Мартыненко Борис Константинович

УЧЕБНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПРОЕКТ РЕАЛИЗАЦИИ АЛГОРИТМИЧЕСКИХ ЯЗЫКОВ: ОБЪЕДИНЕНИЯ, СОПОСТАВЛЯЮЩИЕ ПРЕДЛОЖЕНИЯ И ОКРУЖЕНИЯ

Аннотация

Рассматриваются некоторые нерешённые вопросы реализации семантики алгоритмических языков в контексте обсуждаемого проекта.

Ключевые слова: балансировка видов, конструкции объединённых видов, окружения, параметрическая генерация модулей, приводимые, сопоставляющие предложения.

1. ВВЕДЕНИЕ

Редакция любезно предоставила ряд номеров этого журнала для обсуждения учебного проекта, адресованного студентам университетов, изучающих языки программирования и методы их реализации.

Данная статья заканчивает серию, посвящённую проекту реализации алгоритмических языков на базе объектно-ориентированного описания семантики [2–7].

Эти публикации нельзя считать окончательным ответом на все проблемы, которые, очевидно, встанут при детальной разработке предлагаемой темы, поэтому они носят предварительный характер.

Одна из таких проблем возникла при подготовке статьи по реализации значений объединённых видов и сопоставляющих предложений.

2. ОБЪДИНЁННЫЕ ВИДЫ И СОПОСТАВЛЯЮЩИЕ ПРЕДЛОЖЕНИЯ

В языке Алгол 68 не существует никаких конструкций, исполнение которых непосредственно даёт значения объединённых видов. Значения таких видов возникают только в результате приведений, называемых объединениями.

Объединение есть один из способов приведения видов. Объединение не изменяет вида значения, выдаваемого конструкцией во время счёта, а просто увеличивает свободу его использования. Такое значение должно быть приемлемо не только для одного какого-то вида значения конструкции, но для целого множества видов. Однако после объединения это значение может использоваться примитивным действием только после динамической

проверки его *сопоставляющим предложением*. И в самом деле, с конструкцией вида **UNITED** нельзя запрограммировать никакие примитивные действия (кроме, конечно, присваивания переменной-вида-**UNITED**).

Например:

1) ***union*** **(*bool, char*)** *t, v; t* := *"a"*; *t* := **true**; *v* := *t*.

В этом последовательном предложении описываются две переменные *t*, *v* вида, объединённого из логических и литерных.

Источник первого присваивания *"a"* литерного вида объединяется до вида ***union*** **(*bool, char*)**, перед тем как переменной *t* будет присвоено апостериорное значение источника.

Аналогично второе присваивание той же самой переменной *t* возможно только после приведения значения источника **true** логического вида к виду ***union*** **(*bool, char*)**.

Наконец, последнее присваивание *v* := *t* изменяет значение переменной *v*, после того как его источник *t* разыменовывается.

Напомним, что объединение возможно только в конструкции, занимающей *крепкую* или *сильную* позицию [1].

2) ***union*** **(*int, void*)** *u* : = ***empty*****; ...; *union*** **(*char, int, void*)** (*u*).

В этом примере описывается переменная *u* вида ***union*** **(*int, void*)** с инициализацией *пустым* значением. Это пустое значение вида ***void*** приводится к виду ***union*** **(*int, void*)**, прежде чем переменная *u* получает его в качестве своего значения.

В дальнейшем переменная *u* приводится ещё раз, но теперь уже к виду ***union*** **(*char, int, void*)**, поскольку она занимает сильную позицию, образованную конструкцией *ядро* [1].

Нелишне ещё раз подчеркнуть, что приведения составляют часть действий по исполнению конструкций, причём эти действия связаны с эквивалентными преобразованиями значений – результатов других конструкций. Какие преобразования необходимо выполнить данной конструкции, использующей значение её подконструкции, определяется планом приведений, получаемым объектом-конструкцией (*coercend*) в момент его создания [6, 7].

Объединение как преобразование вида значений, имеет ту особенность, что оно создаёт апостериорное значение просто дополняя априорное значение ещё одним полем – номером вида в составе объединения (`ModeNumber`). Этот номер и используется для выбора исполняемой основы. В то же время доступ к объекту-значению конкретного (необъединённого) вида передаётся использующей его основе анонимно через вершину стека данных.

