Интерпретатор языка программирования

Реализация с помощью компилятора компиляторов Coco/R

Уваров Александр Федорович
2011 г. Москва

Оглавление

Создание интерпретатора Alfa *3*

Основные требования 3

Конструкции языка 3

Строительный участок 6

Файл Alfa.atg без реализации 10

Перечень модулей интерпретатора 16

Схема создания проекта интерпретатора 17

Переменные интерпретатора 17

Представление переменных языка Alfa в интерпретаторе 19

Запуск интерпретатора 32

Обработка ошибок в интерпретаторе 32

Программа Alfa 35

Объявление констант, переменных и серверов 37

Инициализация таблицы процедур и функций 50

Предварительный просмотр исходного текста в интерпретаторе 51

Выражения и операции 55

Операторы 67

Оператор присваивания 68

Выполнение стандартных процедур и функций 82

Выполнение процедуры 85

Выполнение функции 90

Оператор IF 95

Оператор SWITCH 97

Оператор WHILE 100

Оператор FOR 102

Оператор LOOP 106

Оператор READLN 107

Оператор WRITE 111

Оператор WRITELN 113

Оператор EXIT 115

Литература 115

Создание интерпретатора Alfa

Основные требования

Попробуем сформулировать основные требования к создаваемому языку программирования:

- Должен быть простым и ясным, достаточно мощным для своего назначения;
- Язык должен быть процедурным (поддержка пользовательских процедур и функций с передачей параметров по ссылке и по значению), типизированным;
- Программа на языке должна состоять из единственного модуля;
- Предусмотреть константы и переменные основных типов - вещественного, целого, строкового, булевского. Предусмотреть переменные - файлового, серверного, процедурного типа, COM типа, одномерные массивы перечисленных типов;
- Предусмотреть переменные серверного типа - специальный объект сервер, с помощью методов которого, можно удаленно выполнять различные действия на сервере - запускать и останавливать службы и процессы, определять их текущее состояние, перезагружать сервер, изменять *IP* адрес, определять доступность любого компьютера или сервера в сети, работать с файлами, папками, общими ресурсами;
- Предусмотреть набор операторов, который должен включать присваивание, вызов процедур и функций, условия и циклы;
- Предусмотреть стандартные процедуры и функции для работы с числами, строками, массивами, папками и файлами, а также функции для отправки сообщений электронной почты по протоколу *SMTP*, функции передачи и приёма файлов по протоколу *FTP*, функции для работы с *INI* файлами;
- Предусмотреть ввод и вывод на консоль, чтение и запись информации в текстовый файл;
- Предусмотреть выполнение кода написанного на других языках программирования *JScript*, *VBScript*;
- В записи программы разрешить комментарии, которые могут быть вложенными;
- Большие и малые буквы должны различаться;
- Интерпретатор языка программирования будет “чистым” интерпретатором, без преобразования исходного текста программы и исполнять строки исходного текста «как есть».

Конструкции языка

Возьмем за основу создаваемого языка – старый, добрый Pascal, но для нас принципиально важно, с существенными изменениями. Язык Pascal – достаточно простой и выразительный язык, но опять же «но».

Рассмотрим основные конструкции языков программирования C++ и Pascal, Delphi.

C++	***Pascal, Delphi***
Циклы	

while (<условие>)
  <оператор>;
 	
while (<условие>) {
  <операторы>;
}
	while <условие>
	  <оператор>;
	
	while <условие>
	  begin
	  <операторы>;
	  end;

do
  <оператор>;
while (<условие>);
 	
do {
  <операторы>;
} while (<условие>);
	repeat
	  <оператор> | <операторы>;
	until <условие>;

for (<иниц>; <условие>; <изм_усл>)
  <оператор>;
 	
for (<иниц>; <условие>; <изм_усл>) {
  <операторы>;
}
	for <сч> := <нач_зн> to <кон_зн> do
	  <оператор>;
	
	for <сч> := <нач_зн> to <кон_зн> do
	  begin
	  <операторы>;
	  end;

Условные операторы	

if (<выражение>)
  <оператор>;
else
  <оператор>;
 	
if (<выражение>) {
  <операторы>;
}
else {
  <операторы>;
}
	if <выражение>
	  <оператор>
	else
	  <оператор>;
	
	if (<выражение>)
	  begin
	  <операторы>;
	  end
	else
	  begin
	  <операторы>;
	  end;

switch (<выражение>) {
  case <конст_1>:
  <оператор>;|<операторы>;
  case <конст_2>:
  <оператор>;|<операторы>;
  case <конст_n>:
  <оператор>;|<операторы>;
  default:
  <операторы_по_умолчанию>;
}
	case <выражение> of
	  <конст_1>:
	  <оператор>;
	  <конст_2>:
	  <оператор>;
	  <конст_n>:
	  <оператор>;
	  else
	  <оператор>;
	end;
	
	case <выражение> of
	  <конст_1>:
	  begin

	<операторы>;
	end;
	<конст_2>:
	begin
	<операторы>;
	end;
	<конст_n>:
	begin
	<операторы>;
	end;
	else
	begin
	<операторы>;
	end;
	end;

В операторах циклах и в операторах условий в C++, Pascal и Delphi, если нужно поместить несколько операторов, то эти операторы помещаются в блок кода. В C++ это { … }, в Pascal и Delphi это begin … end. Для нашего языка программирования неприемлемо. Почему? Потому такие синтаксические конструкции сильно усложняют анализ исходного кода программы. Мы должны выбрать синтаксис основных конструкций гораздо проще, но без потери выразительности и ясности. За основу возьмем основные конструкции языка программирования Pascal, чуть-чуть из C++ модифицируем их и посмотрим, что получится.

Определение программы:

```
program <Имя>;
const
  <Определение_констант>;
var
  <Определение_переменных>;
<Определение_процедур_и_функций>;
begin
  <Операторы>;
end.
```

Определение процедуры:

```
procedure <Имя> (<Список_формальных_параметров>);
const
  <Определение_констант>;
var
  <Определение_переменных>;
begin
  <Операторы>;
end;
```

Определение функции:

```
function <Имя> (<Список_формальных_параметров>): <Тип_результата>;
const
  <Определение_констант>;
var
  <Определение_переменных>;
```

```
begin
  <Операторы>;
end;
```

Определение условного оператора if:

```
if <Условие> then
  <Операторы>;
endif;

if <Условие> then
  <Операторы>;
else
  <Операторы>;
endif;
```

Определение оператора выбора switch:

```
switch <Выражение>
  case <Значение>
    <Операторы>;
  endcase;
  …
  case <Значение>
    <Операторы>;
  endcase;
  default:
    <Операторы>;
endswitch;
```

Определение оператора цикла for:

```
for <Переменная_цикла> := <Начальное_значение> to <Конечное_значение>
  <Операторы>;
endfor;
```

Определение оператора цикла while:

```
while <Условие>
  <Операторы>;
endwhile;
```

Определение оператора цикла loop:

```
loop
  <Операторы>;
endloop;
```

Строительный участок

Для создания интерпретатора *Alfa*, нам понадобятся следующие инструменты: *Microsoft Visual Studio 2008 с SP1*, компилятор компиляторов *Coco/R*, а так же для удобства работы с файлом *ATG Coco/R plug-in for Visual Studio*.

Заходим на основную страницу *Coco/R* университета г. Линца по ссылке: *http://ssw.jku.at/coco/#Docu*

Рис. 1 Основная страница Coco/R

Создаем каталог *Coco*. На основной странице переходим по ссылке *Documentation*, скачиваем в каталог *Coco/Doc* следующие файлы:

License.txt
UserManual.pdf
DataStructures.pdf
TestSuite.zip
JMLC`03 paper
Tutorial

На основной странице переходим по ссылке Coco/R for C#, скачиваем в каталог *Coco* следующие файлы:

Coco.exe
Scanner.frame
Parser.frame

И наконец, с основной страницы по ссылке *Tools -> Tools for Coco/R -> Coco/R plug-in for Visual Studio -> Further information and Download -> Coco/R-Plugin* скачиваем файл:

CocoPlugin.zip

Разархивируем файл например в каталог *Plugin*. Запускаем *Microsoft Visual Studio 2008*. Устанавливаем скачанный *Plugin*. Добавляем *Coco/R* в *Microsoft Visual Studio 2008* в качестве внешнего инструмента, для этого в меню выбираем пункт *Сервис –> Внешние инструменты (См. рис. 2)*.

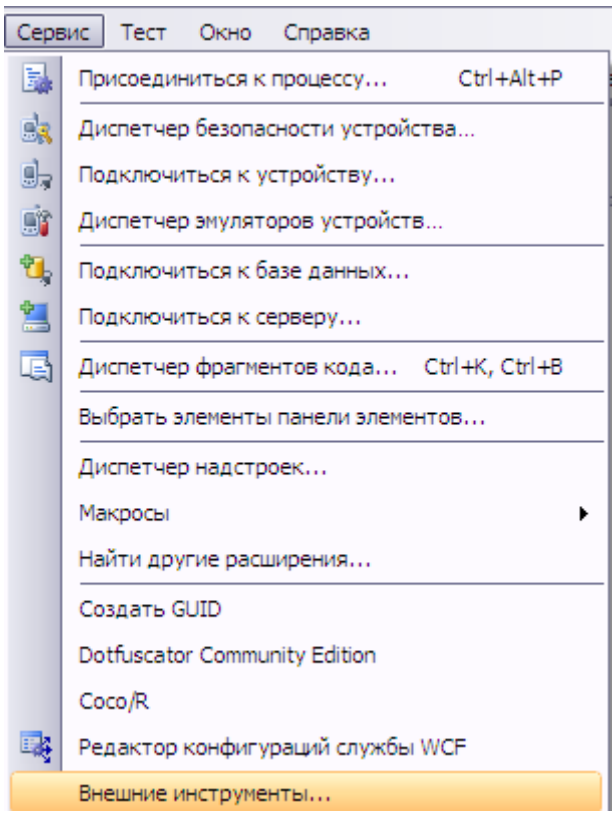

Рис. 2 Пункт меню Сервис -> Внешние инструменты

Производим соответствующие настройки. В окне «Внешние инструменты» заполняем поля по порядку: имя приложения, полный путь, аргументы командной строки, исходный каталог (См. рис. 3).

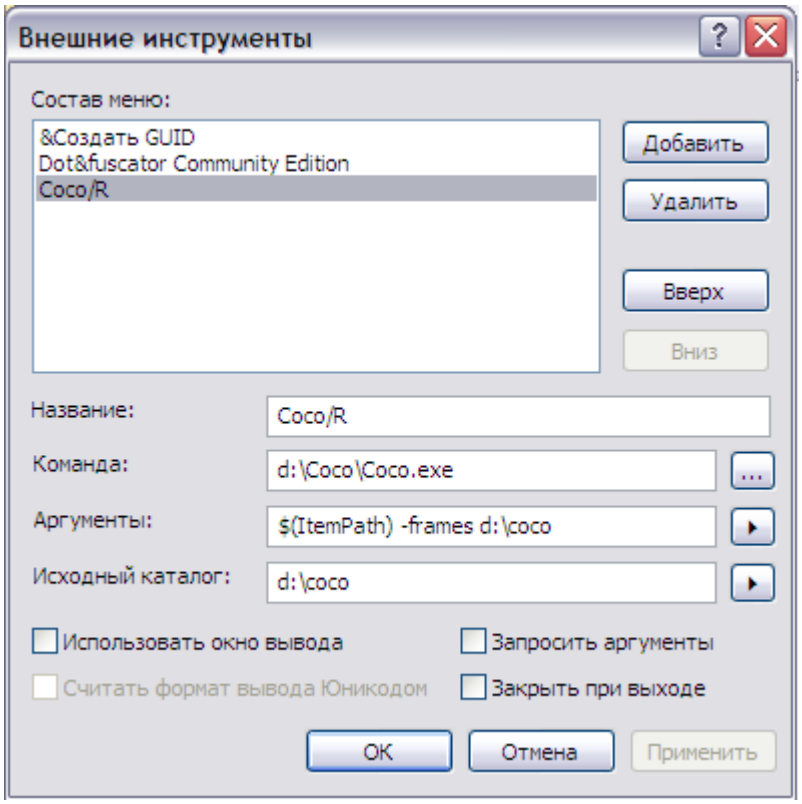

Рис. 3 Окно "Внешние инструменты"

Последнее что нам осталось, это произвести некоторые изменения в файлах *Scanner.frame* и *Parser.frame*. Проводим изменения в файле *Scanner.frame*. Ниже приведен фрагмент файла, измененные строки исходного кода выделены курсивом.

```
-->declarations

      public Buffer buffer; // scanner buffer

      public Token t;          // current token
      public int ch;           // current input character
      public int pos;          // byte position of current character
      public int col;          // column number of current character
      public int line;         // line number of current character
      public int oldEols;      // EOLs that appeared in a comment;

      static readonly Hashtable start; // maps first token character to start
                                       // state

      public Token tokens;     // list of tokens already peeked (first token
                               // is a dummy)
      public Token pt;         // current peek token
```

Зачем нужны эти изменения. Первое изменение нам нужно для того чтобы данные класса *Buffer* сканера были доступны. В процессе интерпретации исходного текста нам нужно будет запоминать исходную позицию начала процедур и функций, начало цикла (и т.д.) и возвращаться сканером на исходные запомненные позиции в процессе интерпретации.

```
//*************************************************************
//* Function: SavePosProg
//* Notes: Сохранение текущей позиции сканера в исходном тексте
//*        программы
//*

private ObjLabel SavePosProg () {
  return (new ObjLabel(32, la.pos, la.col, la.line, scanner.oldEols));
}

//*** End of SavePosProg ***********

//*************************************************************
//* Function: RestorPosProg
//* Notes: Восстановление текущей позиции сканера в исходном
//*        тексте программы

private void RestorPosProg (ObjLabel Label) {
  scanner.buffer.Pos = Label.pos;
  scanner.ch = Label.ch;
  scanner.pos = Label.pos;
  scanner.col = Label.col;
  scanner.line = Label.line;
  scanner.oldEols = Label.oldEols;
  scanner.pt = scanner.tokens = new Token();
  t = la = new Token();
}

//*** End of RestorPosProg ***********
```

Проводим изменения в файле *Parser.frame*. Ниже приведен фрагмент файла, изменения выделены курсивом.

```
-->declarations

      public Parser(Scanner scanner) {
        this.scanner = scanner;
        errors = new Errors();
      }

      void SynErr (int n) {
//        if (errDist >= minErrDist) errors.SynErr(la.line, la.col, n);
//          errDist = 0;
        errors.SynErr(la.line, la.col, n);
        throw new Exception("");
      }

      public void SemErr (string msg) {
//        if (errDist >= minErrDist) errors.SemErr(t.line, t.col, msg);
//          errDist = 0;
        errors.SemErr(t.line, t.col, msg);
        throw new Exception("");
      }
```

Второе изменение нам нужно для того, чтобы изменить стратегию обработки ошибок. Инструмент *Coco/R* в первую очередь предназначен для создания компиляторов. В основном современные компиляторы однопроходные и поэтому очень важно на этапе компиляции обнаружить как можно больше ошибок (синтаксических, семантических и т.д.). Для интерпретатора ситуация обработки ошибок другая, при обнаружении ошибки, нет смысла интерпретировать и исполнять исходный текст программы, нужно локализовать место ошибки, выдать сообщение с указанием места ошибки в исходном тексте программы и выйти из интерпретатора. Выводим сообщение об ошибке в поток и генерируем повторное исключение с пустой строкой сообщения. В основном модуле интерпретатора *Program.cs* перехватываем все исключения, и анализируем было это исключение обработано или нет.

```
      try {
        …
        Scanner scanner = new Scanner(DestName);
        Parser parser = new Parser(scanner);
        …
        parser.Parse();
        …
        return (parser.Result);
      }
      catch (Exception e){
        if ((e.Message != null) && (e.Message != ""))
          Console.WriteLine("Error! " + e.Message);
        …
        return 1;
      }
```

Файл Alfa.atg без реализации

После создания требований, определения основных конструкций языка программирования и создания строительного участка, мы должны перейти к созданию спецификации языка программирования в терминах и правилах компилятора компиляторов Coco/R. Для описания

синтаксиса языка используются Расширенные Бэкуса-Наура формы (РБНФ). Инструкция и правила изложены в документе “Compiler Generator Coco/R. User Manual. – University of Linz.” Ниже представлено описание языка Alfa без реализации.

```
COMPILER Alfa

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_".
  digit = "0123456789".
  hexDigit = digit + "ABCDEFabcdef".
  notDigit = ANY - digit.
  eol = '\u000a'.
  cr  = '\r'.
  lf  = '\n'.
  tab = '\t'.
  newLine = cr + eol.
  stringCh = ANY - '"' - newLine.
  notNewLine = ANY - newLine.
  ws = " " + tab + '\u000b' + '\u000c'.

TOKENS
  ident  = letter {letter | digit}.
  intCon =
    (
      digit {digit} | digit {digit} CONTEXT ("." notDigit)
      |
      ("0x" | "0X") hexDigit {hexDigit}
    ) .

  realCon =
    "." digit {digit} [("e" | "E") ["+" | "-"] digit {digit}]
    |
    digit {digit}
    (
      "." digit {digit} [("e" | "E" ) ["+" | "-"] digit {digit}]
      |
      ("e" | "E") ["+" | "-"] digit {digit}
    ).

  stringLit   = '"' { stringCh | "\"\"" } '"' .
  markerbeg = "<!--".
  markerend = "-->".
  true = "true".
  false = "false".
  xor = "xor".
  or = "or".
  and = "and".
  shl = "shl".
  shr = "shr".
  not = "not".
  real = "real".
  int = "int".
  bool = "bool".
  nil = "nil".
  string = "string".
  file = "file".
  server = "server".
  comobj = "comobj".
  proc = "proc".
  func = "func".
  array = "array".
  of = "of".
```

```
  program = "program".
  begin = "begin".
  declare = "declare".
  end = "end".
  const = "const".
  var = "var".
  if = "if".
  then = "then".
  else = "else".
  endif = "endif".
  switch = "switch".
  endswitch = "endswitch".
  case = "case".
  endcase = "endcase".
  default = "default".
  for = "for".
  to = "to".
  endfor = "endfor".
  while = "while".
  endwhile = "endwhile".
  loop = "loop".
  endloop = "endloop".
  readln = "readln".
  writeln = "writeln".
  write = "write".
  exit = "exit".
  return = "return".
  procedure = "procedure".
  function = "function".
  EQ = "=".
  NE = "<>".
  LT = "<".
  LE = "<=".
  GT = ">".
  GE = ">=".
  DIV = "/".
  MUL = "*".
  PLUS = "+".
  MINUS = "-".
  scolon = ";".
  lpar   = "(".
  lbrack = "[".
  dot = ".".
  point = "->".

PRAGMAS
  ppOnWait = "#" {ws} "on_wait" {notNewLine} newLine.
  ppOffWait = "#" {ws} "off_wait" {notNewLine} newLine.
  ppOnLog = "#" {ws} "on_log" {notNewLine} newLine.
  ppOffLog = "#" {ws} "off_log" {notNewLine} newLine.
  ppOnUTF8 = "#" {ws} "on_UTF8log" {notNewLine} newLine.
  ppOffUTF8 = "#" {ws} "off_UTF8log" {notNewLine} newLine.
  ppOnKey = "#" {ws} "on_keywait" {notNewLine} newLine.
  ppOffKey = "#" {ws} "off_keywait" {notNewLine} newLine.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO lf

IGNORE cr + lf + tab

PRODUCTIONS
```

```
Alfa =
  program Ident<out Name>  ";"
  Declarations
  begin
    Operations
    { FuncDeclare<ref ListParam, ref Var> | ProcDeclare<ref ListParam> }
  end "." .

Declarations =
  { const
    { ConstDeclare<out Var> ";" } |
  var
  { VarDeclare<out Var> ";" } }
  { ServDeclare<out oServ, out Name> } .

ConstDeclare<out ObjVar Var> =
  ConstName<out Name>  "=" ConstExpr<out Var> .

ConstExpr<out ObjVar Var>
=
  [ Sign<out TypeOper> ] ( IntDef<out IntValue> | RealDef<out RealValue> )
  |
  StringChar<out Value>
  |
  StringScript<out Value>
  |
  true
  |
  false .

VarDeclare<out ObjVar Var>
=
 ["@"] Ident<out Name> ":" Type<out ValueVarType, out DimArr>
   [ "=" ( ConstExpr<out Var1> | ListParam<ref ListPar> )] .

ServDeclare<out ObjServ oServ, out string Name> =
  declare "@" ServName<out Name> ":" server
    Ident<out NameField> "=" StringChar<out Value> ";"
    Ident<out NameField> "=" StringChar<out Value> ";"
    Ident<out NameField> "=" StringChar<out Value> ";"
    Ident<out NameField> "=" StringChar<out Value> ";"
    Ident<out NameField> "=" intCon ";"
    Ident<out NameField> "=" intCon ";"
  end ";" .

ProcDeclare <ref ArrayList ListParam> =
  procedure Ident<out Name> FormalParameters<ref ListFormalPar> ";"
  { const
    { ConstDeclare<out Var> ";" }
  |
  var
    { VarDeclare<out Var> ";" } }
  begin
    Operations
  end ";" .

FuncDeclare <ref ArrayList ListParam, ref ObjVar Result> =
  function Ident<out Name> FormalParameters<ref ListFormalPar> ":"
    Type<out ValueVarType, out DimArr> ";"
  { const
    { ConstDeclare<out Var> ";" }
  |
```

```
  var
    { VarDeclare<out Var> ";" } }
  begin
    Operations
  end ";" .

FormalParameters<ref ArrayList ListFormalPar>
=
  "(" [ FormalPar<out Var> { ";" FormalPar<out Var> } ] ")" | "()" .

FormalPar<out ObjVar Var>
=
  [ var ] ( "@" ServName<out Name> | Ident<out Name> )
    ":" Type<out ValueVarType, out DimArr> .

Type<out TypeVarValue ValueVarType, out int DimArr> =
  real | int | bool | string | file | server | comobj | proc | func
  |
  array [ Expr<out Var> ] of Type<out ValueVarType, out DimArr> .

Operations = [ Operation { Operation } ] .

Operation = Assign | If | Switch | For | While | Loop | ReadLn | Write |
            WriteLn | Exit .

Assign =
  Variable<ref NameVar> ( ":=" Expr<out Var> | ListParam<ref ListPar> ) ";" .

Exit = exit ";" .

If =
  if Expr<out Var> then
    Operations
  [ else
    Operations ]
  endif ";" .

Switch =
  switch Expr<out Var>
    CaseSection<ref Var, out Result>
    { CaseSection<ref Var, out Result> }
    [ default ":" Operations ]
  endswitch ";" .

CaseSection<ref ObjVar Var1, out bool Result> =
  case Expr<out Var> ":"
    Operations
  endcase ";" .

For =
  for Ident<out Name> ":=" Expr<out Var> to Expr<out Var>
    Operations
  endfor ";" .

While =
  while Expr<out Var>
    Operations
  endwhile ";" .

Loop =
  loop
    Operations
```

```
  endloop ";" .

ReadLn = readln [ ListParam<ref ListPar> ] ";" .

WriteLn = writeln [ ListParam<ref ListPar> ] ";" .

Write = write [ ListParam<ref ListPar> ] ";" .

ListParam<ref ArrayList ListPar> =
  "(" [ Param<out Var> { "," Param<out Var> } ] ")" | "()" .

Expr<out ObjVar Var> =
  SimpleExpr<out Var> [ RelatOper<out TypeOper> SimpleExpr<out Var1> ] .

SimpleExpr<out ObjVar Var> =
  [ Sign<out TypeOper> ] Term<out Var> { AddOper<out TypeOper> Term<out Var1>
} .


Term<out ObjVar Var> =
  Factor<out Var> { MulOper<out TypeOper> Factor<out Var1> } .

Factor<out ObjVar Var> =
  ConstUnSigned<out Var>
  |
  Variable<ref NameVar> [ ListParam<ref ListPar> ]
  |
  not Factor<out Var>
  |
  "(" Expr<out Var> ")" .

RelatOper<out int TypeOper> = "=" | "<>" | "<" | "<=" | ">" | ">=" .

AddOper<out int TypeOper> = "+" | "-" | or | xor .

MulOper<out int TypeOper> = "*" | "/" | and | shl | shr .

ConstUnSigned<out ObjVar Var> =
  RealDef<out RealValue>
  |
  IntDef<out IntValue>
  |
  StringChar<out Value>
  |
  StringScript<out Value>
  |
  true
  |
  false
  |
  nil .

Variable<ref DescrNameVar NameVar> =
 ["@"] Ident<out Name> [( "."  Ident <out NameField> |
   "->" Ident <out NameMethod> | "[" Expr <out Var> "]" )] .

Param<out ObjVar Var> = Expr<out Var> .

IntDef<out int IntValue> = intCon .

RealDef<out double RealValue> = realCon .
```

```
ConstName<out string Name> = Ident<out Name> .

ServName<out string Name> = IdentServ<out Name> .

StringScript<out string Value> = markerbeg {ANY} markerend .

StringChar<out string Value> = stringLit .

Sign<out int TypeOper> = MINUS | PLUS .

Ident<out string Name> = ident .

IdentServ<out string Name> = ident .

END Alfa.
```

Перечень модулей интерпретатора

Интерпретатор Alfa состоит из следующих модулей:

Имя файла	**Назначение**
Program.cs	Основной модуль интерпретатора. Кодировка: Кириллица (Windows) – кодовая страница 1251.
Parser.cs	Синтаксический анализатор. Создается Coco/R на основе файла Alfa.atg. Кодировка: Юникод (UTF-8, с сигнатурой), кодовая страница 65001.
Scanner.cs	Лексический анализатор. Создается Coco/R на основе файла Alfa.atg. Кодировка: Юникод (UTF-8, с сигнатурой), кодовая страница 65001.
TableSymb.cs	Модуль определения классов таблиц, переменных, процедур, функций и т.п. интерпретатора Кодировка: Кириллица (Windows), кодовая страница 1251.
OM.cs	Модуль вспомогательных классов интерпретатора. Кодировка: Кириллица (Windows), кодовая страница 1251.
Alfa.atg	Файл исходного описания языка Alfa для Coco/R. Кодировка: Юникод (UTF-8, с сигнатурой), кодовая страница 65001 .
saRemoteServ.dll	Внутренний сервер автоматизации,

	позволяет выполнять команды на удаленном сервере. Примечание: Сервер автоматизации перед использованием должен быть зарегистрирован в системе. Для этого нужно выполнить следующую командную строку из каталога, где находится модуль saRemoteServ: *RegSvr32 saRemoteServ.dll*

Таблица 1 Список файлов проекта

Схема создания проекта интерпретатора

В *MS Visual Studio* создаем проект консольного приложения. Создаем файл *Alfa.atg*. Запускаем приложение *Coco/R* в *MS Visual Studio*, выполнив команду *Сервис->Coco/R*. Получаем два файла *Scanner.cs* и *Parser.cs*. Компилируем и компонуем проект, на выходе получаем *Alfa.exe* – выполняемый файл интерпретатора.

