Name: Extern Logic Interpreter
The principle of operation: sequential interpretation
Form: dynamically linked library (DLL)

The Purpose

Extern Logic Interpreter (hereinafter referred to as ELI) was developed to extend logical manipulation outside the application code itself. Its direct purpose is to abstract the program as much as possible from logical operations that could be easily modified without recompiling the main program. Ideally, an application that uses ELI can consist only of implemented mechanisms designed to interact with the environment (e.g., features for working with files, outputting information to the screen, etc.), along with special wrapper functions that allow ELI to access these mechanisms. The interpreter itself will be responsible for linking all of this to a specific algorithm.

The Operating Mechanism

ELI is a line interpreter that runs commands with a syntax similar to high-level programming languages such as C++. Line translation is mostly performed sequentially, except for the user-defined function (UDF), conditions, and loops, for which the bodies are parsed separately before the main script translation starts.

The unit of translation is a script. A script consists of lines separated by a ";" character. ELI can receive the script text directly from the main application or load it from an external file. There can be several translation units, their code is added to the translation using the #include directive or the Run() function.

Before translation, the interpreter renders the script: it selects string constants and code snippets in the text in curly brackets "{}". Code snippets are replaced in the script text with special identifiers, and their bodies get into a special stack. Such snippets are called deferred, and their translation starts only on a direct request.

Important: There can be only one script in a file.
Important: The script file should not include empty lines at the end.

The script text may contain comments. A commented line must begin with two forward slash characters "//" and end with a semicolon ";".

ELI provides the user with the following functionality:

Directives: direct commands to the interpreter.
Variables: internal variables of the interpreter that exist during the script's execution. Scope: the current context.
Simple and complex mathematical operations with numbers.
Loops.
Conditions.
Functions: pointers to wrapper functions from the main applicator. Enables the use of algorithms contained in the main application. The functions can pass the result of their work from the main application to the script. Scope: global, i.e. until the library is disabled.
Procedures: code snippets that are invoked (and translated) only when directly requested. As opposed to functions, they do not return values. Scope: global.

Objects: abstractions that represent a set of typed data and enable operating with them as a whole using properties and methods. Scope: global.

Parameter stack: a layer between the main application and the ELI, used for data exchange. It is a dynamically changing set of typed structures that characterize the concept of “parameter”.

Message stack output: errors and service messages that occur during the translation process are saved to a special string variable that can be passed to the main application using the appropriate interpreted function.

Variable stack output: the contents of the variable stack are saved in a formatted form to a string variable.

Function stack output: the contents of the wrapper function stack are saved to a string variable. Not only the functions described in the interpreter library are saved, but also those stated in the main application.

The output of the parameter stack.

The output of the object stack.

Translation log: an option that is set when the script is launched. Enables logging of strings to a file and saves error messages there. Logging slows down the translation process.

Interpreter log: an option set using a special interpreter function or the ELI::SetDebug() method. Enables logging of the ELI itself, and directs the output to a file or STDOUT. Logging significantly slows down the interpreter.

Implementation

ELI is written using C++, with std::vector<std::wstring> containers for storing script strings, as well as special classes that describe stacks of variables, parameters, functions, procedures, objects, and individual code snippets. Instances of these classes are included in the **ELI** class. The string encoding is ANSI. The internal language is essentially a wrapper over C++ code. Each string is interpreted into a set of C++ commands using a special algorithm and then executed. Using a special method of the ELI class, the application passes the script text (or the name of the file to be translated) to the interpreter, as well as an optional set of input parameters. The ELI interprets the strings and then returns the result of the process (type **const wchar_t***), if necessary. In case of an error, it returns the constant string “-err-“.

Integration

The main ELI code is built as a dynamically linked library that must be connected to the main application. In order to use the ELI functionality, it is necessary to import an object of the ELI class from the interpreter library. This is done using the abstract interface defined in the header file **eli_interface.h**, which is distributed with the interpreter.

The file contains the **ELI_INTERFACE**, which is the parent of the **ELI** class defined in the dll. The file also defines two factory functions that are used to create and destroy interfaces.

```
//-----------------------------------------------------------------------------------------------------------------
__declspec(dllexport) int _stdcall GetELIInterface(ELI__INTERFACE **eInterface);
typedef int (__stdcall *GETELIINTERFACE)(ELI__INTERFACE **eInterface);

__declspec(dllexport) int __stdcall FreeELIInterface(ELI__INTERFACE **eInterface);
typedef int (__stdcall *FREEELIINTERFACE)(ELI_INTERFACE **eInterface);
//-----------------------------------------------------------------------------------------------------------------
```

The functions return 1 if they are executed successfully, or 0 if they fail.

After adding the eli_interface.h file to the main application project, it is necessary to define pointers to the factory functions and initialize them.

```
//---------------------------------------------------------------------------------------------------------------------
GETELIINTERFACE GetELI;
FREEELIINTERFACE FreeELI;

ELI_INTERFACE *eIface;

GetELI = (GETELIINTERFACE) GetProcAddress(dllhandle, "GetELIInterface");
FreeELI = (FREEELIINTERFACE) GetProcAddress(dllhandle, "FreeELIInterface");
//---------------------------------------------------------------------------------------------------------------------
```

After that, initialize the interface object using the previously defined function.

```
//---------------------------------------------------------------------------------------------------------------------
GetELI(&eIface);
//---------------------------------------------------------------------------------------------------------------------
```

Now it is enough to use the **elface** to access the methods of the **ELI** class that are described inside the library. Thus, several ELI instances can be used in an application, which will act independently of each other.