Основная форма сопоставляющего предложения представляется следующей схемой (см. рис. 1).

Здесь *выясняющее предложение* есть последовательное предложение вида, *объединённого* из нескольких видов, которых должно быть не меньше двух. Его результат, как всегда, доставляется последней основой выясняющего предложения и должен быть вида **union ($\mathbf{m_1}$, ..., $\mathbf{m_n}$)**.

Выясняющее предложение не содержит *определяющих* вхождений меток, и потому исполняется ровно один раз.

Часть, следующая за символом **in**, называемая *главной частью*, состоит из двух или более основ (U_k) с предшествующими спецификациями, каждая из которых включает формальный описатель вида ($\mathbf{m}_k$) и

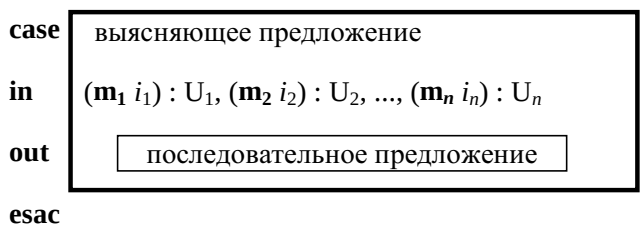

Рис. 1. Блочная структура сопоставляющего предложения

необязательный идентификатор (i_k), областью действия (range [1]) которого является основа U_k ($1 \le k \le n$, $n \ge 2$). При этом считается, что этот идентификатор, если он используется, обладает *конкретным* значением[1] выясняющего предложения, которое может использоваться в соответствующей основе.

Если *конкретный* вид значения выясняющего предложения, совпадает с видом, специфицированным формальным описателем $\mathbf{m}_k$ перед основой U_k, то исполняется именно эта основа, и её значение служит априорным результатом всего сопоставляющего предложения. В противном случае, априорный результат сопоставляющего предложения доставляется последовательным предложением, следующим за символом **out**. Эта часть называется *продолжением* выясняющего предложения, так как последовательное out-предложение может начинаться на основу, представленную подобным же сопоставляющим предложением.

Выясняющее предложение занимает *раскрытую* позицию[2] [1], в то время как одна из выбираемых основ главной части или out-предложение занимают позицию того же сорта, что и само сопоставляющее предложение. Остальные конструкции в составе сопоставляющего предложения находятся в *сильной* позиции.

Балансировка видов подконструкций сопоставляющего предложения, то есть выбор сортов позиций основ в in- и out- части и определение апостериорных видов выясняющего предложения, основ главной части и out-предложения, является задачей синтаксического анализа и воплощается в планах приведений этих конструкций [6]. В частности, выяснение, подходит ли вид значения выясняющего предложения хотя бы одной спецификации главной части или нет, относится к этапу синтаксического анализа и фиксируется в поле данных (`choice: Boolean`) объекта-конструкции ‘сопоставляющее предложение’ (см. п. 4).

Если choice есть истина, то выполняется основа с номером i, где i – номер вида объединения, а иначе если существует продолжение выбирающего предложения, то оно выполняется в окружении, устанавливаемом сопоставляющим предложением.

Что бы ни было выбрано для исполнения, область действия результата сопоставляющего предложения должна быть не новее, чем область действия (scope [1]) окружения, в котором оно исполняется.

Напомним, что в Алголе 68 приведения выполняются над результатами последних основ последовательных предложений перед передачей их объемлющей конструкции. Причём приведения касаются только *приводимых (coercend)* конструкций, к которым по синтаксису языка Алгол 68 относятся все основы, кроме ЗАМКНУТЫХ[3] предложений, *переходов* и *псевдоимён*.

3. ПРЕДСТАВЛЕНИЕ ЗНАЧЕНИЙ ОБЪЕДИНЁНЫХ ВИДОВ

Значения объединённых видов, в отличие от обычных значений необъединённых видов, имеют дополнительное поле данных `ModeNumber`, представляющее номер вида в множестве видов, входящих в данное объединение.

Как всегда, указатель на приводимое значение находится в `UV`, независимо от того, является оно уже объединённым раннее или нет.

В случае, когда значение приводимого не является значением объединённого вида, исполнение приведения путём объединения заключается в том, чтобы снабдить его дополнительным полем – номером вида в объединении.