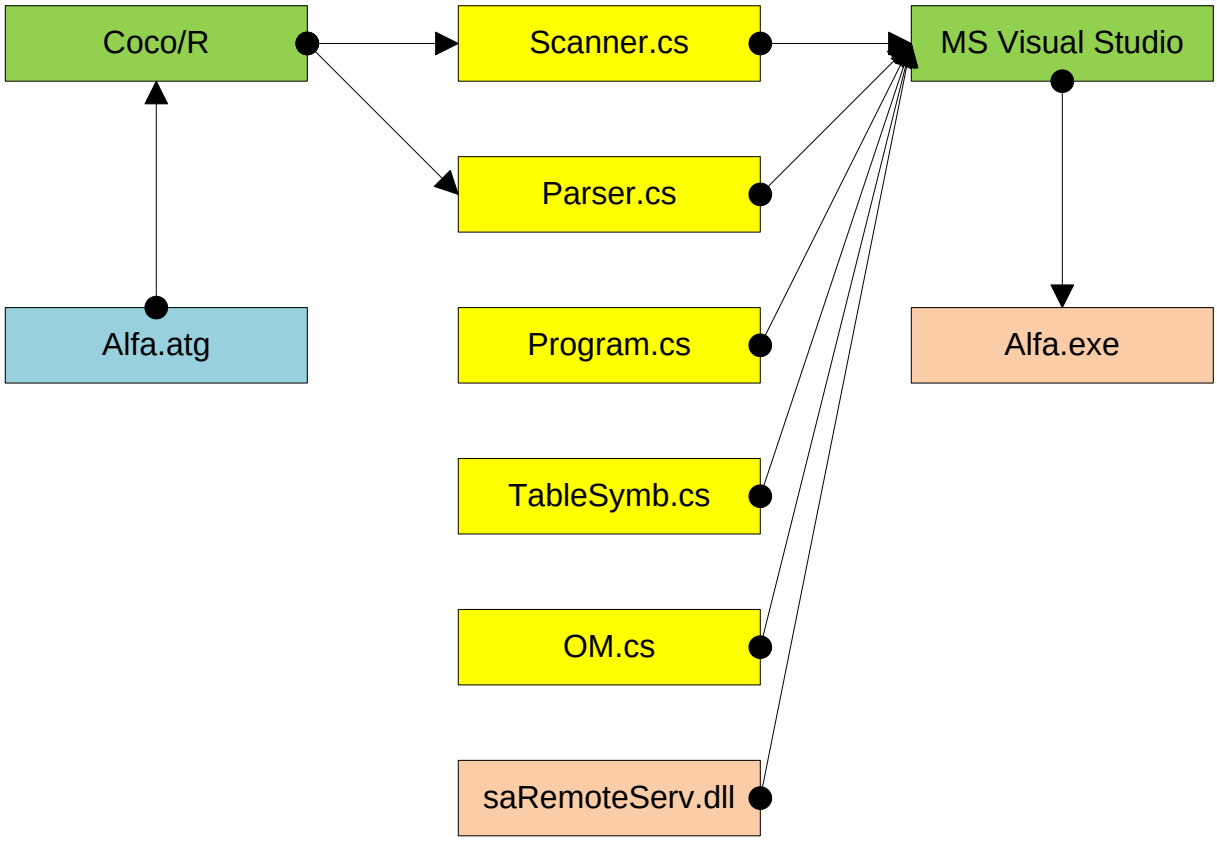

Рис. 4 Схема проекта

Переменные интерпретатора

Переменные интерпретатора определяются в файле *Alfa.atg*, сразу за ключевым словом *COMPILER*. Далее приведено описание основных переменных интерпретатора с кратким комментарием.

```
[DllImport("kernel32.dll")]
static extern IntPtr GetConsoleWindow ();

[DllImport("user32.dll")]
static extern bool ShowWindow (IntPtr hWnd, int nCmdShow);

const int SW_HIDE = 0;
const int SW_SHOW = 5;

// таблица переменных
private TableObj<ObjVar> VarTable = new TableObj<ObjVar>();
// таблица глобальных констант
private TableObj<ObjVar> ConstTable = new TableObj<ObjVar>();
// таблица локальных переменных
private TableObj<ObjVar> VarLocalTable = new TableObj<ObjVar>();
// таблица меток
private TableObj<ObjLabel> LabelTable = new TableObj<ObjLabel>();
// таблица процедур и функций
private TableObj<ObjFunc> FuncTable = new TableObj<ObjFunc>();
// таблица методов серверного объекта
private TableObj<ObjFunc> MethodTable = new TableObj<ObjFunc>();
// стек локальных переменных процедуры или функции
private Stack VarLocalStack = new Stack();
// стек подпрограмм
private Stack GoSubStack = new Stack();
// стек цикла For
private Stack ForStack = new Stack();
// стек цикла While
private Stack WhileStack = new Stack();
// стек цикла Loop
private Stack LoopStack = new Stack();
// объект таймера
private Timer oTimer = null;
// об'ект вывода сообщений
private OM oOM = null;
// об'ект вывода на консоль статуса ожидания
private OutCharWait OCW = new OutCharWait();
// об'ект файла инициализации
private saIniFile oIni = null;
// об'ект E-Mail
private Mail oMailServ = null;
// об'ект FTP
private FTP oFTPServ = null;
// об'ект выполнения скриптов на JScript, VBScript
private Script oScript = null;
// имя файла протокола
private string namefilelog = "";
// имя файла исходного текста программы в кодировке Win-1251
private string namefilesrc = "";
// флаг вывода на консоль статуса ожидания
private bool FOutWait = true;
// флаг вывода сообщений в файл протокола
private bool FOutLog = true;
// флаг кодировки файла протокола в UTF8
private bool FUTF8 = false;
// флаг ожидания нажатия клавиши для закрытия консоли,
//   при возникновении исключения
public static bool FKey = true;
// текущий выполняемый оператор
private int Oper = 0;
// флаг работы с локальными переменными
private bool FlagLocal = false;
```

```
// код завершения программы
public static int Result = 0;
// массив параметров командной строки
public string[] Args;
// ссылка на объект потока исходного текста программы в памяти,
// если файл исходного текста был зашифрован
public MemoryStream Mem = null;
// флаг работы с зашифрованным файлом
public bool FCrypt = false;
// флаг видимости консоли приложения
public bool FConsole = true;
// сообщение об последней ошибке после выполнения стандартных функций
public string MessErr = "";

// имя файла протокола
public string NameFileLog
{
  get
  {
    return namefilelog;
  }
  set
  {
    if (namefilelog != value) {
      namefilelog = value;
    }
  }
}

// имя файла исходного текста программы в кодировке Win-1251
public string NameFileSrc
{
  get
  {
    return namefilesrc;
  }
  set
  {
    if (namefilesrc != value) {
      namefilesrc = value;
    }
  }
}
```

Представление переменных языка Alfa в интерпретаторе

Одним из ключевых моментов при построении интерпретатора, является представление переменных языка программирования. Чтобы оперировать переменной, нужно знать о ней, по крайней мере следующее: имя переменной, тип (константа или переменная), тип значения переменной (целое, строковое, плавающее и т.д.), само значение переменной или ссылка на объект, который отображает значение переменной.

Для представления переменных языка *Alfa* в интерпретаторе используется структура имеющая тип *ObjVar* (модуль *TableSymb.cs*) описание которой приведено далее.

```
  // объект переменной
public struct ObjVar {
  // объект переменной
  public Object Obj;
  // тип COM объекта
  public Type tp;
```

```
// имя переменной
public string Name;
// тип переменной
public TypeVar VarType;
// тип значения переменной
public TypeVarValue ValueType;

// конструктор класса
public ObjVar (string VarName, TypeVar Type, TypeVarValue ValType,
               int Value) {
  this.Obj = null;
  this.tp = null;
  this.Name = VarName;
  this.VarType = Type;
  this.ValueType = ValType;
  switch (ValType) {
    case TypeVarValue.RealArr:
      this.Obj = new double[Value];
      break;
    case TypeVarValue.IntArr:
      this.Obj = new int[Value];
      break;
    case TypeVarValue.StrArr:
      this.Obj = new string[Value];
      break;
    case TypeVarValue.BoolArr:
      this.Obj = new bool[Value];
      break;
    case TypeVarValue.FileArr:
      this.Obj = new ObjFile[Value];
      break;
    case TypeVarValue.ServerArr:
      this.Obj = new ObjServ[Value];
      break;
    case TypeVarValue.ComObjArr:
      this.Obj = new ObjVar[Value];
      break;
    case TypeVarValue.ProcArr:
    case TypeVarValue.FuncArr:
      this.Obj = new string[Value];
      break;
    case TypeVarValue.Real:
      this.Obj = 0.0;
      break;
    case TypeVarValue.Int:
      this.Obj = 0;
      break;
    case TypeVarValue.Str:
      this.Obj = "";
      break;
    case TypeVarValue.Bool:
      this.Obj = false;
      break;
    case TypeVarValue.File:
      this.Obj = new ObjFile();
      break;
    case TypeVarValue.Server:
      this.Obj = new ObjServ();
      break;
    case TypeVarValue.ComObj:
      this.Obj = null;
      this.tp = null;
```

```
        break;
      case TypeVarValue.Proc:
        this.Obj = "";
        break;
      case TypeVarValue.Func:
        this.Obj = "";
        break;
    }
  }

  // конструктор класса
  public ObjVar (string VarName, TypeVar Type, TypeVarValue ValType) {
    this.tp = null;
    this.Name = VarName;
    this.VarType = Type;
    this.ValueType = ValType;
    switch (ValType) {
      case TypeVarValue.Real:
        this.Obj = 0.0;
        break;
      case TypeVarValue.Int:
        this.Obj = 0;
        break;
      case TypeVarValue.Str:
        this.Obj = "";
        break;
      case TypeVarValue.Bool:
        this.Obj = false;
        break;
      default:
        Obj = null;
        break;
    }
  }

  // сравнение типов значений объектов по присваиванию
  public bool CompareType (TypeVarValue Type) {
    bool Res = false;

    switch (this.ValueType) {
      case TypeVarValue.RealArr:
        if (Type == TypeVarValue.RealArr || Type == TypeVarValue.Real ||
            Type == TypeVarValue.Int)
          Res = true;
        break;
      case TypeVarValue.IntArr:
        if (Type == TypeVarValue.IntArr || Type == TypeVarValue.Int)
          Res = true;
        break;
      case TypeVarValue.StrArr:
        if (Type == TypeVarValue.StrArr || Type == TypeVarValue.Str)
          Res = true;
        break;
      case TypeVarValue.BoolArr:
        if (Type == TypeVarValue.BoolArr || Type == TypeVarValue.Bool)
          Res = true;
        break;
      case TypeVarValue.FileArr:
        if (Type == TypeVarValue.FileArr || Type == TypeVarValue.File ||
            Type == TypeVarValue.None)
          Res = true;
        break;
```

```
case TypeVarValue.ServerArr:
  if (Type == TypeVarValue.ServerArr ||
      Type == TypeVarValue.Server ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.ComObjArr:
  if (Type == TypeVarValue.ComObjArr ||
      Type == TypeVarValue.ComObj ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.ProcArr:
  if (Type == TypeVarValue.ProcArr ||
      Type == TypeVarValue.Proc ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.FuncArr:
  if (Type == TypeVarValue.FuncArr ||
      Type == TypeVarValue.Func ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.Real:
  if (Type == TypeVarValue.Real ||
      Type == TypeVarValue.Int)
    Res = true;
  break;
case TypeVarValue.Int:
  if (Type == TypeVarValue.Int)
    Res = true;
  break;
case TypeVarValue.Str:
  if (Type == TypeVarValue.Str ||
      Type == TypeVarValue.Proc ||
      Type == TypeVarValue.Func)
    Res = true;
  break;
case TypeVarValue.Bool:
  if (Type == TypeVarValue.Bool)
    Res = true;
  break;
case TypeVarValue.File:
  if (Type == TypeVarValue.File ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.Server:
  if (Type == TypeVarValue.Server ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.ComObj:
  if (Type == TypeVarValue.ComObj ||
      Type == TypeVarValue.None)
    Res = true;
  break;
case TypeVarValue.Proc:
  if (Type == TypeVarValue.Proc ||
      Type == TypeVarValue.Str ||
      Type == TypeVarValue.None)
```

```
          Res = true;
        break;
      case TypeVarValue.Func:
        if (Type == TypeVarValue.Func ||
            Type == TypeVarValue.Str ||
            Type == TypeVarValue.None)
          Res = true;
        break;
    }
    return Res;
}

// записать значение элемента массива
public void SetValueArr (int Index, double Value) {
    ((double[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, int Value) {
    ((int[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, string Value) {
    ((string[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, bool Value) {
    ((bool[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, ObjFile Value) {
    ((ObjFile[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, ObjServ Value) {
    ((ObjServ[])this.Obj)[Index] = Value;
}

public void SetValueArr (int Index, ObjVar Value) {
    ((ObjVar[])this.Obj)[Index] = Value;
}

// считать значение элемента массива
public ObjVar ValueArray (int Index) {
    ObjVar Var = new ObjVar();
    Var.VarType = TypeVar.Var;
    switch (this.ValueType) {
      case TypeVarValue.RealArr:
        Var.ValueType = TypeVarValue.Real;
        if (this.Obj == null)
          break;
        Var.Obj = ((double[])this.Obj)[Index];
        break;
      case TypeVarValue.IntArr:
        Var.ValueType = TypeVarValue.Int;
        if (this.Obj == null)
          break;
        Var.Obj = ((int[])this.Obj)[Index];
        break;
      case TypeVarValue.StrArr:
        Var.ValueType = TypeVarValue.Str;
        if (this.Obj == null)
          break;
```

```
        Var.Obj = ((string[])this.Obj)[Index];
        break;
      case TypeVarValue.BoolArr:
        Var.ValueType = TypeVarValue.Bool;
        if (this.Obj == null)
          break;
        Var.Obj = ((bool[])this.Obj)[Index];
        break;
      case TypeVarValue.FileArr:
        Var.ValueType = TypeVarValue.File;
        if (this.Obj == null)
          break;
        Var.Obj = ((ObjFile[])this.Obj)[Index];
        break;
      case TypeVarValue.ServerArr:
        Var.ValueType = TypeVarValue.Server;
        if (this.Obj == null)
          break;
        Var.Obj = ((ObjServ[])this.Obj)[Index];
        break;
      case TypeVarValue.ComObjArr:
        Var.ValueType = TypeVarValue.ComObj;
        if (this.Obj == null)
          break;
        Var = ((ObjVar[])this.Obj)[Index];
        break;
      case TypeVarValue.ProcArr:
        Var.ValueType = TypeVarValue.Proc;
        if (this.Obj == null)
          break;
        Var.Obj = ((string[])this.Obj)[Index];
        break;
      case TypeVarValue.FuncArr:
        Var.ValueType = TypeVarValue.Func;
        if (this.Obj == null)
          break;
        Var.Obj = ((string[])this.Obj)[Index];
        break;
    }
    return Var;
  }

  // определение длины массива
  public int LenArr
  {
    get
    {
      if (this.Obj == null)
        return (-1);
      switch (this.ValueType) {
        case TypeVarValue.RealArr:
          return ((double[])this.Obj).Length;
        case TypeVarValue.IntArr:
          return ((int[])this.Obj).Length;
        case TypeVarValue.StrArr:
          return ((string[])this.Obj).Length;
        case TypeVarValue.BoolArr:
          return ((bool[])this.Obj).Length;
        case TypeVarValue.FileArr:
          return ((ObjFile[])this.Obj).Length;
        case TypeVarValue.ServerArr:
          return ((ObjServ[])this.Obj).Length;
```

```
            case TypeVarValue.ComObjArr:
              return ((ObjVar[])this.Obj).Length;
            case TypeVarValue.ProcArr:
            case TypeVarValue.FuncArr:
              return ((string[])this.Obj).Length;
            default:
              return 0;
          }
        }
      }
    }
```

Структура *ObjVar* состоит из следующих полей – объект переменной, тип COM объекта, имя переменной, тип переменной, тип значения переменной.

Допустимые значения типа переменной *VarType* из перечисления *TypeVar*: *TypeVar.None*, *TypeVar.Const*, *TypeVar.Var*, *TypeVar.VarLocal*.

Значение	**Назначение**
TypeVar.None	Неопределено
TypeVar.Const	Константа
TypeVar.Var	Переменная любого типа
TypeVar.VarLocal	Параметр – значение, передаваемое в процедуру или в функцию.

Таблица 2 Значения типов переменных

Перечисление *TypeVar* представлено далее.

```
// определение типов переменных
public enum TypeVar {
  None,
  Var,
  VarLocal,
  Const,
  Func,
  Proc
};
```

Допустимые типы значений переменной *ValueType* из перечисления *TypeVarValue* - все, кроме *TypeVarValue.Array*.

Значение	**Назначение**
TypeVarValue.None	Неопределено
TypeVarValue.RealArr	Значение переменной - массив вещественных чисел
TypeVarValue.IntArr	Значение переменной - массив целых чисел
TypeVarValue.StrArr	Значение переменной - массив строк
TypeVarValue.BoolArr	Значение переменной - массив логических значений
TypeVarValue.FileArr	Значение переменной - массив файловых переменных
TypeVarValue.ServerArr	Значение переменной - массив серверных переменных
TypeVarValue.ComObjArr	Значение переменной - массив COM переменных
TypeVarValue.ProcArr	Значение переменной - массив процедурных типов (тип - процедура)
TypeVarValue.FuncArr	Значение переменной - массив процедурных типов (тип - функция)
TypeVarValue.Real	Значение переменной – вещественное число
TypeVarValue.Int	Значение переменной – число
TypeVarValue.Str	Значение переменной – строка
TypeVarValue.Bool	Значение переменной - булевское значение
TypeVarValue.File	Файловая переменная

TypeVarValue.Server	Серверная переменная
TypeVarValue.ComObj	COM переменная
TypeVarValue.Proc	Переменная процедурного типа (тип - процедура)
TypeVarValue.Func	Переменная процедурного типа (тип - функция)
TypeVarValue.Var	Переменная любого типа

Таблица 3 Типы значений переменной

Перечисление *TypeVarValue* представлено далее.

```
// определение типа значения переменной
public enum TypeVarValue {
  None,
  Array,
  RealArr,
  IntArr,
  StrArr,
  BoolArr,
  FileArr,
  ServerArr,
  ComObjArr,
  ProcArr,
  FuncArr,
  Real,
  Int,
  Str,
  Bool,
  File,
  Server,
  ComObj,
  Proc,
  Func,
  Var
};
```

Рис. 5 Представление переменной

Объекты переменных, серверов, файловых переменных, COM переменных, метки процедур и функций сохраняются в таблицах переменных, реализованных на базе обобщенного класса TableObj (определен в *TableSymb.cs*) .

```
  // класс таблицы объектов переменных
public class TableObj <TVar> {
  // объект таблицы переменных
  public Dictionary<string, TVar> Table;

  // добавить объект переменной в таблицу
  public void Add (string Name, TVar Obj) {
    Table.Add(Name, Obj);
  }
```

```
    // проверка на существование объекта переменной по имени
    public bool IsName (string Name) {
      return Table.ContainsKey(Name);
    }

    // определить число объектов в таблице
    public int Count () {
      return Table.Count;
    }

    // получить объект переменной по имени
    public TVar GetObj (string Name) {
      return Table[Name];
    }

    // конструктор класса
    public TableObj () {
      Table = new Dictionary<string, TVar>();
    }

    // получить (установить) объект переменной по имени
    public TVar this[string Name] {
      get {
        return Table[Name];
      }
      set {
        Table[Name] = value;
      }
    }
  }
```

Представление файловой переменной в отличие от представление простых (типа int, real, bool, string) переменных немного сложнее. В начале создается экземпляр файлового объекта типа ObjFile, а ссылка на созданный объект помещается в поле *Obj* структуры *ObjVar* и сохраняется в таблице переменных типа *TableObj*.

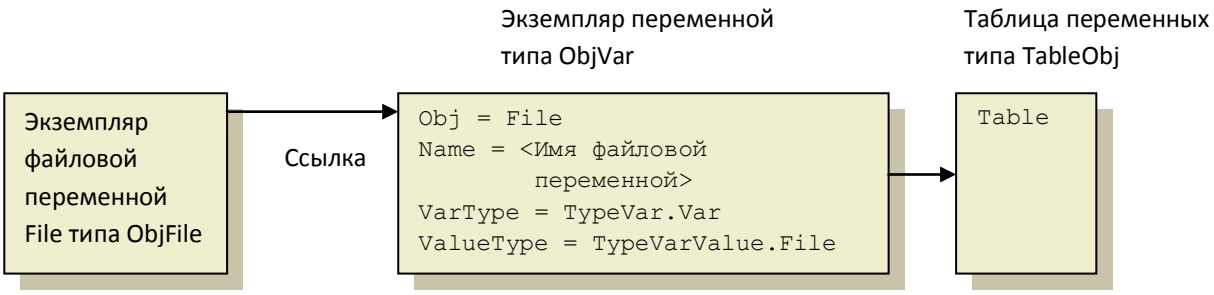

Рис. 6 Представление файловой переменной

Структура *ObjFile* приведена ниже.

```
  // объект файла
  public struct ObjFile {
    // имя файловой переменной
    public string Name;
    // полный путь к файлу
    public string Path;
    // доступ к файлу
```

```
    public FileAccess Access;
    // режим
    public bool Append;
    // объект файлового потока для чтения
    public StreamReader oFR;
    // объект файлового потока для записи
    public StreamWriter oFW;

    public ObjFile (string Name, string Path, FileAccess Access, bool Append)
{
      this.oFW = null;
      this.oFR = null;
      this.Name = Name;
      this.Path = Path;
      this.Access = Access;
      this.Append = Append;
      if (Access == FileAccess.Read)
        oFR = new StreamReader(Path, Encoding.GetEncoding(1251));
      else
        oFW = new StreamWriter(Path, Append, Encoding.GetEncoding(1251));
    }
  }
```

Представление серверной переменной аналогично представлению файловой переменной. Реальный объект сервер, представляет собой внутренний сервер автоматизации типа *saRS*, который представлен полем *oRS* и осуществляет все необходимые действия по работе с удаленным сервером.

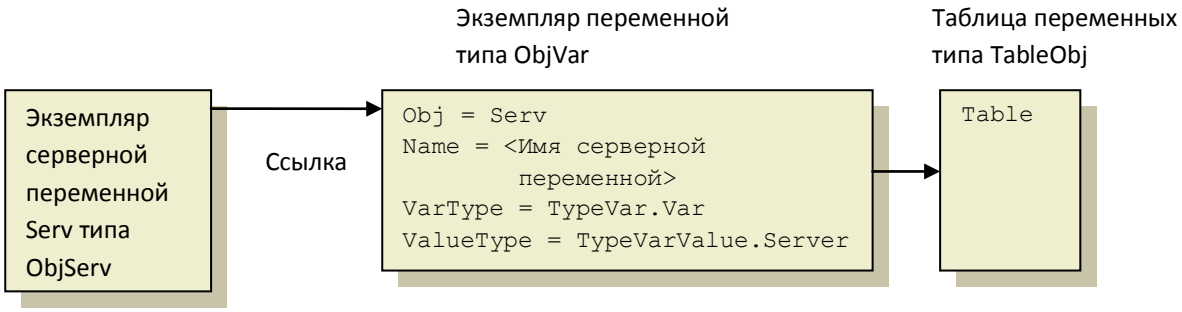

Рис. 6 Представление серверной переменной

Структура *ObjServ* представлена ниже.

```
  // объект описания сервера
  public class ObjServ {
    // объект сервера
    public saRS oRS;
    // имя переменной сервера
    public string Name;
    // наименование сервера
    public string Title;
    // строка состояния сервера
    public string Status;
    // время ожидания перезагрузки сервера
    public int WaitReboot;

    // имя сервера
    public string Host
```

```
{
  get
  {
    if (oRS == null)
      return "";
    else
      return oRS.Host;
  }
  set
  {
    if (oRS != null)
      oRS.Host = value;
  }
}

// пользователь
public string User
{
  get
  {
    if (oRS == null)
      return "";
    else
      return oRS.User;
  }
  set
  {
    if (oRS != null)
      oRS.User = value;
  }
}

// пароль
public string Password
{
  get
  {
    if (oRS == null)
      return "";
    else
      return oRS.Password;
  }
  set
  {
    if (oRS != null)
      oRS.Password = value;
  }
}

// интервал ожидания
public int IntervalWait
{
  get
  {
    if (oRS == null)
      return 0;
    else
      return oRS.IntervalWait;
  }
  set
  {
    if (oRS != null)
```

```
        oRS.IntervalWait = value;
    }
}

// конструкторы класса
public ObjServ() {
    this.oRS = new saRS();
    this.oRS.Host = ".";
    this.oRS.Password = "";
    this.oRS.User = "";
    this.oRS.IntervalWait = 60;
    this.Name = "";
    this.Title = "";
    this.WaitReboot = 300;
    this.Status = "";
}

// конструкторы класса
public ObjServ(string ServNameVar) {
    this.oRS = new saRS();
    this.oRS.Host = ".";
    this.oRS.Password = "";
    this.oRS.User = "";
    this.oRS.IntervalWait = 60;
    this.Name = ServNameVar;
    this.Title = "";
    this.WaitReboot = 300;
    this.Status = "";
}

public ObjServ (string ServNameVar, string ServTitle, string ServHost,
                string ServUser, string ServPassword,
                int ServIntervalWait, int ServWaitReboot) {
    // создаем объект удаленного сервера
    this.oRS = new saRS();
    this.oRS.Host = ServHost;
    this.oRS.User = ServUser;
    this.oRS.Password = ServPassword;
    this.oRS.IntervalWait = ServIntervalWait;
    this.Name = ServNameVar;
    this.Title = ServTitle;
    this.WaitReboot = ServWaitReboot;
    this.Status = "";
}

// определение типа поля
public TypeVarValue GetTypeField (string NameField) {
    switch (NameField) {
        case "Host":
            return TypeVarValue.Str;
        case "User":
            return TypeVarValue.Str;
        case "Password":
            return TypeVarValue.Str;
        case "Title":
            return TypeVarValue.Str;
        case "IntervalWait":
            return TypeVarValue.Int;
        case "WaitReboot":
            return TypeVarValue.Int;
        default:
            return TypeVarValue.None;
```

```
      }
    }

  } // ObjServ
```

