destination for following repeating assignments if:
 - it is used as the left side of a usual assignment (`a = b`, `a += b`, `a *= b` etc.);
 E.g.:

```
Text = New memo( THIS, "TEXT", "VH")
...Set_width(200) // equal to Text.Set_width(200)
...Set_anchor_bottom(TRUE) // same as Text.Set_anchor_bottom(TRUE)
```
 - it is used as a destination in a usual sending data statement (`a << b`, `a[] << b`), including a case when the sending operator is replacing a method write call;
 E.g.:

```
Lines[] << "#!/bin/bash"
..[] << "echo run Wine"
..[] << "wine"
```
 - if a pseudo-operator `'..'` is used in place of a single "dot" operation in a chain of unnamings / indexing operations applied to an expression. E.g.:

```
A.Field1[indexA]...Do_something(params)
...Color = RED
...Foo(params)
```

In the example above, in two last lines, the symbol `'..'` is replaced by a compiler with an expression **A.Field1[indexA]**

I. 3. Expressions

Priority	Operator	Description	Group
1	-	Negation	Arithmetic operations (with numbers), result is a number
2	* / %	Multiplication, division, rest of division	
3	+ -	Adding, subtracting	
4	IN !IN	Check for presence a value in a set or array, check for an object belonging to a class	Comparing numbers, strings, classes, result is Boolean or integer
5	LIKE !LIKE	Compares two strings without case sensitivity and interpreting special symbols '?' and '%' in the second operand as patterns 'any character' and 'any amount of any symbols'	
6	< <= > >= == != <>	Comparisons	
7	~	Bitwise NOT	Bitwise operations with integers, result again is integer
8	& ^	Bitwise AND, eXclusive OR(xor)	
9		Bitwise OR	
10	!	Logical NOT	
11	&&	Logical AND	

12		Logical OR	Logical operations with Boolean, result also is Boolean
----	--	------------	---

- Concatenation operation has no special sign (just write operands sequentially).
- If any bitwise operations are used in a module, then a modifier **BITWISE** must be added for the module.
- Using both bitwise and arithmetic operations in the same expression is not allowed.
- bitwise shift and rotate operations are implemented with embedded functions *ShiftL/left*, *ShiftR/right*, *RotateL/left*, *RotateR/right*, there are no separate binary operation signs for them.
- it is allowed to write chains of comparisons.
 $a < b \leq c == 1 > d$ is the same as:
 $a < b \ \&\& \ b \leq c \ \&\& \ c == 1 \ \&\& \ 1 > d$

I. 3. a. Checking the presence of an item

Operators **IN** and **!IN** can be used in expressions:

- for checking if a scalar value is present in an array of values (and an array to the right can be either an array variable or a construction in form [value1, value2, ..., valueN])
- to check is a substring is present in a string
- also in SQL-expressions the operation **IN** can be used (but for opposite operation, SQL syntax is used: **NOT IN**)

In case of a constructed array, all its items must be constant expressions. And in case when the left value is a string, it is not allowed to construct an array from a single string value. Either a string should be to the right of **IN** / **!IN** (the case 2).

Operations **IN** / **!IN** are not applicable to **REAL** numbers (still it is not allowed to compare **REAL** values for equality/inequality).

I. 4. Other simple statements

- To return from a function before its code ends, a statement **==>** used.
 ◦ Such statement can be written either separately or following another simple statement (in the same line of code).
 E.g.:

```
CASE R > 0 ? RESULT = R ==> ;
```

- To terminate or continue a loop forcibly, keywords **BREAK** and **CONTINUE** can be used, together with a loop label, e.g.:

```
FOR x IN [1 TO 100] :  
  do_something  
  CASE condition ? BREAK x;  
;
```

- It is not allowed to use **BREAK** / **CONTINUE** in the **FOR**, **INFINITE** loop. It is only possible to terminate such loop by the return statement (**==>**) or in result of an exception firing.
- Using a label in **BREAK** / **CONTINUE** statements allows to continue or terminate an outer loop from an inner loop directly (but it is necessary to write first **BREAK** statements for all the nested loops first:
BREAK k, BREAK j, CONTINUE i)

I. 5. Conditional statement CASE

- An expression in the **CASE** statement should be of Boolean, integer, enumerated *or string* type.
- Expressions of other types (e.g., **REAL**, structures, objects, etc.) are not allowed.
- The header is always finished by the symbol **'?'**.
- In case of the Boolean condition, a block of code follows which is executed in case when the condition is true. And an **ELSE** block can be following (see below).
- In other cases each branch is starting from a constant array of constant values in square brackets:
[array of values]: code
- Constants (literal or named) should be listed in the array having a type correspondent to the type of the condition in the header.
- The value can be a single, or it can be represented with a list of values, or by a range **X TO Y** (which is allowed in case of integer condition).
- All the items in all sets or arrays must be distinct.
- In case of enumerated condition, all the enumeration items must be listed in arrays starting branches, and in such case it is not allowed to use **ELSE** branch.

- If a common part **ELSE** is present, it is executed if the expression in the header were not matching any constant listed, and there was no a branch executed.
- Usually the **ELSE** keyword is starting a new line. But it is allowed to place it in the same line as the **CASE** statement if the entire CASE statement fits a single line (80 symbols). In such case the **ELSE** keyword must be separated from the previous statement (then-clause) by spaces;
- Symbol ';' ends entire **CASE** block, together with all its branches and ELSE clause.

A **BREAK** statement is not used to finish a branch.
After any branch execution entire **CASE** statement is finished.

Classic example - solving a square equation for real numbers:

STRUCTURE {roots_sq|uare_equation} :

```
INT N|umber_solutions
REAL X1|_solution
REAL X2|_solution .
```

FUNCTION Square_eq|uation(

```
REAL A|_coefficient,
REAL B|_coefficient,
REAL c|_coefficient) ==> {roots_sq}
:
CASE A.Near(0) ? ==> ;
REAL d|iscriminant = B * B - 4 * A * C
CASE d.Sign ? [0]: RESULT.N = 1
                RESULT.X1 = -B / (2 * A)
                RESULT.X2 = RESULT.X1
[1]: d = d.Sqrt
    RESULT.X1 = (-B - d) / (2 * A)
    RESULT.X2 = (-B + d) / (2 * A)
    RESULT.N = 2 ; .
```

Especial version of the **CASE ?** statement - with a condition in the header but with sequential calculation of conditions in square brackets until either the first one is found with value **TRUE** (in such case a correspondent branch is executed only), or until the **ELSE** part (or the end of the **CASE** statement) is found.

In such case symbols '?' are written after each expression rather than ':'. .

FUNCTION Square_eq2|uation(

```
REAL A|_coefficient,
REAL B|_coefficient,
REAL c|_coefficient) ==> {roots_sq}
:
----- 'sequentional version'
REAL d|iscriminant = B * B - 4 * A * C
CASE ?
[A.Near(0)]? ==>
[d.Near(0)]? RESULT.N = 1
            RESULT.X1 = -B / (2 * A)
            RESULT.X2 = RESULT.X1
[d > 0] ? d = d.Sqrt
    RESULT.X1 = (-B - d) / (2 * A)
    RESULT.X2 = (-B + d) / (2 * A)
    RESULT.N = 2 ; .
```

Also there is a special variant **CASE ?** purposed to use in generic functions. See detailed in a correspondent section.

I. 6. FOR loop statements

A loop statement **FOR** enumerating array items or a range values - is a single possible controlled loop statement:

```
FOR variable IN array[range] : body ;
or
FOR variable IN [range] : body ;
or
FOR variable ENUM|ERATE method: body ;
```

- The loop variable of such loop statement is always has the same type as a enumerating array items data type.
- In case of a range without an array, the variable type is integer;
- It:
 - Does not require declaration.
 - Does not require a long synonym.
 - Can not be used out of the loop, except in another **FOR** loop.

- It is used as a label in statements **BREAK variable** and **CONTINUE variable**, related to the loop.
- A variable of a **FOR** loop with a range can not be changed in a loop body (and used as a usual variable).
- In case of empty range all the items are enumerated from the start (index 0) to the end (index *).
- To indexes change direction and bounds, it is necessary to use an explicit range of indexes of an array or a range without an array: **[X TO Y]** or **[Y DOWNT0 X]**, where **X** and **Y** are integer expressions.
- In case of a loop on an array items, it is possible to apply an embedded function **INDEX** to it (it returns an index of the variable in the enumerating array).
- Bounds of indices enumerating are calculated before the loop, and even if size of an array is changed during a loop, this does not affect enumerated indices. As a result, this can lead to violating array bounds, and an array variable will be set to **NONE** in such cases.
- For the variant **FOR x ENUM y : ... ;** the "y" should be a call to a method which does not have parameters and return a Boolean. Such method should return **TRUE** to stop iterations, and it is called before each loop iteration. Therefore the loop variable ("x") is changed from 0 to some maximal value and then its value becomes greater the limit, the loop is stopped. The default limit value is 1_000_000 but it can be redefined explicitly:
FOR x ENUM(MAX 1000) method : ... ;

To create an unguided infinitive loop, a special block statement is used:

```
FOR, INFINITE : loop body ;
```

- The **FOR, INFINITE** loop can be terminated only by a return from the function statement **==>**.
- Statements **CONTINUE / BREAK** can not be used.
- A class where statements **FOR, INFINITE** are used, must be marked with a modifier **INFINITIVE**.

A loop variable if it is integer can (as well as any integer variable) to have an attribute **INDEXING type** or **INDEXING (list, of, types)**. See more detailed in a section devoted to control of types of indexing arrays.

In case when a **FOR** block is ending with the **BREAK** statement, its final sequence of statements can be separated with the special statement **DONE**. E.g.:

```
FOR a IN A[] :
CASE !a.satisfying_conditions ? CONTINUE a ;
DONE // we've found desired item
a.do_something
do_something_more
BREAK a ;
```

The **DONE** statement decreases the counting nesting level of nesting statements (as the statement **FOR** would already finished).

I. 6. a. About assigning a value to a loop variable:

- If a loop variable is changed in a certain range of values (not lists items of an array), then it is not allowed to change it in the loop;
- In case when the loop variable iterates on all (or part of) items of an array (e.g. string or integer or other type array except an array of structures) – assigning new value to such variable changes it but does not affect the array item enumerating. E.g.:
FOR s IN Strings_array[]:
 s = s.Trim
 ... work with trimmed String_array[s.INDEX] ...
;
- In case of an array of structures, the variable of the loop actually is a macros providing the access to a current item array. So, changing its fields or assigning another structure to it changes a correspondent array item as well as it was accessing;
- In any case this does not affect the result of an expression **Variable.INDEX** returning an integer index of an array item currently iterating.

I. 7. PUSH statements block

A **PUSH** statement is purposed to create procedural "brackets" with saving some state before and obligatory restoring this state after the block. Or to call some **PUSH**-method and to guarantee that at the end of the **PUSH** block a correspondent **POP**-method will be called.

Restoring is always guaranteed:

- When the block is left by a **BREAK / CONTINUE** statement.
- When a function is left by a **==>** (return) statement.

- When an exceptional situation occur (e.g. when an operation is interrupting of a too long operation by the user, or an operation resulting too deep recursion of function calls)

Syntax:

```
PUSH variable = expression : body ;
or:
PUSH variable : body ;
or:
PUSH method(parameters) : body ;
```

- On entry into the block a value of a variable is saved. E.g. in a local stack or in a local dynamic array.
- Then new value is assigned to it (if specified).
- When the finishing bracket is achieved (it is executed even in case of a failure), the value of the variable is restored.
- Any simple or object data type scalar variable or an array item is allowed to be used as a **PUSH** block variable.

A special method with modifier **POP(method2)** can be called in the **PUSH** statement. At the end of such **PUSH** block, the method2 specified in parentheses (of the **POP** modifier of a method called in the **PUSH**), is called obligatory.

Such method2 should not have parameters, and it can not be called directly (as well as opening method, having the **POP** modifier, which therefore can be called in the **PUSH** statements). E.g.:

```
PUSH db.Transaction :
    // some database operations
    db.Commit
;
```

I. 8. DEBUG statements block

A **DEBUG** block is intended for a temporary code injection which is required for debug purposes. In such blocks, levels of nesting blocks are not controlled. A compiler is always warning about such blocks not removed from a code. Such blocks should be removed when debugging is finished.

In **DEBUG** blocks it is possible to declare local variables as usual. But it is not allowed to use such variables out of **DEBUG** blocks (though it is allowed to use it in another **DEBUG** block in the same function).

```
DEBUG :
    тело ;
```

If there are required additional classes to compile **DEBUG** blocks, it is recommended to use a special directive **IMPORT** with the modifier **DEBUG**.

For the **DEBUG** statement:

- nesting levels are not taking into account both for the **DEBUG** block itself and for its nesting code, too;
- it is allowed to omit colon:
DEBUG count_passes += 1 ;
- if the first statement is the output to console with the first output string literal, than it is allowed to omit the symbol << :
DEBUG: "test!"#NL ;
DEBUG "another test" ;

I. 9. SILENT statements block

The block **SILENT** purposed to prevent the compiler from warnings on some block of code. E.g. about **DEBUG** statements in code.

```
SILENT:
    DEBUG :
        body ;
;
```

I. 10. LIMIT statements block

The block **SILENT** purposed to prevent the compiler from warnings on some block of code. E.g. about **DEBUG** statements in code.

```
LIMIT TIME(n) MSEC:
    body
;
```

There are following variants possible:

- **LIMIT** TIME(n) units: ... ;
is a limit by time (n - is fractional numeric expression). Units can be one of following identifiers:
 - **MSEC** || **MILLISECONDS**,
 - **SEC**|**ONDES**,

- **MINUTES**,
- **HOURS**,
- **DAYS**;
- **LIMIT FUNCTIONS(n) : ... ;**
restricts executing by amount of functions calls;
- **LIMIT LOOPS(n) : ... ;**
restricts executing by amount of FOR loops iterations;

Checks for possible exceeding of a restriction specified are done (usually) each 65535 loop iterations done in the AL-IV code. So, final exceed of a limit can be discovered later to the actual exceeding after that 64K iterations executed between checks.

In case of a limit exceeding the code in a block stops running and running continued from a statement next to the block. And if some PUSH statements were executing in between, all these are finished correctly with a correspondent POP operation (e.g. restoring a variable value).

L. 11. LIKE statement

(Do not mean the comparison operation LIKE between two strings in an expression!)

The **LIKE** statement is intended to insert earlier written code without changes in a position of the LIKE statement. An inserted code should be placed between two block comments (on the same nesting level). A label of the first block comment (together with a function name) is used to identify the code to insert. There are rules:

- A reused block of code should be marked with a modifier **REUSED**.
- If a block is inserted from another function, the name of the function is written following the **LIKE** keyword.
- A multi-dot is written next (amount of dots is not restricted, at least three should present).
- It is possible to insert code **only from the same class**.
- Inserted code should be located above the **LIKE** statement.
- An inserted block should not have **LIKE** statements itself (recursion or nesting are not possible like in macro).
- An inserted code is injected "as is", its syntax and semantics are controlled only while compiling the LIKE statement. So do not duplicate variable names, declarations etc. to avoid compiler errors.
- A reused block can be inserted so many times as it is needed, without restrictions.
- For a time while a block is inserted by the **LIKE** statement, a counter of nesting **FOR/CASE** levels is reset to zero, so there are no problems of total nesting level exceeding allowed amount (3 nesting levels).

An example:

```
...
----- 'find a dot'
pos = s.Find(".")
CASE pos < 0 ? pos = s.Len ;
----- 'end of find'
...
...
LIKE ..... 'find a dot' // place in the same function
or
LIKE Method1 ..... 'find a dot' // place in another function
```

Additionally, while calling the **LIKE** statement, it is possible to replace some fragments of code with other:

```
LIKE [method].....'label', BUT (pattern1 ==> replacement1) [*N1] [, (pattern2 ==> replacement2) [*N2] ]
```

Lists of replacements are enclosed into parentheses and listed via comma. A replacement should have a multiplier following its right bracket in form *** number** if there are several such patterns in code. The multiplier can be omitted only in case when only a single such pattern is there.

All the replacements are done just before compiling the LIKE statement having such replacements defined. Both pattern and replacement strings can be enclosed into apostrophes. Only the string right to the symbol ==> can be empty, but patterns always should not be empty strings.

And please remember:

- It is better to use **LIKE 'name'** then just to copy/paste a code.
- **BUT** It is better to move a common code into separate functions if this is possible (then to use **LIKE**).

So do not overuse the **LIKE** statement without necessity.

L. 12. REVERT statement

The **REVERT** statement is intended to insert earlier written sequence of assignments reverting its direction of assignment in a position of the **REVERT**

statement. An inserted code should be placed between two block comments (on the same nesting level). A label of the first block comment (together with a function name) is used to identify the code to insert. There are rules:

- A reused block of code should be marked with a modifier **REUSED**.
- If an inverting code block is from another function, the name of the function is written following the **REVERT** keyword.
- A multi-dot is written next (amount of dots is not restricted, at least three should present).
- It is not possible to revert code from another class.
- Inverted code should be located above the **REVERT** statement.
- An inverted block should contain only assignment statements, in those destination and source parts can be reverted (at least using correspondent getters / setters).
- An inverted code should not use local variables (including parameters and a special variable **RESULT**).
- A reused block can be inserted so many times as it is needed, without restrictions.