Объединение уже объединённого значения приводит лишь к расширению первоначального множества видов объединения и перенумерации видов в расширенном множе-

[1] То есть значением необъединённого вида.
[2] В ней допускаются лишь разыменования и распроцедуривания.
[3] То есть замкнутых, условных, вариантных, сопоставляющих и циклических предложений.

стве. Другими словами, если объединение как приведение исполняется над значением, уже являющимся объединённым, то в нём уже имеется подходящее поле: необходимо лишь сменить его значение на другой номер в расширенном объединении.

Значения объединённых видов представляются как наследники типа `TValue` и определяются в зависимости от состава видов, входящих в объединение.

Например, объединение вида, специфицированное описателем **`union(bool,real)`**, определяется следующим предописанием типов:

```
type PUnionBooleanRealValue = ^TUnionBooleanRealValue;
     TUnionBooleanRealValue = object (TValue)
    { Унаследованное поле данных:
      Scope : integer; }
      ModeNumber : integer; { Номер конкретного вида в объединении }
      ParticularValue : PValue; { Значение конкретного вида }
     constructor Init (m : integer; v1 : PBooleanValue; v2 : PRealValue);
      function Show : PChar; virtual;
      function GetParticularValue : PValue; virtual;
      function GetModeNumber : integer; virtual;
     end;
```

и реализацией методов:

```
constructor TUnionBooleanRealValue.Init
            (m : integer; v1 : PBooleanValue; v2 : PRealValue);
 { Только один вариант вида возможен из двух:
   1: bool, когда v1 <> Nil и v2 = Nil;
   2: real, когда v2 <> Nil и v1 = Nil. }
 begin ModeNumber := m;
       if v1 <> Nil then ParticularValue := v1
                    else ParticularValue := v2;
       Scope := ParticularValue^.GetScope
 end;
 function TUnionBooleanRealValue.GetParticularValue : PValue;
 begin if ModeNumber = 1
       then {конкретное значение bool}
            GetParticularValue := PBooleanValue(ParticularValue)
       else {конкретное значение real}
            GetParticularValue :=  PRealValue(ParticularValue)
 end;
 function TUnionBooleanRealValue.GetModeNumber : integer;
 begin GetModeNumber := ModeNumber end;
 function TUnionBooleanRealValue.Show : PChar;
 var s : string; Bf, Bf1: array [0..127] of char; pv : PChar;
 begin str (ModeNumber, s);
       s := '('+ s + ', ';
  if ModeNumber = 1 then pv := PBooleanValue (ParticularValue)^.Show
                    else pv := PRealValue (ParticularValue)^.Show;
  StrPCopy (Bf, s);
  StrCat (Bf, pv);
  StrPCopy (Bf1, ')'); StrCat (Bf, Bf1);
  Show := Bf
 end;
```

Кроме того, объекты-значения конкретного вида `boolean` и `real` пополняются методами, реализующими объединение.

```
procedure TRealValue.uniting_to_unionBooleanReal;
 { Объединение конкретного вида bool к виду union (bool, real) }
 var pv : PBooleanValue;
     u : PUnionBooleanRealValue;
 begin pv := PBooleanValue (UV);
       u := New (PUnionBooleanRealValue, Init (1, pv, Nil));
       UV := u
 end;
 procedure TBooleanValue.uniting_to_unionBooleanReal;
 { Объединение конкретного вида real к виду union (bool, real) }
 var pv : PRealValue;
     u : PUnionBooleanRealValue;
 begin pv := PRealValue (UV);
       u := New (PUnionBooleanRealValue, Init (2, Nil, pv));
       UV := u
 end;
```