List of methods of the ELI_INTERFACE class

const wchar_t * __stdcall GetVersion() – returns the current versionn of ELI.

const wchar_t * __stdcall ShowVarStack() – returns the contents of the variable stack in text form.

const wchar_t * __stdcall ShowObjStack() – returns the contents of the object stack in text form.

const wchar_t * __stdcall ShowClassStack() – returns the contents of the class stack in text form.

const wchar_t * __stdcall ShowProcStack() – returns the contents of the procedure stack in text form.

const wchar_t * __stdcall ShowFuncStack() – returns the contents of the function stack in text form.

const wchar_t * __stdcall ShowParamStack() – returns the contents of the parameter stack in text form.

const wchar_t * __stdcall ShowFragmentStack() – returns the contents of the stack of deferred code snippets.

const wchar_t * __stdcall ShowInfoMessages() – returns a list of interpreter messages.

const wchar_t * __stdcall RunScript(const wchar_t *imptext, const wchar_t *parameter, bool log) – starts the script translation. The log argument is a flag for using the log. The parameter argument can contain several values separated by the “|” character.

const wchar_t * __stdcall RunScriptFromFile(const wchar_t *filepath, const wchar_t *parameter, bool log) – starts thetranslation from a file. The log argument is a flag for using the log. The parameter argument can contain several values separated by the “|” character. The file path can be either absolute or conditional (.\<file_name>). In this case, the search is performed in the directory where the interpreter dll is located.

void __stdcall SetDebug(bool enable_dbg, bool in_file) – enables and disables the recording of the diagnostic log of the interpreter itself, and also indicates where exactly it should be written.

bool __stdcall DebugEnabled() – checks if the logging mode is used.

void __stdcall AddFunction(const wchar_t *name, const wchar_t *params, func_ptr fptr) – adds a function to the stack.

void __stdcall DeleteFunction(const wchar_t *name) – removes a function from the stack.

void __stdcall CallFunction(const wchar_t *name) – requests the function.

wchar_t * __stdcall GetFunctionResult(const wchar_t *name) – converts the result of a function from the ELI stack to a string and returns a pointer to it. In case of error, returns NULL.

void __stdcall SetFunctionResult(const wchar_t *name, const wchar_t* result) – writes the result of the function to the ELI stack.

void __stdcall SetParam(const wchar_t *name, const wchar_t *new_val) – adds a new parameter to the stack or changes the value of an existing parameter.

int __stdcall GetParamToInt(const wchar_t *name) – returns the value of the parameter as an integer.

float __stdcall GetParamToFloat(const wchar_t *name) – returns the parameter value as float.

const wchar_t * __stdcall GetParamToStr(const wchar_t *name) – returns the parameter value as const wchar_t*. In case of error, returns NULL.

const wchar_t * __stdcall GetCurrentFuncName() – returns the name of the current wrapper function that ELI is currently processing. It can be used to tell a wrapper function from an external library that was connected during the script's execution what internal name ELI gave it.

void __stdcall AddToLog(const wchar_t *msg) – adds an entry to the list of interpreter messages.

Defining a wrapper function

In order for the interpreter to use the code of the main application, this code must be wrapped in special functions. Wrapper functions should not be members of a class or defined in a class with a **static** specifier.

Important: the type of the result and arguments of the wrapper function are strictly regulated because the pointer is defined in the eli_interface.h file is used to work with these functions:

```
//-----------------------------------------------------------------------------------------------------------------
typedef void (__stdcall *func_ptr)(void*);
//-----------------------------------------------------------------------------------------------------------------
```

The void* argument is used to specify a pointer to the instance of the ELI object that requested function. This allows different instances of the interpreter to access the functions of the main application independently of each other.

```
//-----------------------------------------------------------------------------------------------------------------
void __stdcall foo(void *p)
{
  //convert the pointer p to the ELI_INTERFACE type
  ELI_INTERFACE *ELI = (ELI_INTERFACE*)p;

  int x = ELI->GetParamToInt("pInd"); //obtain a parameter from the dll stack

  /*some code of the main application*/

  ELI->SeFunctionResult("_foo", "0"); //setting the required result to the stack
  }
//-----------------------------------------------------------------------------------------------------------------
```

In this example, the pInd parameter is read from the stack and converter to an integer type. After that, the code of the main application is executed, and the result is passed to the interpreter library using the interface, which ELI will pass to the script.

Important: using the ELI_INTERFACE::SetFubnctionResult() method at the end of the function body is mandatory. Each wrapper function must return a result to the stack, so the interpreter will expect it. The type of the result is always a string, the interpreter will convert it to the requested type, if necessary.

After the definition, the function is needed to be added to the stack.

```
//-----------------------------------------------------------------------------------------------------------------
ELI->AddFunction("_foo", "num pInd", &foo);
//-----------------------------------------------------------------------------------------------------------------
```

“_foo” is the name of the function that ELI will use. The “_” character is mandatory; it is used by the interpreter to determine what is to be written in the script text, including the function name and arguments. The function name is case insensitive.

The “num pInd” string describes the list of function arguments and their type. If there are several arguments, they must be separated by commas. Spaces are placed only between the type of

argument and its name. A single argument is not separated by a comma. Argument names are case-insensitive and are not strictly regulated.

Now, when the interpreter detects an expression of the type "_foo(12)" in the body of the script, it will access the function stack, find the _foo() function there, check the number and type of arguments that the script passes to it, and invoke the foo() wrapper function from the main application, and pass it a pointer to its instance. Then it will get the result and return it to the line of the script in which the function was originally invoked.

In addition, any data can always be passed to the script anywhere in the application code using the parameter stack.

```
//-----------------------------------------------------------------------------------------------------------------
ELI->SetParam(«NewParamName», <значення>);
//-----------------------------------------------------------------------------------------------------------------
```

The ELI_INTERFACE::SetParam() method adds a new parameter to the stack only if it does not contain an element with the same name. If the name match, the interpreter will update the value of the existing parameter. This behavior is used to save resources. In the body of the script, a parameter can be retrieved from the stack using the corresponding built-in function. The input parameters of the script are also included in the parameter stack with predefined names of the INPRM<order number> type.

Syntax of the internal language

Data types

The ELI internal language operates with two types of data:
num – numeric type.
sym – character type.

The numeric type is used to operate with both integers and real numbers. By default, all numeric values are treated by the interpreter as real. The separator between the integer and fractional parts is the dot ".". Number precision: 3 decimal places. Conversion to integer form is performed using the built-in functions _round() and _int(), or automatically if required by the algorithm. If the function argument is of type int, for example, the variable of type num that was passed to the function will be automatically converted to an integer type.

The internal ELI language is a language with non-strict type matching. If the expression does not strictly specify that a numeric type should be used, the interpreter will treat the value as text. Thus, for example, a string variable can be assigned the value 21, and it will be converted to text. However, the reverse operation, e.g., setting a numeric variable to the value '22', will result in an error and stop the translation.