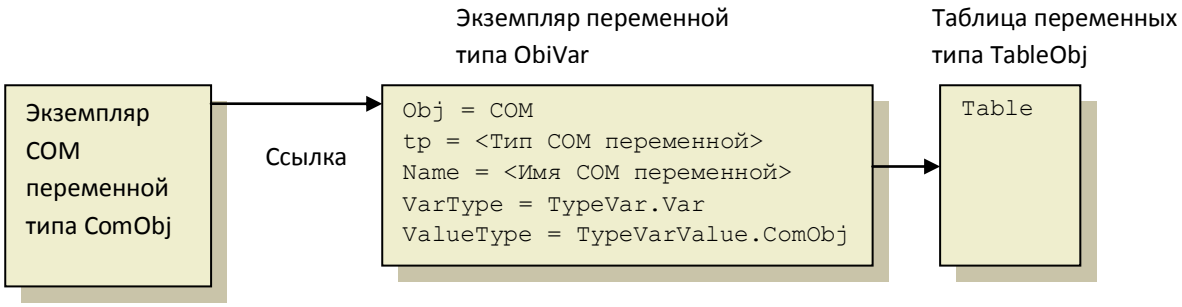

Рис. 8 Представление COM переменной

Пример создания переменных в интерпретаторе.

```
// Создание константы
// создаем экземпляр переменной конструктором по умолчанию
ObjVar Var = new ObjVar();
// получаем имя переменной
…
// присваиваем имя переменной
Var.Name = Name;
// присваиваем тип переменной
Var.VarType = TypeVar.Const;
// присваиваем тип значения переменной
Var.ValueType = …
// присваиваем значение в зависимости от типа
…
// помещаем в таблицу переменных
if (!VarTable.IsNameVar(Var.Name))
  VarTable.Add(Var.Name, Var);
else {
  // выдать сообщение об ошибке "invalid ConstName"
  ErrorMess(203);
}


// Создание переменной
// получаем имя и тип переменной
…
// создаем экземпляр переменной с помощью перегруженного конструктора
Var = new ObjVar(Name, TypeVar.Var, ValueVarType, DimArr);
// присваиваем значение в зависимости от типа
…
// помещаем в таблицу переменных
if (!VarTable.IsNameVar(Var.Name)) {
  VarTable.Add(Var.Name, Var);
}
else {
  // выдать сообщение об ошибке "invalid VarName"
  ErrorMess(204);
}
```

Запуск интерпретатора

Запуск интерпретатора осуществляется в модуле *Program.cs*. Часть исходного кода, осуществляющая запуск интерпретатора, приведена ниже.

```
Scanner scanner = new Scanner(DestName);
Parser parser = new Parser(scanner);
parser.Parse();
```

Рассмотрим подробнее эти три строки исходного кода. Создаем объект *scanner*, вызывая перегруженный конструктор, передавая ему в качестве входного параметра, имя файла исходного текста программы. Создаем объект *parser,* передавая конструктору ссылку на вновь созданный объект *scanner*. Вызываем метод *Parse()*, объекта parser. Метод *Parse()* в свою очередь создает объект *la* типа *Token*, инициализирует поле *la.val*, считывает токен из входного потока вызывая функцию *Get()* и передает управление основной функции интерпретатора *Alfa()*. Метод *Parse()* реализован в *Parse.cs,* исходный код которого приведен ниже.

```
public void Parse () {
  la = new Token();
  la.val = "";
  Get();
  Alfa();
  Expect(0);
}
```

Обработка ошибок в интерпретаторе

Мы знаем, что любая программа может потенциально содержать множество ошибок самого различного рода. Ошибки могут быть лексическими, синтаксическими, семантическими и логическими. *Coco/R*, при компиляции файла описания языка, автоматически генерирует класс обработки ошибок *Errors*, который обрабатывает лексические и синтаксические ошибки. За обработку семантических и логических ошибок, должны отвечать мы. Ниже представлена функция вывода сообщений об ошибках *ErrorMess()*, которая используется в разных частях нашего интерпретатора.

```
//*************************************************************
//* Function: ErrorMessStr
//* Notes: Вывод строки сообщения об ошибке в файл протокола и на консоль
//*

private void ErrorMessStr (string s) {
  int line;
  int col;
  // 0=line, 1=column, 2=text
  string errMsgFormat = "Error, line {0} col {1}: {2}";

  if (t.line == 0) {
    line = la.line;
    col = la.col;
  }
  else {
    line = t.line;
    col = t.col;
  }
  OutLog("", false);
  OutLog(String.Format(errMsgFormat, Convert.ToString(line),
         Convert.ToString(col), s), false);
  Console.WriteLine("");
```

```
  Console.WriteLine(errMsgFormat, line, col, s);
  OutErrStr(line, col);
  LogSaveToFile();
  throw new Exception("");
}

//*** End of ErrorMessStr ***********

//***********************************************************
//* Function: ErrorMess
//* Notes: Формирование и вывод сообщения об ошибке в файл протокола и на
//*        консоль
//*

private void ErrorMess (int n) {
  string s;

  switch (n) {
    case 200: s = "invalid Label"; break;
    case 201: s = "invalid Var"; break;
    case 202: s = "invalid ServName"; break;
    case 203: s = "invalid ConstName"; break;
    case 204: s = "invalid VarName"; break;
    case 205: s = "invalid count Param"; break;
    case 206: s = "dividing by zero"; break;
    case 207: s = "invalid number is read"; break;
    case 208: s = "invalid boolean is read"; break;
    case 209: s = "invalid Type"; break;
    case 210: s = "invalid RelatOper"; break;
    case 211: s = "invalid MulOper"; break;
    case 212: s = "invalid operator if"; break;
    case 213: s = "invalid operator for, absent endfor"; break;
    case 214: s = "invalid operator while, absent endwhile"; break;
    case 215: s = "invalid Type result function"; break;
    case 216: s = "invalid Name function or procedure"; break;
    case 217: s = "invalid Name, not function"; break;
    case 218: s = "invalid operator loop, absent endloop"; break;
    case 219: s = "invalid operator exit"; break;
    case 220: s = "invalid definition function"; break;
    case 221: s = "invalid definition procedure"; break;
    case 222: s = "it needs function"; break;
    case 223:
      s = "invalid assigment statement, requirement variable";
     break;
    case 224: s = "error open file"; break;
    case 225: s = "invalid Type parameter"; break;
    case 226: s = "invalid operator case"; break;
    case 227: s = "invalid operator switch"; break;
    case 228: s = "invalid parameter"; break;
    case 229: s = "invalid index array"; break;
    case 230: s = "invalid variable operator for"; break;
    case 231: s = "invalid type dimension array"; break;
    case 232: s = "invalid FieldName"; break;
    case 233: s = "invalid dimension array"; break;
    case 234: s = "invalid quantity of elements of initialization"; break;
    case 235:
     s = "invalid initial initialization for the given type of a variable";
     break;
    case 236: s = "invalid Name method"; break;
    case 237: s = "the parametre - a variable is required"; break;
    case 238: s = "the marker of the end of a script is not found"; break;
    case 239: s = "Title expected"; break;
```

```
      case 240: s = "Host expected"; break;
      case 241: s = "User expected"; break;
      case 242: s = "Password expected"; break;
      case 243: s = "IntervalWait expected"; break;
      case 244: s = "WaitReboot expected"; break;
      case 245: s = "function with such name already exists"; break;
      case 246: s = "procedure with such name already exists"; break;
      case 247: s = "value of a variable is not defined"; break;
      case 248: s = "property cannot transfer parametres"; break;
      case 249: s = "for performance it is required COM object"; break;
      case 250: s = "the object method cannot appropriate value"; break;
      case 251: s = "invalid ComObj"; break;
      case 252: s = "error of creation of object - ComObj"; break;
      default: s = "error " + Convert.ToString(n); break;
    }
    ErrorMessStr(s);
  }

  //*** End of ErrorMess ***********
```

Функция ErrorMess() выводит собщение об ошибке, а так же строку исходного текста с указателем на ошибочную позицию. Вывод осуществляется с помощью функции OutErrStr(). Исходный код функции представлен ниже.

```
  //******************************
  //* Function: OutStrErr
  //* Notes: Вывод строки исходного текста с указанием позиции ошибки
  //*        на консоль и в файл протокола

  public void OutErrStr(int Line, int Col)
  {
    FileStream fs;
    StreamReader FIn;
    Encoding Enc = Encoding.GetEncoding("windows-1251");
    string Str = "";
    string StrPtr = "";
    int LineStr = 0;

    // открываем файл исходного текста программы для чтения
    try {
      if (!FCrypt) {
        fs = new FileStream(NameFileSrc, FileMode.Open, FileAccess.Read,
                            FileShare.Read);
        FIn = new StreamReader(fs, Enc);
      }
      else {
        // установить положение в потоке на начало
        Mem.Seek(0, SeekOrigin.Begin);
        FIn = new StreamReader(Mem, System.Text.Encoding.UTF8);
      }
    }
    catch (Exception e) {
      Console.WriteLine(
        "Ошибка открытия файла: " + NameFileSrc + " " + e.Message);
      return;
    }

    try {
      while (!FIn.EndOfStream) {
        Str = FIn.ReadLine();
        LineStr++;
```

```
        // если найдена ошибочная строка, то выходим из цикла
        if (LineStr == Line)
          break;
      }
    }
    finally {
      // закрываем выходной файл
      FIn.Close();
    }
    // выводим строку ошибки и указатель на ошибочную позицию
    // в файл протокола и на консоль
    OutLog("->" + Str, false);
    StrPtr = "^";
    OutLog(StrPtr.PadLeft(Col + 2), false);
    StrPtr = "|";
    OutLog(StrPtr.PadLeft(Col + 2), false);
    Console.WriteLine("->" + Str);
    StrPtr = "^";
    Console.WriteLine(StrPtr.PadLeft(Col + 2));
    StrPtr = "|";
    Console.WriteLine(StrPtr.PadLeft(Col + 2));
  } // OutErrStr

  //*** End of OuttErrStr ***********
```

Программа Alfa

Синтаксис определения программы следующий:

```
Alfa =
  program Ident<out Name> ";"
  Declarations
  begin
    Operations
  { FuncDeclare<ref ListParam, ref Var> | ProcDeclare<ref ListParam> }
  end "." .
```

Определение программы реализуется функцией *Alfa()*. Функция *Alfa()* является основной функцией в интерпретаторе. С её вызова начинается исполнение исходного кода программы. Если посмотреть на синтаксис определения программы, можно заметить ряд несуразностей. Определение начинается с ключевого слова “*program*” затем идет идентификатор программы, далее декларация констант, переменных, серверов. Начало программы “*begin*”, последовательность операторов, определение процедур и функций и наконец, конец программы ключевое слово “*end*.”. Позвольте, спросите Вы, как же так? Определение процедур и функций должны быть после определения констант и переменных? Да это так. Более того, определение процедур и функций фиктивно и код определения в функции *Alfa()* не используется. Почему? Мы пишем интерпретатор и любой код, встретившийся в исходном тексте программы мы должны исполнять, а исполнять пока нечего, вызовы процедур и функций в исходном тексте программы будут только после ключевого слова “*begin*”. Вызовы *FuncDeclare()* и *ProcDeclare()* делаем фиктивными для того, чтобы не получить ошибку от *Coco/R* - продукция есть, а использования нет. Код функции *Alfa()* приведен ниже.

```
Alfa

          (.
             string Name = "";
             ArrayList ListParam = new ArrayList();
```

```
            ObjVar Var = new ObjVar();
            // инициализировать таблицу процедур и функций
            IniFuncTable(ref FuncTable);
            // инициализировать таблицу методов серверного объекта
            IniMethodTable(ref MethodTable);
            // создать переменную result
            VarTable.Add("result", new ObjVar("result", TypeVar.Var,
                         TypeVarValue.Int));
        .)

=
  program Ident<out Name>  ";"

        (.
            // скрыть окно консоли
            if (!FConsole) {
              var handle = GetConsoleWindow();
              ShowWindow(handle, SW_HIDE);
            }
            if (FOutLog) {
              // создаем об'ект вывода сообщений
              oOM = new OM(namefilelog, Name);
              // установить кодировку файла протокола
              if (FUTF8)
                // кодировка UTF8
                oOM.UTF8 = true;
              else
                // кодировка WIN-1251
                oOM.UTF8 = false;
            }

        .)

  Declarations

        (.
            // найти и поместить метки процедур и функций в таблицу
            // (позицию начала определений процедур и функций в исходном
            //  тексте)
            RetrievalLabel();
        .)

  begin
  Operations

        (.
            // если текущий оператор Exit, то выйти из программы
            if (Oper == _exit) {
              Oper = 0;
              la.kind = 0;
              // получить возвращаемый признак программы
              Var = VarTable["result"];
              Result = (int)Var.Obj;
              return;
            }
            goto ExitProg;
        .)

  { FuncDeclare<ref ListParam, ref Var> | ProcDeclare<ref ListParam> }

        (.
            ExitProg:
```

```
            // получить возвращаемый признак программы
            Var = VarTable["result"];
            Result = (int)Var.Obj;
          .)

  end "." .
```

В функции *Alfa()* выполняем следующее. Инициализируем таблицу стандартных процедур и функций. Для этого вызываем функцию *IniFuncTable()*, которая инициализирует объект таблицы *FuncTable*, данными стандартных процедур и функций. Инициализируем таблицу методов серверного объекта, вызывая функцию *IniMethodTable()*, где в качестве параметра передаем ссылку на таблицу методов серверного объекта *MethodTable*. Создаем преопределенную переменную *result*, типа *int*. Предопределенная переменная *result* возвращает код завершения программы. Считываем из исходного текста программы ключевое слово "*program*" и имя программы. Если установлен флаг вывода сообщений программы в файл протокола, создаем объект вывода сообщений *oOM*. Устанавливаем признак кодировки файла протокола. Считываем объявления констант, переменных, серверов с помощью вызова функции *Declarations()*. Находим с помощью функции *RetrievalLabel()* начало объявления каждой процедуры или функции и помещаем в объект таблицы *FuncTable*. Переходим к началу программы, считываем ключевое слово "*begin*", выполняем последовательность операторов, с помощью вызова функции *Operations()*. Если текущий оператор exit, получаем возвращаемый признак программы (считываем значение предопределенной переменной *result*) и немедленно выходим из программы. Иначе выполняем исходный код программы, пока не встретится конец программы "*end*.", и по окончанию возвращаем код завершения программы (по умолчанию 0).

Объявление констант, переменных и серверов

Синтаксис объявления констант, переменных и серверов следующий:

```
Declarations =
  { const
    { ConstDeclare<out Var> ";" } |
  var
  { VarDeclare<out Var> ";" } }
  { ServDeclare<out oServ, out Name> } .

ConstDeclare<out ObjVar Var> =
  ConstName<out Name>  "=" ConstExpr<out Var> .

ConstExpr<out ObjVar Var> =
  [ Sign<out TypeOper> ] ( IntDef<out IntValue> | RealDef<out RealValue>
)
  |
  StringChar<out Value>
  |
  StringScript<out Value>
  |
  true
  |
  false .

VarDeclare<out ObjVar Var> =
 ["@"] Ident<out Name> ":" Type<out ValueVarType, out DimArr>
   [ "=" ( ConstExpr<out Var1> | ListParam<ref ListPar> )] .

ServDeclare<out ObjServ oServ, out string Name> =
```

```
        declare "@" ServName<out Name> ":" server
          Ident<out NameField> "=" StringChar<out Value> ";"
          Ident<out NameField> "=" StringChar<out Value> ";"
          Ident<out NameField> "=" StringChar<out Value> ";"
          Ident<out NameField> "=" StringChar<out Value> ";"
          Ident<out NameField> "=" intCon ";"
          Ident<out NameField> "=" intCon ";"
        end ";" .
```

Объявление констант, переменных и серверов следует за ключевым словом *program* и индетификатором программы. В начале объявляются константы, далее переменные и затем следует декларация серверов. Объявлением констант предшествует ключевое слово *const*, переменным *var*.

Пример объявлений:

```
program Test;
const
  NameService = "Служба Telnet";
var
  @Serv : server;
  @SP_Serv : server = ("Тестовый", "10.33.44.128", "adm", "pass", 30, 180);
  Data : file;
  Stat : bool;
  N : int;
  S : string;

declare @UAF : server
  Title = "Компьютер UAF";
  Host = ".";
  User = "";
  Password = "";
  IntervalWait = 60;
  WaitReboot = 300;
end;

begin

  …

end.
```

Декларация констант, переменных и серверов, реализована в функции *Declarations()*, исходный код которой приведен ниже.

```
Declarations

          (.
             string Name = "";
             ObjServ oServ = new ObjServ();
             ObjVar Var = new ObjVar();
          .)

=
  { const
  { ConstDeclare<out Var> ";"

          (.
             if (!VarTable.IsName(Var.Name)) {
               VarTable.Add(Var.Name, Var);
```

```
              ConstTable.Add(Var.Name, Var);
            }
            else {
              // выдать сообщение об ошибке "invalid ConstName"
              ErrorMess(203);
            }
          .)

  } |
  var
  { VarDeclare<out Var> ";"

          (.
            if (!VarTable.IsName(Var.Name)) {
              VarTable.Add(Var.Name, Var);
            }
            else {
              // выдать сообщение об ошибке "invalid VarName"
              ErrorMess(204);
            }
          .)

  } }
  {

          (. oServ = new ObjServ(); .)

    ServDeclare<out oServ, out Name>

          (.
            Var.Name = Name;
            Var.VarType = TypeVar.Var;
            Var.ValueType = TypeVarValue.Server;
            if (!VarTable.IsName(Var.Name)) {
              Var.Obj = oServ;
              VarTable.Add(Var.Name, Var);
            }
            else {
              // выдать сообщение об ошибке "invalid ServName"
              ErrorMess(202);
            }
          .)

  } .
```

Объявление константы *ConstDeclare* состоит, из идентификатора константы *ConsName*, следующим за ним знака равенства *“=”* и константным выражением *ConsExpr*. Объявление константы реализуется функцией *ConsDeclarare()* исходный код которой, приведен ниже.

```
ConstDeclare<out ObjVar Var>

          (. string Name = ""; .)

=
  ConstName<out Name>
  "=" ConstExpr<out Var>

          (.
            Var.Name = Name;
            Var.VarType = TypeVar.Const;
          .) .
```

Константным выражением *ConstExpr* может быть: число без знака или с ним, строка символов, булевское значение (*true* или *false*). Предусмотрено альтернативное представление любой последовательности символов “как есть” включая управляющие символы и символы кавычек.

```
StringScript = "<!--" {ANY} "-->" .
```

Такая последовательность начинается с символов `"<!--"` и заканчивается символами `"-->"`.

Исходный код функции константного выражения *ConstExpr()* представлен ниже.

```
ConstExpr<out ObjVar Var>

          (.
             double RealValue = 0.0;
             int IntValue = 0;
             int TypeOper = 0;
             string Value = "";
             Var = new ObjVar();
          .)

=
   [ Sign<out TypeOper> ]
   (
     IntDef<out IntValue>

          (.
             Var.ValueType = TypeVarValue.Int;
             if (TypeOper == _MINUS)
               Var.Obj = -IntValue;
             else
               Var.Obj = IntValue;
          .)
     |
     RealDef<out RealValue>

          (.
             Var.ValueType = TypeVarValue.Real;
             if (TypeOper == _MINUS)
               Var.Obj = -RealValue;
             else
               Var.Obj = RealValue;
          .)
   )
   |
   StringChar<out Value>

          (.
             Var.ValueType = TypeVarValue.Str;
             Var.Obj = Value;
          .)

   |
   StringScript<out Value>

          (.
             Var.ValueType = TypeVarValue.Str;
             Var.Obj = Value;
          .)
```

```
  |
  true

          (.
             Var.ValueType = TypeVarValue.Bool;
             Var.Obj = true;
          .)

  |
  false

          (.
             Var.ValueType = TypeVarValue.Bool;
             Var.Obj = false;
          .) .
```

Примеры объявление констант:

```
const
  E = 0xFFFF;
  N = 100;
  FlagOff = false;
  Str = "строка";
  F = 3.14;
  StrScript =
    <!--
      Function Main()
        Dim fso, fldr
        Set fso = CreateObject("Scripting.FileSystemObject")
        Set fldr = fso.CreateFolder("C:\MyTest1")
        Main = "Выполнено"
      End Function
    -->;
```

Объявление переменной *VarDeclare* состоит из идентификатора переменной *Ident* следующий за ним знак *":"* и тип переменной *Type*. Если тип переменной представляет собой массив или сервер, его можно инициализировать, используя синтаксис объявления массива или серверной переменной, за которой следует операция присваивания и разделенный запятыми список значений, заключенный в круглые скобки.

Объявление переменной реализуется функцией *VarDeclare()*, исходный код которой, приведен далее.

```
VarDeclare<out ObjVar Var>

          (.
             bool Flag = false;
             int DimArr = 0;
             int Count = 0;
             int IntervalWait = 0;
             int WaitReboot = 0;
             string Title = "";
             string Host = "";
             string User = "";
             string Password = "";
             string Name = "";
             ObjVar Var1 = new ObjVar();
             ObjServ oServ;
             ArrayList ListPar = new ArrayList();
             TypeVarValue ValueVarType = new TypeVarValue();
          .)
```

```
=
 ["@"]

          (.
              if (t.val == "@")
                Flag = true;
          .)

 Ident<out Name> ":" Type<out ValueVarType, out DimArr>

          (.
             if (Flag)
               Name = "@" + Name;
             if ((ValueVarType == TypeVarValue.RealArr ||
                  ValueVarType == TypeVarValue.IntArr ||
                  ValueVarType == TypeVarValue.StrArr ||
                  ValueVarType == TypeVarValue.BoolArr ||
                  ValueVarType == TypeVarValue.FileArr ||
                  ValueVarType == TypeVarValue.ServerArr ||
                  ValueVarType == TypeVarValue.ComObjArr ||
                  ValueVarType == TypeVarValue.ProcArr ||
                  ValueVarType == TypeVarValue.FuncArr ) && DimArr < 0)
               // выдать сообщение "invalid dimension array"
               ErrorMess(233);
             // в имени переменной сервера должен быть первый символ @
             else if (ValueVarType == TypeVarValue.Server ||
                      ValueVarType == TypeVarValue.ServerArr) {
               if (Name[0] != '@')
                 // выдать сообщение "invalid ServName"
                 ErrorMess(202);
             }
             Var = new ObjVar(Name, TypeVar.Var, ValueVarType, DimArr);
          .)

[ "=" ( ConstExpr<out Var1>

           (.
              if (Var.ValueType == TypeVarValue.ComObj) {
                if (Var1.ValueType != TypeVarValue.Str)
                  // выдать сообщение об ошибке
                  // "invalid initial initialization for the given type of a
                  // variable"
                  ErrorMess(235);
                try {
                  Name = Var.Name;
                  // создаем COM объект
                  Var = ComObj.Create((string)Var1.Obj);
                  Var.Name = Name;
                }
                catch {
                  // выдать сообщение об ошибке
                  // "error of creation of object - ComObj"
                  ErrorMess(252);
                }
              }
              else if (Var.CompareType(Var1.ValueType)) {
                if (Var.ValueType == TypeVarValue.Real &&
                    Var1.ValueType == TypeVarValue.Int)
                  Var.Obj = Convert.ToDouble(Var1.Obj);
                else
                  Var.Obj = Var1.Obj;
```

```
      }
      else
       // выдать сообщение об ошибке
       // "invalid initial initialization for the given type of a
       // variable"
       ErrorMess(235);
  .)

|
ListParam<ref ListPar>

 (.
    // инициализация массива или объекта server – начальными
    // значениями
    if (ValueVarType == TypeVarValue.RealArr ||
        ValueVarType == TypeVarValue.IntArr ||
        ValueVarType == TypeVarValue.StrArr ||
        ValueVarType == TypeVarValue.ComObjArr ||
        ValueVarType == TypeVarValue.ServerArr ||
        ValueVarType == TypeVarValue.ProcArr ||
        ValueVarType == TypeVarValue.FuncArr ||
        ValueVarType == TypeVarValue.Server) {
      Count = ListPar.Count;
      if (Count > 0) {
        if (ValueVarType == TypeVarValue.Server) {
          if (Count != 6)
            // выдать сообщение "invalid quantity of elements of
            // initialization"
            ErrorMess(234);
        }
        else if (Count != DimArr) {
          // выдать сообщение "invalid quantity of elements of
          // initialization"
          ErrorMess(234);
        }
        for (var i = 0; i <= Count - 1; i++) {
          Var1 = (ObjVar)ListPar[i];
          switch (ValueVarType) {
            case TypeVarValue.RealArr:
              if (Var1.ValueType == TypeVarValue.Real)
                ((double[])Var.Obj)[i] = (double)Var1.Obj;
              else
                // выдать сообщение об ошибке "invalid Type
                // parameter"
                ErrorMess(225);
                break;
            case TypeVarValue.IntArr:
              if (Var1.ValueType == TypeVarValue.Int)
                ((int[])Var.Obj)[i] = (int)Var1.Obj;
              else
                // выдать сообщение об ошибке "invalid Type
                // parameter"
                ErrorMess(225);
                break;
            case TypeVarValue.StrArr:
              if (Var1.ValueType == TypeVarValue.Str)
                ((string[])Var.Obj)[i] = (string)Var1.Obj;
              else
                // выдать сообщение об ошибке "invalid Type
                // parameter"
                ErrorMess(225);
                break;
```

```
case TypeVarValue.BoolArr:
  if (Var1.ValueType == TypeVarValue.Bool)
    ((bool[])Var.Obj)[i] = (bool)Var1.Obj;
  else
    // выдать сообщение об ошибке "invalid Type
    // parameter"
    ErrorMess(225);
  break;
case TypeVarValue.ComObjArr: {
  if (Var1.ValueType != TypeVarValue.Str)
    // выдать сообщение об ошибке
    // "invalid initial initialization for the given type
    // of a variable"
    ErrorMess(235);
  try {
    // создаем COM объект
    ((ObjVar[])Var.Obj)[i] =
      ComObj.Create((string)Var1.Obj);
  }
  catch {
    // выдать сообщение об ошибке
    // "error of creation of object - ComObj"
    ErrorMess(252);
  }
  break;
}
case TypeVarValue.ServerArr: {
  if (Var1.ValueType != TypeVarValue.Server)
    // выдать сообщение об ошибке
    // "invalid initial initialization for the given type
    // of a variable"
    ErrorMess(235);
  ((ObjServ[])Var.Obj)[i] = (ObjServ)Var1.Obj;
  break;
}
case TypeVarValue.ProcArr:
case TypeVarValue.FuncArr:
  if (Var1.ValueType == TypeVarValue.Str)
    ((string[])Var.Obj)[i] = (string)Var1.Obj;
  else
    // выдать сообщение об ошибке "invalid Type
    // parameter"
    ErrorMess(225);
  break;
case TypeVarValue.Server: {
  switch (i) {
    case 0:
      if (Var1.ValueType == TypeVarValue.Str)
        Title = (string)Var1.Obj;
      else
        // выдать сообщение об ошибке "invalid Type
        // parameter"
        ErrorMess(225);
      break;
    case 1:
      if (Var1.ValueType == TypeVarValue.Str)
        Host = (string)Var1.Obj;
      else
        // выдать сообщение об ошибке "invalid Type
        // parameter"
        ErrorMess(225);
      break;
```

```
                        case 2:
                          if (Var1.ValueType == TypeVarValue.Str)
                            User = (string)Var1.Obj;
                          else
                            // выдать сообщение об ошибке "invalid Type
                            // parameter"
                            ErrorMess(225);
                          break;
                        case 3:
                          if (Var1.ValueType == TypeVarValue.Str)
                            Password = (string)Var1.Obj;
                          else
                            // выдать сообщение об ошибке "invalid Type
                            // parameter"
                            ErrorMess(225);
                          break;
                        case 4:
                          if (Var1.ValueType == TypeVarValue.Int)
                            IntervalWait = (int)Var1.Obj;
                          else
                            // выдать сообщение об ошибке "invalid Type
                            // parameter"
                            ErrorMess(225);
                          break;
                        case 5:
                          if (Var1.ValueType == TypeVarValue.Int)
                            WaitReboot = (int)Var1.Obj;
                          else
                            // выдать сообщение об ошибке "invalid Type
                            // parameter"
                            ErrorMess(225);
                          break;
                      }
                      break;
                    }
                  }
                }
                if (ValueVarType == TypeVarValue.Server) {
                  oServ = new ObjServ(Name, Title, Host,
                                      User, Password, IntervalWait,
                                      WaitReboot);
                  Var.Obj = oServ;
                }
              }
              else
                // выдать сообщение об ошибке
                // "invalid initial initialization for the given type of a
                // variable"
                ErrorMess(235);
            .)

 )] .
```