В процедуре `Coercing` *объединение*, представленное комментарием под литерой `'c'` в [7], теперь раскрыто для случаев объединения *логического* и *вещественного* до **union (bool, real)** и зашифровано литерами `'r'` и `'s'`.

```
Unit COERCION;
 { Все процедуры, реализующие приведения, получают априорное значение в
UV и выдают апостериорное значение на выходе тоже в UV }
interface
uses objects, Strings,
     VALUES, PLAIN_VALUES, UNION, ENVIRON, AUXILIARY, REF_PLAIN;
     procedure Coercing (cc : string);
implementation
 { Интерпретатор приведений }
 procedure Coercing (cc : string);
 var i: integer;
 begin
  for i := 1 to length (cc) do
  if cc [i] = 'a' then {разыменование} UV^.deref else
  if cc [i]='b' then {распроцедуривание} else
   if cc [i]='d' then {обобщение целого до вещественного} UV^.widening
else
  if cc [i]='e' then {обобщение вещественного до комплексного} else
  if cc [i]='f' then {обобщение массива логических до битового} else
  if cc [i]='g' then {обобщение массива литерных до байтового} else
   if cc [i]='i' then {векторизация скалярного до одномерного массива}
else
  if cc [i]='j' then {векторизация имени скалярного до
                      одномерного массива имён} else
  if cc [i]='l' then {векторизация массива до
                      массива большей размерности на 1} else
  if cc [i]='m' then {векторизация имени массива до
                      массива имён на 1 большей размерности} else
  if cc [i]='o' then {опустошение прямое } else
  if cc [i]='p' then {опустошение после раскрытия} else
  if cc [i]='r' then {объединение логического до union (bool, real)}
                     PBooleanValue(UV)^.uniting_to_unionBooleanReal else
  if cc [i]='s' then {объединение вещественного до union (bool, real)}
                    PRealValue (UV)^.uniting_to_unionBooleanReal
 end;
end.
```

Рассмотрим программу с описанием тождества, использующего объединение:

```
.begin .union(.int, .bool) u1 = 3, u2 = .false; ... .end
```

Её образ на инструментальном (промежуточном) языке Free Pascal представляется следующим образом:

```
program UnionBooleanReal_Itentity_Declaration;
uses CRT, objects, Strings,
     VALUES, PLAIN_VALUES, STANDART, FORMULAS, CONSTRUCTS, CLAUSES,
     ENVIRON, DECLARATIONS, COERCION, DENOTATIONS;
var
_id, _id1, _id2, _bd, _u1, _u2: string;
id { Конструкция 'изображение целого' 3 }: PIntegralDenotation;
bd { Конструкция 'изображение логического'.true }: PBooleanDenotation;
cp1 { план приведений конструкции 'изображение целого' 3 },
cp2 {план приведений конструкции 'изображение логического'.false }: string;
IdentityDeclaration { Конструкции 'описание тождества' }:
                    PIdentityDeclaration;
u1, u2 : PTag; { тег конструкции 'описание тождества' }
tl: PTagList; { список тегов конструкций 'описание тождества'}
Addr_u1 { Адрес значения u1}, Addr_u2 { Адрес значения u2 } : PAddr;
cl0 : PConstructList; { Список конструкций последовательного предложения }
Range0 : PRange;  { Последовательное предложение - блок уровня 0 }
main: PClosedClause; { Замкнутое предложение - собственно программа }
begin ClrScr;
 writeln (#13#10'   Тестирование программы Алгола 68:');
 writeln ('   .begin .union (.int,.bool) u1 = 3, u2 = .false; ... .end');
 writeln (#13#10'   ПРОСТРАНСТВО ДАННЫХ:'#10#13);
 { СОЗДАНИЕ СТЕКА ДАННЫХ }
   Stack := New (PStack, Init (1));
   writeln ('   Стек на ', Stack^.Limit, ' участок');
 { СОЗДАНИЕ ТАБЛИЦЫ DISPLAY }
   Display := New (PDisplay, Init (1));
   writeln ('   Display на ', Display^.Limit,' участок'#13#10);
 { СОЗДАНИЕ СЕМАНТИЧЕСКОГО ДЕРЕВА ПРОГРАММЫ }
 { Создание конструкций блока 0 }
   { Изображение целого }
    _id := '3';
     cp1:= 'ds';
     id := New (PIntegralDenotation, Init (@_id[1], 3,''));
   { Изображение логического }
    _bd := 'false'; cp2 := 'r';
     bd := New (PBooleanDenotation, Init (@_bd[1], false, ''));
    { Создание Tag''a u1 = 3}
     _u1 := 'u1'; cp1 := 'ds';
      u1 := New (PTag, Init (true, _u1, id, cp1));
    { Создание Tag''a u2= .false }
     _u2 := 'u2'; cp2 := 'r';
      u2 := New (PTag, Init (true, _u2, bd, cp2));
   { Создание конструкции 'описания тождества' вида .union(.int, .bool) }
     { Создание коллекции тегов }
       tl := New (PTagList, Init (2, 0));
       tl^.Insert (u1);
       tl^.Insert (u2);
```

```
     IdentityDeclaration := New (PIdentityDeclaration,
                                 Init ('.union(.int,.bool)', tl));
    { Создание списка конструкций блока 0 }
      cl0 := New (PConstructList, Init (1, 0));
      cl0^.Insert (IdentityDeclaration);
    { Создание блока 0 }
      Range0 := New (PRange, Init (0, 10, cl0));
    { Создание программы }
      main := New (PClosedClause, Init (Range0));
     writeln (#13#10'   СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:'#13#10,
              main^.Show); readln;
 writeln (#13#10'ЗАПУСК ПРОГРАММЫ ... '#13#10);
 main^.Run;
 writeln (#13#10'ПРОГРАММА ВЫПОЛНЕНА!')
end.
```

```
Тестирование программы Алгола 68:
.begin .union (.int,.bool) u1 = 3, u2 = .false; ... .end

ПРОСТРАНСТВО ДАННЫХ:
Стек на 1 участок
Display на 1 участок

СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:
BEGIN(0)
  [0] .union(.int, .bool) u1 = 3, u2 = false
END(0)

ЗАПУСК ПРОГРАММЫ ...
априорное значение источника = 3
Выполнено widening:  3.000000000000000E+000
TUnionBooleanRealValue.uniting из Real начала ...
Конкретное значение вида Real =  3.000000000000000E+000
Объединённое из Real = (2,  3.000000000000000E+000)
TUnionBooleanRealValue.uniting из Real кончила
апостериорное значение источника = (2,  3.000000000000000E+000 )
априорное значение источника = false
TUnionBooleanRealValue.uniting из Boolean начала ...
Конкретное значение вида Boolean = false
TUnionBooleanRealValue.uniting из Boolean кончила
апостериорное значение источника = (1, false)
ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ИСПОЛНЕНИЯ TTagList.Run
Display [0] ::
[0] u1 => (2,  3.000000000000000E+000)
[1] u2 => (1, false)
ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ
ПУСТО
ПРОГРАММА ВЫПОЛНЕНА!
```