An example:

```
----- 'save form coordinates '
config.Set_i("Left", Left)
config.Set_i("Top", Top)
config.Set_i("Width", Width)
config.Set_i("Left", Height)
----- 'end of save'
...
...
REVERT ..... 'save form coordinates'
```

II. Variables, constants and data types

II. 1. Simple data types

II. 1. a. Names of data types, data types classification

There are **predefined simple data types** which can not be redefined:

- **BOOL** - Boolean data type, with values **TRUE** and **FALSE**.
- **BYTE** - byte data type, with values from 0 to 255. Using bytes in a class requires adding a modifier **BYTES** to the class header.
- **INT|EGER** - integer numbers of usual precision.
- **REAL** - real numbers with default precision.
- **STR|ING** - string type (chain of characters).

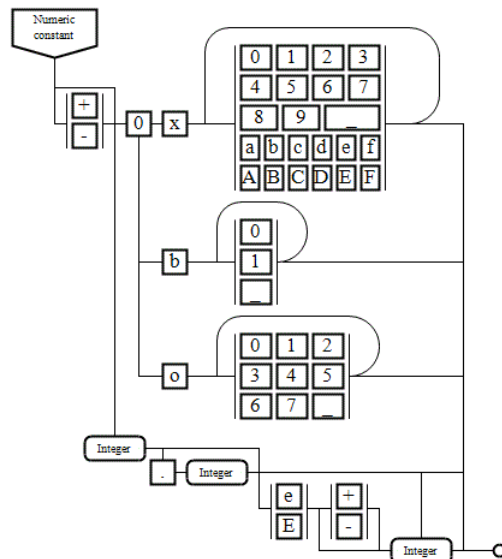
For data types a **precision is never specified**:

- For variables of base data types always a precision is used which is defined by a compiler developer (or it is controlled by project options, if the compiler allows to do so).

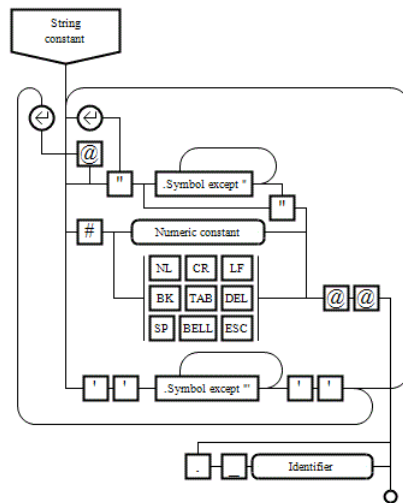
When a **BYTE** data type is used in a class, it should be marked in the header by the modifier **BYTES**, and can not be marked as **SAFE**.

II. 1. a. Writing constants of base data types

- Integer constants in decimal base system: **[-|+]<0...9>...**
1_345 = 1345
(underscores are ignored and can be used to separate groups of digits).
- In binary system:
0b_0100_1010_0011_1001
(Case of a letter used to define a base, is ignored: B=b, X=x, except the octal system)
- In hexadecimal system (x=X):
0xFFEA_d44a
- In octal system (only small letter 'o'):
0o_7_342_001
- Real constants: **{0...9}.{0...9}[e[+|-]{0...9}]**
E.g.:
-2.73e3
11_E-04



- String constants: `"{symbol except "}"` where symbol is any printable character or space, double quotes are never written. To write it, it is necessary to use a symbol constant `#QU`). E.g.:
`#QU "This text is enclosed into double quotes" #QU`
- There are several named string constants written in form `#NAME:`
 - `#NL` - new line (usually this are two symbols `#CR#LF` with codes 13 and 10);
 - `#CR` - caret return (13);
 - `#LF` - line feed (10);
 - `#BK` - backspace (8);
 - `#TAB` - tabulation (9);
 - `#SP` - space (32);
 - `#BELL` - sound (7);
 - `#QU` - quotation mark (34);
 - `#ESC` - special "escape" symbol (27);
- Also a symbol can be encoded in form `#number`, e.g., `#32` equals to `#0x20` and equals to `#SP` and `" "`;
- To concatenate character constants and literal strings there are no operation symbols using.
E.g.:
`"This text is finished by a caret return" #CR`
- If a line of code is starting from a quotation symbol, the line is a continuation of the previous line. So, to write a very long string constant, not fit into a single line of code, it is sufficient to divide the string onto two (or more) strings, and write its continuation in another string. E.g.:
`"This is very long string of text. It is so long that "`
`"it can not be written in a single line of code."`
- If a string constant is starting from a symbol `'@'`, then for all continued lines symbol `#NL` is inserted in between, so new line characters are inserted into a long string constant in places where the line is continued in next lines. E.g.:
`@ "This string constant contains 3 lines: the 1st,"`
`"the 2nd,"`
`"and the 3rd"`
- A symbol `@@` before a string literal cancels previous symbol `@-` until the end of the concatenation of strings or until the next symbol `@`.
- As a continuation of a string constant in a separate line of code a literal in form `'text'` can be used (bounded with doubled apostrophes). Such string literal can contain any characters except a combination `['']` (doubled apostrophes).



II. 1. c. Enumerations

A enumerated data type defines a **set of named constants**. The set itself is also named to make it possible to refer to it by the name.

- In enumeration declaration, its items are written comma separated. A declaration:
`ENUM {names|_of_enumeration} : values .`
- A enumeration type name is always enclosed into figure brackets and starts from a lowercase letter.
- Each enumeration item name is enclosed into apostrophes.
- An item name can have short and small form as usual (separating short name and the rest part by a symbol '|').
- Names must be unique in the class (among all the enumerations in the class).

```
ENUM {color|s_of_traffic_light} :
    'RED COLOR',
    'YELLOW COLOR',
    'GREEN COLOR' .
```

Operations on enumerations:

- It is not allowed to assign an integer value or value from another enumeration to a variable of such type.
- NONE-value for a enumeration type value is the first item in the enumeration (corresponds to 0 in the internal representation);
- Value of variable (or constant, or expression) of a enumerated type can be converted to an integer using embedded function `Int`.
- Back conversion of a numeric value to an enumerated type is not possible except via a custom function (e.g. using `CASE` statement).
- Embedded function `Name` returns for an enumerated item its name by value with apostrophes (the first name of the item if there are several ones there).

A enumeration can be used as a range for indexes of a fixed array:

```
STR LName[s_of_lights[{color}]]
In such case the enumeration items only can be used as indexes of such fixed array:
LName['R'] = "Red color"
```

Enumeration items are not ordered and can not be compared using operations `<`, `<=`, `>`, `>=` (only `==` and `!=` are allowed). Enumeration items can form a constant array, and for pair "item - array of items" it is possible to use operations `IN` and `!IN`:

```
CASE A IN ['R', 'Y'] ? ... ;
In case of naming conflict between enumeration items from different enumerations, those should be qualified using names of enumerations:
{enumeration_name}. 'ENUMERATION_ITEM'.
In case when names of enumerations are matching, a class name should also be added to a specification:
{Class_name}. {enumeration_name}. 'ENUMERATION_ITEM'.
```

When a `CASE` statement condition has a enumerated type:

- it is not allowed to use the `ELSE` branch,
- all the items from the enumerated data type must be listed,
- more then a single constant can be used in a single branch:
`CASE Greek ?`
`['ALPHA', 'BETHA']: ...`

```

['GAMMA', 'DELTA']: ...
['EPSILON']: ...
['DZETHA', 'ETHA', 'TETHA', 'IOTHA', 'KAPPA']: ...
... ;

```

II. 2. Variables and arrays

II. 2. a. Declaration of variables

A declaration of a variable is starting from a specification of its data type in **{figure}** brackets (or from one of reserved data type names: **BOOL**, **BYTE**, **INT**, **REAL**, **STR**).

- When a variable is declaring its full name should be at least 8 characters length.
 - *EXCLUSIONS:*
 1. Loop variables (*FOR*) name length is not restricted;
 2. Integer variables having explicitly specified attribute *INDEXING*;
- Short part of the variable name is separated from the rest of name with a symbol '|'.
- For arrays, its size is specified in **[square]** brackets.
 - For dynamic arrays brackets are empty.
 - For fixed arrays indexed by integer index, a constant is specified defining the array size (index of the last item + 1).
 - For fixed arrays with indexes from an enumeration, a name of the enumerated type is specified, in figure brackets as usual:
BOOL My_array_of_flags[{traffic}]
- Then (only for integer scalar variables and arrays) an attribute **INDEXING type** or **INDEXING (list,of,types)** can be specified. Specifying such attribute means that types specified will be compared to types of arrays items when such arrays are indexed using such variables (or simple expressions like X+Y). Mismatching of such types leads to error messages. Also, similar comparison occurs while assigning one variable to another (when both have the *INDEXING* attribute), and in case when a simple expression X+Y or X-Y is assigned and one of X and Y are *INDEXING* variables and the destination also is an *INDEXING* variable. And while passing such expression or variable in place or a parameter having such attribute, too.
- Then variable **MODIFIERS** are following if these are necessary (**READ**, **INIT**, **MAXLEN[n]**).
- A declaration can continuing with an assignment of an initial constant value, if this is suitable (not in all cases, see about classes, structures, global object variable).

A variable always has an initial value. If there are no explicit initial value specified, a default initial value is set.

- For an object, this is **NONE**-value of a correspondent class (see "Classes").
- For a string this is an empty string.
- For a dynamic array this is an empty array.
- For enumerations, collections, numbers this is arithmetic 0 value (for enumeration types this means always the first item in the enumeration).
- A structure is initialized with a structure having zero-valued fields.

Naming variables rules:

- A short name is an identifier of 1 or more characters,
- A long name is an identifier from 8 to 80 characters,
- In the code both short and long names can be used equally.

Case of letters in a variable name:

- A **Public** variable or class field name is starting from an uppercase letter (and can have modifier **READ** making it read-only).
- **CONSTANTS** usually contain only capital letters.
- **private** variables and class fields usually start from lowercase letter.

II. 2. b. Field modifiers

If some modifiers are required for a variable, these are written in a declaration following its names and dimensions.

- **READ|ONLY** - is used for module level variables, fields of classes (and also for function parameters).
 - For a module variable it enables only reading the variable from other modules.
 - For a class field, it becomes read only except its own module and code of methods of inherited classes.
- **INIT|IALIZE** - for a class field, it is required to be initialized while creating an object (see "Classes").

- **DEPRECATED('text')** - such field is deprecated though it is still is supported. Text should contain information about an alternative (this can be not only a filed, but a function, or another class etc.) It is recommended to use an alternative as quick as possible, still in following versions the field can become unsupported.
- **ABANDONED('text')** - the field is abandoned. Using it leads to a compiling error. The text should contain information about an alternative.

Additionally to explicit modifiers above, there are also implicit variables modifiers:

- A function parameter declared in the function header but not used in the function body, should have a substring "dummy" temp in its name (in any register case).
- A variable declared in a function body and (even if a value was assigned to it) not used in expressions also should have the substring "dummy" in its name.
- Object local variable, to which a new object variable was assigned, which were not added to an array of strong references, has not an owner specified, should have a substring " temp" in its name (in any register case). E.g.,:
`load|er_Temp = {Text_file}{Path = path}`

II. 2. c. Named constants declaration

Constants can be declared only on a module level, and only of embedded simple data types (**BYTE**, **BOOL**, **INT**, **REAL**, **STR**).

- Declaration of a single constant in a single line:
`CONST type_name {Name_of_block} : CONSTANT_NAME = expression .`
- Declaration of several constants:
`CONST type_name :
 CONSTANT_NAME_1 = expression
 CONSTANT_NAME_2 = expression
 ...
 CONSTANT_NAME_N = expression .`

It is recommended to use uppercased names for constants.

A block of constants can be named (its name is specified in {figure brackets} following the type name). Such name of a block of constants can be used while specifying restrictions on functions parameters.

II. 2. d. Arrays declaration

Array dimensions are written in square brackets following the variable names.

- Arrays can be only one-dimensional (if necessary special indexing methods can be used to create multi-dimensional arrays of some data types).
- An array variable can be fixed or dynamic.
- In square brackets nothing are written for a dynamic array:
`STR Lines | Lines_if_text[]`
- Fixed array can have several dimensions.
- For a fixed array an integer constant expression is written in square brackets, or a name of a enumeration type. For an integer, the value is an array size (it is greater by 1 then an index on the given dimension). An example (an array of 16 items):
`REAL A | Array_of_values[16]`
- while using an array even as a whole object all its square brackets are listed always.
 E.g., `A[]`.

Initial values for items of arrays are always provided as well as for all other variables.

- A dynamic size array initially is empty.
- A fixed array of class objects initially is filled by **NONE** objects of the correspondent class.
- A fixed array of numbers, enumerations, collections is initially filled with arithmetic zeroes.
- A fixed array of structures initially is filled with NONE-values of correspondent structures (i.e., all its numeric fields are 0, string are empty etc., and this is defined recursively for fields also been structures).

II. 2. e. Array constructor

An array constructor is a sequence of items in square brackets in a form:

`[ value, value, ... value ]`
 Only constants are allowed as values.

An array constructor can be used as the second operand of an operation checking if the first operand is present in the second array **A IN B** (and in the opposite case: **A !IN B**).

II. 2. f. Using arrays

Main specific operations with arrays:

- Reading and writing items of an array. In the left side of an assignment statement or as an operand of an expression a name of an array is specified and for each of its dimensions an expression is written in square brackets, which represents an integer index of an item of the array.
E.g.: **A[i] = B**
- Adding an item to the end of a dynamic array with an operation **<<**, e.g.:
X[] << Value
- Calling one of embedded functions designed under dynamic arrays::
 - Adding an item at the end of an array:
X[] << s
 - Inserting an item at the position i:
X[].Insert(i, Value)
 - Deleting an item at position i:
X[].Delete(i)
 - Deleting several (Count) items from the index i:
X[].Delete_range(i, Count)
 - Deleting all the items equal to R:
X[].Remove(R)
 - Finding of an index of the first item equal to R:
X[].Find(v)
 - Checking if the value R is containing in the array X[]:
present = R IN X[]
 - The same as above (if not present, -1 is returned):
CASE X[].Find(R) < 0 ? not_found ;
 - Getting amount of items in an array:
count = X[].Count
 - Deleting all the items from an array:
X[].Clear
 - Allocating exactly new_size items (deleting ambiguous or adding NONE items):
X[].Allocate(new_size)
- When an item is read out of array bounds, then a **NONE** value of correspondent type is returned.
- Assigning a value out of array bounds is ignored.

Passing an array as a parameter to a function:

- Square brackets for an array parameter (dynamic or fixed).
- In place of a dynamic array only a dynamic array can be passed.
- In place of a fixed array formal parameter only a fixed array or a sub-range of a dynamic or a fixed array can be passed:
A[first TO last]
- To specify a formal parameter of a function as a fixed array with integer indexes, the symbol '*' is written in square brackets.
- In place of a fixed array with indexes given by a enumerated data type, only a fixed array with same type of indexes can be passed only.

It is not allowed:

- Assigning an array as a whole variable.
- Returning an array as a result of a function (it is necessary to declare an array parameter of a desired type and pass results with it).

Special indexing arrays with the symbol '*':

- The '*' symbol in the square brackets following the array name in a position where an expression operand is possible (e.g. a variable) means the index of the last item in the array (equal to Count-1);
- So, **A[*]** is the last item in the array A, and **A[*-1]** accesses the penultimate item (in the position Count-2).

II. 3. Functions

II. 3. a. Function header

FUNCTION names (parameters) ==> result_type , modifiers :
statements .

FUNCTION names (parameters) ==> result_type , modifiers :

- A static function is starting from a keyword **FUNCTION** or **FUN**.
- A class method is starting with the keyword **METHOD**.
- If the method is overriding its predecessor method from its class ancestor, the keyword **OVERRIDE** is used.

- For a constructor and a destructor, a keyword **CONSTRUCT** or **DESTRUCT** are used, correspondently (and in such case there are no function name(s), parameters, result types and so on - just a colon symbol).
- To redefine standard operations +, -, *, / special form of a function is used starting from the keyword **OPERATOR**.
- Testing functions are starting from the keyword **TEST**.

FUNCTION names (parameters) ==> result_type , modifiers :

- Functions can have two or more names (at least one name should be not less then 8 characters).
- If a function has two names, the short name is written first, than after a separator '|' - the rest of the name. When a method of a predecessor class is overriding, only the first (short) name of the method overriding is specified.
- Public function name starts from an uppercase letter. To make public a method or function which name is starting from a lowercase letter, use the modifier **PUBLIC** (see below).
- Function accessible only in the class itself (and its descendants) is named from a lowercase letter always.
- **OPERATOR** has not names, and its parameters are specified in the form **TYPE OPERATION TYPE** (or, for unary "-" operation, in the form **OPERATION TYPE**), and so its parameters definition is used as its unique name (it is allowed to redefine operators several times for the same operation but with different types of parameters).
- It is possible to declare a single no-named method in a class (in the header the dot is used in place of name). Parameters for such method are enclosed in square brackets. If a setter is defined for such method, then the method call can be used in the left side of assignment statements.