Variables

A variable is defined using a type entry:

```
//-----------------------------------------------------------------------------------------------------------------
$n = num;
$s = sym;
//-----------------------------------------------------------------------------------------------------------------
```

Where $n is the name of the variable, num is the type. This is followed by the value that initializes the variable. The absence of a value leads to initialization with the default value. A numeric variable $n, equal to 0.00, and a string valuable $s, which is an empty string, i.e. “ “, will be added to the stack after the string is translated. A variable can be initialized with a constant value, a scalar expression, another variable, an object property, and the result of an object function or method. A string variable can be initialized with the values of numeric variables, and they will be converted to text.

```
//-------------------------------------------------------------------------------------------------------------------
$n = num;
$s = sym 10$n;
//-------------------------------------------------------------------------------------------------------------------
```

The variable $s will be initialized with the string value “100.00”, which is the result of concatenating the string constant “10” and the value of the numeric variable $n, which is “0.000”. In addition, the $str = sym <arg1> + <arg2> construct perceives the expression on the right side as the string “2+2”, but the result of the $str = 2 + 2 construct is “22”, that is, concatenation.

The “$” character is a mandatory part of the name; it is a flag that tells ELI that the variable name follows. It is allowed to initialize one variable per line.

The data type in the variable definition can be omitted and the construction of the type $var = <value> can be used, in which case the data type will be selected for the variable based on the calculation of the right side of the expression.

To set the value of a variable, use the following construct:

```
//-------------------------------------------------------------------------------------------------------------------
$n = 12;
$s = some text;
//-------------------------------------------------------------------------------------------------------------------
```

The right side of the expression can contain: a constant, a scalar expression valid for this data type (product, multiplication, concatenation, etc.), an object property or method, and the result of a function. In this case, the variable $s will be assigned the value “sometext”, i.e. the interpreter will omit the space. However, by enclosing the value in single quotes, the interpreter will consider this code snippet a constant, so it will set the variable to “some text”.

String and numeric constants are monitored by the interpreter at the stage of parsing the script text and placed on the variable stack under a special name. Subsequently, when a string containing a variable like this is translated, ELI places the value from the stack instead. Constants are stored on the stack in one instance. If the same string constant occurs several times in the script code, the interpreter will substitute the value of one corresponding stack element in all cases.

Important: be careful when using the “$” character in string constants, as the ELI parser will recognize it as a variable marker.

String variables, except for concatenation (denoted by the “+” character), support direct sequential concatenation of values. In general, the interpreter considers the entire contents of the right side of the expression as text when operating with a string type. Any characters (except for special string characters), string constants, values of functions, methods, properties, and variables will be converted to the sym type and included in one string. The inclusion in the string occurs

sequentially in the order in which the values were arranged in the expression. Spaces (if they are not in string constants) will be ignored. Any characters of mathematical operations (except “+”) are considered normal characters.

Operations

ELI supports all basic mathematical operations for the numeric type (“+”, “-“, “*”, “/”), including the priority operation “()”. In addition, comparison operations are available for numbers, such as:

= set value;
== equal to;
> greater than;
< less than;
>= greater than or equal to;
<= less than or equal to;
!= not equal;

The increment (“++”) and decrement (“-- ”) operations can also be applied to numeric variables.

For a string type, only the concatenation “+” and setting the value “=” are allowed.

Conditions

```
//-------------------------------------------------------------------------------------------------------------
if (<condition>)
  {
    <action list>
  }
else if (<condition>)
  {
    <action list>
  }
else
  {
    <action list>
  }
//-------------------------------------------------------------------------------------------------------------
```

ELI supports branched conditions (if – else if – else), as well as nested conditions.

The syntax of the construct is as follows:

```
//-------------------------------------------------------------------------------------------------------------
$i = 5.15;

if ($i == 5)
  {
    $i = 0;
  }
```

```
else if ($i > 5)
  {
    $i = 0;
  }
else
  {
    $i = 1;
  }
//-------------------------------------------------------------------------------------------------------------
```

The parameters of the expression being checked can be a number, a numeric variable, or an object property, as well as functions and methods of objects whose results can be converted to a numeric type. When calculating, the fractional part of the values is also considered, so in the above example, $i will be greater than 5. The left and the right sides of an expression can be compared using any comparison operation. If there is no comparison operation symbol in the expression, the expression will be checked for validity. The expression is considered valid if the result of its calculation is greater than 0.

Select construct

```
//-------------------------------------------------------------------------------------------------------------
select (<parameter>)
{
  when <value1> then {<action list>}
  when <value2> then {<action list>}
  ...
}
//-------------------------------------------------------------------------------------------------------------
```

An alternative to branched if-else conditions is the select construct. Its essence lies in checking the passed input parameter for compliance with the values contained in the executive block. The check is performed using the special service word **when**, while the description of the action to be performed, in case the equality, is located after the service word **then**.

The parameter of the select construct can be: a variable, a function result, an object property, or the result of its method. Scalar expressions are not allowed. The values that are compared with the select parameter must be constants. The syntax allows mixing numeric and string constants within a construct, so the parameter passed to select can be compared both with the number 2 and with the string constant “two”. Obviously, a numeric variable cannot be compared to a string, so in this case, the interpreter will simply discard this when…then block and move on to the next one.

In the executive block of the select construct, in addition to the when…then blocks, any other actions can be to be executed after all the check blocks.

```
//-------------------------------------------------------------------------------------------------------------
$res = _foo(); //$res receives certain values;

select ($res)
{
  when 10 then {$res = $res + 2;}
  when 21 then {_return(2);}
```

```
  _return(0)
}
//-----------------------------------------------------------------------------------------------------------------
```

The scope of the visibility of the constructive block is global. This means that all operations with the variable $res from the previous case, if the first revision block is valid, influence the results of the following revision blocks.

Loops

The interpreter provides the user with three types of loops: **for**, **while,** and **count** (the last one is a simplified implementation of for). Loops can be nested. The parameters of loop conditions (except for count) can be numbers, numeric variables, or object properties, as well as functions or object methods that return a result that can be interpreted as a number. Variables defined in the loop will be initialized at each iteration.

