Мартыненко Борис Константинович

УЧЕБНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПРОЕКТ РЕАЛИЗАЦИИ АЛГОРИТМИЧЕСКИХ ЯЗЫКОВ: ЗНАЧЕНИЯ И КОНСТРУКЦИИ

Аннотация

Рассматривается представление значений простых видов в форме объектов-контейнеров и реализация элементарных конструктов, источников таких значений, в форме объектов-конструкций, то есть изображений соответствующих видов, а также формул над простыми значениями со стандартными операциями.

Описывается реализация последовательных предложений, входящих в состав всех других предложений, таких как замкнутые, выбирающие (по логическому условию, вариантные целому или по виду) и циклические предложения.

Ключевые слова: значение, конструкт, окружение, последовательное предложение, стандартная операция, сцена, участок, формула над простыми значениями.

1. ВВЕДЕНИЕ

В [1] по существу была сформулирована тема моделирования гипотетического вычислителя, используемого в [2] для описания семантики алгоритмического языка типа Алгол 68 в терминах понятий объектно-ориентированного программирования (ООП). При этом объекты: *конструкт*, *значение*, *участок*, *окружение* или *сцена*, с которыми работает этот гипотетический вычислитель, отображаются в соответствующие классы объектов.

Взаимодействие между конструкциями и значениями вытекает из описания смысла программы в терминах действий гипотетического вычислителя. Модель вычислителя имеет дело с программой, представленной в виде семантического дерева объектов-конструкций. Её исполнение сводится к вызовам полиморфного метода Run, относящегося к составляющим подконструкциям программы, в результате чего и появляются значения.

Здесь мы рассмотрим представление значений простых видов (логических, целых, и т. д.) в форме объектов-контейнеров и реализацию элементарных конструктов, источников таких значений, в форме объектов-конструкций, то есть изображений соответствующих видов, а также формул над простыми значениями со стандартными операциями.

Также опишем реализацию последовательных предложений, входящих в состав других предложений, таких как замкнутые, выбирающие (по логическому условию, вариантные по целому или по виду) и циклические предложения. Заметим, что конструкция «программа» сама является замкнутым предложением.

Все эксперименты были проведены на компьютере с использованием системы программирования FREE PASCAL [3].

2. ЗНАЧЕНИЯ

Значения любых видов определяются как абстрактный родовой класс в модуле

```
Unit VALUES;
{ Реализация объектов-значений }
interface
uses objects, Strings;
type
  { Абстрактный объект - родовой класс для значений любых видов }
     PValue = ^TValue;
     TValue = object (TObject)
      Scope : integer;
      constructor Init (level : integer);
      function Show : PChar; virtual;
      function GetScope : integer;
     end;
  implementation
  {---------------- TValue ---------------- }
  constructor TValue.Init (level : integer);
  begin Scope := level end;
  function TValue.GetScope : integer;
  begin GetScope := Scope end;
  function TValue.Show : PChar;
  begin abstract end;
end
```

Значения простых видов определяются в модуле **PLAIN_VALUES** как объекты-контейнеры.

```
Unit PLAIN_VALUES;
{ Реализация объектов-значений простых видов }
interface
uses objects, Strings, VALUES;
{ Представление контейнеров значений простых видов }
type
 { Логические значения }
  PBoolean = ^Boolean;
  PBooleanValue = ^TBooleanValue;
  TBooleanValue = object (TValue)
      Value : Boolean;
      constructor Init (v : Boolean);
      function Show : PChar; virtual;
      function GetValue : Boolean; virtual;
      procedure PutValue (var v : Boolean); virtual;
     end;
{ Целые значения }
    PInteger = ^integer;
    PIntegralValue = ^TIntegralValue;
    TIntegralValue = object (TValue)
     Value : integer;
     constructor Init (v : integer);
     {Использовани: IntegralValue := New (PIntegralValue, Init (v)) }
     destructor Done; virtual;
     {Использовани: dispose (IntegralValue, Done) }
     function Show : PChar; virtual;
     function GetValue: integer; virtual;
     procedure PutValue (var v : integer); virtual;
    end;
          . . .
```

```
implementation
{ РЕАЛИЗАЦИЯ КОНТЕЙНЕРОВ ЗНАЧЕНИЙ ПРОСТЫХ ВИДОВ }
{ Логические значения }
  constructor TBooleanValue.Init (v : Boolean);
  begin inherited Init (0); Value := v end;
  function TBooleanValue.Show : PChar;
  var Bf : array [0..127] of char;
  begin
   if Value then StrPCopy (Bf, 'true') else StrPCopy (Bf, 'false');
   Show := Bf
  end;
  function TBooleanValue.GetValue: Boolean;
  begin GetValue := Value end;
  procedure TBooleanValue.PutValue (var v : Boolean);
  begin Value := v end;
{ Целые значения }
  constructor TIntegralValue.Init (v : integer);
  begin inherited Init (0); Value := v end;
  function TIntegralValue.Show : PChar;
  var s : string; Bf: array [0..127] of char;
  begin str (Value, s); StrPCopy (Bf, s); Show := Bf end;
  function TIntegralValue.GetValue: integer;
  begin GetValue := Value end;
  procedure TIntegralValue.PutValue (var v : integer);
  begin Value := v end;
          . . .
end
```