Тип переменной, определяется как вещественный (*real*), целый (*int*), строковый (*string*), булевский (*bool*), файловый (*file*) тип, серверный (*server*) тип, процедурный (proc, func) тип, COM (comobj) тип, а так же как одномерный массив, выше перечисленных типов. Исходный код функции определения типа переменной *Type()*, представлен ниже.

```
Type<out TypeVarValue ValueVarType, out int DimArr>
```

```
          (.
             ObjVar Var = new ObjVar();
             DimArr = 0;
             ValueVarType = TypeVarValue.None;
          .)

=
  real
          (. ValueVarType = TypeVarValue.Real; .)
  |
  int
          (. ValueVarType = TypeVarValue.Int; .)
  |
  bool
          (. ValueVarType = TypeVarValue.Bool; .)
  |
  string
          (. ValueVarType = TypeVarValue.Str; .)
  |
  file
          (. ValueVarType = TypeVarValue.File; .)
  |
  server
          (. ValueVarType = TypeVarValue.Server; .)
  |
  comobj
          (. ValueVarType = TypeVarValue.ComObj; .)
  |
  proc
          (. ValueVarType = TypeVarValue.Proc; .)
  |
  func
          (. ValueVarType = TypeVarValue.Func; .)
  |
  array [ Expr<out Var>

          (.
             if (Var.ValueType != TypeVarValue.Int)
               // выдать сообщение об ошибке "invalid type dimension array"
               ErrorMess(231);
          .)

  ] of Type<out ValueVarType, out DimArr>

          (.
             switch (ValueVarType) {
               case TypeVarValue.Real:
                 ValueVarType = TypeVarValue.RealArr;
                 break;
               case TypeVarValue.Int:
                 ValueVarType = TypeVarValue.IntArr;
                 break;
               case TypeVarValue.Str:
                 ValueVarType = TypeVarValue.StrArr;
                 break;
               case TypeVarValue.Bool:
                 ValueVarType = TypeVarValue.BoolArr;
                 break;
               case TypeVarValue.File:
                 ValueVarType = TypeVarValue.FileArr;
                 break;
               case TypeVarValue.Server:
```

```
                ValueVarType = TypeVarValue.ServerArr;
                break;
              case TypeVarValue.ComObj:
                ValueVarType = TypeVarValue.ComObjArr;
                break;
              case TypeVarValue.Proc:
                ValueVarType = TypeVarValue.ProcArr;
                break;
              case TypeVarValue.Func:
                ValueVarType = TypeVarValue.FuncArr;
                break;
              default:
                // выдать сообщение об ошибке "invalid Type"
                ErrorMess(209);
                break;
            }
            if (Var.Obj != null)
              DimArr = (int)Var.Obj;
            else
              DimArr = 0;
          .) .
```

Примеры объявление переменных:

```
var
  F : real = 3.14;
  N : int;
  FlagOff : bool = true;
  Str : string;
  TextFile : file;
  @Serv : server = ("Локальный", ".", "", "", 30, 180);
  FS : comobj = "Scripting.FileSystemObject";
  F1 : func = "Cos";
  P1 : proc;
```

Примеры объявление массивов:

```
var
  N : array 10 of int;
  FlagArr : array 9 of bool;
  StrArr : array 12 of string;
  RealArr : array 20 of real;
  FileArr : array 5 of file;
  ServArr : array 7 of server;
  COMArr : array 2 of comobj;
  ProcArr : array 9 of proc;
  FuncArr : array 11 of func;
```

Пример инициализации массивов:

```
var
  RealArray : array 4 of real = (1.2, 2.3, 5.7, 9.56);
  IntArray : array 5 of int = (10, 20, 30, 40, 50);
  BoolArray : array 3 of bool = (true, false, true);
  StrArray : array 4 of string = ("один", "два", "три", "четыре");
  COMArray : array 2 of comobj = ("Scripting.FileSystemObject", "WScript.Shell");
  FuncArray : array 2 of func = ("Sin", "Cos");
  ProcArray : array 2 of proc = ("Sin", "Cos");
```

Исходный код функции объявления сервера *ServDeclare()* представлен ниже.

```
ServDeclare<out ObjServ oServ, out string Name>

          (.
             string NameField = "";
             string Value = "";
             string Title = "";
             string Host = "";
             string User = "";
             string Password = "";
             int IntervalWait = 0;
             int WaitReboot = 0;
          .)

=
  declare "@" ServName<out Name> ":" server
    Ident<out NameField>

          (.
             if (NameField != "Title")
               // выдать сообщение об ошибке "Title expected"
               ErrorMess(239);
          .)

      "=" StringChar<out Value> ";"

          (.
             Title = Value;
          .)

    Ident<out NameField>

          (.
             if (NameField != "Host")
               // выдать сообщение об ошибке "Host expected"
               ErrorMess(240);
          .)

      "=" StringChar<out Value> ";"

          (.
             Host = Value;
          .)

    Ident<out NameField>

          (.
             if (NameField != "User")
               // выдать сообщение об ошибке "User expected"
               ErrorMess(241);
          .)

      "=" StringChar<out Value> ";"

          (.
             User = Value;
          .)

    Ident<out NameField>

          (.
             if (NameField != "Password")
               // выдать сообщение об ошибке "Password expected"
```

```
                ErrorMess(242);
            .)

        "=" StringChar<out Value> ";"

            (.
               Password = Value;
            .)

    Ident<out NameField>

            (.
               if (NameField != "IntervalWait")
                 // выдать сообщение об ошибке "IntervalWait expected"
                 ErrorMess(243);
            .)

        "=" intCon

            (.
               IntervalWait = Convert.ToInt32(t.val);
            .)

        ";"
    Ident<out NameField>

            (.
               if (NameField != "WaitReboot")
                 // выдать сообщение об ошибке "WaitReboot expected"
                 ErrorMess(244);
            .)

        "=" intCon

            (.
               WaitReboot = Convert.ToInt32(t.val);
               oServ = new ObjServ(Name, Title, Host,
                                   User, Password, IntervalWait,
                                   WaitReboot);
            .)

        ";"
  end ";" .
```

Примеры декларации серверов:

```
declare @CUMR04_TO : server
  Title = "Сервер телеобработки";
  Host = "10.23.248.15";
  User = "adm";
  Password = "pass";
  IntervalWait = 60;
  WaitReboot = 360;
end;

declare @CUMR04A : server
  Title = "Основной сервер приложений";
  Host = "10.23.248.11";
  User = "adm";
  Password = "pass";
  IntervalWait = 60;
  WaitReboot = 360;
end;
```

Инициализация таблицы процедур и функций

В любом языке программирования есть стандартные процедуры и функции, которые расширяют функциональность языка при работе с различными данными – числами, строками, файлами и т.д. Язык *Alfa* не является исключением. Для того чтобы выполнить стандартную процедуру или функцию интерпретатору нужно знать не только имя, но и количество и тип передаваемых параметров, а также, если это функция - тип возвращаемого результата. Занесение в объект *FuncTable* информации о стандартных процедурах и функциях, выполняется с помощью функции *IniFuncTalle()* . Функция описана в *Alfa.atg*. Часть исходного кода приведена ниже.

```
 //*************************************************************
//* Function: IniFuncTable
//* Notes: Инициализация таблицы функций и процедур
//*

private void IniFuncTable (ref TableObj<ObjFunc> FuncTable) {
  FuncTable.Add("Len", new ObjFunc(0, "Len", TypeVar.Func, 1, "var",
                TypeVarValue.Int));
  FuncTable.Add("Clear", new ObjFunc(1, "Clear", TypeVar.Proc, 1, "array",
                TypeVarValue.None));
  FuncTable.Add("Reverse", new ObjFunc(2, "Reverse", TypeVar.Proc, 1,
                "array", TypeVarValue.None));
  FuncTable.Add("Sort", new ObjFunc(3, "Sort", TypeVar.Proc, 1, "array",
                TypeVarValue.None));
  FuncTable.Add("BinarySearch", new ObjFunc(4, "BinarySearch",
                TypeVar.Func, 2, "array,var", TypeVarValue.Int));
  FuncTable.Add("IndexOf", new ObjFunc(5, "IndexOf", TypeVar.Func, 2,
                "array,var", TypeVarValue.Int));
  FuncTable.Add("CopyToArray", new ObjFunc(6, "CopyToArray", TypeVar.Proc,
                2, "array,array", TypeVarValue.None));
  FuncTable.Add("CopyOfRange", new ObjFunc(7, "CopyOfRange", TypeVar.Proc,
                5, "array,int,array,int,int", TypeVarValue.None));
  FuncTable.Add("ApostStr", new ObjFunc(8, "ApostStr", TypeVar.Func, 1,
                "string", TypeVarValue.Str));
  FuncTable.Add("StrToInt", new ObjFunc(9, "StrToInt", TypeVar.Func, 1,
                "string", TypeVarValue.Int));
  FuncTable.Add("IntToStr", new ObjFunc(10, "IntToStr", TypeVar.Func, 1,
                "int", TypeVarValue.Str));
  FuncTable.Add("Length", new ObjFunc(11, "Length", TypeVar.Func, 1,
                "string", TypeVarValue.Int));
    …
    …
}

//*** End of IniFuncTable ***********
```

В функции *IniFuncTable()*, вызывается метод *Add()* объекта таблицы *FuncTable* с передачей двух параметров, имя процедуры или функции и экземпляр объекта типа *ObjFunc*. Экземпляр объекта создается с помощью перегруженного конструктора класса *ObjFunc* и передачей шести параметров (ID, имя, тип, число входных параметров, строка типов входных параметров, тип возвращаемого значения). Код конструктора приведен ниже.

```
  // конструктор класса
  public ObjFunc (int ID, string FuncName, TypeVar FuncType,
                  int FuncNumPar, string FuncTypeInParam,
                  TypeVarValue FuncTypeReturn) {
    this.ID = ID;
    this.Name = FuncName;
    this.Type = FuncType;
```

```
    this.NumPar = FuncNumPar;
    this.TypeInParam = FuncTypeInParam;
    this.TypeReturn = FuncTypeReturn;
    this.FExternal = true;
    this.LabelFunc = new ObjLabel();
  }
```

Предварительный просмотр исходного текста в интерпретаторе

При выполнении программы важно знать местоположение объявления каждой процедуры или функции определенной в программе, чтобы обеспечить их более быстрый вызов. Если этот шаг не выполнить, то для каждого вызова процедуры или функции потребуется дополнительный поиск в исходном коде программы. Функция, выполняющая предварительный просмотр программы называется *RetrievalLabel()* описание ее находится в *Alfa.atg*. Далее приведен ее код.

```
//*************************************************************
//* Function: RetrievalLabel
//* Notes: Поиск процедур и функций, запись информации в таблицу
//*

private void RetrievalLabel () {
  string NameProc = "";
  TypeVar Type;
  ObjLabel PosBeg;
  ObjLabel LabelProc;
  ObjFunc Proc;

  PosBeg = SavePosProg();
  while (la.kind != _EOF) {
    SkipScript();
    if (la.kind == _function) {
      LabelProc = SavePosProg();
      Type = TypeVar.Func;
      Get();
      if (la.kind == _ident)
        NameProc = la.val;
      else {
        // выдать сообщение об ошибке "invalid Name function or procedure"
        ErrorMess(216);
      }
      Get();
      // ищем конец функции
      while (la.kind != _EOF) {
        SkipScript();
        if (la.kind == _end) {
          Get();
          if (la.val == ";") {
            Get();
            break;
          }
          else
            // выдать сообщение об ошибке "invalid definition funchion"
            ErrorMess(220);
        }
        Get();
      }
      if (la.kind == _EOF)
        // выдать сообщение об ошибке "invalid definition funchion"
        ErrorMess(220);
    }
    else if (la.kind == _procedure) {
```

```
        LabelProc = SavePosProg();
        Type = TypeVar.Proc;
        Get();
        if (la.kind == _ident)
          NameProc = la.val;
        else
          // выдать сообщение об ошибке "invalid Name function or procedure"
          ErrorMess(216);
        Get();
        // ищем конец процедуры
        while (la.kind != _EOF) {
          SkipScript();
          if (la.kind == _end) {
            Get();
            if (la.val == ";") {
              Get();
              break;
            }
            else
              // выдать сообщение об ошибке "invalid definition procedure"
              ErrorMess(220);
          }
          Get();
        }
        if (la.kind == _EOF)
          // выдать сообщение об ошибке "invalid definition procedure"
          ErrorMess(220);
      }
      else {
        Get();
        continue;
      }
      // если не существует в таблице, то записать
      if (!FuncTable.IsName(NameProc)) {
        Proc = new ObjFunc(NameProc, Type, LabelProc);
        FuncTable[NameProc] = Proc;
        if (la.kind == _begin)
          break;
        else
          continue;
      }
      else {
        if (Type == TypeVar.Func)
          // выдать сообщение об ошибке
          // "Function with such name already exists"
          ErrorMess(245);
        else
          // выдать сообщение об ошибке
          // "Procedure with such name already exists"
          ErrorMess(246);
      }
    }
    if (la.kind == _EOF) {
      RestorPosProg(PosBeg);
      Get();
    }
  }

  //*** End of RetrievalLabel ***********
```

Функция *RetrievalLabel()* работает следующим образом. Запоминаем текущее положение сканера. В цикле, пока не конец файла исходного текста программы, считываем код признака следующего

токена. Если это функция, запоминаем текущую позицию, устанавливаем тип. Считываем следующий токен. Следующий токен должен быть именем функции, если это не так, выдаем сообщение об ошибке и выходим из программы. Запоминаем имя найденной функции. Далее в цикле ищем конец функции т.е. “*end;*”. Если обнаружен конец файла, выдаем сообщение об ошибке и выходим из программы. Проверяем существует ли, имя функции в таблице *FuncTable*, если нет, то заносим информацию в таблицу. Если процедура с таким именем уже существует в таблице *FuncTable*, выдаем сообщение об ошибке и выходим из программы. Проверяем код признака следующего токена. Если он равен *_begin* (обнаружена опреаторная скобка начала программы), то выходим из функции, иначе на начало цикла определения объявлений функций и процедур. И наконец, если определения функции нет, и обнаружен конец файла, восстанавлеваем состояние сканера и выходим из функции *RetrievalLabel()*. Для процедуры выполняются точно такие же действия. Текущая позиция сканера, процедуры или функции сохраняются в переменных имеющих тип *ObjLabel*. *ObjLabel* представляет собой структуру описание которой приведено ниже.

```
// объект метки
public struct ObjLabel {
  public int ch;
  public int pos;
  public int col;
  public int line;
  public int oldEols;

  // конструктор класса
  public ObjLabel (int Ch, int Pos, int Col, int Line, int OldEols) {
    this.ch = Ch;
    this.pos = Pos;
    this.col = Col;
    this.line = Line;
    this.oldEols = OldEols;
  }
}
```

Данные о объявленной процедуре или функции помещаются в переменную *Proc* типа *ObjFunc*. *ObjFunc* представляет собой структуру, часть описания которой приведено далее.

```
// объект спецификации процедуры или функции
public struct ObjFunc {
  // идентификатор процедуры или функции
  public int ID;
  // имя процедуры или функции
  public string Name;
  // тип (процедура или функция)
  public TypeVar Type;
  // количество входных параметров
  public int NumPar;
  // строка типов входных параметров
  public string TypeInParam;
  // тип возвращаемого результата
  public TypeVarValue TypeReturn;
  // флаг вызова внешних процедур и функций
  public bool FExternal;
  // метка процедуры или фунции
  public ObjLabel LabelFunc;

  // конструктор класса
```

```
public ObjFunc (int ID, string FuncName, TypeVar FuncType,
                int FuncNumPar, string FuncTypeInParam,
                TypeVarValue FuncTypeReturn) {
  this.ID = ID;
  this.Name = FuncName;
  this.Type = FuncType;
  this.NumPar = FuncNumPar;
  this.TypeInParam = FuncTypeInParam;
  this.TypeReturn = FuncTypeReturn;
  this.FExternal = true;
  this.LabelFunc = new ObjLabel();
}

public ObjFunc (string FuncName, TypeVar FuncType, ObjLabel Label) {
  this.ID = 0;
  this.Name = FuncName;
  this.Type = FuncType;
  this.NumPar = 0;
  this.TypeInParam = "";
  this.TypeReturn = TypeVarValue.None;
  this.LabelFunc = Label;
  this.FExternal = false;
}

// вернуть тип указанного входного параметра
public TypeVarValue TypeParam (int IndexParam) {
  string[] StrType;
  StrType = TypeInParam.Split(',');
  if (IndexParam <= NumPar - 1) {
    switch (StrType[IndexParam]) {
      case "real":
        return TypeVarValue.Real;
      case "int":
        return TypeVarValue.Int;
      case "string":
        return TypeVarValue.Str;
      case "bool":
        return TypeVarValue.Bool;
      case "real_array":
        return TypeVarValue.RealArr;
      case "int_array":
        return TypeVarValue.IntArr;
      case "string_array":
        return TypeVarValue.StrArr;
      case "bool_array":
        return TypeVarValue.BoolArr;
      case "file_array":
        return TypeVarValue.FileArr;
      case "server_array":
        return TypeVarValue.ServerArr;
      case "comobj_array":
        return TypeVarValue.ComObjArr;
      case "proc_array":
        return TypeVarValue.ProcArr;
      case "func_array":
        return TypeVarValue.FuncArr;
      case "array":
        return TypeVarValue.Array;
      case "file":
        return TypeVarValue.File;
      case "server":
        return TypeVarValue.Server;
```

```
            case "comobj":
              return TypeVarValue.ComObj;
            case "var":
              return TypeVarValue.Var;
            default:
              return TypeVarValue.None;
          }
        }
        else
          return TypeVarValue.None;
      }
    }
```

Переменная *Proc* сохраняется в объекте *FuncTable* обощенного класса *TableObj*.

Выражения и операции

Выражение обозначает правило вычислений, дающих при выполнении выражения некоторое значение. Значение выражения зависит от значений констант, переменных выражения и от операций и функций, включенных в выражение.

Синтаксис выражения и операций приведен ниже.

```
Expr<out ObjVar Var> =
  SimpleExpr<out Var> [ RelatOper<out TypeOper> SimpleExpr<out Var1> ] .

SimpleExpr<out ObjVar Var> =
  [ Sign<out TypeOper> ] Term<out Var> { AddOper<out TypeOper>
  Term<out Var1> } .

Term<out ObjVar Var> =
  Factor<out Var> { MulOper<out TypeOper> Factor<out Var1> } .

Factor<out ObjVar Var> =
  ConstUnSigned<out Var>
  |
  Variable<ref NameVar> [ ListParam<ref ListPar> ]
  |
  not Factor<out Var>
  |
  "(" Expr<out Var> ")" .

RelatOper<out int TypeOper> = "=" | "<>" | "<" | "<=" | ">" | ">=" .

AddOper<out int TypeOper> = "+" | "-" | or | xor .

MulOper<out int TypeOper> = "*" | "/" | and | shl | shr .

ConstUnSigned<out ObjVar Var> =
  RealDef<out RealValue>
  |
  IntDef<out IntValue>
  |
  StringChar<out Value>
  |
  StringScript<out Value>
  |
  true
  |
  false
  |
```

```
        nil .
```

Любая мультипликативная операция *MulOper* в терме *Term* имеет два операнда: один – та часть терма, которая предшествует операции, второй – фактор *Factor*, идущий сразу же за операцией. Аддитивная операция *AddOper* в простом выражении *SimpleExpr* имеет также два операнда: один та часть простого выражения, которая предшествует операции, второй терм, идущий сразу же за операцией. Два операнда любой операции отношения *RelatOper* – простые выражения *SimpleExpr* предшествующие и следующие за самой операцией. Операндом для (одиночного) знака является простое выражение из терма, непосредственно следующего за знаком. Операндом для *not* в некотором факторе является фактор, идущий следом за *not*. Тип любого фактора выводится на основании типов, его составляющих (т.е. переменных или функций). Вычисление фактора, содержащего переменную, задает обращение к этой переменной и обозначает ее значение, если переменная не определена, то это ошибка. Вычисление фактора, содержащего обозначение функции, задает активацию функции (вызов), указанной именем функции. В качестве её формальных параметров подставляются соответствующие фактические параметры. После выполнения фактор возвращает значение результата, если результат не определен, то это ошибка.

Правила задают приоритеты операций в соответствии с их разбиением на четыре класса. Наивысший приоритет – у операции not, затем идут мультипликативные операции *MulOper*, следом – аддитивные операции *AddOper* и, наконец, с наименьшим приоритетом – операции отношения *RelatOper*. Последовательности операций с одинаковым приоритетом выполняются слева направо. Система приоритетов отражена в синтаксисе операций приведенном выше.

Исходный текст вычисления выражения *Expr* представлен ниже.

```
Expr<out ObjVar Var>

          (.
             ObjVar Var1 = new ObjVar();
             int TypeOper = 0;
          .)

=
  SimpleExpr<out Var> [ RelatOper<out TypeOper> SimpleExpr<out Var1>

          (.
             // проверяем типы значений переменных
             if (Var.ValueType != Var1.ValueType) {
               // выдаем сообщение об ошибке "invalid Type" и выходим
               ErrorMess(209);
             }
             // устанавливаем, что результатом является булевское значение
             // для операции отношения
             Var.Name = "";
             Var.VarType = TypeVar.None;
             Var.ValueType = TypeVarValue.Bool;
             if (Var1.ValueType == TypeVarValue.Real) {
               switch (TypeOper) {
                 case _EQ:
                   Var.Obj = ((double)Var.Obj == (double)Var1.Obj);
                   break;
                 case _LT:
                   Var.Obj = ((double)Var.Obj < (double)Var1.Obj);
                   break;
                 case _LE:
```

```
        Var.Obj = ((double)Var.Obj <= (double)Var1.Obj);
        break;
      case _NE:
        Var.Obj = ((double)Var.Obj != (double)Var1.Obj);
        break;
      case _GT:
        Var.Obj = ((double)Var.Obj > (double)Var1.Obj);
        break;
      case _GE:
        Var.Obj = ((double)Var.Obj >= (double)Var1.Obj);
        break;
    }
  }
  else if (Var1.ValueType == TypeVarValue.Int) {
    switch (TypeOper) {
      case _EQ:
        Var.Obj = ((int)Var.Obj == (int)Var1.Obj);
        break;
      case _LT:
        Var.Obj = ((int)Var.Obj < (int)Var1.Obj);
        break;
      case _LE:
        Var.Obj = ((int)Var.Obj <= (int)Var1.Obj);
        break;
      case _NE:
        Var.Obj = ((int)Var.Obj != (int)Var1.Obj);
        break;
      case _GT:
        Var.Obj = ((int)Var.Obj > (int)Var1.Obj);
        break;
      case _GE:
        Var.Obj = ((int)Var.Obj >= (int)Var1.Obj);
        break;
    }
  }
  else if ((Var1.ValueType == TypeVarValue.Str) ||
           (Var1.ValueType == TypeVarValue.Func) ||
           (Var1.ValueType == TypeVarValue.Proc)) {
    switch (TypeOper) {
      case _EQ:
        Var.Obj = ((string)Var.Obj == (string)Var1.Obj);
        break;
      case _NE:
        Var.Obj = ((string)Var.Obj != (string)Var1.Obj);
        break;
      default:
        // выдаем сообщение об ошибке "invalid RelatOper" и выходим
        ErrorMess(210);
        break;
    }
  }
  else if (Var1.ValueType == TypeVarValue.Bool) {
    switch (TypeOper) {
      case _EQ:
        Var.Obj = ((bool)Var.Obj == (bool)Var1.Obj);
        break;
      case _NE:
        Var.Obj = ((bool)Var.Obj != (bool)Var1.Obj);
        break;
      default:
        // выдаем сообщение об ошибке "invalid RelatOper" и выходим
        ErrorMess(210);
```

```
                    break;
                }
              }
              else {
                // выдаем сообщение об ошибке "invalid Type" и выходим
                ErrorMess(209);
              }
            .)

  ] .
```

Рассмотрим работу функции *Expr()*. Создаем и инициализируем локальные переменные. Вычисляем значение левого операнда, вызывая функцию *SimpleExpr()*, результат помещаем в переменную *Var1*. Если присутствует операция отношения, получаем код операции, вызывая функцию *RelatOper()*. Вычисляем значение правого операнда, результат помещаем в переменную *Var*. Сравниваем типы значений левого и правого операндов, если они не совпадают, выдаем сообщение об ошибке и выходим из программы. Помещаем в переменную результата вычисления операции отношения *Var* - значение типа переменной *TypeVarValue.Bool* (т.е. устанавливаем, что результатом является булевское значение). Далее в зависимости от типа значения операнда и от кода операции отношения производим необходимую операцию отношения, результат помещаем в переменную *Var*. Если заданная операция проводится над недопустимым типом операнда, выдаем сообщение об ошибке.

Примеры выражений:

```
X = 15

P <= Q

(I < J) = (J < K)
```

Исходный код вычисления простого выражения *SimpleExpr* представлен ниже.

```
SimpleExpr<out ObjVar Var>

          (.
             int TypeOper = 0;
             ObjVar Var1 = new ObjVar();
          .)

=
  [ Sign<out TypeOper> ] Term<out Var>

          (.
             if (TypeOper == _MINUS) {
               if (Var.ValueType == TypeVarValue.Real)
                 Var.Obj = -((double)Var.Obj);
               else if (Var.ValueType == TypeVarValue.Int)
                 Var.Obj = -((int)Var.Obj);
               else {
                 // выдаем сообщение об ошибке "invalid Type" и выходим
                 ErrorMess(209);
               }
             }
             TypeOper = 0;
          .)

  { AddOper<out TypeOper> Term<out Var1>
```

```
        (.
           if (Var.ValueType != Var1.ValueType) {
             // выдаем сообщение об ошибке "invalid Type" и выходим
             ErrorMess(209);
             break;
           }
           if (Var.ValueType == TypeVarValue.Real) {
             switch (TypeOper) {
               case _PLUS:
                 Var.Obj = (double)Var.Obj + (double)Var1.Obj;
                 break;
               case _MINUS:
                 Var.Obj = (double)Var.Obj - (double)Var1.Obj;
                 break;
             }
           }
           else if (Var.ValueType == TypeVarValue.Int) {
             switch (TypeOper) {
               case _PLUS:
                 Var.Obj = (int)Var.Obj + (int)Var1.Obj;
                 break;
               case _MINUS:
                 Var.Obj = (int)Var.Obj - (int)Var1.Obj;
                 break;
               case _or:
                 Var.Obj = (int)Var.Obj | (int)Var1.Obj;
                 break;
               case _xor:
                 Var.Obj = (int)Var.Obj ^ (int)Var1.Obj;
                 break;
             }
           }
           else if (Var.ValueType == TypeVarValue.Str) {
             if (TypeOper == _PLUS)
               Var.Obj = (string)Var.Obj + (string)Var1.Obj;
             else {
               // выдаем сообщение об ошибке "invalid Type" и выходим
               ErrorMess(209);
               break;
             }
           }
           else if (Var.ValueType == TypeVarValue.Bool) {
             if (TypeOper == _xor)
               Var.Obj = (bool)Var.Obj ^ (bool)Var1.Obj;
             else if (TypeOper == _or)
               Var.Obj = (bool)Var.Obj || (bool)Var1.Obj;
             else {
               // выдаем сообщение об ошибке "invalid Type" и выходим
               ErrorMess(209);
               break;
             }
           }
        .)

  } .
```

Функция *SimpleExpr()* работает следующим образом. Создаем и инициализируем локальные переменные. Если левому операнду, предшествует знак, вызываем функцию *Sign()* для получения кода знака. Вычисляем значение левого операнда, вызывая функцию *Term()*, результат помещаем в переменную *Var*. Сохраняем полученное значение в переменной Var1. Если знак перед

операндом минус и тип переменной числовой, переменной присваиваем отрицательное значение. В цикле пока значение текущей лексемы равно *“or”*, *“+”* (плюс) или *“–“* (минус) получаем код аддитивной операции, вызывая функцию *AddOper()*. Вычисляем значение правого операнда, результат помещаем в переменную *Var*. Сравниваем типы значений левого и правого операндов, если они не совпадают, выдаем сообщение об ошибке и выходим из программы. Далее в зависимости от типа значения операнда и от кода операции отношения производим необходимую аддитивную операцию, результат помещаем в переменную *Var*. Если заданная операция проводится над недопустимым типом операнда, выдаем сообщение об ошибке.