FUNCTION names (parameters) ==> result_type , modifiers :

- Parentheses must be always present.
- For empty parameters list nothing written (and parenthesis also can be omitted).
- Amount of parameters is not restricted but if a function has more then three parameters then all its parameters starting from the fourth (or earlier) should be passed in the form of an assignment
Parameter_name= expression

Parameters are separated with comma.

- First a type of a parameter is written, then its names, then optional: dimensions, modifiers, default value.
- Parameter can have two or more names (short name is separated from the rest of the long name with the symbol '|' as usual).
- Full parameter name should be at least 8 characters length.
- Parameter names of public functions should start from the uppercase letter.

For an array parameter its dimensions are listed following its names.

- For dynamic array, square brackets are empty.
- For fixed arrays with integer indexes, the symbol '*' is written in brackets.
- For fixed arrays indexed by a enumeration, the name of the enumerated type is written, in figure brackets as usual, e.g.:
BOOL A[rray_of_color[{color}]]

Integer parameters can have attribute **INDEXING type** or **INDEXING (list, of, types)**. If such variable is indexing an array then the type of array items is comparing to types specified. When a function is called passing in place of an **INDEXING** parameter a variable (or a simple expression like X+Y where one of argument is an indexing variable), then its **INDEXING** attributes (types) are compared.

Parameters can not have its own modifiers.

Parameters which are not used in the function body, should have a substring "dummy" in its name (in any registry case).

Scalar parameters are always passed by the value (formally) and can not be changed in the function body.

Parameters can have restrictions on values passed or be used in restrictions to other parameters. Parameters can be restricted with lists of constant values of ranges independently, or dependently from values of other parameters passed.. See function modifiers **RESTRICT**, **IF**, **THEN** (below).

For operators, input data types are its parameters. At least one of input data types should be a class or a structure. In the operator body (which is always a single expression) to refer to that type parameters, letters A and B are used (though there are no such letters on the operator header). The "A" letter is used to refer to the first parameter, "B" - to the second. In case of redefining the unary operator "-", the "A" letter is used to refer to the single input data type.

In case of common functions in place of a parameter type a list of possible types enclosed into parenthesis can be found like **{INT, REAL, STR, {color}}**. See detailed in additional parts.

FUNCTION names (parameters) ==> result_type, modifiers :

If a result type is not specified (together with the symbol '==>'), then the function is not returning a result.

- To assign a result to a function, an assignment to a pseudo variable **RESULT** is used.
- As well as for other local variables, the **RESULT** variable is initialized initially with a **NONE** value:
 - **NONE** - for objects,
 - **0** - for numbers,
 - **""** - for strings,
 - first item - for enumerations,
 - **FALSE** - for **BOOL** data type.
- A function can not return an array as a result, only a scalar.
- Special symbol **==>** is also used as an operation of returning from a function. It can follow any simple statement (assignment, function call, etc.)

A sample of a function counting number of dots in a string:

```
FUNCTION Ndots|_count (STR S) ==> INT :
  FOR i IN [0 TO S.Len-1] :
    CASE S[i] == "." ? RESULT+=1 ;
  ; .
```

In case of a common function result type can be represented as a list of types enclosed into figure brackets: **{INT, BOOL}**. See detailed in the Additionally topics.

It is possible to write following the result type returning:
RESULT|some_additional_text_describing_the_returning_value.

FUNCTION names (parameters) ==> result_type, modifiers :

Function modifiers are listed via comma. Following modifiers are possible:

- **RECURSIVE** - function is recursive. If a recursive function does not call itself and it is recursive via a call to a function which is already marked as recursive, the modifier can be omitted.
 - If too deep recursion is detected (usually more then 128 call levels), then execution is returned to the first recursive call level and a function called on that level is ended (returning **NONE** value corresponding to its result type - if the function should return a value).
- **NEW** - function is returning a new object instance.
 - At least one statement in the function is an assignment of a new object creation (or a result of calling another function with the **NEW** modifier) to the pseudo variable **RESULT**.
 - A function with the **NEW** modifier can be called only separately (not as an ordinary operand in an expression), and it is necessary either to assign a result to an strong object reference, or to its **RESULT** variable, or to a temporary variable (having substring temp in its name).
 - If at least once a new object is assigned directly to the **RESULT** pseudo-variable, than the function must have the modifier **NEW**.
- **REPLACE** - for an overridden virtual method of a class **not returning a value**, to inform that the function does not call a base method, so it is totally override. If there are no such modifier for such kind of an override function, then it is required to call a base method (using operator **BASE**) at least once.
- **NATIVE** - a body of the function is a string constant containing a code inserted into resulting (compiled) function as is. (Either a function body is ending with a statement **NATIVE "string"**).
 - A class, having **NATIVE** functions must have the modifier **NATIVE**.
- **SETTER FOR identifier** - function is a setter for a field or a getter method specified by the identifier. Such field or method must be declared just before the setter function. In case of a field, a setter should have a single parameter of the same type as the field. For a case of the getter method, the setter should have number of parameters greater by one then a getter. The last parameter should have the same type as a result of a getter, all other parameter data types should match for getter and setter. The presence of such method for a field does not mean that it is possible to assign a value to the field using assignment statements (except special cases, see **REVERT**, **/debug**).
- **CALLBACK** - function is intended to call it from the class code (and native functions) only, not for direct call of it from the customer code (even not from descendant classes of the class containing the callback function). E.g. an event **mouse_down** in a form: it is necessary to override the method rather than to call it in your form implementation class. The **CALLBACK** modifier also should be used to prevent removing the function in result of optimization.
- **POP(Method2)** - function is intended to be called only in the **PUSH** statement. When such **PUSH** block is ending, the **Method2** specified in parenthesis is called anyway (the **Method2** should not have parameters, and it can not be called any other way except this).
- **DEPRECATED('text')** - the function is deprecated and it is necessary to use an alternative specified in a 'text'. In further versions of a class it is possible that the function will not be supported (and become abandoned or

removed). Either, on some platforms the function can not be implemented or very restricted (in such case it is desired to write the text **'Platform dependent'**).

- **ABANDONED('text')** - the function is no more supported, use an alternative specified in the text in apostrophes. Calling such function leads to a compilation error. The **ABANDONED** modifier can be used in a derived class to abandon a method declared in a base class. But the compiler will control calls of abandoned methods only when those are called for the final class only.
- **STORE(Parameter=value)** - creates a special "invisible" parameter of an integer data type, which creates on a caller class side correspondent integer variable (separately for each call to the function). Such variable is passed implicitly by reference at call. See more detailed in the **Additional section**.
- **FORGET** - when such method is called, all the stored invisible values in the caller object are reset to its initial values. See details in the **Additional section**.
- **TRAP** - a method is intended to use as a debugging trap, which is called when some event on data change occur for a class field. For a field, up to three traps can be set (for operations: read, write and add/insert/delete a value to a dynamic array field). A list of traps installed for a field is specified in its modifier **TRAP(trap1, trap2, trap3)**. A trap type is defining on base of its parameters:
 - for a scalar field, method without parameters is called when the field is accessed for read operation, and a method with a single parameter is called before changing the field, in such case the parameter should be of the same type as the field and it receives a new value assigning to the field;
 - in case of an array field, an **INT** parameter is added where an index of an item accessing is passed; in case when the first parameter is of type **BOOL**, the trap is called before delete / after insert/add operations, and the first **BOOL** parameter receives **TRUE** in case of add/insert operation, or **FALSE** in case of delete operation.
- **RESTRICT name IN [list]** or **RESTRICT name in {constants_block_name}** or **RESTRICT name IS CONST** - restricts a parameter with the given name: it can be only a constant (in the first case only from the list of values specified).
- **IF name IN [list]** - a function parameter with the given name is used to restrict values of another parameter (specified in the next restriction modifier **THEN**). A list can be replaced with a name of a constants block in figure brackets (see **CONSTANT** statement definition).
- **IF name IS CONST** - if a function parameter with the given name gets a constant, then the following restriction (modifier **THEN**) is active.
- **THEN name IN [list]** - a conditional restriction of a given parameter by only values specified in the list (a condition is specified by the previous **IF** modifier).

In case when some redefined operators are used in the function body, it should have the modifier **OPERATORS(list of classes)**. This does not depend on kind of a function (**FUNCTION**, **METHOD**, **CONSTRUCTOR**, **OPERATOR**, etc.) A list of used classes should be specified in parenthesis, in the order in which classes are enumerated while searching an appropriate operator.

II. 3. b. Function body

Function body consist of statements located and it is finished by a dot symbol.

- On any block level the body should not have more then 7 simple statements, 7 declarations of local variables and 7 block statements.
- A block can be separated by comments

```
----- 'label'
```

 (not more then 7 such comments for a block), in such case counting statements of different kinds starts again.

If the requirements above are not satisfied. then it is necessary to mark entire class with the modifier **BAD** (in the class header).

In case of a single assignment statement, in which an expression is assigned to the pseudo variable **RESULT**, the **RESULT** keyword can be omitted, and entire function body has a form

```
: = expression
```

(the space between ':' and '=' is not necessary).

II. 3. c. Calling functions

Parenthesis are necessary to be used only if actual parameters are present.

- Actual parameters are listed in parentheses comma separated in the same order as correspondent formal parameters are declared.
- Object parameter is passed implicitly. It can be used explicitly via a pseudo-variable **THIS**.
- If a function is returning a result, it can be called in a separate statement not assigning returning result. If the result is not required, it

should be "assigned" to a **NONE**, e.g. in form:
NONE = object.Function(parameters)

- Any static function can be called using a prefix style, when its first parameter is moved to a function prefix, e.g.:
x.Sin - instead of **Sin(x)**
s.Lower.Ending(".a14") - instead of **Ending(Lower(s), ".a14")**

III. Classes and objects

III. 1. Classes

III. 1. a. Class declaration

Syntax.

```

/*
CLASS {names}, modifiers:
IMPORT: {class1}...;
BASE CLASS {Base_class}
field1
field 2
...
method 1
method 2
...
----- 'section label1'
...
----- 'section label2 '
...
END
HISTORY
...
*/

```

One file contains exactly one class definition.
All other declarations (enumerations, functions, constants, fields) are located only in classes and belong them.
There are no global (static) class fields, all the fields are members of certain objects.

- The AL-IV code is always located between lines containing symbols of the end and the start of multi-line comments: `/*` and `*/`.
- Class names** are enclosed into figure brackets and always start from an uppercase letter.
- Class name can be separated onto parts by the symbol `'|'`. Before the symbol a short name of the class is defined, after the `'|'` symbol - the rest of a long name of the class.
E.g.: `{My|_class}` defines short name `{My}` and full name `{My_class}` for the same class.
- A long name must not be shorter then 8 characters.
- Class file name (w/o extension) should much the long class name (w/o figure brackets). But violating this rule leads only to a warning, not to an error.

III. 1. b. Class modifiers

A modifier for a class is written following the class names. Such as:

- ABSTRACT** - the class is intended only for deriving new classes on base of this class, it is not allowed to create objects of the class itself.
- BAD** - the class is not satisfying requirements on code style (not more then 3 parameters for a function, not more then 3 nesting levels of block statements, not more then 7 fields in a section of a class, not more then 7 simple statement + 7 block statements in a section of a block in a function code).
- BITWISE** - bitwise logical operations between integer values are used.
- BYTES** - bytes are used in the class.
- DESTRUCTORS** - the class has a destructor.
- NATIVE** - there are native (low-level) functions in the class (which depends on a target platform).
- OPERATORS** - there are operators redefined.
- RECURSIVE** - there are recursive functions in the class.
- STOPPING** - a **STOP** operation is used in the class.
- UNTESTED** - there is an untested code in the class.
- TESTED(nn)** - class was tested for about nn precents.

- **SAFE** - all is OK, class is tested fully and there are no warnings on a safety or style.
- **DEPRECATED('text')** - the class is still supported but its using is deprecated. The text in apostrophes contains information about an alternative which should be used as soon as possible still in future the class can become not supported.
- **ABANDONED('text')** - the class is abandoned. Any reference to it leads to a compilation error. The text contains information about an alternative.
- **INT64 REQUIRED** - it is required for the class to work correctly to use 64 bits integers in a project. If the project has the option /int32 in settings, this is not possible, and the compiler fires an error.
- **INT64 DESIRED** - it is desired for the class that the project should use 64 bits integers. But if it has the option /int32, the compiler warns about this conflict.

III. 1. c. Import section

The first in a class (following the class header) should be specified a statements **IMPORT** if these are required. In an import statement classes are listed which are used (including an ancestor class if the class is not descending from the common class **{Object}**).

IMPORT: {Class1}, {Class2}, ...;
(the semicolon or the dot can be used to end the IMPORT statement).

IMPORT statements can have modifiers:

- **TEST** - classes listed are imported only on the test stage;
- **DEBUG** - only when the **/debug** key is in project options (i.e. the project is building in the debugging mode);
- **FRIENDS** - classes are declared as friendly for the class defining (so, these have access to its hidden fields and methods as those are public).

III. 1. d. Inheritance

To specify an ancestor for a class its declaration is written in a form (immediately after **IMPORT** directives):

```
BASE CLASS <Base class name 1>
```

- If a construction **BASE CLASS...** is absent, then the class has no other ancestors except the common for all classes unnamed class **{}**.
- If there are methods in a class which are overriding correspondent methods of a base class then all those methods are declared starting from a keyword **OVERRIDE**:

```
OVERRIDE some_name(parameters) ==> result_type :  

...  

.
```
- To assign other initial values to fields of a base class and to do something else on a create time of an instance of a class, a procedure is used

```
CONSTRUCT :  

...  

.
```
- To do something special when an object is destroyed, a procedure is used

```
DESTRUCT :  

...  

.
```
- If a destruction procedure is used in a class it is necessary to add a modifier **DESTRUCTORS** to the header of the class.

when a method is redefined in a class:

- A prefix **OVERRIDE** is used instead of the **METHOD**.
- Only one name of the method is specified.
- All the parameters of a base method are listed (in parenthesis, comma separated). First letters of correspondent parameter names must much. Types must be compatible by types and by dimensions (for arrays).
- For methods returning a value:
 - data type of a returning value is specified.
 - An example:

```
OVERRIDE enabled ==> BOOL :  

RESULT =o.Edit1.Text!="" .
```

A redefined method can call a base method (the same named method of the base class).

- To call explicitly a base method in a body of a redefined method a syntax is used: **BASE.**
- All the parameters are passing by a usual way, and the object itself is passed implicitly as usual: **Base(parameters)** .
- A method not returning a value, should at least once call a base method, else it is necessary to add a modifier **REPLACE** to the overriding method.

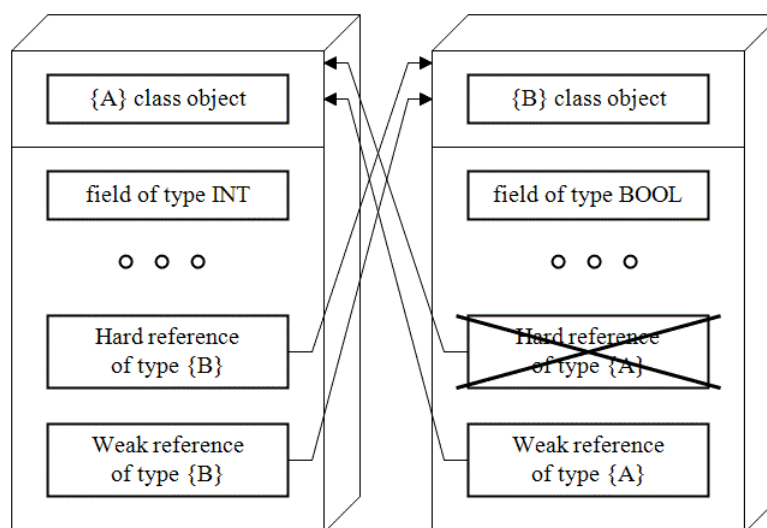
III. 1. e. Objects. Strong and weak references

A variable of a class type (a class instance) is **an object** of the class.

- Objects are declared by specifying a correspondent class name in figure brackets as a type name of the variable.
E.g.:
`{My Class} My_object={My_class}`
- A field or a variable not having trailing underscore characters in its names, is a hard (keeping) reference to an object.
- If an **object_** name has a trailing underscore character, it is defined not to own an object but to be just a weak reference to another object.
- Such **weak** object reference variable links to it just while it exist and there are object variables **referencing** such object strongly (or an owner of the object still exists).
- All the local variables in a function are strong references always.
- When a count of strongly referencing variables becomes zero, the object is destroyed and all the **weak** pointers start pointing to **NONE** value immediately.
- If at create time for an object an owner was specified explicitly (**OWNED BY**), then presence of strong references to it has no effect: it will be destroyed together with its owner, and at that moment all the references to it, either strong or weak will be redirected to the **NONE** object.
- **NONE** - is a special object of a class, having all the object fields equal to **NONE** (and 0 for numbers, "" - for strings, etc.). Writing to fields of such pseudo-object is always ignored, reading is always returning **NONE** and 0 values, calling its methods does not lead no any valuable operations.
- All the object variables initially are set to **NONE**, and if an object is read out of bounds of an array, also **NONE** is returned.