3. КОНСТРУКЦИИ

Конструкции стандартных видов определяются в постоянном модуле **CONSTRUCTS** и представляются как наследники родового класса типа

```
type PConstruct = ^TConstruct;
     TConstruct = object (TObject)
       Representation : PChar;
        constructor Init (r : PChar);
       procedure Run; virtual;
        function Show : PChar; virtual;
         end;
```

с реализацией методов

```
constructor TConstruct.Init (r : PChar);
begin Representation := r end;
procedure TConstruct.Run;
begin abstract end;
function TConstruct.Show : PChar;
begin Show :=  Representation end;
```

- Метод **Init** создаёт конструкт соответствующей конструкции с учётом её вида.
- Метод **Run** реализует исполнение конструкции.

Если конструкция – описание, то эффект её исполнения фиксируется в текущем участке стека в виде последовательности индикаторов, открывающих доступ к значениям, которыми они обладают.

Если конструкция – основа, то её результат фиксируется в административной полиморфной переменной **UV** (Universal Value) в виде указателя на объект, представляющий это значение. Что делать с этим результатом, «знает» над-конструкция, вызвавшая эту основу. Как правило, если результат конструкции используется в дальнейшем, то есть не опустошается, то он передаётся через стек использующей его конструкции.

Случай, когда все конструкции опустошаются, представлен в листинге I.

Случай, когда конструкции не опустошаются, представлен в листинге II.

• Метод **Show** представляет конструкцию в виде строки типа **PChar**, то есть последовательности символов, ограниченной на конце нулевым кодом.

4. ОСНОВЫ

Самые большие по содержанию основы – это предложения:

– замкнутые,
– последовательные,
– совместные,
– параллельные,
– выбирающие (по логическому условию, вариантные целому или по виду),
– циклические.

Последовательные предложения непосредственно входят в состав всех других предложений, за исключением совместных и параллельных. Поэтому стоит начать с описания реализации последовательных предложений.

Последовательные предложения, как и все другие предложения, в описываемой модели базируются на коллекциях конструкций реализуемого языка. Это отличает описываемый подход от представления сложных объектов в системах функционального программирования, таких как Ocaml [4], базирующихся на списках. Такой выбор обоснован, по крайней мере, методологически, преимуществом прямого доступа перед последовательным.

Представление предложений описано в модуле **CLAUSES**[1].

```
unit CLAUSES;
interface
uses objects, Strings, ENVIRON, CONSTRUCTS, VALUES;
type
  { Список конструкций }
 PConstructList = ^TConstructList;
 TConstructList = object (TCollection)
  procedure Run; virtual;
  { Исполнение всех конструкций списка конструкций блока }
  procedure RunWhile; virtual;
  { Исполнение всех конструкций списка конструкций блока
    до первой конструкции перехода в блоке }
  procedure RunFrom (i : integer); virtual;
  { Исполнение всех конструкций списка конструкций блока,
    начиная с i-ой, 0 <= i <= count-1 }
  function Show : PChar; virtual;
  { Создаёт строку, представляющую конструкции списка из входного текста }
 end;
  { ПОСЛЕДОВАТЕЛЬНОЕ ПРЕДЛОЖЕНИЕ }
  PRange = ^TRange;
  TRange = object (TConstruct)
```