The for loop

```
//-----------------------------------------------------------------------------------------------------------------
for (<initial value>, <condition>, <step>)
{<actions>}
//-----------------------------------------------------------------------------------------------------------------
```

Executes the code contained between the curly brackets until the comparison operation of the first and second arguments of the condition is valid. The third parameter is the step by which the value of the first condition parameter changes. The step can be either an integer or a fractional number, and the “+” or “-“ characters indicate the way the first parameter is changed. The second parameter consists of the comparison operation character and a numeric value that defines the limit of the condition. Valid comparison operations are “>”, “<”, “>=”, and “<=”. The first parameter can be a numeric constant, the result of a function or method, or an object property or variable that has a numeric type. If a variable or object property is used, it must be initialized before the loop begins. In addition, their value will change after each iteration, so using an object property may be the wrong option.

```
//-----------------------------------------------------------------------------------------------------------------
for ($var, <= 10, +1)
{<actions>}

for (0, <= 10, -1)
{<actions>}
//-----------------------------------------------------------------------------------------------------------------
```

The while loop

```
//-----------------------------------------------------------------------------------------------------------------
while(<condition>)
{<actions>}
//-----------------------------------------------------------------------------------------------------------------
```

Executes the code contained between the curly brackets until the comparison operation of the first and second parameters is valid. Contrary to the previous loop, while does not contain a step that changes the parameters of the condition. The condition is checked for validity before each iteration. If the condition is invalid before the first iteration, the loop body is not executed. Both parameters of the condition can be numeric constants and variables, results of functions or methods of objects, and properties of objects whose value can be interpreted as a number. Allowed comparison operations: “>”, “<”, “>=”, “<=”, “!=”, and “==”.

```
//-------------------------------------------------------------------------------------------------------------------
$i = 1;
while($i != 100)
{$i = _random(100);}
//-------------------------------------------------------------------------------------------------------------------
```

The count loop

```
//-------------------------------------------------------------------------------------------------------------------
count(<counter>)
{<actions>}
//-------------------------------------------------------------------------------------------------------------------
```

A simplified version of the for loop. A single parameter indicates the number of iterations of the loop body. The parameter can be a numerical constant, the result of a function or method, or a numerical property of the object or a variable that must be initialized with a certain value before the loop begins. The parameter must necessarily be a positive value.

Directives

Directives are service commands of the interpreter that describe certain actions that are mandatory for execution. The following directives are used in ELI:

#begin – mark the beginning of the script. Mandatory.

#end – marks the end of the script. Mandatory.

#include <file name> – includes the code contained in an external file in the script. All variables defined in the attached file are initialized on the currently active stack.

#exit – forced shutdown of the script.

#procedure <procedure name>(<arguments>){<procedure body>} – defines a custom procedure.

#drop procedure <procedure name> – deletes a custom procedure.

#make <variable name> {<specific code>} – assigns the variable the identifier of the deferred code snippet. Later, that identifier can be passed to another variable or object property.

#run <variable name> – executes the code snippet whose identifier is contained in the variable. All variables created during the execution of this snippet will be placed on the stack that is

active for the context where the #run directive was invoked. After translation, the snippet will be removed from the snippet stack.

#class <class name> {class body} – creates a new class of objects.

#modify class <class name> {class body} – modifies an existing class of objects.

#property <property name>=<value> – adds a new private property to the selected class. It is used only inside the code that describes the body of the class.

#public property <property name>=<value> – adds a new public property to the selected class. It is used only inside the code that describes the body of the class.

#method <method name>(<arguments>) {<method body>} – adds a new private method to the selected class. It is used only inside the code that describes the body of the class.

#public method <method name>(<arguments>) {<method body>} – adds a new public method to the selected class. It is used only inside the code that describes the body of the class.

#drop property <property name> – deletes a property from a class.

#drop method <method name> – deletes a method from a class.

#drop class <class name> – deletes a class of objects.

#return <value> – returns the result of executing a class method.

#protect {<specific code>} – executes a protected code snippet. The code contained in this snippet will be executed in any case and will not stop the translation, even due to a critical failure. However, all errors will be added to the translation log.

#trigger <condition> {<specific code>} – defines a trigger. A trigger is a deferred code snippet that will be executed if the condition defined in the trigger is valid. The trigger is defined once and checked for execution after the translation of each line of the script. Execution of the trigger does not affect the translation of the main script, even if errors were found during the translation of the trigger body or an exception was thrown.

#drop trigger <condition> – deletes the trigger.

#set {<directive>} – configures the ELI interpreter. Several service directives are used in the body of the directive.

#cnum | #!cnum – enables/disables parsing of numeric constants. Available only in the body of the #set directive. It is enabled by default.

#csym | #!csym – enables/disables parsing of character constants. Available only in the body of the #set directive. It is enabled by default.

#keepobjects | #!keepobjects – specifies whether to save/not save the contents of the class stack after the script is translated. Available only in the body of the #set directive. It is enabled by default.

#keepclasses | #!keepclasses – specifies whether to save/not save the contents of the object stack after the script is translated.

Functions

ELI can use any functions described in a user application if they are defined in the function stack. In addition, ELI includes several built-in functions:

_random(num pArea) – returns a pseudo-random value from the range pArea.

_round(num pNumber, num pPrecision) – returns the result of rounding the value of the argument pNumber. The pPrecision argument indicates the rounding accuracy (from 0 to 2 decimal places).

_int(num pNumber) – converts the value of the pNumber argument to an integer type.

_strlen(sym pStr) – returns the length of the string.

_streq(sym pStr1, sym pStr2) – compares strings, returns 1 if strings are equal, 0 if not. Case sensitive.

_istreq(sym pStr1, sym pStr2) – compares strings, returns 1 if strings are equal, 0 if not. Case insensitive.

_substr(sym pTargetStr, num pPos, num pCount) – returns a substring.

_return(sym pReturnVal) – ends the script and sets the result of execution.

_throw(sym pException) – ends the translation with the addition of a user-defined exception with arbitrary text to the log.

_free(sym pVarName) – deletes a variable from the stack, returns 1 if successful, 0 if not.

_LoadObjStack(sym pFilePath, num pClear) – loads the contents of the object stack from a file, returns 1 if successful, 0 if not. If pClear > 0, the object stack is cleared before loading. The file path can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located.

_SaveObjStack(sym pFilePath) – saves the contents of the object stack to a file, returns 1 if successful, 0 if not. The file path can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located.

_SaveObjects(sym pFilePath, sym pCategory) – saves all objects from the stack in the specified category to a file. The file path can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located.

_CompactObjStack() – compresses the stack, and deletes from it records that are marked as deleted, that is, those for which the Keep ==0.

_RemoveObjects(sym pCathegory) – deletes all objects with the specified category from the stack.

ClearObjStack() – clears the object stack.

_Run(sym pVarName) – executes the code contained in the variable named pVarName. Corresponds to a sequential request to the **#make** and **#run** directives. It is used when the code to be executed is generated during the script translation, i.e. if another function (_LoadFileToVar(), _ReadIn(), etc.) returns the text to be translated. If the code to be translated is specified directly in the body of the script, use the #make and #run directives. The code is executed within the current context.

_GetParamAsNum(sym pParam) – returns the value of the parameter pParam from the parameter stack converted to a number.