Рис. 1. Протокол исполнения программы **UnionBooleanReal_Itentity_Declaration**

Протокол исполнения описания тождества не нуждается в пояснениях.

4. РЕАЛИЗАЦИЯ КОНСТРУКЦИИ ‘СОПОСТАВЛЯЮЩЕЕ ПРЕДЛОЖЕНИЕ’

Конструкция ‘сопоставляющее предложение’ представляется как объект типа TConformity_clause со следующим предописанием:

```
type PConformity_clause = ^TConformity_clause;
  TConformity_clause = object (TConstruct)
      choice_clause: PConstruct; { Указатель на объект-конструкцию
                                   'выясняющее предложение'}
      in_part: PConstructList; { Указатель на список основ, вариантов выбора
                                 по номеру вида в объединении }
     out_part : PConstruct; { Указатель на объект-конструкцию на случай,
                              когда не один вариант не выбирается }
      choice: Boolean; { Если choice = true, то выполнять i-ю основу,
                         где i – номер вида в объединении, доставляемом
                           значением выясняющего предложения.
                         Иначе — исполнять out-конструкцию.}

      constructor Init(cc: PConstruct; { Указатель на объект-конструкцию
                                         'выясняющее предложение'.}
                       ip: PConstructList; { Указатель на список основ,
                                    вариантов выбора по номеру вида
                                        в объединении. }
                     op: PConstruct; { Указатель на объект-конструкцию на
                                       случай, когда не один вариант
                                       не выбирается. }
                        c: Boolean { Выполнять выбор варианта или нет. }
                       );
     destructor Done; virtual;
     function Show : PChar; virtual;
     procedure Run; virtual;
  end;
```