Примеры простых выражений:

```
X + Y

-X

P or Q

I*J + I
```

Исходный текст вычисления терма *Term* представлен ниже.

```
Term<out ObjVar Var>

          (.
             ObjVar Var1 = new ObjVar();
             int TypeOper = 0;
          .)

=
  Factor<out Var> { MulOper<out TypeOper> Factor<out Var1>

          (.
             if (Var.ValueType != Var1.ValueType) {
               // выдаем сообщение об ошибке "invalid Type" и выходим
               ErrorMess(209);
               break;
             }
             if (Var.ValueType == TypeVarValue.Real) {
               switch (TypeOper) {
                 case _DIV:
                   try {
                     Var.Obj = (double)Var.Obj / (double)Var1.Obj;
                   }
                   catch (Exception e) {
                     ErrorMessStr(e.Message);
                   }
                   break;
                 case _MUL:
                   try {
                     Var.Obj = (double)Var.Obj * (double)Var1.Obj;
                   }
                   catch (Exception e) {
                     ErrorMessStr(e.Message);
                   }
                   break;
                 default:
                   // выдать сообщение об ошибке "invalid MulOper"
                   ErrorMess(211);
                   break;
```

```
            }
          }
          else if (Var.ValueType == TypeVarValue.Int) {
            switch (TypeOper) {
              case _DIV:
                if ((int)Var1.Obj == 0) {
                  // выдать сообщение об ошибке "dividing by zero"
                  ErrorMess(206);
                  Var.Obj = 0;
                }
                Var.Obj = (int)Var.Obj / (int)Var1.Obj;
                break;
              case _MUL:
                try {
                  Var.Obj = (int)Var.Obj * (int)Var1.Obj;
                }
                catch (Exception e) {
                  ErrorMessStr(e.Message);
                }
                break;
              case _and:
                Var.Obj = (int)Var.Obj & (int)Var1.Obj;
                break;
              case _shl:
                Var.Obj = (int)Var.Obj << (int)Var1.Obj;
                break;
              case _shr:
                Var.Obj = (int)Var.Obj >> (int)Var1.Obj;
                break;
              default:
                // выдать сообщение об ошибке "invalid MulOper"
                ErrorMess(211);
                break;
            }
          }
          else if ((Var.ValueType == TypeVarValue.Bool) &&
                   (TypeOper == _and)) {
            Var.Obj = (bool)Var.Obj && (bool)Var1.Obj;
          }
          else {
            // выдать сообщение об ошибке "invalid Type"
            ErrorMess(209);
            break;
          }
        .)

  } .
```

Функция *Term()* работает следующим образом. Создаем и инициализируем локальные переменные. Вычисляем значение левого операнда, вызывая функцию *Factor()*, результат помещаем в переменную *Var*. Сохраняем полученное значение в переменной *Var1*. В цикле пока значение текущей лексемы равно “*and*”, “*” (умножить) или “/” (разделить) получаем код мультипликативной операции, вызывая функцию *MulOper()*. Вычисляем значение правого операнда, результат помещаем в переменную *Var*. Сравниваем типы значений левого и правого операндов, если они не совпадают, выдаем сообщение об ошибке и выходим из программы. Далее в зависимости от типа значения операнда и от кода операции отношения производим необходимую мультипликативную операцию, результат помещаем в переменную *Var*. Если

заданная операция проводится над недопустимым типом операнда, выдаем сообщение об ошибке.

Примеры термов:

```
X * Y

I/(I-1)

Q and not P

(X <= Y) and (Y < W)
```

Исходный код вычисления фактора *Factor* представлен ниже.

```
Factor<out ObjVar Var>

          (.
             bool FlagParam = false;
             string Name = "";
             int Index = 0;
             TypeVarValue ValueType;
             DescrNameVar NameVar = new DescrNameVar();
             ArrayList ListPar = new ArrayList();
             Var = new ObjVar();
          .)

=
  ConstUnSigned<out Var>
  |
  Variable<ref NameVar> [ ListParam<ref ListPar>

          (.
             FlagParam = true;

          .)

    ]

          (.
             Name = NameVar.Name;
             // передано имя функции со списком параметров
             if (NameVar.Type == TypeNameVar.Var && FlagParam) {
               ValueType = TypeVarValue.Func;
             }
             else {
               // получить значение переменной
               if (!FlagLocal) {
                 if (!VarTable.IsName(Name))
                   // выдать сообщение об ошибке "invalid VarName"
                   ErrorMess(204);
                 Var = VarTable[Name];
               }
               else {
                 if (!VarLocalTable.IsName(Name))
                   // выдать сообщение об ошибке "invalid VarName"
                   ErrorMess(204);
                 Var = VarLocalTable[Name];
               }
               ValueType = Var.ValueType;
             }
```

```
switch (NameVar.Type) {
  case TypeNameVar.Var: {
    if (!FlagParam)
      break;
    switch (ValueType) {
      case TypeVarValue.Func: {
        ObjVar Var1;
        // осуществляем проверку, переменная по имени Name,
        // является ли процедурным типом
        if (!FuncTable.IsName(Name)) {
          if (!FlagLocal) {
            if (VarTable.IsName(Name)) {
              Var1 = VarTable[Name];
              if ((Var1.ValueType != TypeVarValue.Proc) &&
                  (Var1.ValueType != TypeVarValue.Func)) {
                // выдать сообщение об ошибке
                // "invalid Type"
                ErrorMess(209);
              }
              else {
                Name = (string)Var1.Obj;
                if (String.IsNullOrEmpty(Name))
                  // выдать сообщение об ошибке
                  // "value of a variable is not defined"
                  ErrorMess(247);
              }
            }
            else
              // выдать сообщение об ошибке "invalid Name
              // function or procedure"
              ErrorMess(216);
          }
          else {
            if (VarLocalTable.IsName(Name)) {
              Var1 = VarLocalTable[Name];
              if ((Var1.ValueType != TypeVarValue.Proc) &&
                  (Var1.ValueType != TypeVarValue.Func)) {
                // выдать сообщение об ошибке
                // "invalid Type"
                ErrorMess(209);
              }
              else {
                Name = (string)Var1.Obj;
                if (String.IsNullOrEmpty(Name))
                  // выдать сообщение об ошибке
                  // "value of a variable is not defined"
                  ErrorMess(247);
              }
            }
            else
              // выдать сообщение об ошибке "invalid Name
              // function or procedure"
              ErrorMess(216);
          }
          if (!FuncTable.IsName(Name))
            // выдать сообщение об ошибке "invalid Name function
            // or procedure"
            ErrorMess(216);
        }
        ObjFunc Func = (ObjFunc)FuncTable[Name];
        if (Func.FExternal)
          // выполнить указанную стандартную функцию
```

```
          Var = RunExtFunc(Name, ref FuncTable, ref ListPar);
        else {
          // сохранить информацию о точке возврата из
          // подпрограммы в стеке
          PushGoSub();
          // встать на выполняемую подпрограмму в исходном тексте
          RestorPosProg(Func.LabelFunc);
          Get();
          if (Func.Type == TypeVar.Func)
            // выполнить разбор и выполнение функции
            FuncDeclare (ref ListPar, ref Var);
          // встать на точку возврата
          PopGoSub();
          Get();
        }
        break;
      }
      case TypeVarValue.Proc: {
        // выдать сообщение об ошибке "invalid Type"
        ErrorMess(209);
        break;
      }
    }
    break;
  }
  case TypeNameVar.Array: {
    Index = NameVar.Index;
    if (Index < 0 || Index > Var.LenArr - 1)
      // выдать сообщение об ошибке "invalid index array"
      ErrorMess(229);
    // получить значение элемента массива
    Var = Var.ValueArray(Index);
    break;
  }
  case TypeNameVar.Field: {
    if (FlagParam)
      // выдать сообщение об ошибке
      // "property cannot transfer parametres"
      ErrorMess(248);
    switch (ValueType) {
      case TypeVarValue.ComObj: {
        if (Var.tp == null)
          // выдать сообщение об ошибке
          // "for performance it is required COM object"
          ErrorMess(249);
        try {
          Var = ComObj.Invoke(ref Var, NameVar, ref ListPar,
                              true);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
        }
        break;
      }
      case TypeVarValue.Server: {
        ObjServ oServ;
        if (Var.Obj == null)
          // выдать сообщение об ошибке
          // "value of a variable is not defined"
          ErrorMess(247);
        oServ = (ObjServ)Var.Obj;
        // присваиваем значение полю объекта
```

```
            switch (NameVar.NameMethod) {
              case "Host":
                Var.ValueType = TypeVarValue.Str;
                Var.Obj = (string)oServ.Host;
                break;
              case "User":
                Var.ValueType = TypeVarValue.Str;
                Var.Obj = (string)oServ.User;
                break;
              case "Password":
                Var.ValueType = TypeVarValue.Str;
                Var.Obj = (string)oServ.Password;
                break;
              case "IntervalWait":
                Var.ValueType = TypeVarValue.Int;
                Var.Obj = (int)oServ.IntervalWait;
                break;
              case "Title":
                Var.ValueType = TypeVarValue.Str;
                Var.Obj = (string)oServ.Title;
                break;
              case "WaitReboot":
                Var.ValueType = TypeVarValue.Int;
                Var.Obj = (int)oServ.WaitReboot;
                break;
              default:
                // выдать сообщение об ошибке "invalid FieldName"
                ErrorMess(232);
                break;
            }
            break;
          }
        }
        break;
      }
      case TypeNameVar.Method: {
        switch (ValueType) {
          case TypeVarValue.ComObj: {
            if (Var.tp == null)
              // выдать сообщение об ошибке
              // "for performance it is required COM object"
              ErrorMess(249);
            try {
              Var = ComObj.Invoke(ref Var, NameVar, ref ListPar,
                                  true);
            }
            catch (Exception e) {
              ErrorMessStr(e.Message);
            }
            break;
          }
          case TypeVarValue.Server: {
            Var = RunMethodServ(ref Var, NameVar.NameMethod,
                                ref ListPar);
            break;
          }
        }
        break;
      }
    }

.)
```

```
  |
  not Factor<out Var>

          (.
             if (Var.ValueType == TypeVarValue.Int)
               Var.Obj = ~((int)Var.Obj);
             else if (Var.ValueType == TypeVarValue.Bool)
               Var.Obj = !((bool)Var.Obj);
             else {
               // выдать сообщение об ошибке "invalid Type"
               ErrorMess(209);
             }
          .)

  |
  "(" Expr<out Var> ")" .
```

Фактор согласно синтаксиса может быть:

- числовым, строковым или булевским значением;
- значением функции;
- значением переменной;
- значением свойства или метода объектной переменной;
- значением операции отрицания (not) самого фактора;
- значением выражения помещенного в скобки.

Примеры факторов:

```
15
“Строка символов”
true
X
@ServName.User
not F
(X + Y + Z)
```

Исходный текст получения кода операции отношения *RelatOper* представлен ниже.

```
RelatOper<out int TypeOper>
          (. TypeOper = 0; .)
=
  "="
          (. TypeOper = _EQ; .)
  |
  "<>"
          (. TypeOper = _NE; .)
  |
  "<"
          (. TypeOper = _LT; .)
  |
  "<="
          (. TypeOper = _LE; .)
  |
```

```
  ">"
          (. TypeOper = _GT; .)
  |
  ">="
          (. TypeOper = _GE; .) .
```

Исходный текст получения кода аддитивной операции *AddOper* представлен ниже.

```
AddOper<out int TypeOper>
          (. TypeOper = 0; .)
=
  "+"
          (. TypeOper = _PLUS; .)
  |
  "-"
          (. TypeOper = _MINUS; .)
  |
  or
          (. TypeOper = _or; .)
  |
  xor
          (. TypeOper = _xor; .) .
```

Исходный текст получения кода мультипликативной операции *MulOper* представлен ниже.

```
MulOper<out int TypeOper>
          (. TypeOper = 0; .)
=
  "*"
          (. TypeOper = _MUL; .)
  |
  "/"
          (. TypeOper = _DIV; .)
  |
  and
          (. TypeOper = _and; .)

  |
  shl
          (. TypeOper = _shl; .)
  |
  shr
          (. TypeOper = _shr; .) .
```

Операторы

Операторы обозначают алгоритмические действия, они выполняемые. Операторы объединяются в раздел операторов. Последовательность операторов, разделенных точкой с запятой, означает поочередное выполнение действий, заданных составляющими операторами.

Синтаксис операторов следующий:

```
Operations = [ Operation { Operation } ] .

Operations = [ Operation
          (.
             if (Oper == _exit)
               return;
          .)
              { Operation
          (.
```

```
            if (Oper == _exit)
              return;
        .)
            } ] .

Operation = Assign | If | Switch | For | While | Loop | ReadLn | Write |
            WriteLn | Exit .
```

Оператор присваивания

Оператор присваивания заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть совместимым по присваиванию с переменной.

Синтаксис оператора следующий:

```
Assign =
  Variable<ref NameVar> ( ":=" Expr<out Var> | ListParam<ref ListPar> )
  ";" .

Variable<ref DescrNameVar NameVar> =
 ["@"] Ident<out Name> [( "."  Ident <out NameField> |
   "->" Ident <out NameMethod> | "[" Expr <out Var> "]" )] .
```

Оператор выполняет следующие действия:

- Присваивает значение свойству объектной серверной или COM переменной;

```
@Serv.Password := "123";
Sel.Size := 15;
```

- Осуществляет вызов методов объектной серверной или COM переменной;

```
@Serv->PingStatus();
Sel->TypeText("Привет");
```

- Присваивает переменные;

```
@Serv1 := @Serv2;
Sel1 := Sel2;
```

- Присваивает переменным значения;

```
var
  A : int;
  B : int;

A := 5;
B := A;
```

- Присваивает переменной значение функции или значение метода объектных переменных;

```
S := IntToStr(5);
Flag := @Serv->Connect();
```

- Присваивает объектным переменным или массивам объектных переменных значение *nil*

```
var
  @UAF : array 2 of server = (
    CreateServObj("Сервер приложений 1", "10.33.44.125",
                  "adm1", "pass1", 60, 320),
    CreateServObj("Сервер приложений 2", "10.33.44.126",
                  "adm2", "pass2", 60, 320)
  );
  @A : server;
  @B : server;

begin
  @B := nil;
  @B := @A;
  @UAF := nil;
  writeln("Переменная @UAF имеет значение: ", not IsNull(@UAF));
end;
```

- Присваивает значения элементам массива;

```
var
  Arr : array 20 of int;

Arr[1] := 12;
```

- Осуществляет вызов процедур или функций.

```
MyProc();
MyFunc(A, B, D);
```

В исходном тексте программы переменная может быть представлена в следующем виде:

- переменная - *A*
  (переменная может быть любой: целое, массив, COM переменная и т.д., кроме серверной переменной, серверная переменная должна начинаться с символа ‘@’)

- серверная переменная - *@A*

- свойство переменной - *A.B*
  (свойство COM переменной)
  @A.B
  (свойство серверной переменной)

- метод переменной - *A->B()*
  (метод COM переменной)
  @A->B()
  (метод серверной переменной)

- элемент массива - *A[B]*

- процедура или функция – *A()*

С помощью функции *Variable()*, определяются основные атрибуты переменной в соответствии с синтаксисом, которые заносятся в структуру типа *DescrNameVar* и потом используются в функции *Assign()*. Исходный текст функции *Variable()* приведен ниже.

```
// определение типа имени переменной
public enum TypeNameVar {
  // неопределено
  None,
  // переменная
  Var,
  // элемент массива
  Array,
  // свойство, поле - объекта
  Field,
  // метод объекта
  Method
};

// описатель имени переменной
public struct DescrNameVar {
  // имя переменной
  public string Name;
  // имя метода (свойства)
  public string NameMethod;
  // значение индекса массива
  public int Index;
  // тип имени переменной
  public TypeNameVar Type;
}

Variable<ref DescrNameVar NameVar>

          (.
             bool FlagIndex = false;
             bool Flag = false;
             string Name = "";
             string NameField = "";
             string NameMethod = "";
             ObjVar Var = new ObjVar();
          .)
=
 ["@"]
          (.
              if (t.val == "@")
                Flag = true;
          .)

 Ident<out Name>

          (.
             if (Flag)
               Name = "@" + Name;
          .)
```

```
 [( "."  Ident <out NameField> |
    "->" Ident <out NameMethod> |
    "[" Expr <out Var> "]"

          (.
             if (Var.ValueType != TypeVarValue.Int)
               // выдать сообщение "invalid index array"
               ErrorMess(229);
             if ((int)Var.Obj < 0)
               // выдать сообщение "invalid index array"
               ErrorMess(229);
             NameVar.Index = (int)Var.Obj;
             FlagIndex = true;
          .)
 )]
          (.
             NameVar.Name = Name;
             if (FlagIndex) {
               NameVar.Type = TypeNameVar.Array;
             }
             else if (NameField != "") {
               NameVar.NameMethod = NameField;
               NameVar.Type = TypeNameVar.Field;
             }
             else if (NameMethod != "") {
               NameVar.NameMethod = NameMethod;
               NameVar.Type = TypeNameVar.Method;
             }
             else {
               NameVar.Type = TypeNameVar.Var;
             }

          .) .
```

Функция *Variable()* достаточно проста. При вызове функции передается один параметр, ссылка на структуру типа *DescrNameVar*. Определяется имя переменной. Далее, в зависимости от следующего за именем переменной, токена (который может быть - “.”, “->”, “[”), считывается следующая лексема, которая является соответственно: именем поля (свойства), именем метода или значением индекса массива. Полученные данные записываются в соответствующие поля структуры *NameVar* типа *DescrNameVar*.

Исходный код функции *Assign()* приведен ниже.

```
Assign

          (.
             string Name = "";
             string NameField = "";
             int Index = 0;
             bool ElemArr = false;
             ObjVar Var = new ObjVar();
             ObjVar Var1 = new ObjVar();
             DescrNameVar NameVar = new DescrNameVar();
          .)

=
```

```
Variable<ref NameVar> (

        (.
           // выделить индекс элемента массива
           if (NameVar.Type == TypeNameVar.Array) {
             // установить флаг
             ElemArr = true;
             // индекс элемента массива
             Index = NameVar.Index;
           }
           // имя переменной
           Name = NameVar.Name;
           if (!FlagLocal) {
             if (VarTable.IsName(Name)) {
               Var1 = VarTable[Name];
               if (Var1.VarType == TypeVar.Const)
                 // выдать сообщение об ошибке
                 // "invalid assigment statement, requirement variable"
                 ErrorMess(223);
             }
             else
               // выдать сообщение об ошибке "invalid Var"
               ErrorMess(201);
           }
           else {
             if (VarLocalTable.IsName(Name)) {
               Var1 = VarLocalTable[Name];
               if (Var1.VarType == TypeVar.Const)
                 // выдать сообщение об ошибке
                 // "invalid assigment statement, requirement variable"
                 ErrorMess(223);
             }
             else
               // выдать сообщение об ошибке "invalid Var"
               ErrorMess(201);
           }
        .)

                    ":=" Expr<out Var>

        (.
           if (Var1.ValueType == TypeVarValue.Server ||
               Var1.ValueType == TypeVarValue.ServerArr) {
             if (Name[0] != '@')
               // выдать сообщение "invalid ServName"
               ErrorMess(202);
           }
           switch (NameVar.Type) {
             case TypeNameVar.Var:
             case TypeNameVar.Array:
               if (!Var1.CompareType(Var.ValueType)) {
                 // выдать сообщение об ошибке "invalid Type"
                 ErrorMess(209);
               }
               if (ElemArr) {
                 if (Index < 0 || Index > Var1.LenArr - 1)
                   // выдать сообщение об ошибке "invalid index array"
                   ErrorMess(229);
               }
               switch (Var.ValueType) {
                 case TypeVarValue.Real:
                   if (ElemArr)
```

```
      Var1.SetValueArr(Index, (double)Var.Obj);
    else
      Var1.Obj = (double)Var.Obj;
    break;
  case TypeVarValue.Int:
    if (ElemArr) {
      if (Var1.ValueType == TypeVarValue.RealArr)
        Var1.SetValueArr(Index,
                         Convert.ToDouble((int)Var.Obj));
      else
        Var1.SetValueArr(Index, (int)Var.Obj);
    }
    else {
      if (Var1.ValueType == TypeVarValue.Real)
        Var1.Obj = Convert.ToDouble((int)Var.Obj);
      else
        Var1.Obj = (int)Var.Obj;
    }
    break;
  case TypeVarValue.Str:
    if (ElemArr)
      Var1.SetValueArr(Index, (string)Var.Obj);
    else
      Var1.Obj = (string)Var.Obj;
    break;
  case TypeVarValue.Bool:
    if (ElemArr)
      Var1.SetValueArr(Index, (bool)Var.Obj);
    else
      Var1.Obj = (bool)Var.Obj;
    break;
  case TypeVarValue.RealArr:
    ControlDimArr(Var1, Var);
    Var1.Obj = (double[])Var.Obj;
    break;
  case TypeVarValue.IntArr:
    ControlDimArr(Var1, Var);
    Var1.Obj = (int[])Var.Obj;
    break;
  case TypeVarValue.StrArr:
    ControlDimArr(Var1, Var);
    Var1.Obj = (string[])Var.Obj;
    break;
  case TypeVarValue.BoolArr:
    ControlDimArr(Var1, Var);
    Var1.Obj = (bool[])Var.Obj;
    break;
  case TypeVarValue.ComObj:
    Var1.Obj = Var.Obj;
    Var1.tp = Var.tp;
    break;
  case TypeVarValue.File:
    if (ElemArr)
      Var1.SetValueArr(Index, (ObjFile)Var.Obj);
    else
      Var1.Obj = (ObjFile)Var.Obj;
    break;
  case TypeVarValue.Server:
    if (ElemArr)
      Var1.SetValueArr(Index, (ObjServ)Var.Obj);
    else
      Var1.Obj = (ObjServ)Var.Obj;
```

```
  break;
case TypeVarValue.Proc:
case TypeVarValue.Func:
  if (ElemArr)
    Var1.SetValueArr(Index, (string)Var.Obj);
  else
    Var1.Obj = (string)Var.Obj;
  break;
case TypeVarValue.FileArr:
  ControlDimArr(Var1, Var);
  Var1.Obj = (ObjFile[])Var.Obj;
  break;
case TypeVarValue.ServerArr:
  ControlDimArr(Var1, Var);
  Var1.Obj = (ObjServ[])Var.Obj;
  break;
case TypeVarValue.ComObjArr:
  ControlDimArr(Var1, Var);
  Var1.Obj = (ObjVar[])Var.Obj;
  break;
case TypeVarValue.ProcArr:
case TypeVarValue.FuncArr:
  ControlDimArr(Var1, Var);
  Var1.Obj = (string[])Var.Obj;
  break;
case TypeVarValue.None:
  switch (Var1.ValueType) {
    case TypeVarValue.ComObjArr: {
      int Len;
      Len = Var1.LenArr;
      for (int i = 0; i < Len; i++) {
        (((ObjVar[])Var1.Obj)[i]).Obj = null;
        (((ObjVar[])Var1.Obj)[i]).tp = null;
      }
      Var1.Obj = null;
      Var1.tp = null;
      break;
    }
    case TypeVarValue.ComObj:
      Var1.Obj = null;
      Var1.tp = null;
      break;
    case TypeVarValue.FileArr: {
      int Len;
      Len = Var1.LenArr;
      for (int i = 0; i < Len; i++) {
        (((ObjFile[])Var1.Obj)[i]).oFR = null;
        (((ObjFile[])Var1.Obj)[i]).oFW = null;
      }
      Var1.Obj = null;
      break;
    }
    case TypeVarValue.File: {
      ObjFile File;
      File = (ObjFile)Var1.Obj;
      File.oFR = null;
      File.oFW = null;
      Var1.Obj = null;
      break;
    }
    case TypeVarValue.ServerArr: {
      int Len;
```

```
            Len = Var1.LenArr;
            for (int i = 0; i < Len; i++) {
              (((ObjServ[])Var1.Obj)[i]).oRS = null;
            }
            Var1.Obj = null;
            break;
          }
          case TypeVarValue.Server:
            ObjServ Serv;
            Serv = (ObjServ)Var1.Obj;
            Serv.oRS = null;
            Var1.Obj = null;
            break;
          case TypeVarValue.ProcArr:
          case TypeVarValue.FuncArr:
          case TypeVarValue.Proc:
          case TypeVarValue.Func:
            Var1.Obj = null;
            break;
          default:
            // выдать сообщение об ошибке "invalid Type"
            ErrorMess(209);
            break;
        }
        break;
      }
      if (!FlagLocal)
        VarTable[Var1.Name] = Var1;
      else
        VarLocalTable[Var1.Name] = Var1;
      break;
    case TypeNameVar.Field:
      if (Var1.ValueType == TypeVarValue.Server) {
        NameField = NameVar.NameMethod;
        // присвоить значение свойству Server объекта
        // проверка на совпадение типов поля сервера и
        // присваиваемого выражения
        if (oServ.GetTypeField(NameField) != Var.ValueType)
          // выдать сообщение об ошибке "invalid Type"
          ErrorMess(209);
        // получить объект сервер
        oServ = (ObjServ)Var1.Obj;
        // присваиваем значение полю объекта
        switch (NameField) {
          case "Host":
            oServ.oRS.Host = (string)Var.Obj;
            break;
          case "User":
            oServ.oRS.User = (string)Var.Obj;
            break;
          case "Password":
            oServ.oRS.Password = (string)Var.Obj;
            break;
          case "IntervalWait":
            oServ.oRS.IntervalWait = (int)Var.Obj;
            break;
          case "Title":
            oServ.Title = (string)Var.Obj;
            break;
          case "WaitReboot":
            oServ.WaitReboot = (int)Var.Obj;
            break;
```

```
            default:
              // выдать сообщение об ошибке "invalid FieldName"
              ErrorMess(232);
              break;
          }
          Var1.Obj = oServ;
        }
        else {
          // присвоить значение свойству COM объекта
          ArrayList Par = new ArrayList();
          if (Var1.tp == null)
            // выдать сообщение об ошибке
            // "for performance it is required COM object"
            ErrorMess(249);
          Par.Add(Var);
          try {
            ComObj.Invoke(ref Var1, NameVar, ref Par, false);
          }
          catch (Exception e) {
            ErrorMessStr(e.Message);
          }
        }
        // сохраняем переменную в таблице
        if (!FlagLocal) {
          VarTable[Var1.Name] = Var1;
        }
        else {
          VarLocalTable[Var1.Name] = Var1;
        }
        break;
      case TypeNameVar.Method:
        // выдать сообщение об ошибке
        // "the object method cannot appropriate value"
        ErrorMess(250);
        break;
    }
.)

              |
(.

   ArrayList ListPar = new ArrayList();

.)
              ListParam<ref ListPar>

(.
   // имя переменной
   Name = NameVar.Name;
   switch (NameVar.Type) {
     case TypeNameVar.Method:
       if (!FlagLocal) {
         if (VarTable.IsName(Name)) {
           Var1 = VarTable[Name];
         }
         else
           // выдать сообщение об ошибке "invalid Var"
           ErrorMess(201);
       }
       else {
         if (VarLocalTable.IsName(Name)) {
           Var1 = VarLocalTable[Name];
```

```
      }
      else
        // выдать сообщение об ошибке "invalid Var"
        ErrorMess(201);
    }
    try {
      ComObj.Invoke(ref Var1, NameVar, ref ListPar, false);
    }
    catch (Exception e) {
      ErrorMessStr(e.Message);
    }
    break;
  case TypeNameVar.Var:
    // осуществляем проверку, переменная по имени Name,
    // является ли процедурным типом
    if (!FuncTable.IsName(Name)) {
      if (!FlagLocal) {
        if (VarTable.IsName(Name)) {
          Var1 = VarTable[Name];
          if ((Var1.ValueType != TypeVarValue.Proc) &&
              (Var1.ValueType != TypeVarValue.Func)) {
            // выдать сообщение об ошибке
            // "invalid Type"
            ErrorMess(209);
          }
          else {
            Name = (string)Var1.Obj;
            if (String.IsNullOrEmpty(Name))
              // выдать сообщение об ошибке
              // "value of a variable is not defined"
              ErrorMess(247);
          }
        }
        else
          // выдать сообщение об ошибке
          // "invalid Name function or procedure"
          ErrorMess(216);
      }
      else {
        if (VarLocalTable.IsName(Name)) {
          Var1 = VarLocalTable[Name];
          if ((Var1.ValueType != TypeVarValue.Proc) &&
              (Var1.ValueType != TypeVarValue.Func)) {
            // выдать сообщение об ошибке
            // "invalid Type"
            ErrorMess(209);
          }
          else {
            Name = (string)Var1.Obj;
            if (String.IsNullOrEmpty(Name))
              // выдать сообщение об ошибке
              // "value of a variable is not defined"
              ErrorMess(247);
          }
        }
        else
          // выдать сообщение об ошибке
          // "invalid Name function or procedure"
          ErrorMess(216);
      }
      if (!FuncTable.IsName(Name))
```

```
                // выдать сообщение об ошибке
                // "invalid Name function or procedure"
                ErrorMess(216);
            }
            ObjFunc Func = (ObjFunc)FuncTable[Name];
            if (Func.FExternal)
              // выполнить указанную стандартную процедуру или функцию
              RunExtFunc(Name, ref FuncTable, ref ListPar);
            else {
              // сохранить информацию о точке возврата из подпрограммы в
              // стеке
              PushGoSub();
              // встать на выполняемую подпрограмму в исходном тексте
              RestorPosProg(Func.LabelFunc);
              Get();
              if (Func.Type == TypeVar.Func)
                // выполнить разбор и выполнение функции
                FuncDeclare (ref ListPar, ref Var);
              else
                // выполнить разбор и выполнение процедуры
                ProcDeclare (ref ListPar);
              // встать на точку возврата
              PopGoSub();
              Get();
            }
            break;
          default:
            // выдать сообщение об ошибке "invalid Type"
            ErrorMess(209);
            break;
        }
    .)

                )  ";" .
```