[1] Другие предложения здесь не обсуждаются и не демонстрируются в этом модуле.

```
    Level: integer; {Уровень блока. Если Level = -1, то блок не локализующий}
    Appetite : integer; { Статический аппетит блока }
    ConstructList : PConstructList;    { Список конструкций блока }
    constructor Init{Инициализация блока уровнем lev и списком конструкций cl}
                  (lev: integer;    { Блочный уровень }
                   app: integer;    { Статический аппетит блока }
                   cl: PConstructList { Коллекция конструкций блока (фраз
                                        последовательного предложения )});
    function Show : PChar; virtual;
    { Создаёт строку, представляющую конструкции блока из входного текста }
    procedure Run; virtual;
    { Исполнение конструкций последовательного предложения с выдачей
      результата его последней основы) }
  end;
{ Сцены и переходы } ...
  implementation
{ Список конструкций }
  procedure TConstructList.Run;
    procedure ExecItem (Item : PConstruct); far;
    begin Item^.Run; {Результат в UV} end;
  begin ForEach (@ExecItem) end;
  procedure TConstructList.RunFrom (i : integer);
  var j : integer;
  begin
    for j := i to count - 1 do PConstruct (At(j))^.Run;
  end;
  procedure TConstructList.RunWhile;
  var j : integer;
  begin
    for j := 0 to count - 1 do
        begin
     PConstruct (At(j))^.Run;
      if Jump_elaborated then j := count { прерывание цикла!}
    end;
  end;
  function TConstructList.Show : PChar;
  var s : string; i : integer;
      Bf : array [0..4095] of char; B : array [0..127] of char;
    procedure PrintItem (Item: PConstruct); far;
    begin
     inc (i); str (i, s); StrPCopy (B, s);
     StrCat (Bf, #10#13'     ['); StrCat (Bf,B); StrCat (Bf, '] ');
     StrCat (Bf, Item^.Show)
    end;
  begin
   i := -1; Bf[0] := #0; ForEach (@PrintItem);
   if Bf[0] = #0 then StrPCopy (Bf,'   EMPTY'); Show := Bf
  end;
{ Последовательное предложение }
 constructor TRange.Init(lev: integer;{ Блочный уровень. Если Level = -1, то
                                        блок не локализующий }
                         app: integer;{ Статический аппетит блока }
                         cl: PConstructList {Коллекция конструкций блока(фраз
                                             последовательного предложения)}
                          ); { Инициализация блока уровнем lev и списком
                               конструкций cl }
```

```
 begin Level := lev; Appetite := app; ConstructList := cl end;
 function TRange.Show : PChar;
 var Bf : array [0 .. 1023] of Char;
     Bf1 : array [0 .. 1023] of Char; s : string;
 begin str (Level, s);
       StrPCopy (Bf,#10#13'   BEGIN(' + s); StrCat (Bf,')');
       StrCat (Bf, ConstructList^.Show);
       StrPCopy(Bf1,#10#13'   END('+s); StrCat (Bf1,')');
       Show := StrCat (Bf, Bf1)
 end;
 procedure TRange.Run;
 { Исполнение конструкций последовательного предложения
   с выдачей результата его последней основы }
 var Locale : PLocale;
 begin
   { Создание нового участка }
       Locale := New (PLocale, Init (Level, Appetite, Appetite));
   { Присоединение его к текущему окружению }
       Stack^.AddLocale (locale);
       Jump_elaborated := false;
   { Исполнение коллекции фраз блока }
       ConstructList^.RunWhile;
   { Удаление участка, созданного данным блоком, с вершины стека }
       Stack^.RemoveTopLocale;
    end;
  ...
end.
```

Семантическое дерево программы собирается из блоков, то есть последовательных предложений, в порядке от внутренних к внешним блокам.

Описания сцен включены именно в модуль **CLAUSES**, а не в **VALUES**, потому что они используются в конструкциях перехода, осуществляющих навигацию по конструкциям текущего блока или объемлющих его. В вышеприведённом тексте этого модуля они не показаны, но включены в следующий параграф.

5. ПЕРЕХОДЫ И СЦЕНЫ

Как описано в [1], сцены используются для навигации между конструкциями программы, такими как переходы и вызовы. В рассматриваемой модели семантики они представляются как наследники объектов-значений типа

```
PScene = ^TScene;
TScene = object (TValue)
  Labels : string;       { Последовательность меток помеченной основы
                           или идентификатор процедуры }
  Block : PRange;        { Блок, список конструкций которого непосредственно
                           содержит помеченные основы или тела процедур -
                            цели передач управления }
  UnitNumber : integer; { Номер помеченной основы или тела процедуры
                          относительно списка блока - цели передачи
                          управления });
  constructor Init (level : integer; L : string; B : PRange; U : integer);
  function Show : PChar; virtual;
 end;
```

со следующей реализацией методов

```
constructor TScene.Init
     (level : integer; { Уровень блока - цели передачи управления }
      L : string; { Последовательность меток или идентификатор процедуры }
      B : PRange; { Блок – цель передачи управления }
      U : integer { Номер конструкции в блоке – цели передачи управления });
begin inherited Init (level); Labels := L; Block := B; UnitNumber := U end;
   function TScene.Show : PChar;
    var Bf : array [0..127] of char;
   begin StrPCopy (Bf, Labels + ': ') end;
```

Назначения полей объектов-сцен указано в комментариях.