и реализацией методов

```
constructor TConformity_clause.Init (cc: PConstruct;
                                     ip: PConstructList;
                                     op : PConstruct;
                                     c: boolean);
begin choice_clause := cc; in_part := ip; out_part := op; choice := c end;
destructor TConformity_clause.Done;
begin end;
function TConformity_clause.Show : PChar;
  var Bf : array [0 .. 1023] of Char;
  begin
       StrPCopy (Bf,{#13#10}'.case ');
       StrCat (Bf, choice_clause^.Show);
       StrCat (Bf, #13#10'.in ');
       StrCat (Bf, in_part^.Show);
       if out_part <> Nil then
                         begin StrCat (Bf, #13#10'.out ');
                           StrCat (Bf, out_part^.Show)
                         end;
       StrCat (Bf, #13#10'.esac');
       Show := Bf;
  end;
procedure TConformity_clause.Run;
  var i : integer;
  begin
   if (out_part <> Nil) and (not choice)
```

```
    then begin out_part^.Run end
    else
      begin
         choice_clause^.Run;
         i := PUnionValue(UV)^.ModeNumber;
         if (1 <= i) and (i <= in_part^.Count)
         then begin in_part^.RunUnit(i-1);
                    UV := PUnionValue(UV)^.GetParticularValue
              end;
      end;
  end;
```

Здесь используется метод RunUnit объекта типа TConstructList со следующим предописанием:

```
procedure RunUnit (i : integer); virtual;
```

и реализацией

```
procedure TConstructList.RunUnit (i : integer);
{ Исполнение основы номер i  из списка основ }
begin PConstruct (At(i))^.Run end;
```

Что апостериорные виды основ U_1, U_2, ..., U_n соответствуют видам, специфицированным формальными описателями $\mathbf{m_1}$, ..., $\mathbf{m_n}$, гарантирует конвертер, выполняющий синтаксический анализ входной программы.

При пропуске тестовой программы, включающей сопоставляющее предложение, обнаружилось, что модуль ENVIRON, реализующий окружение, не учитывает должным образом виды значений, размещаемых в стеке[1]. Как следствие, передача и использование конкретного значения выясняющего предложения в *i*-ой основе его главной части ещё потребует детальной разработки после пересмотра этого модуля. При этом следует использовать анонимный способ передачи конкретного значения в основу, используя вершину стека данных, с последующим доступом к этому значению в основе, о которой идёт речь, через его статический адрес (см. TAddr).

5. ЕЩЁ ОДНА ЗАДАЧА ПРОЕКТА

Задача разработки метода параметрической[2] генерации модулей, описывающих семантику реализуемого языка, может быть выделена в качестве отдельной задачи проекта. Причём она разделяется на три части, касающиеся представлений:

1) значений стандартных видов и видов, использующихся в частной программе пользователя;

2) конструкций стандартных видов и видов, использующихся в частной программе пользователя;

3) объектов, образующих окружение исполняемой программы.

При решении этой задачи можно использовать механизм макро-подстановок.

[1] См., например, описания объектов типа TAddr. Причиной этого недосмотра было желание иметь этот модуль независимым от видов, используемых в программах пользователя. Разумеется, это утопия.

[2] Параметр – вид значений и конструкций.

6. ЗАКЛЮЧЕНИЕ

Целью публикаций [2–7] было передать идею реализации алгоритмических языков, основанную на методе описания языков, представленном в [1], и использования объектно-ориентированных систем программирования, таких как Free Pascal [8].

Проделанные эксперименты, надеюсь, дают достаточно пищи для размышлений и передают не только «интонацию», но и «мотив» темы. «Аранжировать» мелодию – дело руководителя команды разработчиков проекта, если таковая найдётся.

Литература

1. Под ред. *А. ван Вейнгаарден, Б. Майу, Дж. Пек, К. Костер* и др. Пересмотренное сообщение об Алголе 68. М., 1979.

2. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков // Компьютерные инструменты в образовании, 2008. № 5. С. 3–18.

3. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков: значения и конструкции // Компьютерные инструменты в образовании, 2009. № 1. С. 10–25.

4. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков: описания и окружения // Компьютерные инструменты в образовании, 2009. № 2. С. 12–29.

5. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков: генераторы и имена // Компьютерные инструменты в образовании, 2009. № 3. С. 3–17.

6. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков: приведения // Компьютерные инструменты в образовании, 2009. № 4. С. 5–29.

7. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков: условные и вариантные предложения // Компьютерные инструменты в образовании, 2009. № 5. С. 7–16.

8. *Michaël Van Canneyt*. Reference guide for Free Pascal. 2002.

Abstract

Some unresolved questions of implementation of semantics of algorithmic languages in a context of the discussed project are considered.

Мартыненко Борис Константинович,
доктор физико-математических наук, профессор кафедры информатики математико-механического факультета СПбГУ,

mbk@ctinet.ru