Да, чуть не забыли! Прежде чем, перейти к описанию работы функции *Assign()*, нужно уточнить область видимости констант и переменных. Константы, объявленные за пределами функций и процедур, являются глобальными, общедоступными константами, они видны в теле пользовательских процедур и функций. Переменные в отличие от констант, всегда локальны. Переменные, объявленные в программе, не видны в теле пользовательских процедур и функций и наоборот, переменные, объявленные в пользовательских процедурах и функциях, не видны в теле программы. Что это значит? Это значит, что когда мы выполняем код пользовательской функции или процедуры, все константы и переменные находятся в таблице *VarLocalTable*, а когда выполняем код программы, то в таблице *VarTable*. Определить состояние можно по значению глобального флага *FlagLocal*.

Выполнение функции *Assign()* функционально можно разделить на три части:

- Определение атрибутов переменной;
- Выполнение операции присваивания;
- Выполнение процедуры (функции) или метода объекта.

Определяем и инициализируем локальные переменные функции. Вызываем функцию *Variable()* передавая в качестве параметра ссылку на переменную *NameVar* типа *DescrNameVar*. В

переменной *NameVar* содержатся атрибуты переменной, которой присваевается значение или это имя вызываемой функции или свойство объекта. Что именно, покажет дальнейший анализ.

Проверяем поле *NameVar.Type*, если оно равно значению *TypeNameVar.Array*, то переменной которой присваевается значение, является элементом массива. Устанавливаем флаг *ElemArr* и считываем значение индекса элемента массива в переменную *Index*. Извлекаем из таблицы переменных по имени *Name* саму переменную и сохраняем в *Var1*. Считываем следующий токен.

Если следующий токен равен операции присваивания *':=",* то вызываем функцию вычисления выражения *Expr()* передавая в качестве выходного параметра переменную *Var* в которую будет помещен результат вычисления выражения. В итоге на данном этапе мы получаем: две переменные: *Var1* – переменная которой мы присваиваем значение и переменная *Var* – само значение. Далее, в зависимости от *NameVar.Type* в операторе switch, переменной *Var1* присваиваем значение переменной *Var*, не забывая выполнить все необходимые проверки (совместимость переменных по присваиванию, значение индекса массива, размерность массива и т.д.). Полученное значение переменной *Var1* сохраняем в таблице переменных *VarTable* или *VarLocalTable* в зависимости от значения глобального флага *FlagLocal*.

Иначе, если текущая лексема равна *"("* или *"()"* - вызваем функцию *ListParam(),* где в качестве входного параметра передаем ссылку на список параметров *ListPar*. Функция *ListParam()* формирует список фактических параметров, вызываемой процедуры, функции или метода. Далее в зависимости от значения *NameVar.Type* которое может принимать в данном случае значения *TypeNameVar.Method* (вызов метода объекта) или *TypeNameVar.Var* (вызов функции, процедуры или выполнение процедурного типа) выполняем нижеследующие действия.

Если переменная *NameVar.Type* имеет значение *TypeNameVar.Method*. Считываем из таблицы переменных, значение объектной переменной и помещаем в *Var1*. Выполняем метод объектной переменной путем вызова статического метода класса *ComObj.Invoke(ref Var1, NameVar, ref ListPar, false)* передавая в качестве параметров ссылку на объектную переменную *Var1*, атрибуты переменной *NameVar*, ссылку на список фактических параметров *ListPar* и признак того, что не нужно возвращать результат выполнения метода, который равен *false*.

Если переменная *NameVar.Type* имеет значение *TypeNameVar.Var*. Осуществляем проверку является ли имя переменной *Name* = *NameVar.Name* процедурной переменной, если да, то в *Name* помещаем имя процедуры или функции на которую указывает переменная. Проверяем идентификатор *Name* на наличие в объекте таблицы процедур и функций *FuncTable*. Если процедуры или функции с именем *Name* не найдено, выдаем сообщение об ошибке и выходим из программы. В объекте процедуры или функции по имени *Name*, проверяем поле *FExternal*, если оно равно *true*, то это стандартная процедура или функция. Выполняем ее запуск с помощью вызова функции *RunExtFunc()*, передавая три параметра: имя *Name*, ссылку на объект таблицы *FuncTable* и список фактических параметров *ListPar*. Если поле *FExternal* равно *false*, то вызываемая процедура или функция определена в исходном тексте программы. Сохраняем информацию о точке возврата из подпрограммы в стеке с помощью вызова функции *PushGoSub()* (сохраняем текущую позицию сканера в исходном тексте программы). Устанавливаем позицию сканера на выполняемую подпрограмму в исходном тексте программы с помощью вызова функции *RestorPosProg()* , передавая в качестве параметра, объект метки процедуры или функции *ObjLabel*. Считываем следующую лексему сканером, вызывая функцию *Get()* парсера. В объекте процедуры

или функции по имени *Name*, проверяем поле *Type*, если поле *Type* равно *TypeVar.Func* выполняем функцию *FuncDeclare()*, иначе выполняем функцию *ProcDeclare()*. Функция *FuncDeclare()* производит разбор и выполнение функции декларированной в исходном тексте программы. Функции *FuncDeclare()* передается два входных параметра, ссылка на список фактических параметров *ListPar* и ссылка на объект переменной возвращаемого результата *Var*. Функция *ProcDeclare()* производит разбор и выполнение процедуры. В качестве входного параметра передается ссылка на список фактических параметров. Далее, востанавливаем позицию сканера до вызова процедуры или функции с помощью вызова функции *PopGoSub()*. Считывем следующую лексему и выходим из функции Assign().

Метод *CompareType()* класса *ObjVar*, выполняет сравнение типов значений объектов переменных по присваиванию и возвращает булевский результат равный *true*, если типы значений совместимы и *false* - в противном случае. Исходный код метода приведен ниже.

```
// сравнение типов значений объектов по присваиванию
public bool CompareType (TypeVarValue Type) {
  bool Res = false;

  switch (this.ValueType) {
    case TypeVarValue.RealArr:
      if (Type == TypeVarValue.RealArr || Type == TypeVarValue.Real ||
          Type == TypeVarValue.Int)
        Res = true;
      break;
    case TypeVarValue.IntArr:
      if (Type == TypeVarValue.IntArr || Type == TypeVarValue.Int)
        Res = true;
      break;
    case TypeVarValue.StrArr:
      if (Type == TypeVarValue.StrArr || Type == TypeVarValue.Str)
        Res = true;
      break;
    case TypeVarValue.BoolArr:
      if (Type == TypeVarValue.BoolArr || Type == TypeVarValue.Bool)
        Res = true;
      break;
    case TypeVarValue.FileArr:
      if (Type == TypeVarValue.FileArr || Type == TypeVarValue.File ||
          Type == TypeVarValue.None)
        Res = true;
      break;
    case TypeVarValue.ServerArr:
      if (Type == TypeVarValue.ServerArr ||
          Type == TypeVarValue.Server ||
          Type == TypeVarValue.None)
        Res = true;
      break;
    case TypeVarValue.ComObjArr:
      if (Type == TypeVarValue.ComObjArr ||
          Type == TypeVarValue.ComObj ||
          Type == TypeVarValue.None)
        Res = true;
      break;
    case TypeVarValue.ProcArr:
      if (Type == TypeVarValue.ProcArr ||
          Type == TypeVarValue.Proc ||
          Type == TypeVarValue.None)
        Res = true;
```

```
          break;
        case TypeVarValue.FuncArr:
          if (Type == TypeVarValue.FuncArr ||
              Type == TypeVarValue.Func ||
              Type == TypeVarValue.None)
            Res = true;
          break;
        case TypeVarValue.Real:
          if (Type == TypeVarValue.Real || Type == TypeVarValue.Int)
            Res = true;
          break;
        case TypeVarValue.Int:
          if (Type == TypeVarValue.Int)
            Res = true;
          break;
        case TypeVarValue.Str:
          if (Type == TypeVarValue.Str || Type == TypeVarValue.Proc ||
              Type == TypeVarValue.Func)
            Res = true;
          break;
        case TypeVarValue.Bool:
          if (Type == TypeVarValue.Bool)
            Res = true;
          break;
        case TypeVarValue.File:
          if (Type == TypeVarValue.File || Type == TypeVarValue.None)
            Res = true;
          break;
        case TypeVarValue.Server:
          if (Type == TypeVarValue.Server || Type == TypeVarValue.None)
            Res = true;
          break;
        case TypeVarValue.ComObj:
          if (Type == TypeVarValue.ComObj || Type == TypeVarValue.None)
            Res = true;
          break;
        case TypeVarValue.Proc:
          if (Type == TypeVarValue.Proc || Type == TypeVarValue.Str ||
              Type == TypeVarValue.None)
            Res = true;
          break;
        case TypeVarValue.Func:
          if (Type == TypeVarValue.Func || Type == TypeVarValue.Str ||
              Type == TypeVarValue.None)
            Res = true;
          break;
      }
      return Res;
    }
```

Функция *ControlDimArr()* осуществляет контроль размерности массивов при операции присваивания и реализует следующее правило - количество элементов в массиве (*размерность*), не является частью его типа, это количество задается при объявлении массива и не может быть изменено впоследствии, кроме как присваиванием значения функции, которая возвращает массив того же типа. Исходный код функции приведен ниже.

```
//*************************************************************
//* Function: ControlDimArr
//* Notes: Контроль размерности массивов при операции присваивания
//*
```

```
//*   Будет выполнятся операция Var1 := Var
//*        Возможны следующие ситуации:
//*          1. Var1.Name = "result" - возвращаемый результат функции
//*                          (массив);
//*          2. Var1 - массив;
//*          3. Var.Name = "result" - возвращаемый результат функции
//*                         (массив);
//*          4. Var - массив.
//*             Комбинация 1, 3 - исключена
//*        Характерные признаки при анализе:
//*          1. Переменная имеет имя (Var.Name <> пусто);
//*          2. Переменная - результат функции, имени не имеет
//*                           (Var.Name = пусто).

private void ControlDimArr (ObjVar Var1, ObjVar Var) {
  if ((Var1.Name == "result") && (!String.IsNullOrEmpty(Var.Name))) {
    if ((Var1.LenArr > 0) && (Var1.LenArr != Var.LenArr))
      // выдать сообщение об ошибке "invalid dimension array"
      ErrorMess(233);
  }
  else {
    if ((!String.IsNullOrEmpty(Var1.Name)) &&
        (!String.IsNullOrEmpty(Var.Name)) && (Var.Name != "result")) {
      if (Var1.LenArr != Var.LenArr)
        // выдать сообщение об ошибке "invalid dimension array"
        ErrorMess(233);
    }
  }
}

//*** End of ControlDimArr ***********
```

Выполнение стандартных процедур и функций

Выполнение стандартных процедур и функций, осуществляется с помощью функции *RunExtFunc()*. Функции *RunExtFunc()* передаются три входных параметра: имя выполняемой процедуры или функции *NameFunc*, ссылка на таблицу определений процедур и функций *FuncTable* и ссылку на список передаваемых параметров *ListParam*. Функция *RunExtFunc(),* возвращает объект типа *ObjVar,* результат выполнения вызываемой процедуры или функции. Часть исходного кода функции приведена ниже.

```
//************************************************************
//* Function: RunExtFunc
//* Notes: Выполнение заданной стандартной процедуры или функции
//*

private ObjVar RunExtFunc (string NameFunc,
                           ref TableObj<ObjFunc> FuncTable,
                           ref ArrayList ListParam) {
  ObjVar Var = new ObjVar();
  ObjVar Result = new ObjVar();
  ObjFunc Func;
  Func = ControlParamList(NameFunc, ref FuncTable, ref ListParam);
  Result.VarType = TypeVar.None;
  Result.ValueType = Func.TypeReturn;

  switch (Func.ID) {
    // "Len" - определение размерности массива или длины строки
    case 0:
      try {
        if (((ObjVar)ListParam[0]).Obj == null) {
```

```
        Result.Obj = (int)(0);
        break;
      }
      switch (((ObjVar)ListParam[0]).ValueType) {
        case TypeVarValue.BoolArr:
        case TypeVarValue.RealArr:
        case TypeVarValue.IntArr:
        case TypeVarValue.StrArr:
        case TypeVarValue.FileArr:
        case TypeVarValue.ServerArr:
        case TypeVarValue.ComObjArr:
        case TypeVarValue.FuncArr:
        case TypeVarValue.ProcArr:
          Result.Obj = ((ObjVar)ListParam[0]).LenArr;
          break;
        case TypeVarValue.Str:
          Result.Obj = ((string)((ObjVar)ListParam[0]).Obj).Length;
          break;
        default:
          // выдать сообщение об ошибке "invalid Type parameter"
          ErrorMess(225);
          break;
      }
    }
    catch (Exception e) {
      ErrorMessStr("Len. " + e.Message);
    }
    break;
  // "Clear" - присваивание элементам массива значений по умолчанию
  case 1:
    Var = (ObjVar)ListParam[0];
    try {
      switch (Var.ValueType) {
        case TypeVarValue.RealArr:
          Array.Clear((double[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.IntArr:
          Array.Clear((int[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.StrArr:
          Array.Clear((string[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.BoolArr:
          Array.Clear((bool[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.FileArr:
          Array.Clear((ObjFile[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.ServerArr:
          Array.Clear((ObjServ[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.ComObjArr:
          Array.Clear((ObjVar[])Var.Obj, 0, Var.LenArr);
          break;
        case TypeVarValue.FuncArr:
        case TypeVarValue.ProcArr:
          Array.Clear((string[])Var.Obj, 0, Var.LenArr);
          break;
      }
    }
    catch (Exception e) {
      ErrorMessStr("Clear. " + e.Message);
```

```
        }
        break;

      …

    }
    return Result;
  }

  //*** End of RunExtFunc ***********
```

Во время своей работы, функция *RunExtFunc()* осуществляет контроль передаваемых параметров вызываемой стандартной процедуры или функции. Контроль передаваемых параметров осуществляется функцией *ControlParamList()*, исходный код которой приведен ниже.

```
  //*************************************************************
  //* Function: ControlParamList
  //* Notes: Контроль списка параметров процедуры или функции
  //*

  private ObjFunc ControlParamList (string NameFunc,
                                    ref TableObj<ObjFunc> FuncTable,
                                    ref ArrayList ListParam) {
    int NumPar;
    ObjFunc Func;

    if (!FuncTable.IsName(NameFunc))
      // выдать сообщение об ошибке "invalid Name function or procedure"
      ErrorMess(216);
    Func = FuncTable[NameFunc];
    if (Func.NumPar == -1)
      return Func;
    // определяем количество переданных параметров
    NumPar = ListParam.Count;
    if (NumPar == 0)
      return Func;
    else if (NumPar != Func.NumPar) {
      // выдать сообщение об ошибке "invalid count Param"
      ErrorMess(205);
    }
    // проверить типы передаваемых параметров
    for (var i = 0; i < NumPar; i++) {
      TypeVarValue VarValueFunc = Func.TypeParam(i);
      TypeVarValue VarValueParam = ((ObjVar)ListParam[i]).ValueType;
      if (VarValueFunc == TypeVarValue.Var)
        continue;
      if (VarValueFunc == TypeVarValue.Array) {
        if ((VarValueParam != TypeVarValue.RealArr) &&
            (VarValueParam != TypeVarValue.IntArr) &&
            (VarValueParam != TypeVarValue.StrArr) &&
            (VarValueParam != TypeVarValue.BoolArr) &&
            (VarValueParam != TypeVarValue.FileArr) &&
            (VarValueParam != TypeVarValue.ServerArr) &&
            (VarValueParam != TypeVarValue.ComObjArr) &&
            (VarValueParam != TypeVarValue.ProcArr) &&
            (VarValueParam != TypeVarValue.FuncArr))
          // выдать сообщение об ошибке "invalid Type parameter"
          ErrorMess(225);
      }
      else if (VarValueFunc != VarValueParam)
        // выдать сообщение об ошибке "invalid Type parameter"
```

```
        ErrorMess(225);
    }
    return Func;
  }

  //*** End of ControlParamList ***********
```

Выполнение процедуры

Синтаксис определения процедуры следующий:

```
ProcDeclare <ref ArrayList ListParam> =
  procedure Ident<out Name> FormalParameters<ref ListFormalPar> ";"
  [ const
    { ConstDeclare<out Var> ";" }
  |
  var
    { VarDeclare<out Var> ";" }
  ]
  begin
    Operations
  end ";" .

FormalParameters<ref ArrayList ListFormalPar> =
  "(" [ FormalPar<out Var> { ";" FormalPar<out Var> } ] ")" | "()" .

FormalPar<out ObjVar Var> =  [ var ] ( "@" ServName<out Name> |
  Ident<out Name> ) ":" Type<out ValueVarType, out DimArr> .
```

Объявление процедур и функций в программе осуществляется до ключевого слова *begin*. В описании синтаксиса языка *Alfa* это не отражено, но, тем не менее - это так. Приведем пример программы состоящей из нескольких исходных строк и попытаемся разобраться, как это все работает.

```
program TestMess;

procedure OutMess (Mess : string);
begin
  writeln(Mess);
end;

begin
  OutMess("Привет!");
end.
```

Интерпретатор запускает функцию *Alfa().* Функция *Alfa()* в процессе работы, в свою очередь запускает функцию *RetrievalLabel()*, которая найдет описание процедуры *OutMess()* и сохранит информацию о имени найденной процедуры и ее местоположении в исходном тексте программы, в таблице процедур или функций *FuncTable*. Далее, функция *Alfa()*, осуществит выполнение операторов, запуская функцию *Operations()*. Так как у нас всего лишь один выполняемый оператор (вызов процедуры *OutMess()*), выполнится функция *Assign()* интерпретатора. Функция *Assign()* определит имя вызываемой процедуры и список фактических передаваемых параметров. Сохранит текущее положение сканера (т.е начало оператора "*end*."). Для выполнения пользовательской процедуры *OutMess()*, функция *Assign()* установит текущее

положение сканера на начало объявления процедуры *OutMess()* в исходном тексте программы и вызовет функцию *ProcDeclare()*, передавая в качестве параметра, ссылку на сформированный список фактических параметров вызываемой процедуры. После завершения работы функции *ProcDeclare()* (выполнения процедуры *OutMess()* в исходном тексте программы), функция *Assign()* восстановит предыдущее состояние сканера. Прочитает лексему *"end."*. Передаст управление функции *Alfa()*. Функция *Alfa()* в свою очередь обнаружив конец программы, завершит свое выполнение. Исходный код функции *ProcDeclare()* приведен далее.

```
ProcDeclare <ref ArrayList ListParam>

          (.
             string NameProc = "";
             string Name = "";
             int NumPar = 0;
             bool InFunc = false;
             TypeVarValue ValueType = new TypeVarValue();
             ArrayList ListFormalPar = new ArrayList();
             ObjVar Var = new ObjVar();
          .)

=
  procedure Ident<out Name>

          (. NameProc = Name; .)

  FormalParameters<ref ListFormalPar> ";"

          (.
             // проверить находимся внутри тела другой функции или процедуры
             if (VarLocalTable.Count() > 0)
               InFunc = true;
             else
               InFunc = false;
             // поместить в стек таблицу локальных переменных
             VarLocalStack.Push(VarLocalTable);
             VarLocalTable = new TableObj<ObjVar>();
             if (ListParam.Count != ListFormalPar.Count)
               // выдать сообщение об ошибке "invalid count Param"
               ErrorMess(205);
             NumPar = ListParam.Count;
             // проверить типы передаваемых параметров и присвоить именам
             // фактических параметров, имена формальных параметров
             // переменные занести в таблицу локальных переменных
             for (var i = 0; i < NumPar; i++) {
               ObjVar Param = (ObjVar)ListParam[i];
               ObjVar FormalPar = (ObjVar)ListFormalPar[i];
               if (Param.VarType == TypeVar.Var) {
                 if (FormalPar.VarType != TypeVar.Var &&
                     FormalPar.VarType != TypeVar.VarLocal)
                   // выдать сообщение об ошибке "invalid Type"
                   ErrorMess(209);
               }
               if (Param.ValueType != FormalPar.ValueType)
                 // выдать сообщение об ошибке "invalid Type"
                 ErrorMess(209);
               Var = Param;
               ValueType = FormalPar.ValueType;
               // сравнение размерности массива фактического параметра
               // с размерностью массива формального параметра
               if ((ValueType == TypeVarValue.RealArr ||
```

```
      ValueType == TypeVarValue.IntArr ||
      ValueType == TypeVarValue.StrArr ||
      ValueType == TypeVarValue.FileArr ||
      ValueType == TypeVarValue.ServerArr ||
      ValueType == TypeVarValue.ComObjArr ||
      ValueType == TypeVarValue.ProcArr ||
      ValueType == TypeVarValue.FuncArr) &&
                   (FormalPar.LenArr != 0)) {
  if (FormalPar.LenArr != Var.LenArr)
    // выдать сообщение об ошибке "invalid dimension array"
    ErrorMess(233);
}
// если параметр определен как массив - значение,
// то создаем копию массива и помещаем в переменную Var
if (FormalPar.VarType == TypeVar.VarLocal) {
  ValueType = Param.ValueType;
  if ((ValueType == TypeVarValue.RealArr) ||
      (ValueType == TypeVarValue.IntArr) ||
      (ValueType == TypeVarValue.StrArr) ||
      (ValueType == TypeVarValue.BoolArr)) {
    int LenByte = 0;
    Var = new ObjVar();
    Var.VarType = TypeVar.VarLocal;
    Var.ValueType = ValueType;
    switch (ValueType) {
      case TypeVarValue.RealArr:
        try {
          Var.Obj = new double[Param.LenArr];
          LenByte = System.Buffer.ByteLength(
                      (double[])Var.Obj);
          System.Buffer.BlockCopy((double[])Param.Obj, 0,
                                  (double[])Var.Obj, 0,
                                  LenByte);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
        }
        break;
      case TypeVarValue.IntArr:
        try {
          Var.Obj = new int[Param.LenArr];
          LenByte = System.Buffer.ByteLength((int[])Var.Obj);
          System.Buffer.BlockCopy((int[])Param.Obj, 0,
                                  (int[])Var.Obj, 0, LenByte);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
        }
        break;
      case TypeVarValue.StrArr:
        try {
          Var.Obj = new string[Param.LenArr];
          ((string[])Param.Obj).CopyTo((string[])Var.Obj, 0);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
        }
        break;
      case TypeVarValue.BoolArr:
        try {
          Var.Obj = new bool[Param.LenArr];
          LenByte = System.Buffer.ByteLength((bool[])Var.Obj);
```

```
                        System.Buffer.BlockCopy((bool[])Param.Obj, 0,
                                                (bool[])Var.Obj, 0, LenByte);
                      }
                      catch (Exception e) {
                        ErrorMessStr(e.Message);
                      }
                      break;
                    }
                  }
                }
                Var.Name = FormalPar.Name;
                if (!VarLocalTable.IsName(Var.Name)) {
                  VarLocalTable.Add(Var.Name, Var);
                }
                else
                  // выдать сообщение об ошибке "invalid VarName"
                  ErrorMess(204);
              }
            .)

{ const
  { ConstDeclare<out Var> ";"

            (.
              if (!VarLocalTable.IsName(Var.Name))
                VarLocalTable.Add(Var.Name, Var);
              else
                // выдать сообщение об ошибке "invalid ConstName"
                ErrorMess(203);
            .)

  }
|
var
  { VarDeclare<out Var> ";"

            (.
              if (!VarLocalTable.IsName(Var.Name)) {
                VarLocalTable.Add(Var.Name, Var);
              }
              else
                // выдать сообщение об ошибке "invalid VarName"
                ErrorMess(204);
            .)

  } }
begin

            (.
              // установить, что работа осуществляется с локальными переменными
              FlagLocal = true;
              // добавить глобальные константы в таблицу локальных переменных
              // процедуры
              foreach (KeyValuePair<string, ObjVar> ObjConst in
                       ConstTable.Table) {
                if (!VarLocalTable.IsName(ObjConst.Key))
                  VarLocalTable.Add(ObjConst.Key, ObjConst.Value);
              }
            .)

Operations
```

```
        (.
           for (var i = 0; i < NumPar; i++) {
             ObjVar FormalPar = (ObjVar)ListFormalPar[i];
             if ((FormalPar.VarType == TypeVar.Var) ||
                 (FormalPar.ValueType == TypeVarValue.File) ||
                 (FormalPar.ValueType == TypeVarValue.Server) ||
                 (FormalPar.ValueType == TypeVarValue.ComObj) ||
                 (FormalPar.ValueType == TypeVarValue.FileArr) ||
                 (FormalPar.ValueType == TypeVarValue.ServerArr) ||
                 (FormalPar.ValueType == TypeVarValue.ComObjArr) ||
                 (FormalPar.ValueType == TypeVarValue.ProcArr) ||
                 (FormalPar.ValueType == TypeVarValue.FuncArr)) {
               // получить имя формального параметра
               Name = FormalPar.Name;
               // получить локальный объект переменной
               Var = VarLocalTable[Name];
               // изменить имя переменной на имя фактического параметра
               Var.Name = ((ObjVar)ListParam[i]).Name;
               // занести в таблицу переменных
               VarTable[Var.Name] = Var;
             }
           }
           // вытащить из стека таблицу локальных переменных
           VarLocalTable = (TableObj<ObjVar>)VarLocalStack.Pop();
           // если не находимся внутри функции или процедуры
           if (!InFunc)
             FlagLocal = false;
           if (Oper == _exit) {
             Oper = 0;
             return;
           }
        .)

  end ";" .
```