Указатель **Block** в паре с целым **UnitNumber** представляет статическое описание программой точки, аналог машинного адреса команды, с которой начинается исполнение соответствующего линейного участка программы при программировании в машинных кодах.

Конструкция перехода представляется как наследник типа **TConstruct**

```
PJump = ^TJump;
TJump = object (TConstruct)
   Scene : PScene; { Сцена, указывающая на цель конструкции перехода }
   constructor Init (s : PScene);
   function Show : PChar; virtual;
   procedure Run; virtual; { Исполнение перехода }
  end;
```

со следующей реализацией методов

```
constructor TJump.Init (s : PScene);
   begin Scene := s end;
   function TJump.Show : PChar;
   var Bf, Bf1 : array [0..127] of char; L : string;
   begin
         StrPCopy (Bf, '.goto ');
         L := Scene^.Labels;
         StrPCopy (Bf1, L);
         StrCat (Bf, Bf1);
         Show := Bf
   end;
   procedure TJump.Run;
   var j : integer; B : PRange; CL : PConstructList;
   begin
     j := Scene^.UnitNumber;
     B := Scene^.Block;
     CL := B^.ConstructList;
     CL^.RunFrom (j);
   { Здесь в самый раз прервать исполнение всех остальных элементов списка
     основ данного блока! }
     Jump_elaborated := true
   end;
```

Здесь **Jump_elaborated** – логическая переменная административной системы (модуль **ENVIRON**), специально предназначенная для обслуживания переходов. Когда она равна **true**, все последующие основы текущего блока, следующие за конструкцией перехода, не исполняются. После этого текущей конструкцией становится основа – цель перехода. При этом участки блоков промежуточных уровней удаляются из стека. За этим следит упомянутая выше административная система.

Заметим, что при построении семантического дерева программы существует проблема синхронизации выстраивания конструкций, связанная с переходами.

С одной стороны, значение сцены должно появиться раньше, чем конструкция перехода, в которой она используется.

С другой стороны, в этот момент конструкция перехода не может быть вставлена в список конструкций блока, так как значение сцены перехода опирается на указатель блока, частью которого этот переход является, а этот блок ещё не сформирован. Этот «клинч» разрывается следующим образом.

Сначала формируется список конструкций блока с переходами, в которых используется сцена со значнием **Nil** в качестве указателя на блок – цель перехода; затем этот список вставляется в блок; далее, конструкции переходов доопределяются со сценами, которые используют уже существующий указатель на блок; и, наконец, эти конструкции перехода вставляются в список конструкций блока на своё место. Эта вставка выполняется с помощью метода **AtPut** (номер элемента в списке, указатель на объект-конструкцию). Именно так это было сделано в вышеприведённом примере (*см.* листинг I ниже).

6. ИЗОБРАЖЕНИЯ

Изображения являются *элементарными* конструкциями.

Правило исполнения элементарной конструкции определяет её результат (значение) непосредственно, без ссылки на правила исполнения других конструкций.

Собственно значение изображения фиксируется в административной переменной **UV** в виде указателя на контейнер соответствующего вида.

Например, изображение логического представляется следующим образом

```
PBooleanDenotation = ^TBooleanDenotation;
TBooleanDenotation = object (TConstruct)
  Value : Boolean;
  constructor Init (r : PChar; v : Boolean);
  function Show : PChar; virtual;
  procedure Run; virtual;
 end;
```

с реализацией методов

```
constructor TBooleanDenotation.Init (r : PChar; v : Boolean);
begin inherited Init (r); Value := v end;
function TBooleanDenotation.Show : PChar;
begin Show := Representation end;
procedure TBooleanDenotation.Run;
begin UV := New (PIntegralValue, Init (Value)) end;
```

Изображение целого представляется следующим образом

```
PIntegralDenotation = ^TIntegralDenotation;
TIntegralDenotation = object (TConstruct)
  Value : integer;
  constructor Init (r : PChar; v : integer);
  function Show : PChar; virtual;
  procedure Run; virtual;
 end;
```