_GetParamAsStr(sym pParam) – returns the value of the parameter pParam from the parameter stack converted to a string.

_SetParam(sym pParam, sym pVal) – adds a parameter with the name pParam and the value pVal to the stack. If a parameter with this name is already on the stack, its value will be rewritten.

_LoadFileToVar(sym pFile, sym pTarget) – loads the contents of a text file into a variable named pTarget. The file path can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located.

_SaveVarToFile(sym pTarget, sym pFile) – saves the value of the variable to the specified file. The path to the file can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located. If the file path does not exist, a new file will be created.

_SaveFragmentToFile(sym pTarget, sym pFile) – saves the deferred fragment with the identifier contained in the pTarget variable to the specified file. The file path can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located. If the file path does not exist, a new file will be created.

Important: the _SaveFragmentToFile() function saves the snippet “as is” without expanding snippets whose identifiers are contained in the text.

_GetConfig(sym pFile, sym pLine) – loads the value of the pLine parameter from the file pFile. The path to the file can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located. The file must have the structure <parameter>=<value>.

_SaveState() – saves the current state of all ELI stacks to the state.log file in the ELI operating directory.

_SaveVarStack(num pLevel) – saves the state of the variable stack to the varstack.log file in the ELI working directory. If the pLevel argument is 0, the contents of the global stack are saved, if 1, the contents of the local stack (i.e., the one that is currently used) are saved.

_WriteOut(sym pStr) – directs the contents of the pStr argument to the standard output stream. In addition to the usual set of valid string values, the function can be assigned six reserved words:

#varstack – output of the variable stack. The currently active stack is used.
#funcstack – output of the function stack.
#prmstack – output of the parameter stack.
#objstack – output of the object stack.
#clstack – output of the class stack.
#procstack – output of the procedure stack.
#frgstack – output of the stack of deferred code snippets.
#endl – output of the end of line character (carriage return).

_ReadIn(sym pVar) – reads characters from the standard input stream and writes them to the variable pVar. The maximum number of characters for input: 4096.

_System(sym pCmd) – executes the Windows command contained in the pCmd argument. Analog of the C++ function system().

_LastError() – returns a description of the last recorded error.

_ConnectLib(sym pPath) – connects an external dll library to the interpreter. Returns the descriptor of the connected library. If unsuccessful, returns -1. The path to the file can be either absolute or conditional (“.\<filename>”). In this case, the search is performed in the directory where the interpreter dll is located.

_FreeLib(num pHandle) – releases the connected external library.

ImportFunc(num pHandle, sym pExtName, sym pInName, sym pArgList) – imports a wrapper function named pExtName from the connected library with the pHandle descriptor and adds it to the interpreter stack under the name pInName and the argument list pArgList. The arguments pInName and pArgList must be constant strings, since pInName uses the function marker character “”, and pArgList contains spaces that will be ignored without using a constant string.

_DebugIntoFile() – enables logging of interpreter actions, and directs output to a file in the ELI operating directory.

_DebugIntoScreen() – enables logging of interpreter actions, and directs output to the standard output stream.

_StopDebug() – disables logging of interpreter actions.

_sleep(num pMsec) – pauses the translation for pMsec milliseconds. Analog of the Sleep() function in C++.

Function arguments are represented by the ELI data types:

num - numeric.
sym - string.

The parameter type only indicates the mechanism for interpreting the value that will be passed to the function in the body of the script; the parameters themselves are stored on the stack without specifying a type. The wrapper functions themselves use methods of the ELI class, such as GetParamToInt(), to get a parameter with the desired type from the stack.

```
//-------------------------------------------------------------------------------------------------------------
$n = _random(10);
//-------------------------------------------------------------------------------------------------------------
```

Functions arguments can be: variables, constants, scalar expressions, properties, and methods of objects, as well as other functions.

Procedures

Procedures do not return values, but their implementation is entirely up to the user. In addition, unlike functions, whose algorithm is strictly limited to their description in the ELI library code (or in the code of a user application), procedures can be created, used, and deleted during script execution. A procedure that is defined within one context (e.g., in the body of another procedure) can be used in any other context until the ELI library is terminated. The procedure name is unique and must not match the name of a function on the stack. All variables that are defined in the body of the procedure are visible only in it and will be destroyed after the procedure completes its work. In the body of the procedure, variables that are not defined in the body cannot be used, because they are located on another variable stack.

Procedure definition:

```
//-------------------------------------------------------------------------------------------------------------
#procedure foo($arg1, $arg2)
{
<{ "code".
}
//-------------------------------------------------------------------------------------------------------------
```

Appeal:

```
//-------------------------------------------------------------------------------------------------------------
:foo(10, 5);
//-------------------------------------------------------------------------------------------------------------
```

Deletion from the stack:

```
//-------------------------------------------------------------------------------------------------------------
#drop foo;
//-------------------------------------------------------------------------------------------------------------
```

A procedure can have an arbitrary number of arguments. Arguments are initialized on the local variable stack of the procedure as variables of type sym. When the procedure is invoked, the

values that are passed to it as arguments will be written to the corresponding variables. The arguments are passed by value. As arguments to a procedure constant, variables, scalar expressions, object properties, and the results of functions or methods can be used.

User-defined procedures are stored on a special stack. When a procedure is defined, two elements are created with the category "procedure" and the name specified during the definition. One element contains a list of arguments, and the second element contains a reference identifier to the body of the procedure, which is saved in the stack of code snippets.

Objects

In the ELI language, an “object” is an abstraction that represents a certain set of records from a special stack. For the interpreter, each element of this stack is a set of typed fields described in the following structure:

```
//-----------------------------------------------------------------------------------------------------------------
struct RESOURCE
{
  UINT Index; //unique index
  std::wstring ObjectCategory; //category of the owner object
  std::wstring ObjectID; //ID of the owner object
  std::wstring PropertyID; //ID of the property
  std::wstring Value; //value
  std::wstring KeepInStack; //specifies whether to keep the resource on the stack after the compact
  std::wstring SaveInFile; //specifies whether to save the resource to a file on disk
};
//-----------------------------------------------------------------------------------------------------------------
```

The object stack is described as std::vector<RESOURCE> and is included in the special class RESOURCESTACK. This way, for the interpreter, an object is a set of elements of the corresponding stack for which the ObjectID field is the same. The PropertyID field defines the property of the object, and the Value field defines the value of this property. The key property of ELI when operating on objects is the ability to add and remove properties on the fly. Conversely, the set of methods is strictly limited. Each object has the following methods:

Create(sym cathegory, sym ctor_args) – creates a new object with the specified category. The ctor_args argument contains a list of arguments for the class constructor.