THE RULE

- A field **F_A** of a class **A** which is a strong reference to an object can not belong to a class **B** which is a direct ancestor of the class **A**.
- If a class **A** has an object field of class **B** which is a strong reference, then the class **B** can not have object fields which are strong references to class **A** or other classes which are direct ancestors or inheritances of **A**.
- When a class **A1** references strongly an object of a class **A2**, and the class **A2** references strongly an object of a class **A3** and so on, and a class **A(n-1)** references strongly an object of a class **A(n)**, then the class **A(n)** can not contain object fields which are strong references to any of previous classes **A1**, **A2**, ..., **A(n-1)**.
- It is not possible to refer with a strong reference to an object of the same class, in part.
- As a result, it is not possible to create **circled chains of strong references of objects to each others**.
- Different kinds of peer networks of objects, graphs, lists etc. can be easy formed using **weak_** only pointers.



- Just created object should be either
 - assigned to an external (for a function) object variable without an underscore at the end of its name (i.e. a strong reference to it should be created),

- or it should be added to an external array of strong references (an operation, **>> array[]**),
- or an owner object should be specified (an operation, **OWNED BY var_ref**),
- or it should be assigned to a (local) variable which has substring 'temp' in its name (in any case, e.g., Example|_TempVar) .
- A variable is an external to a function if it is:
 - either is a field of the class itself,
 - or it is a field of an external variable,
 - or it is a pseudo-variable **RESULT**,
 - or it is a field of the **RESULT**,
 - or it is a field of an object parameter.
- If a newly created object at least once in a function body is assigned to the **RESULT** variable, then the function should be marked as **NEW**.
- Function marked as **NEW** can be called only in a standalone statement like a statement where a new object is created directly. And the new object created with a function marked as **NEW**, also must be assigned to an external variable, added to an external array or assigned to a temporary variable.
- For a **NEW** function result it is not allowed to use **OWNED BY** instruction.

III. 1. f. Fields

Fields of a class are declared very like to local variables but on the same level as class methods, functions and other class declarations.

Field declaration (schematically):

TYPE Name[detailed [size], modifier = initialization

- **TYPE** is either one of basic types (**BOOL**, **BYTE**, **INT|EGER**, **REAL**, **STR|ING**), or a enumeration, structure or class in figure brackets.
- A total field name should be at least 8 symbols (not accounting '|'). If a name starts from a capital letter, then the field is public.
STR Public_s|tring_variable

Hidden fields names start from a lowercase letter.

INT local_i|nteger

If a field name ends with the underline symbol (only objects of classes), then it is a weak reference to an object.

{Matrix} Transform_ation_matrix_

In such case any part of the name separated by the symbol '|' should be ended with the underline.

- **[size]** is an (optional) array size (not applicable to scalars). If square brackets are empty, then the array is dynamic.
STR List|_of_strings[]

Size of a fixed array can be specified either by a numeric constant,

INT Ten_num|bers[10]

or by a name of a enumeration

BOOL Traffic_l|ights[{colors|_of_traffic}]

or by the symbol '*'

BYTE Pixels|_array[*]

(the last is very special case allowing to change an array size from a native code, and the same time to access it from high-level code as a usual fixed array).

Class fields can have modifiers listed comma separated following its declarations (but before an initialization if it is present).

- **READ** or **READONLY**- a field is declared accessible "only for read". Such field can be changed only by methods of the class itself, by its descendants and friend classes;
- **INIT** or **INITIALIZE**- a field requires to be initialized due to object creation. If to create an instance of a class a standard constructor is used in form
destination={Class_name}(Field1=value1, Field2=value2, ...)
and such field is not listed in the assignments list in the parentheses, this leads to a compiling error. If an **INIT**-field is also a read only field, then such explicit constructor can be used only in methods of the class itself, of its descendants or its friends;
- **TRAP(method, ...)** - specifies a list of trap methods for a field (up to 3 traps for a field); it is possible to list only methods of the class itself which do not return any result and:
 - if the field is scalar, then it is allowed to use methods without parameter (a read trap), or methods with a single parameter having the same type as the field itself (a writ trap which is called before assigning a new value, which is passed in the parameter);
 - if a field is an array then it is allowed to use following combinations of parameters:
 - **(INT)** - a trap for read an item with an index (specified by the parameter);

- **(INT, {field type})** - a trap for writing a value into an array item (with a certain index specified by the first parameter, the value assigning is passed as the second parameter) ;
- **(BOOL, INT)** - a trap on changing array size: the first **BOOL** parameter is **TRUE** in case of add/ insert operation (and such trap is called after insertion operation); in case of delete operation the first parameter receives the **FALSE** value (and delete trap is called before deleting the value).
- traps are not purposed to modify program logic, and it is not correct to modify values assigning or reading (though it is still possible to change array size - but do not do so). A task for traps is to simplify searching problem operations while debugging only;
- **CLAMP**- the field array of a fixed size (equal to 2ⁿ) which is clumped while indexing: while accessing an item 256 of an array with size 256, actually the item 0 is accessed, the index 257 is converted to 1 etc.
- **DEPRECATED('text')** - field is deprecated, you are warned if use it;
- **ABANDONED ('text')** - field is abandoned, the compiler fires an error in case when it is used in code.

III. 1. g. Methods

To declare a method rather than a static FUNCTION, the keyword **METHOD** is used. All the class methods are virtual and always can be overridden in a derived class (using keyword **OVERRIDE** in place of the **METHOD**).

- The key peculiarity of a method is: its first parameter is the object of its own class. It is not listed in a formal parameters list. If it is necessary to call it explicitly, the reserved word **THIS** can be used. To access field or methods of the class itself (and to fields of an ancestor class or methods of descending classes), the object **THIS** is used implicitly without writing it in code:

```
CLASS {Example1_using_of_fields_in_method}:
STR Field1_for_example="Hello, this is Field!"
METHOD Print_Field: << Field.
END
```
- There is a polymorphism there: calling a method leads to call a method assigned for a certain class instance, representing an object.
- It is always possible to call from an overridden method the correspondent method of the ancestor class: **BASE** or **BASE(parameters)** . If necessary, several times, in different parts of code of the method overridden (but only in that method).
- A method not having parameters should call the **BASE** method at least once, or otherwise it should have the modifier **REPLACE** (talking to the compiler that the method is replacing its previous implementation totally, and does not require to call the ancestor's implementation).
- The constructor always calls the constructor of the ancestor before running, implicitly. It is not possible to call it's **BASE** explicitly in a **CONSTRUCT**.
- The destructor also calls the ancestor's **DESTRUCT** implicitly, but at the end of its working.

III. 2. Working with objects of classes

III. 2. a. Creating an object instance

Object construction:

```
variable={Class_name}(
Field1=expression, Field2[] << expression, ...)
```

- If a value is assigned to a variable in a declaration statement and actually the same type object is created as the variable declared, then type name of the declared variable can be omitted:

```
stream1_read_temp={File_stream}(
Path=source_path, Mode= 'READ')
```
- If fields initializations are not necessary, parentheses are omitted.
- If an operation **OWNED BY expression** is followed (comma separated), this assigns an owner for the object created.
 - For such object, its strong usage counter does not affect its life time: it exists until its owner is destroyed.
 - If at the creation time the object assigned as an owner is referencing **NONE**, then the object owner is not assigned and it is most possible that the object created will be destroyed on a function end.
- Either an operation of adding the object created to an array of strong references may follow (comma separated):

```
, >> variable[]
```
- An object created should be either assigned to an external (for a function) object field or variable, or it should be added to an external array of strong references using

postfix operation: `, >> array[]`
 or an owner should be assigned to an object using postfix
 operation: `, OWNED BY value`.
 In other cases a variable on the left side of an assignment should
 have a substring 'temp' in its name.

III. 3. Other class level operators

If the class has an ancestor in its hierarchy, this should be specified with the statement **BASE CLASS {Name}**, but after the **IMPORT** statements section where the ancestor class is listed.

A class is always ended with a standalone statement **END** located in a separate line.

Before the **END** statement following statements/blocks can be found:

- enumeration declarations

```
ENUM {name|detailed}:
    'NAME|DETAILED1',
    'NAME|DETAILED2',
    ... .
```
- structure definitions

```
STRUCTURE {name|detailed}:
    Field1
    Field2
    ... .
```
- table definitions

```
TABLEN ame|detailed: {structure}
    NAME "name"
    COUNTER(list)
    NOTNULL(list)
    NAMES(field="name", ...) .
```
- field declarations

```
TYPE Name|detailed [size], modifier= initialization
```
- constructor declaration

```
CONSTRUCT:
    BODY.
```
- destructor declaration

```
DESTRUCT:
    BODY.
```
- static functions declarations

```
FUNCTION Name|detailed(parameters) ==> RESULT_TYPE, modifier1,
modifier2, ... :
    BODY.
```
- methods declarations

```
METHOD Name|detailed(parameters) ==> RESULT_TYPE, modifier1,
modifier2, ... :
    BODY.
```
- overridden methods

```
OVERRIDE Name(parameters) ==> TYPE, modifier1, ...:
    BODY.
```
- test functions

```
TEST Name|detailed (parameters) ==> TYPE:
    BODY.
```
- operator redefining

```
OPERATOR TYPE OPERATION TYPE==> TYPE:
    EXPRESSION.
```
- block comments

```
----- 'text'
```
- native code directives

```
NATIVE: "code specific for a platform" .
```
- to-do statements

```
TODO "text" .
```

III. 4. Ending class. History of changes. DATA[] array.

A class is ended with the directive **END**. If following the **END** statement the first non-empty line starts with the keyword **HISTORY**, then the history of changes of the class is located there. Independently of changes history present or not, all the lines after the **END** directive if these are not empty become accessible from the class code via the pseudo-array of strings **DATA[]**.

If the **HISTORY** directive is present, then the history should be in sufficiently strong form. It consists of blocks **CREATED/ UPDATED** (where the **CREATED** can be the first and a single only if it is present). The block header format is:

```
(' CREATED' | ' UPDATED' ) '(' YYYY-MM-DD )' [ ',' ' VERSION' ']' text
[' BY' 'text' ] ':'
```

The block consists of messages in form:

```
(' ADDED' | ' CHANGED' | ' FIXED' ) ':' ('text' | identifier)
[ ',' ('text' | identifier) ]... (';' | ',')
```

The last message of a block should end with the dot, or it should consist from the dot only.

The history of changes can be ended together with the end of the class text. But in case when the text after the END directive contain also another information additionally to the history, it should be ended with the special directive

HISTORY ENDED

E.g.:

HISTORY:

```
UPDATED(2018-08-15), VER "1.0a":
  ADDED: "This history added";
  CHANGED: "No other changes yet".
```

IV. Structures (STRUCTURE)

IV. 1. Declaration of a structure

Syntax.

```
STRUCTURE {name|_of_structure}:
  field1
  field 2
  ...
  field N.
```

Structures are declared in a class body.

Name of a structure is always start from a small letter.

The name can be separated with the symbol '|': the first part is a short name of the structure. Full name of a structure can not be shorter then 8 characters.

All the structures declared in a class are always accessible for all the classes importing that class.

A structure contains of:

- fields of simple data types of fixed size (**BOOL**, **BYTE**, **INT**, **REAL**, {enumerations}, **STR**)
- fields of structure types (any other structures in a scope of view but recursive nesting is not allowed)
- references onto class instances
- arrays of fixed size of the listed above items
- references to other structures (to include their fields into themselves on the same level), in form **LIKE {structure_type}** (or **LIKE {class_name}.{structure}** if it was declared in another class).

A structure declaration is ended with the dot symbol '.'.

A structure like a class can contain fields of any type. But there are restrictions on fields which are objects of classes: such fields of structures must be weak references onto objects (and its names should be ended with the underline character).

If a class field is a structure, it must not be a weak reference since any variable of a structure type is a single reference onto its structure.

All the structure variables and fields are automatically initialized with a structure (of the corresponding type), which consists of zero fields (for numbers these are zeroes, for strings these are empty, for objects - **NONE** values). A **NONE** value for a structure can be achieved in result of reading from an array out of its bounds.

IV. 2. Working with structures

Structure is a class without methods, objects of which are always referenced by a single reference. So when an instance of a structure is created, it is not possible to use the **OWNED BY** modifier or **>> Array[]** operation unlike for classes.

If one structure variable is assigned to another structure variable, then the right side of the assignment should be ended with a call to the embedded function **Clone** or **Dismiss**. In case when **Clone** is called, the source structure is duplicated while assigning, in case of the **Dismiss** the source variable can become equal to **NONE**.

```
Destination=Source.Clone
Destination=Source.Dismiss
Destination=Source_array[i].Clone
Destination_array[i]= Source_array[i].Clone
Destination_array[] << Source.Dismiss
```

E.g.:

```

STRUCTURE {person|_info} :
  STR F|first_name
  STR L|last_name
  INT Y|ear_birthday
  {education} E|ducation
  STR S|tate_province.
...
{person} Smith|_John={person} (F="John", L= "Smith",
                               Y=1950, E='High', S="NY")

```

A similar constructor can be used to pass an actual parameter to a function in place of a structure parameter.

When a structure is assigning (to another STRUCTURE variable or field), its content just is copied.

When passing a structure as a function parameter, it can be used in the called function only for read, and it is not allowed to change its fields. This differs from rules of using objects of classes, for which it is possible to change its fields in a called function.

Correspondently, when assign entire structure to another structure variable, its possible (and necessary) to call only Clone pseudo-method and not Dismiss.

A structure can be used as a result of a function. When working with the **RESULT** variable of a structure type it is not necessary to initialize it still it is always initialized by the default NONE-value.

Structures can not be passed as parameters or returned as results of native (low-level) functions.

A structure can be created with an operator creating a new object of certain class:

```
{structure-type-name}(field1=expression, field2=expression, ...)
```

For example:

```
{complex} c|complex_var={complex}(Im=5)
```

If the **FOR** loop is enumerating some array of structures then its loop variable actually becomes a macros allowing to access items of the array, both for read and write. I.e. if to assign some value to a field of the loop variable, actually the correspondent array item field is changed:

```

FOR x IN A[]:
  x.Some_field=Some_value.CLONE;
  // in result all the items of the A[] array
  // have field Some_field equal to Some_value

```

It is possible to assign structure variables of different types but having so named fields (which are of compatible types):

```
x, BUT (Fieldx1, Fieldx2, ...) =y, BUT (Fieldy1, Fieldy2, ...)
```

In the left **BUT** clause it is necessary to list fields from **x** which should be skipped (usually, these are fields which are present in **x** but not present in **y**). In the right **BUT** fields from **y** are listed which should not be assigned (usually, these are fields not present in **x**). In a minimal case when both lists are empty, the left **BUT** is written anyway (and without brackets):

```
x, BUT = y
```

Methods **CLONE/DISMISS** are not used when structures of different types are assigned.

Structure variables can be used with SQL statements (**SELECT**, **INSERT**, **UPDATE**, **DELETE**) while working with databases. See more detailed in a section devoted to the embedded SQL support.

IV. 3. About structures implementation in the final code

No assumptions should be made how fields of structures are located in the operative memory at runtime or about its ordering and aligning onto machine words bounds. The same is about its sizes in memory and / or mechanisms of allocating memory for them. Also, any assumptions about their effectiveness (in comparison with classes or sets of variables declared separately) can be totally incorrect.

Structures can be actually implemented as objects, or a subset of simple structures (which do not contain dynamic arrays, strings and object references), which can be implemented as actual structures in a target language.

All what really necessary to know about structures are the fact that structures actually like simple variables are copied while assigned, and it is not possible to refer to them (using any pointers). Though, while passing them as function parameters, actually references are passed (but these are read only like simple data types parameters, and its fields can not be changed in a function).

The obligatory requirement to use pseudo-functions **CLONE / DISMISS** is introduced to provide for a programmer that (s)he always take into account that in that case data are copied rather then referenced (and changes in a destination variable will not affect a source variable). This should decrease amount of errors caused by a misunderstanding that fact.

V. Testing

V. 1. General rules

Class can contain special functions starting with a keyword **TEST** in place of the **FUNCTION**. Such functions are intended to test and executed always on a compilation stage.

- If tests are not executed successfully, a class should be marked with an attribute **UNTESTED**.
- If not all lines of code were executed on a test stage, the class should be marked as **UNTESTED**.
- If at least one statement **ASSERT** was failed, the class is not compiled (like there is an error in code).
- If a class should be marked as **UNTESTED** but it has no such attribute, this concerned as an error, and a project is not compiled.
- A class can be marked as partially tested using the modifier **TESTED(n)** . In such case it is sufficiently to have n percents of lines in code to be tested.

which code should be tested:

- All the lines of code.
- All the conditional branches.

A set of tests is concerned to be "covering" only if all the lines which should be tested were executed during a class testing at least once. If this condition was not satisfied, the class should be marked as **UNTESTED**.

All the sections of code in a **TEST** function should have **ASSERT** statements. The compiler can no control correctness of usage of such statements though, so an absence of **ASSERT** statements leads just to warnings.

what is not necessary to test:

- Native functions.
- Abstract classes.
- Classes not having methods.
- Methods not having statements.