с реализацией методов

```
constructor TIntegralDenotation.Init (r : PChar; v : integer);
begin inherited Init (r); Value := v end;
function TIntegralDenotation.Show : PChar;
begin Show := Representation end;
procedure TIntegralDenotation.Run;
begin UV := New (PIntegralValue, Init (Value)) end;
```

Пример использования изображений целых представлен в листинге I. В нём программа представлена двумя блоками **Range0** и **Range1**. Причём внешний блок **Range0** представлен последовательным предложением из четырёх основ, а именно, изображения целого **001**, блока **Range1**, помеченного изображения целого **L4:4** и изображения целого **5**. В свою очередь, блок **Range1** включает изображение целого **02**, переход **goto L4** и изображение целого **03**. Все основы этой программы опустошаются.

Как показывает протокол, представленный на рис. 1, пространство данных, то есть окружение, состоит из двух пустых участков, поскольку программа состоит из двух блоков, в которых нет ни одного описания[1]. Как отмечено выше, только исполнение конструкций-описаний создают индикаторы, составляющие содержание участка.

Результаты исполнения изображений целых показаны в строчках, помеченных текстами «Результат ExecItem:». Это значения 1, 2, 4 и 5. Они взяты из **UV** . Значение изображения **03** не исполняется в связи с переходом на третью основу внешнего блока. Её индекс в рамках коллекции равен 2.

7. ФОРМУЛЫ

Формулы являются наследниками родового класса **TConstruct** и бывают *унарными*, то есть с одним операндом или *бинарными*, то есть с двумя операндами. Каждый из операндов представляется указателем на конструкцию, исполнение которой доставляет значение соответствующего операнда в административной переменной **UV** , а затем оно переносится в локальное поле значения операнда объекта-конструкции «формула». Операция исполняемой формулы выполняется над этими значениями операндов, и опять значение формулы фиксируется в **UV** .

Листинг I. Создание семантического дерева программы **".begin 001; .begin 02; .goto L4; 03 .end; L4:4; 5 .end"** и протокол её исполнения

```
program Dynamic6;
 { Тестирование переходов в последовальных предложениях Алгола 68:
   .begin 1; .begin 02; .goto L4; 03 .end; L4 : 4; 5 .end
   Поскольку значения основ, коими являются изображения целых,
   опустошаются, то их значения в стек не передаются ! }
uses CRT, objects, Strings,
      VALUES, PLAIN_VALUES, STANDART, CONSTRUCTS, CLAUSES, ENVIRON;
var
    id1s, id2s, id3s, id4s, id5s, L4s : string;
    id1, id2, id3, id4, id5 : PIntegralDenotation;
    Scene : PScene;
    Jump : PJump;
    cl0, cl1 : PConstructList;
    Range0, Range1 : PRange;
```