Функция *ProcDeclare()* работает следующим образом. Создаем и инициализируем локальные переменные. Из входного потока считываем лексему “*procedure*”, имя процедуры *NameProc* и список формальных параметров *ListFormalPar*. Имеются два вида параметров передаваемых в процедуру или функцию: параметры-значения и параметры-переменные, обозначаемые в списке формальных параметров отсутствием или наличием ключевого слова *var*. При формировании списка формальных параметров, параметрам-значениям присваиваем тип переменной *VarType = TypeVar.VarLocal*, а параметрам-переменным *VarType = TypeVar.Var*. Проверяем, находимся ли внутри тела другой функции или процедуры, устанавливаем в соответствии с этим глобальный флаг InFunc. Помещаем в стек таблицу локальных переменных *VarLocalTable*. Вызовы процедур могут быть вложенными, поэтому процедура должна работать только со своими локальными переменными. Создаем таблицу локальных переменных *VarLocalTable*. Сравниваем количество фактических параметров с количеством формальных параметров, в случае несовпадения выдаем сообщение об ошибке и выходим из программы. Далее в цикле, сравниваем типы переменных и типы значений переменных фактических и формальных параметров, если они не равны, выводим сообщение об ошибке и выходим из программы. Если в качестве параметра-значения передается массив, то тогда создаем новый объект локальной переменной и копируем в него значения из фактического передаваемого параметра. Имени переменной из списка фактических параметров подставляем имя переменной из списка формальных параметров. Проверяем на наличие в таблице локальных переменных, если объект переменной уже существует с таким именем, выдаем сообщение об ошибке и выходим из программы, иначе заносим в таблицу локальных

переменных. Считываем объявления констант, переменных и заносим их в таблицу локальных переменных. Устанавливаем флаг *FlagLocal = true*, указывающий, что мы работаем с локальными переменными, т.е. в контексте процедуры. Добавляем глобальные константы в таблицу локальных переменных процедуры. Выполняем последовательность операторов. После завершения выполнения операторов в цикле, для каждой переменной из списка формальных параметров-переменных, извлекаем объект переменной *Var* из таблицы локальных переменных *VarLocalTable*. Присваиваем имя фактического параметра объекту переменной *Var* и сохраняем в таблице переменных *VarTable*. Восстанавливаем предыдущий контекст процедуры, извлекая из стека таблицу локальных переменных. Проверяем флаг вложенности InFunc, если он не установлен, сбрасываем флаг работы с локальными переменными *FlagLocal = false*. Проверяем текущий выполняемый оператор, если это был оператор exit, сбрасываем код текущего выполняемого оператора в 0 и выходим из функции *ProcDeclare()*.

Выполнение функции

Синтаксис определения функции следующий:

```
FuncDeclare <ref ArrayList ListParam, ref ObjVar Result> =
  function Ident<out Name> FormalParameters<ref ListFormalPar> ":"
    Type<out ValueVarType, out DimArr> ";"
  [ const
    { ConstDeclare<out Var> ";" }
  |
  var
    { VarDeclare<out Var> ";" } ]
  begin
    Operations
  end ";" .
```

Главное отличие между процедурами и функциями состоит в том, что функции имеют возвращаемое значение. При вызове функции она выполняется и возвращает значение вызывающему приложению. Чтобы возвратить значение из функции, это значение нужно присвоить специальной переменной. Переменная result – специальная переменная, которая неявно создается в каждой функции.

Пример:

```
// запустить-остановить заданную службу на сервере CUMR04A
function CUMR04A_StartStop (NameService : string; Start : bool) : string;
begin
  result := "";
  if Start then
    if not @CUMR04A->StartService(NameService) then
      result := @CUMR04A->Get_MessErr();
    endif;
  else
    if not @CUMR04A->StopService(NameService) then
      result := @CUMR04A->Get_MessErr();
    endif;
  endif;
end;
```

Функция *FuncDeclare()*, выполняет пользовательские функции. Функция принимает два входных параметра, ссылку на список фактических параметров и ссылку на объект переменной,

возвращающей результат выполнения. Алгоритм функционирования *FuncDeclare()* такой же, как и функции *ProcDeclare()* рассмотренной выше, но с некоторыми отличиями. Создается предопределенная переменная *result*.

```
// создать локальную переменную result
if (!VarLocalTable.IsNameVar("result"))
  VarLocalTable.Add("result", new ObjVar(
                    "result", TypeVar.Var, ValueVarType));
```

При завершении функции *FuncDeclare()*, возвращается результат выполнения пользовательской функции в переменной *result*.

```
// проверить тип возвращаемого значения функцией
Result = VarLocalTable["result"];
if ((Result.ValueType == TypeVarValue.None) ||
    (Result.ValueType != ValueVarType))
  // выдать сообщение об ошибке "invalid Type result function"
  ErrorMess(215);
```

Исходный код функции *FuncDeclare()* приведен далее.

```
FuncDeclare <ref ArrayList ListParam, ref ObjVar Result>

          (.
             string NameFunc = "";
             string Name = "";
             int NumPar = 0;
             int DimArr = 0;
             bool InFunc = false;
             TypeVarValue ValueVarType = new TypeVarValue();
             TypeVarValue ValueType = new TypeVarValue();
             ArrayList ListFormalPar = new ArrayList();
             ObjVar Var = new ObjVar();
          .)

=
  function Ident<out Name>

          (. NameFunc = Name; .)

  FormalParameters<ref ListFormalPar>

          (.
             // проверить находимся внутри тела другой функции или процедуры
             if (VarLocalTable.Count() > 0)
               InFunc = true;
             else
               InFunc = false;
             // поместить в стек таблицу локальных переменных
             VarLocalStack.Push(VarLocalTable);
             VarLocalTable = new TableObj<ObjVar>();
             if (ListParam.Count != ListFormalPar.Count)
               // выдать сообщение об ошибке "invalid count Param"
               ErrorMess(205);
             NumPar = ListParam.Count;
             // проверить типы передаваемых параметров и присвоить именам
             // фактических параметров, имена формальных параметров
             // переменные занести в таблицу локальных переменных
             for (var i = 0; i < NumPar; i++) {
               ObjVar Param = (ObjVar)ListParam[i];
```

```
ObjVar FormalPar = (ObjVar)ListFormalPar[i];
if (Param.VarType == TypeVar.Var) {
  if (FormalPar.VarType != TypeVar.Var &&
      FormalPar.VarType != TypeVar.VarLocal)
    // выдать сообщение об ошибке "invalid Type"
    ErrorMess(209);
}
if (Param.ValueType != FormalPar.ValueType)
  // выдать сообщение об ошибке "invalid Type"
  ErrorMess(209);
Var = Param;
ValueType = FormalPar.ValueType;
// проверка размерности массива фактического параметра
// с размерностью массива формального параметра
if ((ValueType == TypeVarValue.RealArr ||
     ValueType == TypeVarValue.IntArr ||
     ValueType == TypeVarValue.StrArr ||
     ValueType == TypeVarValue.FileArr ||
     ValueType == TypeVarValue.ServerArr ||
     ValueType == TypeVarValue.ComObjArr ||
     ValueType == TypeVarValue.ProcArr ||
     ValueType == TypeVarValue.FuncArr) &&
      (FormalPar.LenArr != 0)) {
  if (FormalPar.LenArr != Var.LenArr)
    // выдать сообщение об ошибке "invalid dimension array"
    ErrorMess(233);
}
// если параметр определен как массив - значение,
// то создаем копию массива и помещаем в переменную Var
if (FormalPar.VarType == TypeVar.VarLocal) {
  ValueType = Param.ValueType;
  if ((ValueType == TypeVarValue.RealArr) ||
      (ValueType == TypeVarValue.IntArr) ||
      (ValueType == TypeVarValue.StrArr) ||
      (ValueType == TypeVarValue.BoolArr)) {
    int LenByte = 0;
    Var = new ObjVar();
    Var.VarType = TypeVar.VarLocal;
    Var.ValueType = ValueType;
    switch (ValueType) {
      case TypeVarValue.RealArr:
        try {
          Var.Obj = new double[Param.LenArr];
          LenByte = System.Buffer.ByteLength(
                      (double[])Var.Obj);
          System.Buffer.BlockCopy((double[])(Param).Obj, 0,
                                  (double[])Var.Obj, 0,
                                  LenByte);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
        }
        break;
      case TypeVarValue.IntArr:
        try {
          Var.Obj = new int[Param.LenArr];
          LenByte = System.Buffer.ByteLength((int[])Var.Obj);
          System.Buffer.BlockCopy((int[])(Param).Obj, 0,
                                  (int[])Var.Obj, 0, LenByte);
        }
        catch (Exception e) {
          ErrorMessStr(e.Message);
```

```
                    }
                    break;
                  case TypeVarValue.StrArr:
                    try {
                      Var.Obj = new string[Param.LenArr];
                      ((string[])Param.Obj).CopyTo((string[])Var.Obj, 0);
                    }
                    catch (Exception e) {
                      ErrorMessStr(e.Message);
                    }
                    break;
                  case TypeVarValue.BoolArr:
                    try {
                      Var.Obj = new bool[Param.LenArr];
                      LenByte = System.Buffer.ByteLength((bool[])Var.Obj);
                      System.Buffer.BlockCopy((bool[])(Param).Obj, 0,
                                              (bool[])Var.Obj, 0, LenByte);
                    }
                    catch (Exception e) {
                      ErrorMessStr(e.Message);
                    }
                    break;
                }
              }
            }
            Var.Name = FormalPar.Name;
            if (!VarLocalTable.IsName(Var.Name)) {
              VarLocalTable.Add(Var.Name, Var);
            }
            else
              // выдать сообщение об ошибке "invalid VarName"
              ErrorMess(204);
          }
        .)

":"
Type<out ValueVarType, out DimArr> ";"

        (.
          if (ValueVarType == TypeVarValue.File)
            // выдать сообщение об ошибке "invalid Type"
            ErrorMess(209);
          // создать локальную переменную result
          if (!VarLocalTable.IsName("result"))
            VarLocalTable.Add("result", new ObjVar(
              "result", TypeVar.Var, ValueVarType, DimArr));
        .)

{ const
  { ConstDeclare<out Var> ";"

        (.
          if (!VarLocalTable.IsName(Var.Name))
            VarLocalTable.Add(Var.Name, Var);
          else {
            // выдать сообщение об ошибке "invalid ConstName"
            ErrorMess(203);
          }
        .)

  }
|
```

```
var
  { VarDeclare<out Var> ";"

        (.
             if (!VarLocalTable.IsName(Var.Name)) {
               VarLocalTable.Add(Var.Name, Var);
             }
             else
               // выдать сообщение об ошибке "invalid VarName"
                ErrorMess(204);
          .)

  } }
begin

        (.
           // установить, что работа осуществляется с локальными переменными
           FlagLocal = true;
           // добавить глобальные константы в таблицу локальных переменных
           // функции
           foreach (KeyValuePair<string, ObjVar> ObjConst in
                    ConstTable.Table) {
             if (!VarLocalTable.IsName(ObjConst.Key))
               VarLocalTable.Add(ObjConst.Key, ObjConst.Value);
           }
        .)

Operations

        (.
           for (var i = 0; i < NumPar; i++) {
             ObjVar FormalPar = (ObjVar)ListFormalPar[i];
             if ((FormalPar.VarType == TypeVar.Var) ||
                  (FormalPar.ValueType == TypeVarValue.File) ||
                  (FormalPar.ValueType == TypeVarValue.Server) ||
                  (FormalPar.ValueType == TypeVarValue.ComObj) ||
                  (FormalPar.ValueType == TypeVarValue.FileArr) ||
                  (FormalPar.ValueType == TypeVarValue.ServerArr) ||
                  (FormalPar.ValueType == TypeVarValue.ComObjArr) ||
                  (FormalPar.ValueType == TypeVarValue.ProcArr) ||
                  (FormalPar.ValueType == TypeVarValue.FuncArr)) {
               // получить имя формального параметра
               Name = FormalPar.Name;
               // получить локальный объект переменной
               Var = VarLocalTable[Name];
               // изменить имя переменной на имя фактического параметра
               Var.Name = ((ObjVar)ListParam[i]).Name;
               // занести в таблицу переменных
               VarTable[Var.Name] = Var;
             }
           }
           // поместить в переменную Result возвращаемый результат
           Result = VarLocalTable["result"];
           // вытащить из стека таблицу локальных переменных
           VarLocalTable = (TableObj<ObjVar>)VarLocalStack.Pop();
           // если не находимся внутри функции или процедуры
           if (!InFunc)
             FlagLocal = false;
           if (Oper == _exit) {
             Oper = 0;
             return;
           }
```

```
        .)
  end ";" .
```

Оператор IF

Синтаксис оператора следующий:

```
      If =
        if Expr<out Var> then
          Operations
        [else
          Operations]
        endif ";" .
```

Выполнение оператора *If*, осуществляется функцией *If()* интерпретатора. Исходный текст оператора *if* представлен далее.

```
If

        (.
           ObjVar Var = new ObjVar();
        .)

=
  if Expr<out Var>

        (.
           if (Var.ValueType != TypeVarValue.Bool)
             // выдать сообщение об ошибке "invalid Type"
             ErrorMess(209);
           if ((bool)Var.Obj) {

        .)

  then Operations

        (.
             if (!FindEndOper(_if, _endif)) {
               // выдать сообщение об ошибке "invalid operator if"
               ErrorMess(212);
             }
           }
           else {
             if (!FindElse()) {
               // выдать сообщение об ошибке "invalid operator if"
               ErrorMess(212);
             }
           }
        .)

  [else Operations

        (.
           // если последний выполняемый оператор exit, то ищем endif
           if (Oper == _exit) {
             if (!FindEndOper(_if, _endif)) {
               // выдать сообщение об ошибке "invalid operator if"
               ErrorMess(212);
             }
```

```
            }
        .)

]
endif ";" .
```

Рассмотрим выполнение функции *If()*. Объявляем и инициализируем локальные переменные функции. Вычисляем выражение, следующее за ключевым словом “*if*”. Значение выражения помещаем в объект переменной *Var*. Проверяем тип значения переменной *Var*. Если тип, не равен *TypeVarValue.Bool* (т.е. не равен булевскому типу), выдаем сообщение об ошибке и выходим из программы. Если значение переменной *Var* равно *true*, выполняем последовательность операторов следующих за ключевым словом “*then*”. Ищем ключевое слово “*endif*” в тексте программы с помощью вызова функции *FindEndOper()* . Если ключевое слово “*endif*”, не найдено, выдаем сообщение об ошибке и выходим из программы. Если значение переменной *Var* равно *false*, то с помощью вызова функции *FindElse*() ищем ключевое слово “*else*” или “*endif*”, если ключевое слово “*else*” не найдено. Если ключевое слово “*else*” присутствует, выполняем последовательность операторов следующих за ним. Если последним выполняемым оператором был оператор exit, то ищем ключевое слово “*endif*” в тексте программы с помощью вызова функции *FindEndOper()* и тем самым завершаем выполнение функции *If()*. Исходный код функций *FindEndOper()* и *FindElse()* представлен далее.

```
//*************************************************************
//* Function: FindEndOper
//* Notes: Нахождение конечного служебного слова в операторе
//*        (Например в if_endif, case_endcase, switch_endswitch,
//*         for_endfor, while_endwhile)

private bool FindEndOper (int BegLex, int EndLex) {
  int Line;
  int Col;
  int Level = 0;

  if (la.kind == EndLex)
    return true;
  Line = t.line;
  Col = t.col;
  Get();
  while (la.kind != _EOF) {
    if (la.kind == BegLex)
      Level++;
    else if (la.kind == EndLex) {
      if (Level == 0)
        break;
      else
        Level--;
    }
    Get();
  }
  if (la.kind != _EOF)
    return true;
  else {
    t.line = Line;
    t.col = Col;
    return false;
  }
}
```

```
  //*** End of FindEndOper ***********

  //*************************************************************
  //* Function: FindElse
  //* Notes: Нахождение Else в конструкции If или
  //*        нахождение EndIf, если Else отсутствует
  //*

  private bool FindElse () {
    int Line;
    int Col;

    if ((la.kind == _else) || (la.kind == _endif))
      return true;
    Line = t.line;
    Col = t.col;
    Get();
    while (la.kind != _EOF) {
      if (la.kind == _if) {
        // ищем EndIf
        if (!FindEndOper(_if, _endif))
          return false;
      }
      else if ((la.kind == _else) || (la.kind == _endif))
        break;
      Get();
    }
    if (la.kind != _EOF)
      return true;
    else {
      t.line = Line;
      t.col = Col;
      return false;
    }
  }

  //*** End of FindElse ***********
```

Оператор SWITCH

Синтаксис оператора следующий:

```
Switch =
  switch Expr<out Var>
    CaseSection<ref Var, out Result>
    { CaseSection<ref Var, out Result> }
    [ default ":" Operations]
  endswitch ";" .

CaseSection<ref ObjVar Var1, out bool Result> =
  case Expr<out Var> ":"
    Operations
  endcase ";" .
```

Реализация оператора switch представлена ниже.

```
Switch

        (.
           bool Result = false;
```

```
            ObjVar Var = new ObjVar();
          .)

=
  switch Expr<out Var>
    CaseSection<ref Var, out Result>

          (.
            if (Result)
              goto End;
          .)

    { CaseSection<ref Var, out Result>

          (.
            if (Result)
              break;
          .)

    }

          (.
            if (Result)
              goto End;
          .)

    [ default ":" Operations]

          (.
            goto End1;
            End:
            if (!FindEndOper(_switch, _endswitch)) {
              // выдать сообщение об ошибке "invalid operator switch"
              ErrorMess(227);
            }
            End1:
          .)

  endswitch ";" .
```

Определяем и инициализирум локальные переменные. Считываем ключевое слово “*switch*” из исходного текста программы. Вычисляем значение выражения оператора *switch*. Вычисленное значение выражения помещаем в переменную *Var*. В синтаксисе оператора *switch* определено, что как миниум одна секция *case* должна присутствовать. Вызываем функцию *CaseSection()* с передачей двух параметров ссылку на переменную *Var*, которая содержит вычисленное значение выражения и булевскую переменную *Result* в качестве возвращаемого результата. Переменная *Result* возвращает результат сравнения выражения *switch* с выражением секции *case*. Проверяем возвращаемое значение *Result*, если оно равно истине, переходим на метку *End* – поиск в исходном тексте программы ключевого слова “*endswitch*”. В цикле (синтаксическая конструкция *{} Coco/R* транслируется как цикл) пока значение следующего токена равно ключевому слову “*case*” вызываем функцию *CaseSection()*, если возвращаемое значение функции *Result* равно истине - выходим из цикла, иначе выполняем цикл до тех пор пока не переберем все секции *case*. При выходе из цикла проверяем значение переменной *Result*, если оно равно истине переходим на метку *End* – завершение обработки оператора *switch*. Если ни одно выражение *case* не совпадает с выражением *switch* и в исходном тексте присутствует ключевое слово “*default*” считываем ключевое ключевое слово “*default*” и “*:*”, выполняем последовательность операторов

следующих за ними вызовая функцию *Operations().* Далее переходим по метке *End1* на чтение ключевого слова “*endswitch*” и “*;*” и завершаем работу оператора *switch*.

Исходный код функции *CaseSections()* представлен ниже.

```
CaseSection<ref ObjVar Var1, out bool Result>

          (.
             ObjVar Var = new ObjVar();
             // результат сравнения выражения switch с выражением case
             Result = false;
          .)

=
  case Expr<out Var> ":"

          (.
             // сравнение типа выражения switch с типом выражения case
             if (Var1.ValueType != Var.ValueType)
               // выдать сообщение об ошибке "invalid Type"
               ErrorMess(209);
             // сравниваем значения выражения switch с выражением case
             switch (Var1.ValueType) {
               case TypeVarValue.Int:
                 if ((int)Var1.Obj == (int)Var.Obj)
                   Result = true;
                 else
                   Result = false;
                 break;
               case TypeVarValue.Str:
                 if ((string)Var1.Obj == (string)Var.Obj)
                   Result = true;
                 else
                   Result = false;
                 break;
               // выдать сообщение об ошибке для запрещенных типов
               default:
                 // выдать сообщение об ошибке "invalid Type"
                 ErrorMess(209);
                 break;
             }
             // выражение switch, не равно case выражению,
             // ищем endcase, операторы внутри case не выполняем
             if (!Result) {
               if (!FindEndOper(_case, _endcase)) {
                 // выдать сообщение об ошибке "invalid operator case"
                 ErrorMess(226);
               }
               goto End;
             }
          .)

    Operations

          (.
             // если последний выполняемый оператор exit, ищем endcase
             if (Oper == _exit) {
               if (!FindEndOper(_case, _endcase)) {
                 // выдать сообщение об ошибке "invalid operator case"
                 ErrorMess(226);
               }
```

```
            }
            End:
         .)

  endcase ";" .
```

Функция *CaseSections()* на входе получает два параметра. Первый параметр *Var1* типа *ObjVar*, ссылка на значение выражения *switch*, второй параметр *Result* булевского типа, результат сравнения выражения *switch* оператора с выражением case оператора. Определяем и инициализируем локальные переменные. Считываем ключевое слово "*case*". Вычисляем значение выражения, следующее за ключевым словом *case*, и помещаем результат в переменную *Var*. Сравниваем типы переменной *Var1* и переменной *Var* (проводим сравнение типа выражения *switch* с типом выражения *case*), если они не равны, выдаем сообщение об ошибке и выходим из программы. Сравниваем значение переменной *Var1* и значение переменной *Var* (проводим сравнение значений выражения *switch* с значением выражения *case* для типов *int* или *string*), если значения совпадают, присваиваем Result = true, иначе *Result = false*. Если результат сравнения отрицательный, находим с помощью вызова функции *FindEndOper()* в исходном тексте ключевое слово "*endcase*" и завершаем работу функции, иначе выполняем операторы в секции *case*. Если при выполнении операторов последний выполняемый оператор *exit* то, находим с помощью вызова функции *FindEndOper()* в исходном тексте ключевое слово "*endcase*" и завершаем работу функции.

Оператор WHILE

Синтаксис оператора следующий:

```
While =
  while Expr<out Var>
    Operations
  endwhile ";" .
```

Циклы *while* могут быть вложенными на нескольких уровнях. В начале цикла информация о местоположении начала цикла помещается в стек. Проверяется условие выполнения цикла, если оно истинно, выполняются операторы внутри цикла. Каждый раз, когда встречается ключевое слово *endwhile*, происходит извлечение информации из стека, проверяется условие выполнения цикла. Если условие выполнения ложно, цикл останавливается, а выполнение программы продолжается со строки, следующей за ключевым словом *endwhile*. В противном случае информация о местоположении начала цикла снова помещается в стек, и выполнение цикла продолжается. После завершения внутреннего цикла его информация извлекается из стека. Если существовал внешний цикл, то на вершину стека попадает именно его информация. Таким образом, каждое ключевое слово *endwhile* автоматически ассоциируется с соответствующим ключевым словом *while*. Исходный код оператора *while* представлен ниже.

```
While

         (.
            ObjVar Var = new ObjVar();
            ObjLabel BegWhile;
         .)

=
  while
```

```
        (.
           Loop:
           // поместить информацию о цикле в стек
           // (местоположение цикла в программе)
           BegWhile = SavePosProg();
           WhileStack.Push(BegWhile);
        .)

  Expr<out Var>

        (.
           if (Var.ValueType != TypeVarValue.Bool) {
               // выдать сообщение об ошибке "invalid Type"
               ErrorMess(209);
           }
           // проверить условие выполнения цикла
           if ((bool)Var.Obj) {
        .)

  Operations

        (.
             // вытащить информацию из стека
             BegWhile = (ObjLabel)WhileStack.Pop();
             //   или оператор Exit, то на выход
             if (Oper == _exit) {
               Oper = 0;
               goto EndWhile;
             }
        .)

  endwhile ";"

        (.
             // перейти на запомненое положение начала цикла
             RestorPosProg(BegWhile);
             // считать следующую лексему
             Get();
             // перейти на выполнение операторов внутри цикла
             goto Loop;
           }
           else {
             // вытащить информацию из стека
             BegWhile = (ObjLabel)WhileStack.Pop();
           }
           // найти следующий оператор за телом цикла
           EndWhile:
           // найти endwhile, в случае неудачи выдать сообщение об ошибке
           if (!FindEndOper(_while, _endwhile)) {
             // выдать сообщение об ошибке "invalid operator while, absent
             // endwhile"
             ErrorMess(214);
           }
           // считать лексему endwhile, которая была найдена
           Get();
           Expect(_scolon);
        .) .
```

Создаем и инициализируем локальные переменные. Считываем ключевое слово *while*, запоминаем текущее местоположение сканера в переменной *BegWhile*, т.е. местоположение

условного выражении я цикла. Помещаем переменную *BegWhile* в стек. Вычисляем условное выражение цикла и помещаем вычисленное значение в переменную *Var*. Если тип вычисленного значения не булевский тип, выдаем сообщение об ошибке и выходим из программы. Проверяем значение выражения *Var*, если оно истинно, выполняем операторы внутри цикла. Извлекаем из стека запомненное положение начала цикла в исходном тексте программы в переменную *BegWhile*. Проверяем, если последний выполняемый оператор *exit* переходим по метке *EndWhile* на завершение работы цикла. Иначе считываем ключевое слово "*endwhile*" и "*;*", восстанавливаем запомненное положение начала цикла. Переходим по метке *Loop* на начало обработки цикла. Если условия цикла не выполняется, извлекаем информацию из стека, ищем с помощью функции *FindEndOper()* ключевое слово "*endwile*", считываем "*;*" и выходим из обработки оператора.