Therefore while testing a class inherited from an abstract class all the not redefined methods of an abstract ancestor should be tested to ensure that a derived class is tested.

It is allowed to place in an abstract class some testing functions with parameters, intended to test partially (or totally) functions of an abstract class.

Tests with parameters are not called automatically but these can be called from other **TEST** functions. In part if to create such parameterized tests in an abstract class these can be used to simplify testing the abstract class together with derivatives of it.

Testing is executing every time when a project is compiling (except cases when results of the previous testing are saved and the code did not changed from which the testing code depends, or tests was completely disabled with the key / \$NOTESTS - but this depends on the compiler).

During a cross-platform compiling to some target platforms (Java/Android) the testing can not be performed on the compiling stage, but using the special compilation option it is possible to build a special (pseudo-console) version of the application purposed to run tests only. But integrating results of testing into the process of compiling is not possible in such case, and results of testign should be controlled manually.

In case of cross-compiling to the Linux (from the windows) testing is possible on the source platform. But it is necessary to take into account, that the reliability of results is not 100% (still platforms are different, the same code can give different results on these two platforms). Therefore cross-platform compiling for Linux is done once for the compiler itself, and later other projects are building on the Linux using this compiler.

V. 2. Syntax

Testing function starts from a header which differs from a function header by a keyword **FUNCTION**.

Testing functions can have parameters but such parameterized tests are not called automatically: these can only be called from other tests.

If some additional classes are required for testing functions, it is necessary to list them in **IMPORT** statement with the **TEST** modifiers. E.g.:

```
IMPORT, TEST: {String_functions} .
```

Like other functions testing function also is separated onto sections with block comments

```
----- 'text'
```

But for testing functions an additional requirement is there: each section should contain at least one **ASSERT** statement.

An **ASSERT** statement is intended to check a Boolean expression value. If a value in an

ASSERT expression

is false, then assertion is failed. When at least one failed assertion is present then the class is concerned untested.

The **ASSERT** statement can have the second (string) argument:
ASSERT x, y - it is calculated and included into the message about a failed assertion statement to simplify understanding the problem.

VI. Embedded SQL support

VI. 1. SQL queries encoding

If a class has a method **write(STR)** , then for its objects it is possible to use operation

```
OBJECT << STRING
```

It is actually just calls the write method.

In case of class **{DB}** (which is intended to work with databases), the operation << should be used to set up an SQL query text.

```
db << SELECT (*), FROM Students a,
      JOIN Students b ON (a.Exam1=b.Exam1),
      ORDER BY (Surname, Firstname, Middelname)
db.Open
```

VI. 2. Database tables structure. TABLE operator

When a text of SQL is set and string expression is started from one of keywords **INSERT** / **DELETE** / **UPDATE** / **SELECT**, then entire expression is treated as an SQL-like statement which creates a final SQL text at run time. And if not all but many of parameters of such SQL-like statement are checked at compile time, minimizing possible problems at run time later.

Such SQL queries a referencing "tables" declared by **TABLE** declarations (on base of **STRUCTURE** s). E.g.:

```
STRUCTURE {abiturient}:
  REAL ID|entity_counter_64bit
  STR Surname|_last_name, MAXLEN[40]
  STR FirstName, MAXLEN[40]
  STR MiddleName, MAXLEN[40]
  INT Exam1|_scores
  INT Exam2|_scores
  INT Exam3|_scores
  {date_time} D|date_in_documents
  BOOL OriginalDocuments
  BOOL AgreeToEnroll.
```

```
TABLE Students|_want_to_be:
  {abiturient}
  NAME "Students"
  COUNTER (ID)
  NOTNULL (Surname, FirstName, D)
  NAMES (D="DateJoin") .
```

VI. 3. SQL-like syntax

Queries are similar to SQL but syntax is different a bit:

- All the keywords are written in uppercase: **SELECT**, **UPDATE**, **INSERT**, **DELETE**, **DISTINCT**, **TOP**, **FROM**, **AS**, **INTO**, **WHERE**, **GROUP**, **ORDER**, **BY**, **IN**, **NOT**, **IS**, **NULL**, **JOIN**, **ON**, **LEFT**, **OUTER**;
- All the parts of an SQL-like statement are separated with comma to simplify splitting long queries onto lines. E.g:

```
SELECT DISTINCT, FROM Students, (*), WHERE Graduate_date BETWEEN {d1}, AND {d2}
```

- A list of selected fields and values as well as lists of grouping and ordered values is enclosed into parentheses;
The list of selecting values can be placed just after **SELECT** word (or **SELECT TOP(n)**, **DISTINCT**) either after the **FROM-JOIN-JOIN** -... part;

- If it is possible, all identifiers first are treated as references to tables, table aliases and field names. And only in cases when these are not matching correspondent identifiers, these are attempted to use as AL-IV variables, functions etc. E.g.:

```
UPDATE Students, SET Exam3=64,
WHERE ID={Ident_to_update}
```

- If it is necessary to specify explicitly that part of text in a statement is an AL-IV text, the **AL-IV expression is enclosed into figure brackets**. In a **WHERE** clause this is the only way in many cases to use values calculated at run time; All AL-IV expressions used directly in an SQL expression should have types **INT** / **REAL** / **BOOL** / **STR** / **{date_time}** / **{enumerated}** (and should to correspond to field types). Other types are not allowed (but **BYTE** in many cases is treated as **INT**);
- To represent counter fields (auto-incrementing) it is necessary to use type **{id}**, still usual integer type precision is not enough to represent long integer value, which are used for counters in some databases;
- AL-IV expressions are automatically converted to SQL compatible strings, to treat those in SQL queries as indirect constants. So it is not necessary (and not desirable) to convert these into strings using such functions as **Bool_sql**, **Int_sql**, **Str_sql**, **Real_sql**, **Date_sql**;
- In the **INSERT** statement, a list of fields and assigned values is specified like in the **UPDATE**:

```
db << INSERT INTO Students,
Surname= {"Origatsu"},
FirstName= {"Pei"},
MiddleName= {"Q"},
Exam1={84} ,
Exam2={81} ,
Exam3={92} ,
D={ Date(2017, 7, 14)},
OriginalDocuments= { TRUE }
```

```
db.Exec
```

- In **INSERT** and **UPDATE** statements it is allowed to specify a variable (or other expression) of correspondent to table **STRUCTURE** to set values, instead of listing all the fields and its new values. E.g.:

```
db << INSERT INTO Students, BY Rec
db.Exec
```

- In the previous statement, it is possible to add a clause **BUT** providing a string array containing names of fields which should be omitted while setting new values, e.g.:

```
db << UPDATE Students, BY Rec, BUT array[]
```

- If a query contains more than a single table, the only way to specify additional tables is in adding **JOIN** parts just after the first table. In such case, all the tables should have unique aliases specified to make it possible to reference its fields. E.g.:

```
db << SELECT FROM Students a,
JOIN Students b ON (a.Exam1=b.Exam1),
( a.ID AS a_ID,
a.FirstName AS a_F,
a.MiddleName AS a_M,
a.Surname AS a_S,
COUNT (b.*) AS CNT_b),
GROUP BY (a_ID)
```

```
db.Open
```

- Nested queries are allowed in operations: **X IN (SELECT...)**, **X NOT IN (SELECT ...)**, **EXISTS(SELECT...)**, **NOT EXISTS(SELECT...)**
- When it is necessary to insert some text into a query, use the pseudo-function **SQL(...)** which operand is an AL-IV expression returning a string (or just a string literal). For example:

```
... WHERE x IN SQL("(1,2,3)"),
AND SQL("GetDate()") BETWEEN b.D1 AND b.D2
```

And the text inserted is always enclosed into parenthesis, so it is possible to write, e.g.:
... WHERE Id IN SQL(List_id[0 TO *].Merge(","))
(not adding string literals "(" and ")" to the list specified.

- Lines of a query can be separated with block comments

```
-----'comment'
```

Part of a query between such comments can be deleted before executing a query calling the method **Remove_after("text")**. This can be used to create queries with a variable code (therefore the compiler checks the entire code of a query). For cases when it is more convenient to specify which of such parts should be left rather than directly remove others, finish such comments with question mark '?' and use the method **Allow("text")**. And even if the method **Remove_not_allowed** did not call explicitly, anyway all the optional parts are deleted just before executing the query (removing all the text between comments and starting with comments

```
-----'text?' which were not present in Allow methods called before and until the next
block comment or to the end of the query).
```

VL 4. Executing queries INSERT, UPDATE, DELETE

Queries **INSERT**, **UPDATE** and **DELETE** do not require results. These are just set into DB-connection (of class **{DB}**, let it will be a variable **db**), and executes by the method **Execute**:

```
db << INSERT INTO Students,
Surname= {stud.Last_name},
FirstName={stud.First_name},
MiddleName={stud.Mid_name},
Exam1={exam[0]},
Exam2={exam[1]},
Exam3={exam[2]},
D={date_exam},
OriginalDocuments={ TRUE}
```

db.Exec

To obtain an identity value of just inserted record use the method `identity(STR Table_name, STR Identity_field)==>{id}` since the correct way to get it is different in different databases;

Similar, to get amount of records inserted or updated, it is necessary to call a method `Row_count`. But this operation can be not available for some kinds of databases, so it is marked as `DEPRECATED('Depends on database kind')`.

When inserting records using the INSERT statement or changing records using the UPDATE, code can be shortened if to use a capability to insert/update database from a structure corresponding a database table:

```
db << INSERT INTO Table_name, BY structure_var
db << UPDATE Table_name, BY structure_var
```

In such case it is possible to skip some fields which names are listed in a string array:

```
db << INSERT INTO Table_name, BY structure_var, BUT null_fields[]
db << UPDATE Table_name, BY structure_var, BUT skip_fields[]
```

If a base is configured to set values to NULLs by default, all skipped fields become NULLs.

VI. 5. Getting results of SELECT statements

To obtain results of a SELECT query:

- it is recommended to use a loop in form `FOR x IN ENUM db.Results:...;`
- in such case it is not necessary to write manually code to check ending of data requested and jumping to the next record on each iteration still all these functions are done by the method `Results==> BOOL;`

```
db.Open
FOR j|dummy ENUM db.Results:
  id=db.CInt("ID")
  name= db.CStr("Name")
  STR r|results_array[] << "id=" id " name=" name ;
db.Close
```

- To get results it is recommended to use either a FOR loop with a preliminary check if there are results actually:

```
db.Open
CASE !db.Ended?
  FOR i|dummy IN [1 TO 999_999] :
    {results_tab} r|results1 << db
    {results_tab} all|_results[] << r
    CASE !db.Next ? BREAK i ;
  ;
db.Close
```

or a loop with a check on each iteration:

```
db.Open
FOR j|dummy IN [0 TO 999_999]:
  CASE db.Ended ? BREAK j ;
  id=db.CInt("ID")
  name= db.CStr("Name")
  STR r|results_array[] << "id=" id " name=" name
  db.Forward ;
db.Close
```

- If it is necessary to get just a single value as a result of a query, there is a set of functions `Open_VInt_close`, `Open_VStr_close` etc. E.g.:
`REAL x|_some_value=db.Open_VReal_close`

When getting results selected:

- it is possible to read query results using the special form of the assignment statement `<<` in which in the left part a variable is specified of a `STRUCTURE` type (corresponding to a table selected), and at the right side a `{DB}` class variable (or of an inherited data type), with additional reference to a table, in form:

```
db.Open
FOR i|dummy ENUM db.Results:
  {results_tab} r|results1 << db, BY TABLE Students
  {results_tab} all|_results[] << r;
db.Close
```

In such statement a variable at the left side must not be the same `STRUCTURE` type as the base of a table Tab but all its fields must have non-conflicting correspondence to that `STRUCTURE` type;

- Either it is possible to use methods of class `{DB}`: `VInt(n)`, `VReal(n)`, `VStr(n)`, `VDate(n)`, `VId(n)` passing an index of a value selected;
- Or to use methods `CBool(name)`, `CInt(n)`, `CReal(name)`, `CStr(name)`, `CDate(name)`, `CId(name)` passing as a parameter a field name constant string (these methods are restricted allowing passing only constant string parameter name). And, this variant is still effective: these methods have a hidden parameter which allows caching a found index of a field selected. In result, from the second call actual value is obtained using a numeric index cached, without searching it again.
- There are also methods `VNull(n)` and `CNull(name)` returning logical "true" if the value obtained for the field is `NULL`.

VI. 6. Transactions

Transactions are implemented via the PUSH-method Transaction. It can be called only in the header of a `PUSH` block statement:

```

PUSH DB.Transaction :
FOR sel IN Containers.Selection[] :
    DB << UPDATE Package,
        SET Barcode = {E_Barcode.Text},
        WHERE Packid = {Containers.[sel,
            .Columns[] .Count-1].Id_from}

    DB.Exec
;
CASE !DB.Commit ?
    Main_Handle_error(
        "{Containers_param}.value_change(E_BARCODE)"
    )
;
;

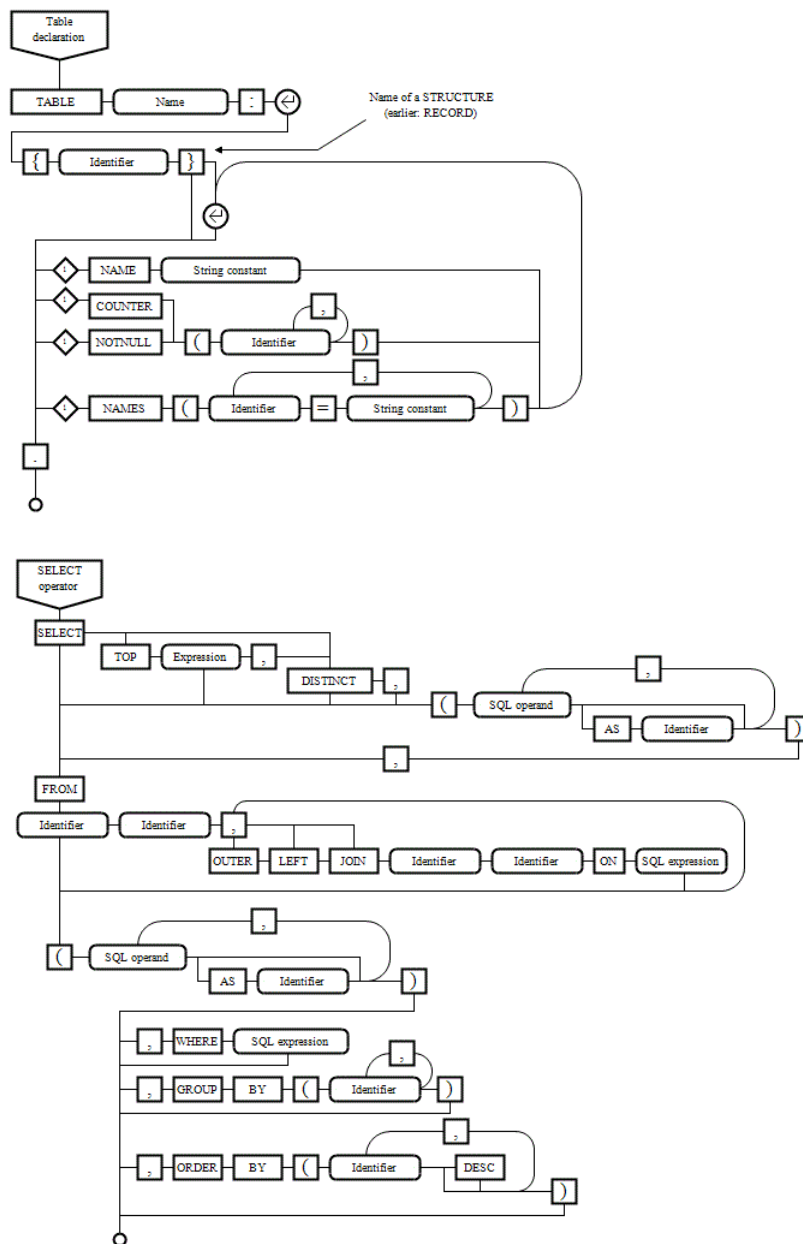
```

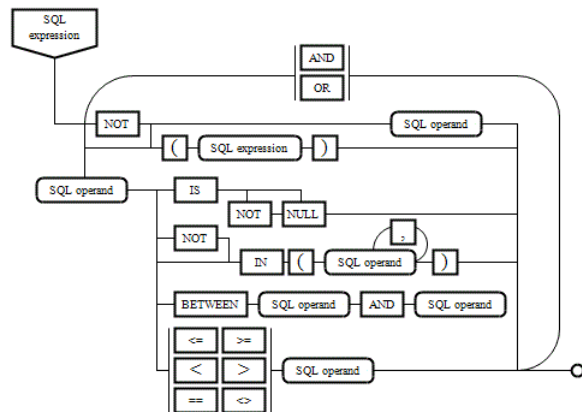
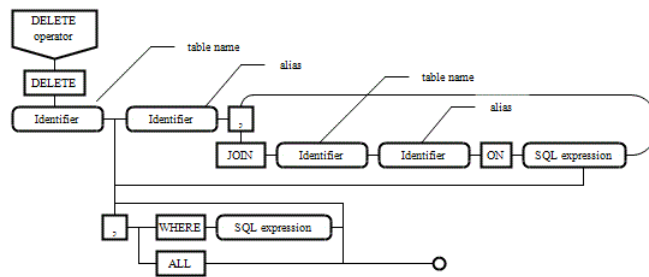
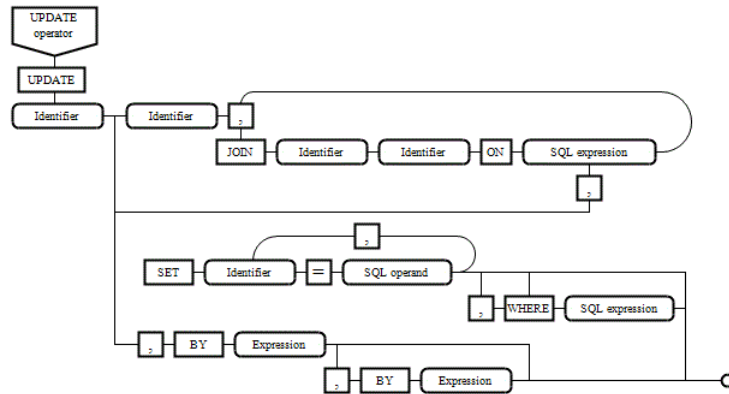
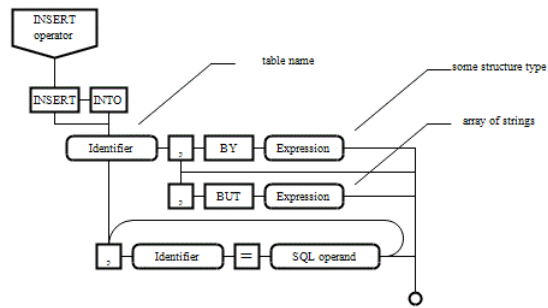
If code was successfully executed until the ';' finishing the PUSH block, then (dependently on the fact if a **Commit** or **Rollback** was called, and which of them was called last), operation either is finishing successfully (operation "commit"), or all changes were reverted (operation "rollback").