[1] Такие окружения называются *нелокализующими.*

```
begin ClrScr; writeln;
writeln ('    Реализация программы:');
writeln ('    .begin 1; .begin 02; .goto L4; 03 .end; L4 : 4; 5 .end'#10#13);
writeln ('    ПРОСТРАНСТВО ДАННЫХ:'#10#13);
 { СОЗДАНИЕ СТЕКА ДАННЫХ }
   Stack := New (PStack, Init (2));
   writeln ('    Стек на ', Stack^.Limit, ' участка');
 { СОЗДАНИЕ ТАБЛИЦЫ DISPLAY }
   Display := New (PDisplay, Init (2));
   writeln ('    Display на ', Display^.Limit,' участка'); writeln;
 { СОЗДАНИЕ ДЕРЕВА ПРОГРАММЫ }
 { ======= Создание конструкций блока 1 ======= }
 id2s := '02';
 id2 := New (PIntegralDenotation, Init (@id2s[1], 2));
  { Создание сцены без указания блока, ибо соответствующий блок (Range0
    ещё не создан! }
 L4s := 'L4';
 Scene := New (PScene, Init (0, L4s, {Range0} Nil, 2));
 id3s := '03';
 id3 := New (PIntegralDenotation, Init (@id3s[1], 3));
 { Создание конструкции перехода пока с не вполне определённой сценой }
 Jump := New (PJump, Init (Scene));
 { Создание списка конструкций блока 1 }
 cl1 := New (PConstructList, Init (5, 0));
 cl1^.Insert (id2);
 cl1^.Insert (Jump); { Переход не вполне определён!}
 cl1^.Insert (id3);  { Эта основа должна пропускаться !!! }
 Range1 := New (PRange, Init (1, 0, cl1));
 { Создание списка конструкций блока 0 }
 id1s := '1';
 id1 := New (PIntegralDenotation, Init (@id1s[1], 1));
 id4s := 'L4: 4';
 id4 := New (PIntegralDenotation, Init (@id4s[1], 4));
 id5s := '5';
 id5 := New (PIntegralDenotation, Init (@id5s[1], 5));
 cl0 := New (PConstructList, Init (4, 0));
 cl0^.Insert (id1);
 cl0^.Insert (Range1);
 cl0^.Insert (id4);
 cl0^.Insert (id5);
 Range0 := New (PRange, Init (0, 0, cl0));
 { Доопределение сцены (Range0, L4: 4) указанием блока Range0 }
 L4s := 'L4';
 Scene := New (PScene, Init (0, L4s, Range0, 2));
 {Доопределение конструкции перехода с доопределённой сценой (Range0, L4: 4)}
 Jump := New (PJump, Init (Scene));
 { Вставка конструкции перехода на своё место в блоке 1 }
 cl1^.AtPut (1, Jump);
 write ('    *** Блок Range0 создан: ', Range0^.Show); writeln;
 writeln ('    ======= Создание семантического дерева программы завершено
               ======= '); writeln;
 writeln ('    Исполнение последовательного предложения 0 ...');
 Range0^.Run;
 writeln ('    СТОП !!!'); readln
end.
```

```
Реализация программы:
.begin 1; .begin 02; .goto L4; 03 .end; L4 : 4; 5 .end

ПРОСТРАНСТВО ДАННЫХ:

Стек на 2 участка
Display на 2 участка

*** Блок Range0 создан:
BEGIN(0)
  [0] 1
  [1]
BEGIN(1)
  [0] 02
  [1] .goto L4
  [2] 03
END(1)
  [2] L4: 4
  [3] 5
END(0)
======= Создание семантического дерева программы завершено =======

Исполнение последовательного предложения 0 ...
TRange.Run начала ...
Stack^.AddLocale (locale):
Stack [0] =
ConstructList:
  [0] 1
  [1]
BEGIN(1)
  [0] 02
  [1] .goto L4
  [2] 03
END(1)
  [2] L4: 4
  [3] 5
РЕЗУЛЬТАТ ExecItem (0): 1
Состояние стека:
Display [0] ::
TRange.Run начала ...
Stack^.AddLocale (locale):
Stack [0] =
Stack [1] =
ConstructList:
  [0] 02
  [1] .goto L4
  [2] 03
РЕЗУЛЬТАТ ExecItem (0): 2
Состояние стека:
Display [0] ::
Display [1] ::
Исполнение перехода ...
j = 2
РЕЗУЛЬТАТ ExecItem (2): 4
Состояние стека:
Display [0] ::
Display [1] ::
РЕЗУЛЬТАТ ExecItem (3): 5
Состояние стека:
Display [0] ::
Display [1] ::
Исполнение перехода закончилось!
РЕЗУЛЬТАТ ExecItem (1): 5
Состояние стека:
Display [0] ::
Display [1] ::
ТЕКУЩЕЕ ОКРУЖЕНИЕ ДО ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

Display [0] ::
Display [1] ::
ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

Display [0] ::
TRange.Run кончила
РЕЗУЛЬТАТ ExecItem (1): 5
Состояние стека:
Display [0] ::
ТЕКУЩЕЕ ОКРУЖЕНИЕ ДО ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

Display [0] ::
ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ
ПУСТО
TRange.Run кончила
СТОП !!!
```

Рис. 1

В случае унарной формулы ссылка на конструкцию, играющей роль левого операнда (*lo*), равна **nil**.

Таким образом, локальные поля значений операндов объекта-конструкции формула заменяют динамические части участков окружений (см. листинг II и рис. 2 ниже).

```
PFormula = ^Tformula;
TFormula = object (TConstruct)
{ Наследуемые поля данных:
  Representation :  Pchar ; }
  Loperand, Roperand : PConstruct;
  constructor Init (r : Pchar; lo, ro : PConstruct);
  function Show : Pchar; virtual;
  procedure Run; virtual;
end;
```

с реализацией методов

```
constructor TFormula.Init (r : Pchar; lo, ro : PConstruct);
begin  Representation := r; Loperand := lo;  Roperand := ro end;
function TFormula.Show : Pchar; virtual;
begin Show := Representation end;
procedure  TFormula.Run;
begin abstract end;
```

Формулы со стандартными операциями являются наследниками родового класса **TFormula**.