Оператор FOR

Синтаксис оператора следующий:

```
For =
  for Variable<out Name> ":=" Expr<out Var> to Expr<out Var>
    Operations
  endfor ";"
```

Цикл *for* допускает только циклы с положительным приращением управляющей переменной, причем с каждой итерацией это приращение равно 1. Циклы *for* могут быть вложенными на нескольких уровнях. В начале цикла вся информация о состоянии управляющей переменной, конечном значении, а также местоположении начала цикла помещается в стек. Каждый раз, когда встречается ключевое слово *endfor*, происходит извлечение информации из стека, управляющая переменная модифицируется, после чего ее значение сравнивается с конечным значением. Если управляющая переменная имеет значение, которое больше конечного, цикл останавливается, а выполнение программы продолжается со строки, следующей за ключевым словом *endfor*. В противном случае модифицированная информация снова помещается в стек, и выполнение цикла продолжается. После завершения внутреннего цикла его информация извлекается из стека. Если существовал внешний цикл, то на вершину стека попадает именно его информация. Таким образом, каждое ключевое слово *endfor* автоматически ассоциируется с соответствующим ключевым словом *for*. Исходный код оператора *for* представлен ниже.

```
For

          (.
             ObjVar Var = new ObjVar();
             ObjVar Var1;
             ObjFor For;
             string Name = "";
          .)

=
  for Ident<out Name>

          (.
             if (!FlagLocal) {
               if (!VarTable.IsName(Name)) {
                 // выдать сообщение об ошибке "invalid VarName"
                 ErrorMess(204);
```

```
            }
            Var1 = VarTable[Name];
          }
          else {
            if (!VarLocalTable.IsName(Name)) {
              // выдать сообщение об ошибке "invalid VarName"
              ErrorMess(204);
            }
            Var1 = VarLocalTable[Name];
          }
        .)

":=" Expr<out Var>

        (.
          if ((Var1.ValueType != Var.ValueType) ||
              (Var1.ValueType != TypeVarValue.Int)) {
              // выдать сообщение об ошибке "invalid Type"
              ErrorMess(209);
          }
          // присвоить переменной цикла начальное значение
          Var1.Obj = (int)Var.Obj;
          if (!FlagLocal)
            VarTable[Name] = Var1;
          else
            VarLocalTable[Name] = Var1;
        .)

to Expr<out Var>

        (.
          if (Var.ValueType != TypeVarValue.Int) {
            // выдать сообщение об ошибке "invalid Type"
            ErrorMess(209);
          }
          // проверить условие выполнения цикла
          if ((int)Var.Obj >= (int)Var1.Obj) {
            // поместить информацию о цикле в стек
            // (переменная цикла, конечное значение, местоположение цикла в
            //  программе)
            PushFor((int)Var1.Obj, (int)Var.Obj);
          Loop:
        .)

Operations

        (.
            // вытащить информацию из стека
            For = PopFor();
            //если оператор Exit, то на выход из цикла
            if (Oper == _exit) {
              Oper = 0;
              goto EndFor;
            }
        .)

endfor ";"

        (.
            // получить переменную цикла
            if (!FlagLocal)
              Var1 = VarTable[Name];
```

```
        else
          Var1 = VarLocalTable[Name];
        // увеличить переменную цикла на 1
        Var1.Obj = (int)Var1.Obj + 1;
        if (!FlagLocal)
          VarTable[Name] = Var1;
        else
          VarLocalTable[Name] = Var1;
        // если переменная цикла больше конечного значения,
        // то на выход
        if ((int)Var1.Obj > For.end)
          return;
        // поместить информацию о цикле в стек
        ForStack.Push(For);
        // перейти на запомненое положение начала цикла
        RestorPosProg(For.Label);
        // считать следующую лексему
        Get();
        // перейти на выполнение операторов внутри цикла
        goto Loop;
      }
      // найти следующий оператор за телом цикла
      EndFor:
      // найти endfor, в случае неудачи выдать сообщение об ошибке
      if (!FindEndOper(_for, _endfor)) {
        // выдать сообщение об ошибке "invalid operator for, absent
        // endfor"
        ErrorMess(213);
      }
      // считать лексему endfor, которая была найдена
      Get();
      Expect(_scolon);
  .) .
```

Создаем и инициализируем локальные переменные. Получаем имя управляющей переменной цикла *Name*. Если управляющая переменная цикла является элементом массива, выдаем сообщение об ошибке и выходим из программы. Получаем объект управляющей переменной цикла *Var1* из локальной таблицы переменных, если флаг *FlagLocal* работы с локальными переменными установлен или из таблицы переменных, если он сброшен. Вычисляем начальное значение управляющей переменной *Var*. Осуществляем следующие проверки, тип значения управляющей переменной цикла и тип начального значения должен совпадать и этот тип должен быть целым значением, если эти условия не выполняются выдаем сообение об ошибке и выходим из программы. Присваиваем управляющей переменной цикла *Var1*, начальное значение цикла *Var* и сохраняем в таблице переменных. Вычисляем выражение конечного значения цикла и помещаем в переменную *Var*. Проверяем тип конечного значения цикла он должен быть целым значением. Далее проверяем условие выполнения цикла. Цикл выполняется, если конечное значение цикла *Var* больше либо равно значению управляющей переменной цикла *Var1*. Если условие выполняется помещаем информацию о цикле в стек - значение переменной цикла, конечное значение и местоположение начала цикла в исходном тексте. Местоположением начала цикла является первый оператор в теле цикла. Выполняем операторы цикла. Извлекаем из стека данные о цикле в переменную *For*. Если последний выполняемый оператор *exit* осуществляем переход на метку *EndFor*. Считываем из исходного текста ключевое слово “*endfor*” и символ “*;*”. Считываем запомненное значение управляющей переменной цикла из таблицы переменных и помещаем в переменную *Var1*. Увеличиваем значение управляющей переменной цикла *Var1* на единицу и запоминаем в таблице переменных. Проверяем условие завершения цикла. Если

значение управляющей переменной цикла *Var1* больше конечного значения цикла, то осуществляем выход из функции *For()*. Помещаем данные о цикле в стек. Устанавливаем позицию сканера на начало цикла, считываем следующую лексему. Осуществляем переход по *Loop* на выполнение операторов внутри цикла. Если условие выполнения цикла ложно, то находим в исходном тексте программы “*endfor*” и “*;*” и завершаем выполнение функции. Функции сохранения информации о цикле *PushFor()* и извлечении её из стека *PopFor()* приведены ниже.

```
//*************************************************************
//* Function: PushFor
//* Notes: Сохранение информации о цикле For
//*

private void PushFor (int Beg, int End) {
  ObjFor Label = new ObjFor(Beg, End, SavePosProg());
  ForStack.Push(Label);
}

//*** End of PushFor ***********

//*************************************************************
//* Function: PopFor
//* Notes: Извлечение информации о цикле For
//*

private ObjFor PopFor () {
  return (ObjFor)ForStack.Pop();
}

//*** End of PopFor ***********
```

Функции *PushFor()* и *PopFor()* для своей работы используют объект цикла типа *ObjFor*, описание которого приведено далее.

```
// объект цикла
public struct ObjFor {
  // переменная счетчик (начальное значение)
  public int beg;
  // конечное значение
  public int end;
  // объект метки
  public ObjLabel Label;

  // конструктор класса
  public ObjFor (int Beg, int End, int Ch, int Pos, int Col, int Line,
                 int OldEols) {
    this.Label = new ObjLabel(Ch, Pos, Col, Line, OldEols);
    this.beg = Beg;
    this.end = End;
  }
  // конструктор класса
  public ObjFor (int Beg, int End, ObjLabel ForLabel) {
    this.Label = ForLabel;
    this.beg = Beg;
    this.end = End;
  }
}
```

Оператор LOOP

Синтаксис оператора следующий:

```
Loop =
  loop
    Operations
  endloop ";" .
```

Оператор цикла *loop* в отличие от своих собратьев *for* и *while*, не имеет, ни счетчика количества итераций, ни условия для завершения. Завершить цикл *loop* можно, только выполнением оператора *exit* в его теле. Оператор *loop* достаточно прост для реализации. Исходный код оператора *loop* приведен ниже.

```
Loop

        (.
           ObjLabel BegLoop;
        .)

=
  loop

        (.
           Loop:
           // поместить информацию о цикле в стек
           // (местоположение цикла в программе)
           BegLoop = SavePosProg();
           LoopStack.Push(BegLoop);
        .)

  Operations

        (.
           // вытащить информацию из стека
           BegLoop = (ObjLabel)LoopStack.Pop();
           //если оператор Exit, то на выход
           if (Oper == _exit) {
             Oper = 0;
             // найти endloop, в случае неудачи выдать сообщение об ошибке
             if (!FindEndOper(_loop, _endloop)) {
               // выдать сообщение об ошибке "invalid operator loop, absent
               // endloop"
               ErrorMess(218);
             }
             // считать лексему endloop, которая была найдена и выйти
             Get();
             Expect(_scolon);
             return;
           }
        .)

  endloop ";"

        (.
           // поместить информацию о цикле в стек
           LoopStack.Push(BegLoop);
           // перейти на запомненое положение начала цикла
           RestorPosProg(BegLoop);
           // считать следующую лексему
           Get();
```

```
            // перейти на выполнение операторов внутри цикла
            goto Loop;
         .) .
```

Считываем ключевое слово “*loop*”. Запоминаем текущее положение сканера в переменной *BegLoop* и помещаем переменную *BegLoop* в стек. Оператор *loop* может быть вложенным. Выполняем операторы в теле цикла. Извлекаем информацию из стека в переменную *BegLoop*. Осуществляем проверку последнего выполняемого оператора. Если последний выполняемый оператор был *exit*, ищем в исходном тексте ключевое слово “*endloop*” с помощью функции *FindEndOper()* и устанавливаем текущую позицию сканера за найденным ключевым словом. Считываем лексему “*;*” и выходим. Иначе считываем “*endloop*” и “*;*”. Помещаем переменную *BegLoop* в стек. Устанавливаем сканер на запомненное положение начала цикла в исходном тексте программы. Считываем следующую лексему. Переходим на начало выполнения цикла (на метку *Loop*).

Оператор READLN

Синтаксис оператора следующий:

```
ReadLn =
  readln [ ListParam<ref ListPar> ] ";" .
```

Оператор *readln* позволяет считывать данные вводимые с клавиатуры или считывать строковые данные из текстового файла, связанного с файловой переменной. Исходный код оператора представлен ниже.

```
ReadLn

         (.
            ArrayList ListPar = new ArrayList();
            ObjFile oFile;
            ObjVar Var;
            string Value;
            int Count = 0;
            bool FlagInp = true;
         .)

=
  readln [ ListParam<ref ListPar> ] ";"

         (.
            Count = ListPar.Count;
            if ((Count == 2) && ((ObjVar)ListPar[0]).ValueType ==
                 TypeVarValue.File) {
              if (!FlagLocal) {
                 if (!VarTable.IsName(((ObjVar)ListPar[0]).Name))
                   // выдать сообщение об ошибке "invalid VarName"
                   ErrorMess(204);
                 Var = VarTable[((ObjVar)ListPar[0]).Name];
                 oFile = (ObjFile)Var.Obj;
              }
              else {
                 if (!VarLocalTable.IsName(((ObjVar)ListPar[0]).Name))
                   // выдать сообщение об ошибке "invalid VarName"
                   ErrorMess(204);
                 Var = VarLocalTable[((ObjVar)ListPar[0]).Name];
```

```
        oFile = (ObjFile)Var.Obj;
      }
      if (oFile.oFR == null)
        // выдать сообщение об ошибке "error open file"
        ErrorMess(224);
      Var = (ObjVar)ListPar[1];
      if (Var.VarType != TypeVar.Var)
        // выдать сообщение об ошибке "invalid Var"
        ErrorMess(201);
      Value = oFile.oFR.ReadLine();
      switch (Var.ValueType) {
        case TypeVarValue.Real:
          try {
            Var.Obj = Convert.ToDouble(Value);
          }
          catch {
            // выдать сообщение об ошибке "invalid number read"
            ErrorMess(207);
          }
          break;
        case TypeVarValue.Int:
          try {
            Var.Obj = Convert.ToInt32(Value);
          }
          catch {
            // выдать сообщение об ошибке "invalid number read"
            ErrorMess(207);
          }
          break;
        case TypeVarValue.Str:
           Var.Obj = Value;
           break;
        case TypeVarValue.Bool:
          try {
            Var.Obj = Convert.ToBoolean(Value);
          }
          catch {
            // выдать сообщение об ошибке "invalid boolean read"
            ErrorMess(208);
          }
          break;
      }
      if (!FlagLocal)
        VarTable[Var.Name] = Var;
      else
        VarLocalTable[Var.Name] = Var;
      return;
    }
    switch (Count) {
      case 0:
        Console.Write("Press any key ... ");
        Console.ReadKey(true);
        Console.WriteLine("");
        break;
      case 1:
        Var = (ObjVar)ListPar[0];
        if (Var.VarType != TypeVar.Var)
          // выдать сообщение об ошибке "invalid Var"
          ErrorMess(201);
        FlagInp = true;
        while (FlagInp) {
          Console.Write("? ");
```

```
        Value = Console.ReadLine();
        switch (Var.ValueType) {
          case TypeVarValue.Real:
            try {
              Var.Obj = Convert.ToDouble(Value);
              FlagInp = false;
            }
            catch {
              Console.WriteLine("");
              Console.WriteLine("Error, invalid number!");
              Console.WriteLine("");
            }
            break;
          case TypeVarValue.Int:
            try {
              Var.Obj = Convert.ToInt32(Value);
              FlagInp = false;
            }
            catch {
              Console.WriteLine("");
              Console.WriteLine("Error, invalid number!");
              Console.WriteLine("");
            }
            break;
          case TypeVarValue.Str:
             Var.Obj = Value;
             FlagInp = false;
             break;
          case TypeVarValue.Bool:
            try {
              Var.Obj = Convert.ToBoolean(Value);
              FlagInp = false;
            }
            catch {
              Console.WriteLine("");
              Console.WriteLine("Error, invalid boolean!");
              Console.WriteLine("");
            }
            break;
        }
      }
      if (!FlagLocal)
        VarTable[Var.Name] = Var;
      else
        VarLocalTable[Var.Name] = Var;
      break;
    case 2:
      if (((ObjVar)ListPar[0]).ValueType != TypeVarValue.Str) {
        // выдать сообщение об ошибке "invalid Type"
        ErrorMess(209);
      }
      Var = (ObjVar)ListPar[1];
      if (Var.VarType != TypeVar.Var)
        // выдать сообщение об ошибке "invalid Var"
        ErrorMess(201);
      FlagInp = true;
      while (FlagInp) {
        Console.Write((string)((ObjVar)ListPar[0]).Obj);
        Value = Console.ReadLine();
        switch (Var.ValueType) {
          case TypeVarValue.Real:
            try {
```

```
                    Var.Obj = Convert.ToDouble(Value);
                    FlagInp = false;
                  }
                  catch {
                    Console.WriteLine("");
                    Console.WriteLine("Error, invalid number!");
                    Console.WriteLine("");
                  }
                  break;
                case TypeVarValue.Int:
                  try {
                    Var.Obj = Convert.ToInt32(Value);
                    FlagInp = false;
                  }
                  catch {
                    Console.WriteLine("");
                    Console.WriteLine("Error, invalid number!");
                    Console.WriteLine("");
                  }
                  break;
                case TypeVarValue.Str:
                  Var.Obj = Value;
                  FlagInp = false;
                  break;
                case TypeVarValue.Bool:
                  try {
                    Var.Obj = Convert.ToBoolean(Value);
                    FlagInp = false;
                  }
                  catch {
                    Console.WriteLine("");
                    Console.WriteLine("Error, invalid boolean!");
                    Console.WriteLine("");
                  }
                  break;
              }
            }
            if (!FlagLocal)
              VarTable[Var.Name] = Var;
            else
              VarLocalTable[Var.Name] = Var;
            break;
          default:
            // выдать сообщение об ошибке "invalid count Param"
            ErrorMess(205);
            break;
        }
      .) .
```

Определяем и инициализируем локальные переменные. Считываем ключевое слово “*readln*” из исходного текста. Считываем список параметров, вызывая функцию *ListParam()* передавая в качестве параметра ссылку на переменную *ListPar*. Считываем “*;*”. Определяем количество переданных параметров оператору *readln*. Если количество переданных параметров равно двум и первый параметр файловая переменная, то в зависимости от значения глобального флага *FlagLocal* получаем объект файловой переменной *oFile* по имени переменной из таблицы *FileTable* или *FileLocalTable*. Выполняем проверку объекта файлового потока для чтения *oFile.oFR*, если он равен *null* выдаем сообщение об ошибке и выходим из программы (чтобы подготовить файл к чтению, необходимо использовать процедуру *OpenFile()* в исходном тексте программы, которая открывает существующий файл для чтения.). Считываем из списка параметров объект переменной

Var определенный вторым параметром. Проверяем тип объекта *Var*, если он не равен *TypeVar.Var* выдаем сообщение об ошибке и выходим из программы. Считываем строку из текстового файла и помещаем ее в переменную *Value*. Далее в зависимости от типа значения переменной Var производим соответствующие преобразования значения переменной Value и присваивание полученного значения переменной *Var*. Заносим объект переменной *Var* в таблицу переменных и выходим из функции. Если первый переданный параметр не является файловой переменной, то осуществляем ввод данных пользователя с клавиатуры. В зависимости от количества переданных параметров оператору *readln* выполняем следующие действия. Если оператор *readln* без параметров, выдаем сообщение на консоль "*Press any key …*" и ждем от пользователя нажатия любой клавиши. Передан один параметр – считываем из списка и помещаем в *Var* объект переменной ввода. Проверяем тип объекта *Var*. Устанавливаем флаг ввода *FlagInp* в *true*. В цикле до тех пор, пока значение флага *FlagInp* равно *true*, выдаем на консоль приглашение к вводу информации - символ «*?*». Считываем строку ввода и помещаем в строковую переменную *Value*. Долее в зависимости от типа значения переменной ввода *Var* производим необходимые преобразования введенного значения к типу переменной ввода. Если происходит исключение при преобразовании, выдаем сообщение об ошибке и переходим на начало цикла ввода данных, иначе сбрасываем флаг ввода *FlagInp* и выходим из цикла ввода. Осуществляем запись переменной *Var* в таблицу переменных и завершаем выполнение оператора *readln*. Передано два параметра: первый параметр - строка сообщения (строковый тип), второй параметр – переменная ввода. Выдаем на консоль строку сообщения, считываем строку пользовательского ввода и проводим все действия описанные выше.

Оператор WRITE

Синтаксис оператора следующий:

```
Write                                                                    =
  write [ ListParam<ref ListPar> ] ";" .
```

Оператор *write* осуществляет вывод данных в символьном виде на консоль или в текстовый файл. В качестве параметра оператору, передается список переменных или выражений любого типа, разделенных символом запятая. Если, в качестве первого параметра, передана файловая переменная, то вывод данных осуществляется в текстовый файл, связанный с этой переменной, иначе вывод осуществляется на консоль. Исходный код оператора представлен ниже.

```
Write

          (.
             ArrayList ListPar = new ArrayList();
             ObjFile oFile;
             ObjVar Var;
             int Count = 0;
             string StrOut = "";
          .)

=
  write [ ListParam<ref ListPar> ] ";"

          (.
             Count = ListPar.Count;
             if ((Count >= 2) && ((ObjVar)ListPar[0]).ValueType ==
                  TypeVarValue.File) {
               if (!FlagLocal) {
```

```
      if (!VarTable.IsName(((ObjVar)ListPar[0]).Name))
        // выдать сообщение об ошибке "invalid VarName"
        ErrorMess(204);
      Var = VarTable[((ObjVar)ListPar[0]).Name];
      oFile = (ObjFile)Var.Obj;
    }
    else {
      if (!VarLocalTable.IsName(((ObjVar)ListPar[0]).Name))
        // выдать сообщение об ошибке "invalid VarName"
        ErrorMess(204);
      Var = VarLocalTable[((ObjVar)ListPar[0]).Name];
      oFile = (ObjFile)Var.Obj;
    }
    if (oFile.oFW == null)
      // выдать сообщение об ошибке "error open file"
      ErrorMess(224);
    for (var i = 1; i <= Count - 1; i++) {
      Var = (ObjVar)ListPar[i];
      switch (Var.ValueType) {
        case TypeVarValue.Real:
          oFile.oFW.Write((double)Var.Obj);
          break;
        case TypeVarValue.Int:
          oFile.oFW.Write((int)Var.Obj);
          break;
        case TypeVarValue.Str:
          oFile.oFW.Write((string)Var.Obj);
          break;
        case TypeVarValue.Bool:
          oFile.oFW.Write((bool)Var.Obj);
          break;
        default:
          // выдать сообщение об ошибке "invalid Type parameter"
          ErrorMess(225);
          break;
        }
    }
    return;
  }
  if (Count > 0)
    for (var i = 0; i <= Count - 1; i++) {
      Var = (ObjVar)ListPar[i];
      switch (Var.ValueType) {
        case TypeVarValue.Real:
          StrOut += Convert.ToString((double)Var.Obj);
          Console.Write((double)Var.Obj);
          break;
        case TypeVarValue.Int:
          StrOut += Convert.ToString((int)Var.Obj);
          Console.Write((int)Var.Obj);
          break;
        case TypeVarValue.Str:
          StrOut += (string)Var.Obj;
          Console.Write((string)Var.Obj);
          break;
        case TypeVarValue.Bool:
          StrOut += Convert.ToString((bool)Var.Obj);
          Console.Write((bool)Var.Obj);
          break;
        default:
          // выдать сообщение об ошибке "invalid Type parameter"
          ErrorMess(225);
```

```
                    break;
                }
                if (i == Count - 1) {
                  OutLog(StrOut);
                }
              }
          .) .
```

Работа оператора осуществляется следующим образом. Определяется количество переданных параметров. Далее в зависимости от количества переданных параметров и и типа первого параметра выполняются различные действия. Если передано два или больше параметра и первый параметр является файловой переменной, получаем объект файловой переменной и осуществляем в цикле вывод в файл связанный с файловой переменной, значения параметров следующих за файловой переменной. Если список параметров не пуст и первый параметр не файловая переменная, вывод осуществляется на консоль. Оператор игнорируется при отсутствии списка передаваемых параметров.

Оператор WRITELN

Синтаксис оператора следующий:

```
WriteLn                                                                =
  writeln [ ListParam<ref ListPar> ] ";" .
```

Оператор *writeln* осуществляет вывод данных в символьном виде на консоль или в текстовый файл заканчивающиеся признаком конца строки. В качестве параметра оператору, передается список переменных или выражений любого типа, разделенных символом запятая. Если в качестве первого параметра передана файловая переменная, то вывод данных осуществляется в текстовый файл, связанный с этой переменной, иначе вывод осуществляется на консоль. Исходный код оператора представлен ниже.

```
WriteLn

          (.
             ArrayList ListPar = new ArrayList();
             ObjFile oFile;
             ObjVar Var;
             int Count = 0;
             string StrOut = "";
          .)

=
  writeln [ ListParam<ref ListPar> ] ";"

          (.
             Count = ListPar.Count;
             if ((Count >= 1) &&
                 ((ObjVar)ListPar[0]).ValueType == TypeVarValue.File) {
               if (!FlagLocal) {
                  if (!VarTable.IsName(((ObjVar)ListPar[0]).Name))
                    // выдать сообщение об ошибке "invalid VarName"
                    ErrorMess(204);
                  Var = VarTable[((ObjVar)ListPar[0]).Name];
                  oFile = (ObjFile)Var.Obj;
               }
               else {
                  if (!VarLocalTable.IsName(((ObjVar)ListPar[0]).Name))
```

```
            // выдать сообщение об ошибке "invalid VarName"
            ErrorMess(204);
          Var = VarLocalTable[((ObjVar)ListPar[0]).Name];
          oFile = (ObjFile)Var.Obj;
        }
        if (oFile.oFW == null)
          // выдать сообщение об ошибке "error open file"
          ErrorMess(224);
        for (var i = 1; i <= Count - 1; i++) {
          Var = (ObjVar)ListPar[i];
          switch (Var.ValueType) {
            case TypeVarValue.Real:
              oFile.oFW.Write((double)Var.Obj);
              break;
            case TypeVarValue.Int:
              oFile.oFW.Write((int)Var.Obj);
              break;
            case TypeVarValue.Str:
              oFile.oFW.Write((string)Var.Obj);
              break;
            case TypeVarValue.Bool:
              oFile.oFW.Write((bool)Var.Obj);
              break;
            default:
              // выдать сообщение об ошибке "invalid Type parameter"
              ErrorMess(225);
              break;
            }
          if (i == Count - 1)
            oFile.oFW.WriteLine("");
        }
        return;
      }
      if (Count > 0)
        for (var i = 0; i <= Count - 1; i++) {
          Var = (ObjVar)ListPar[i];
          switch (Var.ValueType) {
            case TypeVarValue.Real:
              StrOut += Convert.ToString((double)Var.Obj);
              Console.Write((double)Var.Obj);
              break;
            case TypeVarValue.Int:
              StrOut += Convert.ToString((int)Var.Obj);
              Console.Write((int)Var.Obj);
              break;
            case TypeVarValue.Str:
              StrOut += (string)Var.Obj;
              Console.Write((string)Var.Obj);
              break;
            case TypeVarValue.Bool:
              StrOut += Convert.ToString((bool)Var.Obj);
              Console.Write((bool)Var.Obj);
              break;
            default:
              // выдать сообщение об ошибке "invalid Type parameter"
              ErrorMess(225);
              break;
          }
          if (i == Count - 1) {
            OutLog(StrOut, false);
            Console.WriteLine("");
          }
```

```
            }
          else {
            OutLog("", false);
            Console.WriteLine("");
          }
        .) .
```

Оператор EXIT

Синтаксис и реализация представлена ниже.

```
Exit = exit
         (. Oper = _exit; .)
       ";" .
```

Оператор *exit* предназначен для прерывания циклов, выхода из тела программы, процедур и функций. Оператор *exit* самый простой в реализации но, тем не менее, он доставляет массу хлопот. Значение глобальной переменной *Oper,* нужно отслеживать в реализации циклов, процедур и функций, а так же в последовательности выполняемых операторов.

Литература

1. Шилдт Г. Искусство программирования на C++.- СПб.: БХВ-Петербург, 2005.
2. Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. – СПб.: Наука и техника, 2006.
3. Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие. – СПб.: Питер, 2007.
4. Йенсен К., Вирт Н. Паскаль: руководство для пользователя. – М.: Финансы и статистика, 1989.
5. Вирт Н. Построение компиляторов. – М.: ДМК Пресс, 2010.
6. Compiler Generator Coco/R. User Manual. – University of Linz.