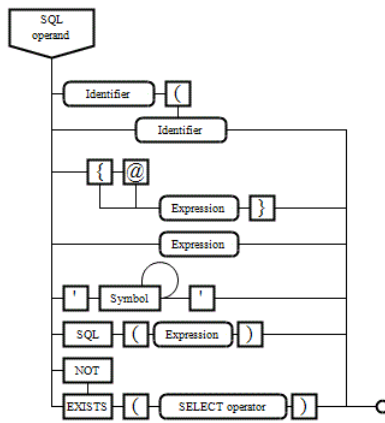
To prevent from the programmer to "forget" writing Commit, the method Transaction has the modifier **WAIT(Commit, Rollback)**. So, if there were no such calls in a PUSH **Transaction** block, the compiler generates an error.

VI. 7. Syntax diagrams

There are syntax diagrams below for statements generating SQL queries:







VII. Additionally

VII. 1. Localization of string resources

The AL-IV programming language supports localization of strings via pseudo-functions in form of

`_identifier("Red square")` or `"Red square"._identifier`

This construction looks like a function call but it is not a call to a function. But it tells to a compiler that a string "Red square" should be placed into an array of localized strings, remember its index in the array and use it to extract the string from the localization array. Actually, another string can be extracted, if it was replacing the original one in result of working of localization operations.

All the names of such string resources (pseudo-functions) should be unique in a class.

A special class `{Localize_str}` is suggested to control localization process (though this is not obligatory and it is always possible to write another class to do this).

When using the class `{Localize_str}` it is necessary at least once (on an application initialization) to call a static method `Localize`, specifying a localization language short name. It is supposed that the name of a current language selected an application itself is storing at some suitable place (e.g. using the class `{Configuration}`). To simplify a setup an application to allow selecting a language by user, a method `List_languages` is provided returning a list of full and short names of available languages (semicolon separated) extracted from names of language files (supposing these have names in form like `English_EN.lng`).

The method `Localize` has an additional parameter `Prefix` which allows translating not all the strings, but only those having names starting from a certain prefix (the `Prefix` is provided without leading underscore character). An empty string means translating all the strings independently of resource names.

When the `Localize` method is called it is first is calling a method `Save_untranslated`, if it was not called before from the application. A translator always can find the primary language file saved by this method, make a copy from it, rename it to `Some-language_SL.lng` and edit it. Strings which should be translated has a form `NAME=value`. It is necessary to replace only values leaving names as is. An edited version of a language file should be saved in UTF-8 format.

By default a directory where an application is located is used as a place to store language files. Or if it is impossible to write there a special application data directory is used. Or it is possible to set such directory explicitly (calling `Set_directory_lang`).

It is important that if new strings to translate were added while developing an application, then these are spread not only to an original language file stored by `Save_untranslated` but to all other (already translated) language files (though not yet translated).

VII. 2. Localization of the language keywords

The AL-IV supports translation of its keywords from any human language. For this purpose a localization technique (from above section) is used partially.

All the English keywords are stored in a text file `Default_.lng` in a directory with source files of the compiler. It is sufficient to copy this text file, rename it e.g. to `Klingon_KL.lng` and replace string values from English to some other. Language keywords have names starting from the letter "K" and these are mainly in the section `[{Translation_to_canonical_keyword}]`.

To use national keywords in your code, a class should be started from a specification ['KL'] or [Language='KL'], following which the first keyword (CLASS) is already translated into the language specified.

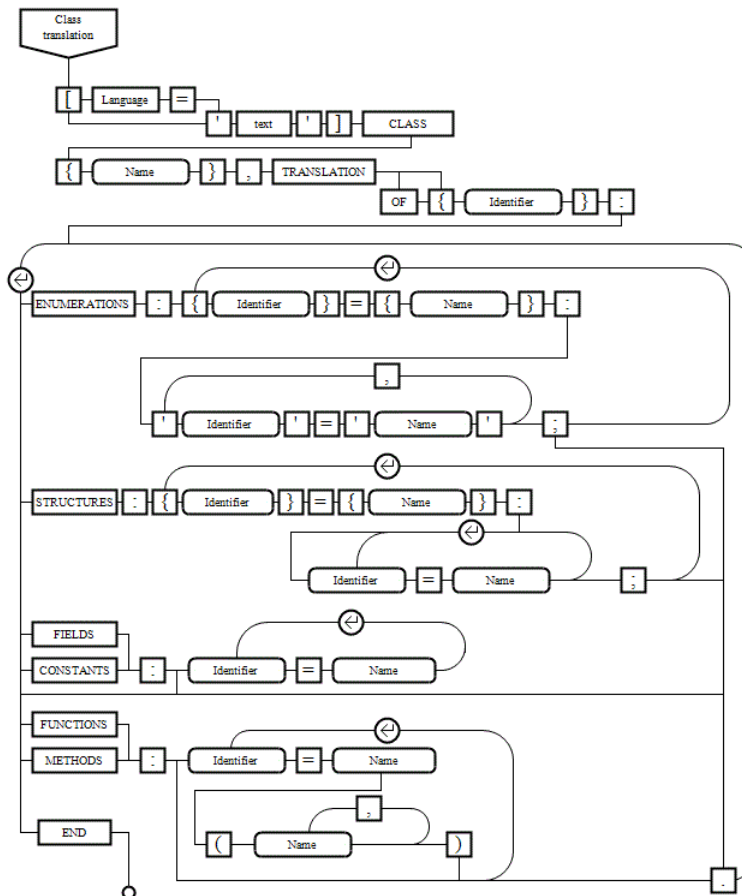
Since a translation language can have some word morph rules or have some variations depending on a lexical context, especially for keywords it is allowed to relate to a single canonical keyword more then a single identifier.

A translation can have a form of several word sequences comma separated, where each sequence have a form root|suffix1|uffix2|..., and any suffix can have a space or start from a space. In result, all the words root, rootsuffix1, rootsuffix2, ... are corresponding to the keyword specified (and a space in a suffix means that two identifiers spaces/tabs separated are treated as the target keyword, too).

Together with the language keywords, encoded characters are translated (#NL, #TAB etc.) and some embedded pseudo-functions such as (.Index, .Len, .Str). And it is allowed to translate whole classes to national languages. To do so, for a translating class a special mirror class is created which have a single modifier **TRANSLATION OF {Class_name}**. It contains only translations for fields, functions, methods, enumerations, structures, constants, placed in correspondent sections.

Then it is sufficient to include such translation class into import and call its methods, functions, fields etc. using translated names.

A class format for a translation class, in form of a diagram:



VII.3. STORE - hidden parameters

A modifier **STORE** for a method creates hidden (for a caller side) integer parameter which value is stored on a caller side.

A name of a parameter is specified in parenthesis, and also its initial value can be specified additionally:
, STORE(Name=value)

For each method call in a calling class a hidden integer field is created for the stored parameter, and this field is passed to the method by a reference implicitly. Passing by a reference means that if the method changes a STORE-parameter, then after it is finished, new value become the value of the hidden field. And on the next call of the same method in the same line of code, this new value will be passed as an additional parameter.

To reset stored hidden parameter values it is possible to call a method having a modifier **FORGET**. Stored parameters created to pass them to methods of the same class as the **FORGET** method will be set to its initial values (specified in the **STORED** modifier).

Modifiers **STORE** and **FORGET** together allow speed up a lot accessing data arrays indexing by constant strings. E.g., while obtaining values of fields of records in **SELECT** results from database, or while accessing items and sub-items of a list view by headers of columns.

An example of accessing listview by column names:

```
FUN Cells|_by_constant_names(
    INT Row|_index,
    STR Name|_of_cell) ==> STR , STORE(I|index_of_name = -1),
                        RESTRICT Name IS CONST
:
CASE I < 0 ? I = Column_index(Name) ;
RESULT = Subitems(Row, I) .
```

In the above example, the **FORGET** modifier should be assigned to all the methods changing columns set and its names. The advantage in speed of the version with **STORE** in contrast to usual method can differ by orders of magnitude since searching of an index is performing only on the first call of the function.

VII. 4. Abandoned and deprecated classes, structures, enumerations, fields, functions

while your classes and libraries are developing, you can find it necessary to re-write some things. Some functions, fields and so on becomes too old, supporting it costs more and more, some things are replaced with others etc.

To make it possible to control on a language level of a process of a distinction of old entities and provide smooth transition to new features, following modifiers are added to the AL-IV:

DEPRECATED ('text') and **ABANDONED** ('text') .

- These modifiers are applicable to classes, structures (**STRUCTURE**), fields of classes and structures, enumerations and functions.
- A text in parentheses (and apostrophes) should contain a name of an alternative or at least a short clarification why the feature is marked as **DEPRECATED** or **ABANDONED**).
- Modifiers **ABANDONED** and **DEPRECATED** are mutually exclusive.
- It is suggested for all things become obsolete to add a modifier **DEPRECATED**, and keep it until the feature become not supported at all, and then replace it to **ABANDONED**.
- When a deprecated feature is used in code, a compiler generates a warning containing the text specified in the modifier.
- If the modifier **ABANDONED** is added then correspondent feature can not be used: compiler will generate an error. For an abandoned function, it is possible to remove its body, for abandoned class - to remove bodies of all the functions.
- The sense in storing information about abandoned features in a class code is in providing an information about an alternative (in the text in parentheses which should contain name of such alternative function, class or other way to get desired functionality).

There is another possible usage of modifiers **DEPRECATED** and **ABANDONED**: these can be used an inherited class to refuse from same functionality of the predecessor class, since it is not working in the descendant class.

For example the class {**Dialog**} which is a descendant of the class {**Form**} do not use the method Show which is declared as **ABANDONED**('Show_modal') . This is done so to prevent a programmer using incorrect method to show a dialog form.

Unfortunately the message from the compiler will not be firing in case when an instance of the class {**Dialog**} is assigned to a variable of the class {**Form**}. In such case the abandoned method actually will be called (and it just will be ignored since an abandoned method always overriding the ancestor method replacing it).

VII. 5. Restrictions to parameter values

A function can have modifiers **RESTRICT** (unconditional restriction on a parameter) and **IF/ THEN** modifiers (conditional restrictions on parameter values depending on values of other parameters).

- All such restrictions on parameters are written following all other function modifiers.
- Amount of restrict modifiers is not restricted.
- If a restriction is specified for a parameter, or the parameter is used in the IF part of a conditional restriction, then it is possible to pass only a constant as a value of such parameter. The only exclusion is a special conditional restriction **IF parameter IS CONSTANT, THEN...**
- Restrictions on parameters are controlled by a compiler at a compile stage. If conditions specified in restrictions are not satisfied, the compiler fires an error.
- A parameter can be restricted or participate in a condition of a restriction only if it has a simple data type (**BOOL**, **BYTE**, **INT**, **REAL**,

STR or a enumeration - **ENUM**) it has a **enumeration** data type, and is not an array.

Unconditional restriction has a form
RESTRICT parameter_name IN [list_of_values]

or (for REAL data type):
RESTRICT parameter_name IN [N1 TO N2]

or (without specifying certain value):
RESTRICT parameter IS CONSTANT

- When a range is used (for real values), a condition is checking $N1 \leq X$
 $\leq N2$, where:
 - X - a value passed to a function,
 - N1 and N2 - bounds of a range[N1 TO N2].
- The last variant does not specify any value checks but requires passing only a constant value as the parameter specified.

A conditional restriction has a form:
IF parameter1 IN [list or range],
THEN parameter2 IN [list or range]

or **part IN [...]** can be replaced with **IS CONSTANT** .

- Both parameters listed accept only constant values (except the case **IF parameter1 IS CONSTANT**).
- If a condition in the **IF** part is satisfied, then a condition in the **THEN** part must be satisfied to avoid compiler errors.
- If the **IF** part condition is not satisfied, then the **THEN** part is just ignored (but the parameter 2 anyway can accept only a constant).

when this can be useful?

E.g., to implement wrappers to the OpenGL library. It is possible to simplify the work just creating simple wrappers to correspondent OpenGL functions and pass just integer constants, but to add restrictions for such parameters to provide checking for only allowed sets of constants. This is much more simple then to create a set of classes or to use enumerated values (since the same values can be used in different functions for different purposes). Restrictions are providing working of classic OpenGL code (e.g. samples from NeHe etc.), almost without any changes (it is sufficient to remove symbols ';' at ends of lines).

VII. 6. Infinitive loop control

- a. On each iteration of any loop, an internal global counter is decreased. when it becomes equal 0, an internal procedure is calling which restores that counter to an initial value (usually 65536), and an additional function can be called in such case.
- b. For visual applications having graphics shell after sufficient time of long operation execution (usually more then 2 seconds) a progress of the operation can start displaying which is showing a percentage an a description of the long operation. And it is possible to cancel the operation by a button on the progress panel.
- c. In case when a programmer did not provide such long operation therefore pressing a button or a menu always is considered as a start of possible such long operation by default. So for such operation also it is possible to display a progress automatically allowing to cancel the operation if it becomes to work too long.

VII. 7. Code optimizations. INLINE insertions

- a. Functions can have the modifier **INLINE**. This is an instruction for the compiler to insert the function code in place a call of it.
- b. If the optimization is on, the compiler itself can replace calls of some functions to its immediate code for sufficiently short functions or for functions used only once. The compiler option **!autoinline** disables such automatic in-lining (but do not prevent in-lining functions marked with the modifier **INLINE**).
- c. In some cases functions can not be in-lined. E.g. it is not allowed to **INLINE** methods of another classes, or functions of another classes which use declarations not imported in the target class. Also, it is not allowed to in-line the code having statements **==>** , **BREAK** or **CONTINUE**.
- d. In some cases, in-lining of functions can lead to degradation of performance. E.g. if several functions having many local variables are in-lined into a single function, then its total amount of local variables can become too large. Since in AL-IV all the local variables must be initialized before running the function code, using such function very often (e.g. in a loop) can lead to slow down of the execution.

VII. 8. Code optimizations. UNROLL for FOR loops

- a. **FOR** loops based on a range defined by constant values can be totally unrolled with the modifier **UNROLL** without parameters (but amount of iterations should not exceed some limit defined by the compiler, current

limit is 1024 for C#, Delphi and Free Pascal). An example of code:
FOR i IN [0 TO 19], UNROLL:...

- b. Any **FOR** loop can be unrolled with an unroll factor specified in parenthesis. The unroll factor should be an integer constant not less than 2.
- c. while unrolling a **FOR** loop, total code size can increase but execution time mostly reduces. But in some cases the performance can reduce rather than increase (in case of C#, this may be a result of its code optimizer which may be can not handle long code sequences).
- d. The compiler option **/!unroll** disables any UNROLLS for **FOR** loops.