Например, формулы со стандартными операциями вида **(integral) boolean** на Паскале представляются типом

```
Pintegral_boolean_Formula =
 ^Tintegral_boolean_Formula;
 Tintegral_boolean_Formula = object (TFormula)
 { Наследуемые поля данных:
   Representation : Pchar;
   Loperand, Roperand : Pconstruct; }
  Routine : Tintegral_boolean_Routine;
  LoperandValue, RoperandValue : PintegralValue;
  constructor Init (r : Pchar;
                    op : Tintegral_boolean_Routine;
                    ro : PConstruct);
  function Show : Pchar; virtual;
  procedure Run; virtual;
 end;
```

с реализацией методов следующим образом

```
constructor  Tintegral_boolean_Formula.Init
              (r : Pchar; op : Tintegral_boolean_Routine;
               ro : Pconstruct );
 begin Representation := r;
       Routine := op;
       Loperand := nil; Roperand := ro
 end;
 function Tintegral_boolean_Formula.Show : Pchar;
 begin Show := Representation end;
 procedure  Tintegral_boolean_Formula.Run;
```

Листинг II. Создание дерева программы
"begin 001 + 02 + 3 .end"
и протокол её исполнения

```
program Dynamic2;
 { Тестирование последовального предложения Алгола 68:
   .begin 001 + 02 + 3 .end }
uses CRT, objects, Strings,
      VALUES, PLAIN_VALUES, STANDART, CONSTRUCTS, CLAUSES, ENVIRON;
var id1s, id2s, id3s, f1s, f2s : string;
    id1, id2, id3 : PintegralDenotation;
    Routine : Tintegral_integral_integral_Routine;
    f1, f2 : Pintegral_integral_integral_Formula;
    cl0 : PconstructList;
    Range0 : Prange;
begin ClrScr; writeln;
   writeln ('   ПРОСТРАНСТВО ДАННЫХ:'#10#13);
 { СОЗДАНИЕ СТЕКА ДАННЫХ }
   Stack := New (Pstack, Init (1));
   writeln ('   Стек на ', Stack^.Limit, ' участка');
 { СОЗДАНИЕ ТАБЛИЦЫ DISPLAY }
   Display := New (Pdisplay, Init (1));
   writeln ('   Display на ', Display^.Limit,' участка');
 { СОЗДАНИЕ ДЕРЕВА ПРОГРАММЫ }
 writeln ('   ======= Создание конструкций блока 0 ======= ');
 id1s := '001';
 id1 := New (PintegralDenotation, Init (@id1s[1], 1));
 writeln ('   Изображение целого ', id1^.Show);
 id2s := '02';
 id2 := New (PintegralDenotation, Init (@id2s[1], 2));
 writeln ('   Изображение целого ', id2^.Show);
 id3s := '3';
 id3 := New (PintegralDenotation, Init (@id3s[1], 3));
 writeln ('   Изображение целого ', id3^.Show);
 Routine := @PlusRoutine;
 f1s := '001 + 02';
 f1 := New (Pintegral_integral_integral_Formula,
            Init (@f1s[1], Routine, id1, id2));
 writeln ('   Формула вида (integral, integral)integral: ',
          f1^.Show);
 f2s := '001 + 02 + 3';
 f2 := New (Pintegral_integral_integral_Formula,
            Init (@f2s[1], Routine, f1, id3));
 writeln ('   Формула вида (integral, integral)integral: ',
          f2^.Show);
 { Создание списка конструкций блока 0 }
 cl0 := New (PconstructList, Init (1, 0));
 cl0^.Insert (f2);
 Range0 := New (Prange, Init (0, 0, cl0));
 Range0^.Show;
 writeln ('   Исполнение последовательного предложения 0 …');
 Range0^.Run;
 writeln ('   СТОП !!!'); readln
 end.
```

```
ПРОСТРАНСТВО ДАННЫХ:

Стек на 1 участка
Display на 1 участка

======= Создание конструкций блока 0 =======

Изображение целого 001
Изображение целого 02
Изображение целого 3
Формула вида (integral, integral)integral: 001 + 02

Формула вида (integral, integral)integral: 001 + 02 + 3


Исполнение последавательного предложения 0 ...

TRange.Run начала ...

Stack^.AddLocale (locale):
Stack [0] =

ConstructList:
[0] 001 + 02 + 3

РЕЗУЛЬТАТ ExecItem: 6

Состояние стека:
Display [0] ::

ТЕКУЩЕЕ ОКРУЖЕНИЕ ДО ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

Display [0] ::

ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ
ПУСТО

TRange.Run кончила

СТОП !!!
```

Рис. 2

```
begin
       Roperand^.Run;
       RoperandValue := PintegralValue(UV);
       UV := Routine(RoperandValue)
end;
```

при условии, что в модуле **CONSTRUCTS** описан тип

```
Tintegral_boolean_Routine = function ( a : PintegralValue ) : PbooleanValue;
```

для представления операций вида **(integral) boolean**, а в модуле **STANDART** имеются соответствующие функции.