Exist() – checks if such an object is present on the stack, returns 1 if there is at least one element of the object stack for which the ObjectID corresponds to the name of the object that invoked this method.

Have(sym prop_name) – checks if the specified property exists in the object.

Add(sym prop_name, sym val) – adds a property to the object.
Although the val argument of the Add() method is defined as sym, it can be used to pass a numeric value. ELI will perform the data type conversion itself.

Remove(sym prop_name) – removes the property named prop_name.

Destroy() – destroys the object.

Keep(sym prop_name, sym istrue) – indicates whether the property should be stored on the stack. The istrue argument can take the value 1 or 0.

Save(sym prop_name, sym istrue) – indicates whether the property should be saved to a file. The istrue argument can take the value 1 or 0.

Execute(sym prop_name) – executes the deferred code snippet whose identifier is contained in the prop_name property. The identifier must first be created using the #make directive. When this method is invoked, a temporary procedure is created with one $this argument, which contains the name of the object.

Important: although the **Execute** method performs the same function as the **#run** directive and the **_Run()** function, they differ. The #run directive and the _Run() function work within the current context, while the Execute method creates a new context, so the code that is deployed when invoking the method is executed within this new context.

Show() – displays all the properties of the object.

GetName() – returns the name of the object.

ExportIn(sym pPropNames, sym pPropVals) – exports the properties of an object to two list objects with the specified names. Each property of the list object pPropNames will have a numeric index, and the value of this property will be the name of the property of the object that is invoked by the method. For the pPropVals object, the property values will contain the property values of the object that invoked the method. In addition, the last property of both lists will be Count, which contains the number of index items in the list.

E.g.:

```
//-------------------------------------------------------------------------------------------------------------
&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
&Hero.ExportIn(x, y);
//-------------------------------------------------------------------------------------------------------------
```

The result of executing &x.Show() will be:

```
//-------------------------------------------------------------------------------------------------------------
0 = Active
1 = Strength
Count = 2
//-------------------------------------------------------------------------------------------------------------
```

The result of executing &y.Show() will be:

```
//-------------------------------------------------------------------------------------------------------------
```

```
0 = 1
1 = 100
Count = 2
//-----------------------------------------------------------------------------------------------------------------
```

The general syntax for operating with objects is as follows:

```
//-----------------------------------------------------------------------------------------------------------------
&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
&Hero.Destroy();
//-----------------------------------------------------------------------------------------------------------------
```

The first string creates an object named &Hero and is categorized as an NPC, the second adds a numeric property Strength to the object with a value of 100, and the third destroys the object. The created object already has two properties by default: Owner with a value of “<none>” and ObjectName with a value that corresponds to the name of the object from the script string ("Hero" in this case).

Object properties can be accessed using the “.” service operator, just as it happens in many programming languages when operating with structure fields. Setting the value of a property is done using the “=” operator.

```
//-----------------------------------------------------------------------------------------------------------------
$x = &Hero.Strength;
&Hero.Strength = $x * 2;
//-----------------------------------------------------------------------------------------------------------------
```

The name of an object is unique, case-sensitive, and can include only letters, numbers, and the “_” character. The name of an object can be determined by the value of a variable (or even several variables) – this construction is called an alias.

```
//-----------------------------------------------------------------------------------------------------------------
$npcname = SuperHero;
&$npcname.Create(Player, '');

$part1 = Bohdan;
$part2 = Hmelnitsky;
&$part1$part2.Create(NPC, '');
//-----------------------------------------------------------------------------------------------------------------
```

Procedures can be passed the name of an object as a parameter. To do this, omit the special character & in the argument in the procedure invocation, and use the &<procedure argument name> construct in the body of the procedure as an object alias.

```
//-----------------------------------------------------------------------------------------------------------------
#procedure objfoo($arg1, $arg2)
{&$arg1.$arg2 = 5;}

&Hero.Create(NPC, '');
&Hero.Add(Strength, 100);
```

```
:objfoo(Hero, Strength);
//-------------------------------------------------------------------------------------------------------------
```

A property alias, like an object alias, can be composed of variable values. However, pay attention when using numeric variables in property or object aliases. The fact is that the values of variables explicitly defined as numbers are stored on the stack with a decimal point, so they will be inserted into the alias in this form.

It should also be remembered that when setting a property value, ELI will first try to convert the right side of the expression to a numeric type. If this is not possible, then the string type will be used.

Class objects

Above, the principles of operating with simple objects were described, the set of properties which is set by the user individually for each instance. In addition, the set of methods for simple objects is strictly limited to standard interpreter methods. These objects can be called structures. However, ELI also supports the ability to create typed objects according to a specific template. Such a template is called a class.

The set of properties of a class is specified by the user when defining it. In the future, when creating an object of this class, all the properties specified during the definition will be added to the object instance automatically. Besides, the class allows defining not only properties but also user-defined methods, which favorably distinguishes the class object from simple objects.

An example of a simple class that defines a list:

```
//-------------------------------------------------------------------------------------------------------------
#class List
{
  #property Next = 0;
  #public method AddItem($val){$x = &$this.Next; &$this.Add($x, $val); &$this.Next = ++1;}
  #public method Count(){#return &$this.Next;}
  #public method Change($ind, $val){&$this.$ind = $val;}
  #public method Get($ind){#return &$this.$ind;}
}
//-------------------------------------------------------------------------------------------------------------
```

Now, after using the standard Create() method with the List argument, an object will be created that includes the Next property with a value of 0.000 and four methods. After each invocation of the AddItem() method, another one with the specified value will be added to the object's properties. Certainly, all the built-in methods that are available to simple objects can be applied to class objects. The class name is case-sensitive, so when executing &obj_1.Create(List, “) and &obj_2.Create(list, ”), two different objects will be created. The class name is unique, so it is impossible to define two classes with the same name, but it is possible to redefine the class by first deleting it with a directive

```
//-------------------------------------------------------------------------------------------------------------
#drop class List;
//-------------------------------------------------------------------------------------------------------------
```

as well as add or remove properties and methods.

```
//-------------------------------------------------------------------------------------------------------------
#modify class List
{
  #drop method Change;
  #public method Clear()
  {
    $i = 0;
    $cnt = &$this.Count();

    for ($i, $i < $cnt, +1)
     {&$this.Remove($i);}

    &$this.Next = 0;
  }
}
//-------------------------------------------------------------------------------------------------------------
```

Important: all properties should be set to their default values when describing a class.