VII. 9. Syntax sugar

a. If there are several sequential assignments in code (or append to a string or to an array) with the same destination variable, then in following statements can be replaced with the **..** symbol (two dots). Note that in case when the destination is an array (in an append statement `array[] << item`), then it is not allowed to omit brackets. For example:

```
A[] << item1 << item2 << item3
..[] << item4 << item5
```

Lines of code with such continued assignments are not accounting while restricting amount of statements in a block, and amount of continued assignments/appends is not limited. And it is allowed to add other language constructions in between the stating statements and continued statements (and between continued statements, too), if such statements are not assignments/appends. E.g.:

```
A[] << item1
CASE condition? ..[] << item2;
```

b. If the target was an object, following assignments to its fields can be replaced with three dots (first two replacing the object, the 3rd - is the normal dot):

```
Button_ok=New_button("OK", "OK")
...Set_width(50)
```

c. It is allowed in case of an assignment to a field specified after a chain of dereferencing operations (separated with dots, e.g. `A.B.C(params).D[index].E`), to specify also which field will be a target for following continued assignment/ append statements. For this, instead of a dot in a correspondent position (after the default target field replaced later with double dots), three dots symbol is used. E.g.:

```
A...B.C[] << item1
...Calculate(param) // equals to: A.Calculate(param)
```

d. If in a single expression there are several sub-expressions of form `A.B[index].C(params).D`, and then entire chain is replicated (except the last field) then the second and following sub-expressions can be written in very short form: `.E`, `.F` etc. Note that leading operation signs like `'-'`, `'!'`, `'~'` are not parts of such expressions and should be written. E.g.:

```
P=pt1.Offset(-PBox.Bounds.Loc.X, -.Y)
```

e. In **DEBUG** statements it is allowed to omit the colon symbol just after the keyword **DEBUG** and also to omit **<<** symbols if the first statement is the console output with the first operand which is a string literal:

```
DEBUG "test" ;
```

f. In console output statements (**<<**) concatenating operands are automatically converted to strings if the compiler successfully can find a conversion function **Strling** (this is almost always correct for integer and real values);

g. A function, which result is defined by a single expression can be written in a very short form:

```
FUN name_of_function(...) ==> {type_of_result}:= expression.
```

(And the space is not required between `'.'` and `'='`, symbol `':='` is allowed too).

But, if the function is returning a new object of a class derived from the resulting value class, it is therefore required to use standard assignment:

```
RESULT={Class}(initializations)
```

VII. 10. Short help on "embedded" functions

The reason why such "embedded" functions exist is that it is not possible staying in the language rules frames to define strictly their parameters/results. E.g. the syntax & semantics do not allow to specify a parameter as "an array of any type". At the same time it is convenient to have a set of same named common functions to manipulate with arrays independent of its items types.

Note: with introducing functions "overloading" the main such reason can be partially eliminated. But therefore there is a problem to refer to "any structure" or "any enumeration" type: the AL-IV is not ready to supply such templates.

Since all the functions in the below table are static, these can be called either in the classic form `foo(x, y, z)` or in the prefix form `x.foo(y, z)`.

Function	Definition	Group
----------	------------	-------

Index(Variable) ==> INT	The index of the current item in a FOR loop, when the Variable is enumerating all or part of array	loops
Name({enum} Item) ==> STR	An enumeration item name. E.g., "RED"	enumerations
First({enum} Item) ==> {enum}	The first enumeration item	
Last({enum} Item) ==> {enum}	The last item	
Previous({enum} Item) ==> {enum}	The previous item	
Next({enum} Item) ==> {enum}	The next item	
Int({enum} Item) ==> INT	Convert the enumeration item to an integer (an index of the item based on 0)	strings
Int(STR S) ==> INT	Get an integer from a string	
Real(STR S) ==> REAL	Get a real value from a string	
Len(STR S) ==> INT	String length	
Find(STR S, STR W) ==> INT	Search an index of the first substring W in the string S (0... Len-1 or -1, when there is no such substring or W== "")	
Str({INT,REAL,BOOL}) ==> STR or S ({INT,REAL,BOOL}) ==> STR	Convert value (BOOL/BYTE/INT/REAL) to the string.	arrays
Count(A[]) ==> INT	Amount of items in the array	
Allocate(A[], INT Size)	Allocating additional items to provide size of the array to be at least given Size. If the size of larger already, it is not changed.	
Find(A[], W) ==> INT	Search an index of the first item W in the array A[] (except a structures array). The result is between 0... Count -1, or -1 if not found.	
Insert(A[], INT I, V)	Inserting the value V at the position I in the array A[]	
Clear(A[])	Clear the dynamic array A[]	
Delete(A[], INT I)	Deleting the item at index I	
Remove(A[], V)	Removing all the values V from the array	
Swap(A[], INT I, INT J)	Exchange items with indexes I and J	
Int(REAL X) ==> INT	Truncate the fractional part of the real value	real numbers
ShiftL(INT N, INT K) ==> INT	Logical shift the N onto the K bits left (if the K<0, shifts onto -K bits right). Bits shifted out are lost, zeroes shifting at the right side.	bitwise shift/rotate operations
ShiftR(INT N, INT K) ==> INT	Logical shift right.	
RotateL(INT N, INT M) ==> INT	Cyclic shift left by 1 bits of lower M bits.	
RotateR(INT N, INT M) ==> INT	Cyclic shift right	
Clone({structure} structure) ==> {structure}	Copying a structure	structures
Dismiss({structure} structure) ==> {structure}	Releasing a structure from its previous store, returning the structure	

All the embedded functions can be identified by capitalized names, e.g.: A [].COUNT

VII. 11. Generic functions

Generic (or overloaded) functions have parameters with variant types of parameters (and sometimes of the RESULT).

A type of a parameter (and may be, of a result) can be represented with a list of types enclosed into parenthesis. E.g.:

```

FUNCTION  Min|imum_of_two_values(
    { INT, REAL, STR}  A|_operand,
    { INT, REAL, STR}  B|_operand )  ==>  {INT, REAL, STR}
:
CASE  A < B ?  RESULT = A  ==>  ;
RESULT = B  .

```

Rules:

1. Only static functions can have variant type of parameters/result, at least, one parameter should be of such type.
2. It is not allowed to define only result to be of the variant type: at least one parameter also should be of such type, too.
3. All the parameters (and if the result is of variant type, then it is too) should have the same amount of variants in lists of types. The order of variants also is important, and all the types with the same index creates a version of the function with certain set of types of parameters/result.
4. The first parameter having variant types list should have all the types in the list different. To decide which version of the generic function is called the first parameter type is analyzed.

In the body of a generic function special variant of the **CASE ?** statement can be used which branches are defined not conditions or values but of types of the first input parameter with variant types:

```

FUNCTION S|string_from_rect_or_point(
  {STR,{rect},{point}} V|value_to_convert) ==> STR
:
CASE ?
{{ 1/STR }}: RESULT = V
{{ 2/{rect} }}: RESULT = S_rect(V)
{{ 3/{point} }}: RESULT = S_point(V) ; .

```

The list of variants determining a branch of such conditional statement contains of items N/type where N is an index of the type in the list of types of the parameter (starting from 1), and type - is the corresponding type of the first variant type parameter.

Like in case of enumerations as conditions, the **ELSE** branch can not be used and all the variants of types should be used.

In differ from usual code, while compiling each variant of such "generic" function, all its branches not corresponding to a currently selected signature are just ignored. So, all local variables declared in such branches become "invisible" for other code.

When such overloaded function is called the compiler searches it between available functions by name(s) and signatures of variable type parameters.

VII. 12. Indexing methods .[] - multi-dimensional arrays

It is possible to declare a single method which name is replaced with the dot '.' and parameters are listed in square brackets: so called .[]-method. To call such method for an object X, following like-array-access notation used:
X.[indexes]

Additionally if for such method a setter method is declared, then such method can be used at the left side of an assignment statement:
X.[indexes]=value

Such methods are used e.g. to implement multi-dimensional arrays like in class **{Matrix|_of_REAL}**:

```
----- 'access Items[] via .[i, j] method'
```

```

METHOD .[ INT I|ndex_from0, INT J|ndex_from0] ==> REAL
:
CASE J < 0 || J >= NColumns ? ==> ;
INT i_j INDEXING REAL = I * NColumns + J
RESULT = Items[i_j] .

```

```

METHOD set_items(
  INT I|ndex_of_row_from0, INT J|ndex_of_column_from0, REAL Value|_to_set),
SETTER FOR .[]
:
CASE J < 0 || J >= NColumns ? ==> ;
INT i_j INDEXING REAL = I * NColumns + J
Items[i_j] = Value .

```

VII. 13. Operators

In the article devoted to functions declaration syntax, operators was already specified (shortly). Now more detailed.

Operator is a static function purposed to be called in place of four arithmetic operations: +, -, *, /. Its declaration differs from the declaration of a usual function:

OPERATOR {type} operation {type2}==> {type3}:....

E.g.:

OPERATOR {complex} * REAL==> {complex}:....

Additionally to operators with two operands, also the operator can be defined for the operation '-' with a single parameter:

OPERATOR - {тип}==> {тип}:....;

At least one of types of input parameters should differ from basis types: **BOOL, BYTE, INT, REAL, STR**. This can be a structure, a class or a enumeration.

To use operators in a function, it should have the modifier **OPERATORS({Class_name})**, where the {Class_name} specifies the class where operators used were defined. If there are several such classes (which operators are used in the function), it is necessary to list all the classes separated by columns:

OPERATORS({Class1}, {Class2}, ...).

Parameter names in the declaration are omitted but names **A** and **B** are supposed for two input parameters. And for single parameter operator (for operation '-'), this parameter is referenced as **A**, too. And **RESULT** is used for returning value, as usual for all functions.

A class containing OPERATORS defined, should have the modifier **OPERATORS**.

An example of the operator code:

```

OPERATOR {Matrix} * REAL ==> {Matrix}, NEW
:
RESULT = New matrix(A.NRows, A.NColumns)
FOR i IN [0 TO RESULT.NRows-1] :
  FOR j IN [0 TO RESULT.NColumns-1] :
    RESULT.[i, j] = B * A.[i, j] ;
; .

```

VII. 14. Multi-threading

Multi-threading in the AL-IV is implemented by the class `{Thread}`, but it provides full data isolation of one thread from data of other threads. Any objects passed to a thread totally disappear from the address space of a thread which passes those data: all the references (both strong and weak) are remapped to NONE-objects of corresponding classes for all the time while these are on another side. And the main way to check if the object actually handled: periodically check if it is still equal to NONE. When not, the handling finished and the object been waiting returned back.

Threads form a hierarchy: if A runs B, then A becomes the master of B, and B depends on A. If the A is finished, B still can continue working but it can not more pass handled objects back: these just auto-destroy in such cases.

The thread object when it is started also disappear actually from the memory of the master thread as an object. But on the side of A, a phantom mirror object is created for it, and references from the original B object are redirected to that phantom object which can be used as a controller. The master thread can get from the controller some information about the status of the thread started (**Running**, **Stopping**, **Terminating**), to pass other objects to its input queue of objects (**Take**), give commands (**Stop**, **Wait**).

To implement some process executing in a separate thread, it is necessary to inherit your own class from `{Thread}` and override the method **execute**. If the thread gives objects for handling them, it should use methods **receive** and **yield** (the last to pass them back when these are ready). The **yield** method can also be used to pass newly created objects, not only received from the master thread.

Sometimes it is desired to get additional info on a progress of executing a task running in a thread running. E.g. to indicate a progress of executing the task in percents. In addition to a regular method of creating objects on the task side and sending them to a master thread (**yield**), it is possible to use the class `{Global_var}`. It is platform dependent, but it allows to create, modify or read integer named values from any thread running very simple, without locks.

VI. 15. Native (low-level) functions

Native functions can be only static (not methods). These should have the modifier **NATIVE**. It is not allowed to use structures as its parameters or data type of the **RESULT** value. Therefore, objects of classes are allowed.

There are two main variants of native functions: containing only native code, and having both normal AL-IV code, and ended with native code (allowing to prepare some parameters in local variables, or check some conditions and to prevent running native code if necessary).

In the first case either string constant or named string constant is following the colon ending the function header. If a string constant is written, it can be multi-line, with the prefix symbol '@' (adding the new line symbol #NL after each line continued).

```

FUNCTION writeln_number|_only_positive(INT N|umber_to_writeln), NATIVE: @
"if X_N < 0 then Exit;"
"writeln(IntToStr(X_N));" .

```

If a named constant is used, then the keyword **NATIVE** must be written first:

```

FUNCTION Some_native_fun, NATIVE: NATIVE NATIVE_string_constant.

```

In the second case, first the AL-IV code is written:

```

FUNCTION writeln_number|_only_local_positive(INT N|umber_to_writeln), NATIVE:
CASE N < 0?=> ;
STR s|tring_to_writeln=N.S
NATIVE @
"writeln(X_s);" .

```

In both implementations of the function `writeln_number` above, the first the parameter is checked for negative value. And both examples do the same result.

The content of a string constant specified as the native function body, is inserted into resulting code almost without changes. And this should be the code on a target programming language. But some peculiarities can be there depending on a target language.

E.g. for the Pascal, the first lines starting from the keyword '**var**' are treating as declarations of local variables and these are inserted before the keyword '**begin**' which is starting final code of a translated native function. (Actually, if the first line is starting from the '**var**', then all the first lines until the last one starting from '**var**' are inserted in the declaration section. Including all intermediate lines even not starting from '**var**'. This allows use e.g. conditional defines in the declaration section).

In all cases, for native functions returning a result, the local variable **RESULT** on the enter point to a native function code, is already initialized with a **NONE** -value (for numbers - with 0, for strings - with the empty string). So, returning from a native function without assigning any value in a code passed with a string, will return such **NONE** value as a function result.

To access its own parameters and local variables, native function in its body (in the string constant) should add the prefix 'X_' to its names. To work with objects and its fields and methods it is necessary to know exactly how to access them in a native code. Usually, fields get prefixes 'F_', methods and functions - 'M_'. Take into attention, that some programming languages do not matter about letters registry case (so, names 'a' and 'A' are equal in the Pascal).

It is possible to find a lot of examples of native code for C#, Pascal and Java in the functions library.

Classes containing native functions should be marked with the modifier **NATIVE**. It is desirable to avoid writing native functions except this really necessary (to optimize for speed critical parts of code, or to access system functions available only from native code).

VIII. Why one more programming language is necessary?

VIII. 1. Real multi-platform support

Yes, it is always possible to create applications which really are multi-platform. To do this, it is necessary to select a language (C++, C#, Java, Pascal) and to use some framework (or classes library). On that way a lot of adventures and disappointments.

Adventures - because authors of these languages and frameworks very often have its own mind about what is important and what is not necessary for you. They go ways which were not known for you before. Be ready that to solve a tiny problem you will spend weeks digging entire Internet, reading dozens of forum pages etc.

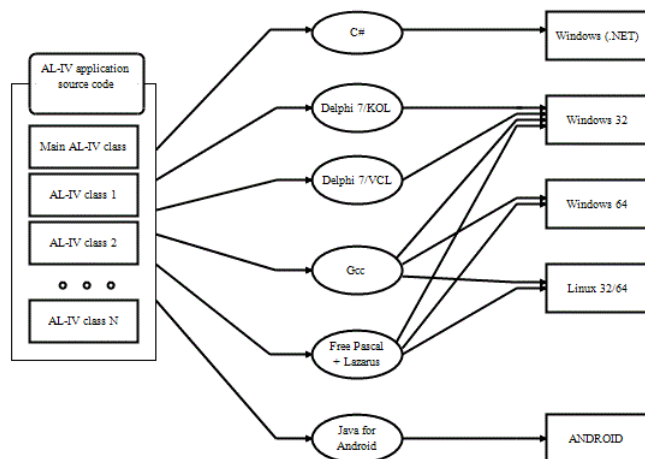
Disappointments - because sometimes you will have to redesign entire project removing desired features due to restrictions of a selected tool. Or even stop using the tool and to search another more comfortable (may be more expensive), and then to re-write all earlier written code.

And what fundamental distinction of the AL-IV in case of multi-platform support, if it is just an extension over one of existing programming languages (under C#, or Delphi, or Free Pascal, or Java)? The peculiarity of the AL-IV is that it is initially is not oriented on a certain platform.

You write your AL-IV code only once. And to launch the application on another platform it is sufficiently to correct configuration files of a project and to call the compiler.

It is just necessary to provide a support of a desired platform. Fortunately, there are not too many platforms. There is a probability that in a finite time we will have support of all the desired platforms.

For this moment (June, 2019) the platforms supported are:



What if one of the supported branches disappear suddenly (e.g. a desired target platform become not supporting or a developer tool become more expensive to pay for it)?

There is an answer on that question. It is in the simplicity of the AL-IV. The compiler for it (which is compiling to a some intermediate language) can be prepared for a short enough time. It is always possible to return to a code generation for C++/Java/Python/... or to make another generator. A task of creating wrapper native classes implementing base libraries functionality for a certain platform can be harder. But this also can be solved since base library also is simple enough and projected to simplify such task as much as possible.

VIII. 2. Safety

Actually all the modern programming languages were developed from ancient primitive assembling languages, allowing not safe operations, address arithmetic, not controlling array bounds etc. There are no existing high level programming languages in which it could not be possible to get an exception in result of an erroneous access to a zero address or in some hard cases to corrupt an arbitrary memory block.

You can use so called "managed" memory (in C++ this is an option, in C# the most of objects are in such memory), or to use none-objects (in the Objective-C). But part of code always will be written in a not safe old style, and you could not fix this (even in case if that code is yours). Even in modern C# and Java you are not free from necessity to provide checking if an object passed to a function is equal to null, or to provide exceptions handling in your code.