Например

```
function OddRoutine ( a : PintegralValue ) : PbooleanValue;
 var z : Boolean; r : PbooleanValue;
 begin z := abs (a^.GetValue) mod 2 = 1;
       r := New (PbooleanValue, Init (z));
       OddRoutine := r
 end;
```

для реализации операции **odd** вида **(integral)boolean**.

Аналогично, формулы со стандартными операциями вида **(integral, integral) integral** представляются следующим образом

```
Pintegral_integral_integral_Formula =
 ^Tintegral_integral_integral_Formula;
 Tintegral_integral_integral_Formula = object (TFormula)
 { Наследуемые поля данных:
   Representation : PChar;
   LOperand, ROperand : PConstruct; }

  Routine : Tintegral_integral_integral_Routine;
  LOperandValue, ROperandValue : PIntegralValue;

  constructor Init (r : PChar;
                    op : Tintegral_integral_integral_Routine;
                    lo, ro : PConstruct);
  function Show : PChar; virtual;
  procedure Run; virtual;
 end;
```

с реализацией методов

```
constructor Tintegral_integral_integral_Formula.Init
              (r : PChar; op : Tintegral_integral_integral_Routine;
               lo, ro : PConstruct );
 begin Representation := r;
       Routine := op;
       LOperand := lo; ROperand := ro end;

 function Tintegral_integral_integral_Formula.Show : PChar;
 begin Show := Representation end;

 procedure Tintegral_integral_integral_Formula.Run;
 begin LOperand^.Run;
       LOperandValue := PIntegralValue (UV);
       ROperand^.Run;
       ROperandValue := PIntegralValue (UV);
       UV := Routine (LOperandValue, ROperandValue)
 end;
```

и с учётом того, что в модуле **CONSTRUCTS** определёны типы всех стандартных функций Алгола 68, в частности, тип

```
Tintegral_integral_integral_Routine = function(a, b : PintegralValue) :
PintegralValue;
```

для представления операции ‘+’, которая на Паскале описывается следующим образом

```
function PlusRoutine ( a, b : PIntegralValue ) : PIntegralValue;
 var x, y, z : integer; r : PIntegralValue;
 begin x := a^.GetValue;
       y := b^.GetValue;
       z := x + y;
       r := New (PIntegralValue, Init (z));
       PlusRoutine := r
 end;
```

Окружение состоит из одного пустого участка. Блок **Range0** содержит одну основу, представленную бинарной формулой f2, левый операнд есть формула f1: **001 + 02**, а правый есть изображение целого **3**. Оба операнда формулы f1 есть изображения целых cd1 = **001** и cd2 = **02**. Обе формулы, о которых идёт речь, имеют одну и ту же стандар-

тную операцию ‘+’ вида **(integral, integral) integral**, реализуемую через функцию PlusRoutine. Реализация таких формул описана на стр. 24.

8. ЗАМЕЧАНИЯ

Хотя все эксперименты и были проведены на компьютере с использованием системы программирования FREE PASCAL [3], их не следует считать образцом для подражания.

Литература

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

2. Под ред. *А. ван Вейнгаарден, Б.Майу, Дж. Пек, К. Костер* и др. Пересмотренное сообщение об Алголе 68. М., 1979. 533 с.

3. *Michaël Van Canneyt*. Reference guide for Free Pascal. 2002. 188 p.

4. *Leroy. X.* The Objective Caml system release 3.11. Documentation and user’s manual. Institut National de Recherche en Informatique et en Automatique. 2008 // http://caml.inria.fr/pub/docs/manual-ocaml/index.html

Abstract

The representation of the plain mode values in the forms of the object-containers and the elementary constructs resulting in values of such a kind, i.e. the plain denotations and the standard mode formulas, are considered.

The implementation of the serial clauses, constituting closed, choice using boolean, integral or united clauses, as well as the loop clauses is described also.

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

mbk@ctinet.ru