Important: changes made using the #modify class directive affect only class objects created after using the directive. Objects that were created earlier remain unchanged.

Important: if a class method participates in an expression to be calculated, it must return a specific result using the #return directive. Otherwise, the interpreter will not be able to form the expression correctly.

Class constructor

It is often necessary that the properties of a class object be defined immediately after its creation. To do this, when defining a class, it is possible to define a method whose name is the same as the class name, and the interpreter will consider it to be the constructor. In the body of the constructor, this method can be used to define and perform actions on any members of the class, but it cannot return a result. The constructor is defined in the same way as any class method. The list of constructor arguments is passed in the second argument of the built-in Create() method.

Important: the constructor must be defined as public, otherwise, the Create() method will not be able to invoke it.

```
//-------------------------------------------------------------------------------------------------------------
#class A
{
 #public method A($x){&$this.X = $x;}
 #property X = 0;
}

&obj.Create(A, 5);
//-------------------------------------------------------------------------------------------------------------
```

If there are several arguments, they must be specified using a string constant.

```
//-------------------------------------------------------------------------------------------------------------------
#class A
{
 #public method A($x, $y, $z){&$this.X = $x; &$this.Y = $y; &$this.Z = $z;}
 #property X = 0;
 #property Y = 0;
 #property Z = 0;
}

&obj.Create(A, '5, 6, 7');
//-------------------------------------------------------------------------------------------------------------------
```

Class destructor

ELI treats all objects as a set of data contained in a translation unit, and usually, it is, so to destroy an object, it is enough to delete all its fields from the object stack using the built-in Destroy() method. However, some of these objects may contain pointers or descriptors that refer to addresses in the heap. This is the case, e.g., if the object's method encapsulates a function from a third-party dll or host application that creates an object using the new operator. Of course, it would be a good idea for the developer of the library (host application) to create a mechanism for clearing the memory of such objects in the heap, but this mechanism still needs to be launched from the ELI context. To do this, a destructor was added to the class functionality.

The destructor is defined in the same way as any class method. It has no arguments. It is not inherited. The destructor will be automatically invoked when requested by the Destroy() method.

Important: the destructor must be defined as public, otherwise, the Destroy() method will not be able to call it.

```
//-------------------------------------------------------------------------------------------------------------------
#class A
{
 #property X = 5;
 #public method ~A(){&$this.X = 0;}
}

&obj.Destroy();
//-------------------------------------------------------------------------------------------------------------------
```

Private and public members of a class

Classes can have both private and public methods and properties. By default, the #property or #method directives define a class member as private. To define a public member, the #public property or #public method directive is needed.

In addition, ELI supports one of the principles of OOP: encapsulation. This means that private members of a class are available only in its methods.

Inheritance of classes

Classes can inherit properties and methods from other classes. Multiple inheritance is not supported. Only public members are inherited. The constructor is also inherited, but its name is inherited from the parent class and is not the default constructor for the child. Nevertheless, the inherited constructor can be invoked like any other method. The junior member class in the hierarchy inherits the constructors of all its members.

The descendant class can define its member with a name that matches the name of the inherited member, with preference given to the member that was defined rather than inherited.

All inherited members are defined as public by default.

```
//-------------------------------------------------------------------------------------------------------------
#class Point
{
 #public method Point($x, $y){&$this.pX = $x; &$this.pY = $y;}
 #public property pX = 0;
 #public property pY = 0;
 #public method SetX($pos){&$this.pX = $pos;}
 #public method SetY($pos){&$this.pY = $pos;}
}

#class 3DPoint : Point
{
 #property pZ = 0;
 #method SetZ($pos){&$this.pZ = $pos;}
}
//-------------------------------------------------------------------------------------------------------------
```

In this example, the 3DPoint class does not have its constructor, but it inherits the constructor from the Point class, just like all other members. Thus, to completely define a point in three coordinates, an object of the 3DPoint class can either call the inherited Point() constructor or use the inherited SetX() and SetY() methods and then call its own private SetZ() method.

Nesting of objects

Class objects can use other class objects as properties. To do this, when describing a class, it is necessary to define a property using the following construct:

```
//-------------------------------------------------------------------------------------------------------------
#class B
{
 #property Prop1 = #class A(0);
}
//-------------------------------------------------------------------------------------------------------------
```

Where A is the name of the class constructor, and the list of constructor arguments is given in brackets. If a class does not have a constructor (or its invocation is not required for some reason), use the following constructor:

```
//-------------------------------------------------------------------------------------------------------------
```

```
#class B
{
  #property Prop1 = #class A;
}
//-------------------------------------------------------------------------------------------------------------------
```

Each time an object of class B is created, the interpreter will create an object of class A with an automatically generated name and provide the Prop1 property with the name of this object. After that, the members of the nested class A object can be accessed through the Prop1 property. E.g.:

```
//-------------------------------------------------------------------------------------------------------------------
#class Point
{
  #public method Point($pos){&$this.Set($pos);}
  #property Pos = 0;
  #public method Set($pos){&$this.Pos = $pos;}
}

#class 2DPoint
{
  #public method 2DPoint($x, $y){&$this.Set($x, $y);}
  #public property X = #class Point(0);
  #public property Y = #class Point(0);
  #method Set($x, $y){&$this.X.Set($x); &$this.Y.Set($y);}
}

&point.Create(2DPoint, '5, 5');

&point.X.Pos = 10; //error - the Pos property is private;
&point.Y.Set(20); //correct;
//-------------------------------------------------------------------------------------------------------------------
```

An automatically generated object differs from a user-generated object in that its Owner service property contains the name of the owner object. When the owner object is deleted, all nested objects associated with it are also deleted.