In case of the AL-IV, the situation is different totally. On the compiler level (and the Alfour language itself) an index value is checking for a bounds when an array is accessing (providing dummy item or ignoring write to the array operation if bounds are exceeding), accessing unassigned objects, it is guaranteed initialization of all the variables, it is preventing infinitive looping and recursion.

There are there embedded testing capabilities (with a control of a covering of code with tests). Programming with AL-IV really become relaxing and regular work, without extreme and adventures.

Exclusions are not possible in the AL-IV. When developing native methods/functions it is recommended to follow this paradigm providing in the final code in case of an exception handling it to provide working with it in a style of post-handling (when errors just are collected in a system array and can be accessed later by a final code, just to get know if there were errors while a function was called. In many cases it is sufficient to make a message about errors or just log it, and it is not necessary to crash).

VIII. 3. Why new syntax?

Really, it is always possible to stay in frames of some of existing syntax rule set, but to change semantics.

while developing new syntax the main purpose was to simplify working with source code using an arbitrary text editor (supporting UTF-8, this was an only requirement for such text editor).

It is actually from here a requirement to write keywords only in the UPPER CASE. In such case source code is much easier to read (and too hard to edit).

What about an absence of a starting bracket of a block statement, and a special symbol ending each simple statement. The absence of these makes text more clear, and simplifies editing it (for operations like re-factoring). And this does not make reading harder.

What about restrictions on amount of nesting blocks, on amount of statements between block comments. These restrictions do not effect a programmer freedom a lot. When we have too many nesting levels, this makes code harder to read and understand. When using several indentation levels, accessible line width becomes too small to fit sufficiently long statements, and we have more often to split lines of code. So, the requirement to move deeply nesting blocks of code to outer methods/functions is very correct and useful.

It is not hard at all to split too long sequences of statements separating these with block comments it is not too hard requirement at all.

The only question left is about a restriction on three only parameters for AL-IV methods/functions. Initially, when such requirement was introducing it was supposed that if the rule become too hard to follow it, this can be changed or removed at all.

But while developing sufficiently complex applications (such as the al-IV compiler, its IDE text editor and some other) this was found that the requirement is not impossible to satisfy, and very useful to make developing easier. For functions/ methods having not more than 3 parameters it is much simpler to remember its parameters rather than to use special IDE to pop-up help on method parameters each time it is used.

The restriction finally was removed and replaced with a requirement to specify parameter names in form of assignment (at least from the fourth). But this was done just to simplify code re-factoring (by moving pieces of code from internal blocks out to separate functions) and adaptation of code earlier written on other programming languages.

VIII. 4. Where is "while" loop statement?

Loops of form "while" / "repeat...until" are not safe, since these can lead to infinitive loops without any chance to leave such loop.

Loops of form FOR i IN [...] are safe for that since these earlier or later but finished (it is possible that it could be a lot of time but not the eternity).

while programming in the Alfour, in place where it could be suitable to use "while" in other languages, use instead the FOR statement setting as a range (for example) [0 TO N] where N is a maximum possible amount of iterations. And the first statement of such FOR statement should be a conditional BREAK on the anti-condition of the loop continuation:

```
CASE !continue_cond ? BREAK i;
```

VIII. 5. Why new memory management rules?

Since the time when a heap of dynamic data was invented, there were only one actually new useful change in memory management: automatic objects destroying on its usage counter value becoming zero.

Therefore it was found immediately that in many times closed chains of references can be created (references from objects to objects increasing its usage counters), which in result prevent objects from automatic freeing still its usage counters never achieve zero value. To fix the problem the so named garbage cleaner was introduced. Unfortunately, such procedure translates all the systems built on base of this technology for releasing actually unused objects into a category of slow and unpredictable. And this means that such programming environments can not be used to create real time systems or even to use in time-critical systems of mass service.

Yes it is always possible to refuse for critical subsystems from the managed memory and to develop in the old style directly controlling all the memory usage. But in such case totally disappear all the advantages of automatic memory management. And this is much more difficult still programmers usually do not do all the work from the scratch but use external libraries of classes and functions. If they can not use automatic releasing objects, then in many cases these should refuse from a lot of libraries. In result, capabilities of programming become restricting, and developing become more expensive and slow.

It is contrary another situation in case when you can use automatically releasing objects but it is not necessary to refuse from real time systems developing. Still a garbage cleaning is not necessary, objects are releasing exactly at the moment when this is necessary. Is not this a dream for a programmer.

Certainly in case of new way it is necessary to change a bit a strategy of objects creating. At least a programmer should think about a life time of an object at the point where it is creating. And we should reserve arrays to store references to dynamic child objects. But this is not too great loose in comparison to a possibility to refuse at all from the garbage collector (and from the garbage in the heap).

VIII. 6. Why embedded SQL statements?

Really, how a desire to make the programming language as simple as possible is combining with embedding of a direct support for SQL expressions?

An answer is following: such support allows to check a lot of SQL semantics on the stage of compiling (rather then at run-time).

Among them we can check: a correspondence of field names in queries to real fields (declared in TABLE definitions), a correctness of its usage (null fields, auto-increment fields, which could not be set in UPDATE statement). And certainly to check the SQL syntax, too. Such checks are done on a compiling stage reducing a possibility that a program which works with data bases, can be run with explicit errors in SQL code.

Such principal - a possibility of a static analysis of a correctness of operations on a compiling stage - makes developing much simpler. Unfortunately what is concerning SQL in modern programming such principal is not applying though working with data bases is one of key technologies in a modern practice. In the AL-IV this is fixed.

In differ to a generic programming the embedding some specialized features into the language (such as SQL, or complex numbers) actually do not increase a minimal necessary programming level.

While writing the code, you just do not use such features, and even could be not know about these existence. While reading an alien code, you should to learn the possibility if you found it in the code.

But learning some additional possibility is much easier than:

- to get know rules of writing generic functions/ methods/ classes;
- to learn a certain definition of a variable declaration and operations with its data type, programmed in a certain generic class including:
 - methods,
 - redefined operations,
 - data type conversions,
 - and often in combination of inheritance from other generic data types.

VIII. 7. Why there are no possibility to control which precision to use for (numeric) variables?

Because this makes your code simpler (and minimizes amount of errors lead from using in calculations of data of almost the same type but with different data precision).

In many of modern programming languages it is possible to define data types dual ways: for values for which the precision is not important, a base data type is used (e.g. Integer). But in cases when it is not enough or otherwise if it is necessary to economy memory, data types with special precision are used (Int64, SmallInt, ShortInt).

In result, a lot of data types and its combinations are present in a code. And we have to take into account situations when a precision of one variable or result of some calculations is not enough to fit in another variable. (Actually, nobody takes into account such things, instead if a possibility is to get incorrect results, then such erroneous results will be obtained, without any reasonable explanation, what was occur and how it can be fixed).

why such thin specifications are necessary if these are just used actually to economy memory. It is much better to reject from those forever, making life simpler for a programmer since he (she) has not to decide each time which variation of a type to use now.

By similar reason there are no unsigned data type in the AL-IV (like in Java, too). And it is not possible to specify bit depth for each variable separately. Only for all the integer or real numbers in a program, specifying compiler options (/int32, /\$REAL=EXTENDED/DOUBLE/SINGLE).

VIII. 8. Where is CHAR data type?

This type became too old and it is not actual for modern conditions. In case of using UTF-8 encoding, one character can occupy more than one byte, so to store it we have to use a string. What we finally have in AL-IV.

In the AL-IV each string character is a substring. So, extracting a symbol using operation like S[n], we actually call the function S.Substring(n, 1).

VIII. 9. Why there are no pointers to STRUCTURES?

Because records are introduced into the language only to allow data aggregation without dynamic data allocations. And these guarantee that a structure is always a single references so when the control flow leave its scope, it immediately releases the occupied memory.

Records are intermediate data types between simple data types and classes. From one side these contain separate fields (and passed to functions by reference actually). From another side these are assigned like simple variables (just copying its content) and can not be modified in functions where these are passed as parameters (i.e. these are always passed as constant values, in notation of C++/Java/Pascal - const).

In the AL-IV it is not possible to work with pointers to structures or its parts.

An absence of pointers and addressing arithmetic essentially increases code safety. Indexing of items in arrays is much more safe operation since the compiler have an information which object (array) code is accessing, and it can provide a control on its bounds exceeding, at least at run time.

VIII. 10. Code brittleness

a. Short names

It is possible to use short names of variables to shorten code. As well in earlier days when programmers used very short (one- two- letters) names for variables and this was not a crime. Only the difference is what the compiler demands you to provide also a long version of the name adding symbols following the '_' sign to fulfill it to at least 8 characters. So if the reader desires to understand what the variable is purposed to (s)he can go to its declaration and read its whole name. Or, in case of a special IDE, just click on the name to get its declaration shown as a hint.

b. Prefix functions calls

It is possible to minimize amount of parenthesis replacing classic call of functions to the prefix form. E.g.:

```
s.Replace_all(" ", ".").Trim.Remove_ending(".").TrimR.Find_last("_")
```

Compare to:

```
Find_lastTrimR(Remove_ending(Trim(Replace_all(s, " ", ".")), "."), "_")
```

The first version (prefix) is shorter a bit and much more understandable. Though both variants are correct in AL-IV.

c. Removing surplus checks

It is not necessary to check always if the object variable is null (in case of the AL-IV, is NONE). The AL-IV compiler add all necessary checks itself, and in case of the NONE value calling its methods or reading/writing fields does not lead to exceptions or other kinds of failures. In most cases while reading values the NONE value of a correspondent type is obtained (0 / "" / FALSE in case of numbers / strings / BOOL values), and nothing done in case of writing.

Also it is not necessary to think about zero division, or operations with NaN operand (the NaN will be returned). If you do not like the result obtained you can make efforts to find a reason and to fix it, but there are no other faults like application crash should occur.

So, in most cases it is enough to write for example:
`CASE sender.{Paint_table}.Count > 0 ?`
 rather then
`CASE sender.{Paint_table} != NONE`
`&& sender.{Paint_table}.Count > 0`
`?...`

And the first variant is much more readable, yeh?

You also do not need to take into account possible zero divisions or calculating results on base of NaN operands. In such case, just NaN will be returned. An application crash is not provided. If a value obtained is not satisfying you, try to find a reason of a failure and fix it. But this is exactly is not a reason to crash entire application at all.

d. LIKE statements

In many cases it is not necessary to do refactoring while code writing only just to reuse a piece of code already written several lines above. Yes, certainly, it is possible to re-design it as a separate function, to come up with a name for it, to guess how to pass parameters to it etc. and all these just to call it a couple times.

In case of the AL-IV it is just possible to bound the code selected to re-write with comments:

```
----- 'do not want to write again', REUSED
...
----- 'end'
```

and later to "call" it:

```
LIKE..... 'do not want to write again'
```

This is looking and working like a macros (which certainly is very bad if this would be a C language) but without any possibility to organize nesting **LIKE**'s or going out of the class code.

e. Formatting block statements

There are not block begin/end or {...} brackets. This allows to economy a couple of lines of code on each nesting block of code. Code becomes compact like the Python but there are no problems there in case when code is reformatted since only ending ';' symbols are defining actual nesting level.

When going onto new line of code there are no special characters contaminating the source. It is sufficient to follow some simple rules like write '(', '[', ',', at the end of a line interrupting or to continue on new line with some operations like '+', '-' etc.

f. Declaring local variables

Local variables are declaring at points where those are assigned first time, in most cases. And (this is important and really differs from the most of known programming languages) these are visible to the end of the function, independently of the nesting level of the block code where the variable was declared. If for some reasons code do not achieve the block where the variable was declared, anyway in all the code below the variable can be safely accessed with the NONE (or 0, or "", or FALSE etc.) value assigned to it in any case.

The only lack is in case when the variable is used to accumulate some value (counter, or collection) in a nested loop which can be run twice or more: certainly, in such case the variable should be initialized before entering the nesting loop. If you forget to do so this can brake your algorithm (but not to crash the application).

g. Automatic releasing of objects no more used

For the most of modern languages this is not the discover. But the AL-IV does not require the garbage collector. This allows to compile code for systems/ target languages/ platforms which do not have one (e.g. Delphi32/ Free Pascal) and still to have the automatic objects releasing on zeroing reference counters on them.

This also makes code shorter still it is not necessary to provide a code releasing objects manually.

Content

[Introduction](#)

- [Important semantic things of the language are:](#)
- [Important syntax things of the language are:](#)
- [The language has lack of:](#)

- [I. 1. Formatting statements](#)
 - [I. 1. a. Continue statement lines](#)
 - [I. 1. b. Block statements](#)
 - [I. 1. c. Comments](#)
 - [I. 1. d. Case sensitivity for keywords, types and other identifier](#)
 - [I. 1. e. Naming](#)
 - [I. 1. f. Modifiers](#)
- [I. 2. Assignment](#)
 - [I. 2. a. Simple assignment statement](#)
 - [I. 2. b. Assignment in combination with arithmetic, logic or bitwise operation](#)
 - [I. 2. c. Data sending operations](#)
 - [I. 2. d. Repeating assignment and sending data operators](#)
- [I. 3. Expressions](#)
 - [I. 3. a. Checking the presence of an item](#)
- [I. 4. Other simple statements](#)
- [I. 5. Conditional statement CASE](#)
- [I. 6. FOR loop statements](#)
 - [I. 6. a. About assigning a value to a loop variable:](#)
- [I. 7. PUSH statements block](#)
- [I. 8. DEBUG statements block](#)
- [I. 9. SILENT statements block](#)
- [I. 10. LIKE statement](#)
- [I. 11. REVERT statement](#)

- [II. 1. Simple data types](#)
 - [II. 1. a. Names of data types, data types classification](#)
 - [II. 1. a. Writing constants of base data types](#)
 - [II. 1. c. Enumerations](#)
- [II. 2. Variables and arrays](#)
 - [II. 2. a. Declaration of variables](#)
 - [II. 2. b. Field modifiers](#)
 - [II. 2. c. Named constants declaration](#)
 - [II. 2. d. Arrays declaration](#)
 - [II. 2. e. Array constructor](#)
 - [II. 2. f. Using arrays](#)
- [II. 3. Functions](#)
 - [II. 3. a. Function header](#)
 - [II. 3. b. Function body](#)
 - [II. 3. c. Calling functions](#)

- [III. 1. Classes](#)
 - [III. 1. a. Class declaration](#)
 - [III. 1. b. Class modifiers](#)
 - [III. 1. c. Import section](#)
 - [III. 1. d. Inheritance](#)
 - [III. 1. e. Objects. Strong and weak references](#)
 - [III. 1. f. Fields](#)
 - [III. 1. g. Methods](#)
- [III. 2. Working with objects of classes](#)
 - [III. 2. a. Creating an object instance](#)
- [III. 3. Other class level operators](#)
- [III. 4. Ending class. History of changes. DATA\[\] array.](#)

- [IV. 1. Declaration of a structure](#)
- [IV. 2. Working with structures](#)

- [IV. 3. About structures implementation in the final code](#)

—

- [V. 1. General rules](#)
- [V. 2. Syntax](#)

—

- [VI. 1. SQL queries encoding](#)
- [VI. 2. Database tables structure. TABLE operator](#)
- [VI. 3. SQL-like syntax](#)
- [VI. 4. Executing queries INSERT, UPDATE, DELETE](#)
- [VI. 5. Getting results of SELECT statements](#)
- [VI. 6. Transactions](#)
- [VI. 7. Syntax diagrams](#)

—

- [VII. 1. Localization of string resources](#)
- [VII. 2. Localization of the language keywords](#)
- [VII. 3. STORE - hidden parameters](#)
- [VII. 4. Abandoned and deprecated classes, structures, enumerations, field S, functions](#)
- [VII. 5. Restrictions to parameter values](#)
- [VII. 6. Infinitive loop control](#)
- [VII. 7. Code optimizations. INLINE insertions](#)
- [VII. 8. Code optimizations. UNROLL for FOR loops](#)
- [VII. 9. Syntax sugar](#)
- [VII. 10. Short help on "embedded" functions](#)
- [VII. 11. Generic functions](#)
- [VII. 12. Indexing methods .\[\] - multi-dimensional arrays](#)
- [VII. 13. Operators](#)
- [VII. 14. Multi-threading](#)
- [VI. 15. Native \(low-level\) functions](#)

—

- [VIII. 1. Real multi-platform support](#)
- [VIII. 2. Safety](#)
- [VIII. 3. Why new syntax?](#)
- [VIII. 4. Where is "while" loop statement?](#)
- [VIII. 5. Why new memory management rules?](#)
- [VIII. 6. Why embedded SQL statements?](#)
- [VIII. 7. Why there are no possibility to control which precision to use f or \(numeric\) variables?](#)
- [VIII. 8. Where is CHAR data type?](#)
- [VIII. 9. Why there are no pointers to STRUCTURES?](#)
- [VIII. 10. Code brittleness](#)
 - [a. Short names](#)
 - [b. Prefix functions calls](#)
 - [c. Removing surplus checks](#)
 - [d. LIKE statements](#)
 - [e. Formatting block statements](#)
 - [f. Declaring local variables](#)
 - [g. Automatic releasing of objects no more used](#)

[Content](#)

[Home](#)