There is another way to create a property object: using the built-in Add() method and passing it as the second argument to a #class Class(<constructor arguments>) construction. The property added by the Add() method will be public.

```
//-------------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class A;
}

&obj.Create(B, '');
&obj.Add(Prop2, #class A);
//-------------------------------------------------------------------------------------------------------------------
```

Important: due to certain properties of the parser, a string constant should be used to pass multiple arguments to the constructor in this case.

```
//-----------------------------------------------------------------------------------------------------------------
#class B
{
  #property Prop1 = #class 2DPoint(2, 2);
}

&obj.Create(B, '');
&obj.Add(Prop2, #class 2DPoint('5, 5'));
//-----------------------------------------------------------------------------------------------------------------
```

Extending the functionality with the help of external libraries

Naturally, ELI is not able to provide the entire range of possibilities that may come to mind. In this case, it provides the ability to extend the functionality by connecting external libraries and using their code in custom scripts. More specifically, this means using the wrapper functions described in external libraries in the same way as it is provided for a user application. It is enough to wrap the code in a wrapper function when designing a library and import it into ELI while the script is running, after which this function will be available for invocation until the interpreter is finished or until the external library is released. For the library to be compatible with ELI, it is necessary to include the **eli_interface.h** header file containing the virtual interface in the project.

```
//-----------------------------------------------------------------------------------------------------------------
$hinst = _connectlib(.\testext.dll);
//-----------------------------------------------------------------------------------------------------------------
```

This code will connect the testext.dll library from the directory where the ELI library is located to the interpreter and pass its descriptor to the $hinst variable. Suppose that the library has a function

```
//-----------------------------------------------------------------------------------------------------------------
__declspec(dllexport) void __stdcall Summ(void *p)
{
  ELI_INTERFACE *ep = (ELI_INTERFACE*)p;

  float num = ep->GetParamToFloat("pNum1") + ep->GetParamToFloat("pNum2");
  wchar_t res[10];
  swprintf(res, "%.2f", num);

  ep->SetFunctionResult(ep->GetCurrentFuncName(), res);
}
//-----------------------------------------------------------------------------------------------------------------
```

that sums two numbers with a decimal point and returns the result. To give the interpreter the ability to invoke this function, include the following code in the script:

```
//-----------------------------------------------------------------------------------------------------------------
_ImportFunc($hinst, Summ, '_summ', 'num pNum1,num pNum2');
```

```
//-------------------------------------------------------------------------------------------------------------
```

In this case, **$hinst** is the descriptor of the external library, **Summ** is the name of the function in the external library, **_summ** is the name that the interpreter will use, and the string **'num pNum1,num pNum2'** defines the list of function parameters. The names of the parameters must, naturally, match those used in the body of the function to be imported. If the import fails, the _ImportFunc() function returns a value less than 1, otherwise, it returns 1.

Important: the GetCurrentFuncName() method in the body of a function from an external library is needed to find out from the interpreter which function name it has assigned to the function from the external library.

Important: the internal name of the function that uses ELI can be anything, but it must be ensured that it does not match the name of a function that already exists on the stack.

Important: the name of the function exported from the DLL may change depending on how the compiler decorates names when using the __stdcall specifier. E.g., the Embarcadero C++ Builder compiler exports the name of the void __stdcall Foo(void*) function as "Foo", and MinGW exports it as "Foo@4". If the interpreter's _ImportFunc() function reports a failure, correct the name of the exported function at the place where _ImportFunc() is invoked.

Now the imported function can be used in the script.

```
//-------------------------------------------------------------------------------------------------------------
$res = _summ(2.2, 3.4);
//-------------------------------------------------------------------------------------------------------------
```

After that, the library can be released if it is no longer needed.

```
//-------------------------------------------------------------------------------------------------------------
_FreeLib($hinst);
//-------------------------------------------------------------------------------------------------------------
```

Important: all functions that have been imported into the interpreter will be deleted from the stack after the invocation of _FreeLib().

All external libraries connected to the interpreter will be automatically released when the ELI library is disabled.

Decorations

In ELI, a decoration is a label that decorates a certain set of language constructs, from variables to entire expressions. In a certain sense, the scenery operates on the principle of the #define preprocessor directive in C++. When translating a string, the decorations contained in it will be processed before all subsequent actions and the decorated constructs will be substituted into the string instead.

```
//-------------------------------------------------------------------------------------------------------------
$x = 10;
?ref = $x;
//-------------------------------------------------------------------------------------------------------------
```

The ?ref decorator replaces a construct consisting of the variable $x. In the future, in all translated strings, the variable $x will be substituted for ?ref. As a result, the following strings will be translated:

```
//-------------------------------------------------------------------------------------------------------------------
$x = 10;
?ref = $x;
$y = ?ref + 5;
//-------------------------------------------------------------------------------------------------------------------
```

The variable $y will be equal to 15. The interpreter will process the right side of the expression, substitute a hidden construct for the decoration, and then calculate the value of the expression.

Decorations are useful for saving template calculation structures, e.g., if a variable value is calculated using a complex formula.

```
//-------------------------------------------------------------------------------------------------------------------
?hip = $a * $a + $b * $b;

$a = 3;
$b = 4;
$sqc = ?hip;
$sqx = ?hip;
//-------------------------------------------------------------------------------------------------------------------
```

This form of writing allows to significantly shorten the code and make it more readable. However, it should be remembered that the interpreter does not monitor the correctness of the decorated structure. If the variables $a or $b in the above example were not previously defined, the interpreter will generate a translation error.

Since decorations contain names rather than values of variables, they are convenient to use when it is necessary to operate on the variable itself, not its value, e.g., in the case of the _free() function. In addition, decorations allow passing a set of arguments, so this construction is legitimate:

```
//-------------------------------------------------------------------------------------------------------------------
$a = здоровенькі;
$b = ' ';
$c = були!;

?ref = $a, $b, $c;

#procedure Hello($x, $y, $z)
{
  _writeout($x $y $z); _writeout(#endl);
}

:Hello(?ref);
//-------------------------------------------------------------------------------------------------------------------
```

As a result, the screen will display “здоровенькі були!”.

Decorations can also be used to create aliases for objects or properties, but such decorations cannot be passed to a procedure. Even the names of methods and functions can be decorated, but such decorations can only be used on the right side of the expression. The interpreter allows using the name of the decoration on the left side of the expression only in one case: when describing the decoration.

This construction is acceptable:

```
//-------------------------------------------------------------------------------------------------------------------
?f = _random;

$a = ?f(10);

#class A
{
  #method Foo(){#return 10;}
}

&cl.Create(A, '');

?f = &cl.Foo();

$b = ?f;
//-------------------------------------------------------------------------------------------------------------------
```

And this one is not:

```
//-------------------------------------------------------------------------------------------------------------------
?f = _writeout;

?f(10);
//-------------------------------------------------------------------------------------------------------------------
```

In addition, it should be remembered that decorations can only hide complete constructs and do not support decorating code snippets. E.g., a class definition or variable definition cannot be decorated. Decorating procedures also makes no sense, because procedures are not used on the right side of an expression.