Интерпретатор «Компонентного Паскаля» для Блэкбокс

Темиргалеев Е. Э.
(29.12.2016, правки 31.05.2017)

1. Постановка задачи

StdInterpreter — штатная реализация сервиса выполнения команд *Dialog.Call* в Блэкбокс 1.6 обеспечивает выполнение последовательности операторов вызова процедур с параметрами типа "целое" и "цепочка литер". Фактически, это сильно урезанное подмножество КП:

```
Command = Call { ";" Call }.
Call = ModuleName "." ProcedureName [ "(" Parameter { "," Parameter } ")" ].
Parameter = Integer | String.
Integer = Decimal | Hex.
Decimal = [ "-" ] Digit { Digit }.
Hex = [ "-" ] HexDigit { HexDigit } "H".
String = " ' " { Char } " ' ".
```

Релизация выполнена посредством библиотеки *Meta*, обеспечивающей динамический доступ к модулям и их элементам.

В ББ1.5 *Meta* обеспечивал только вызов процедур статически известного типа, и *StdInterpreter* поддерживал весьма ограниченное их число:

```
TYPE
   (* command procedure types *)
   Proc = PROCEDURE;
   ProcI = PROCEDURE(x: INTEGER);
   ProcII = PROCEDURE(x, y: INTEGER);
   ProcS = PROCEDURE(s: ARRAY OF CHAR);
   …
   ProcRRII = PROCEDURE(IN s, t: ARRAY OF CHAR; x, y: INTEGER);
   (* corresponding meta values *)
   ProcVal = RECORD (Meta.Value) p: Proc END;
   ProcIVal = RECORD (Meta.Value) p: ProcI END;
   ProcIIVal = RECORD (Meta.Value) p: ProcII END;
   ProcSVal = RECORD (Meta.Value) p: ProcS END;
   …
   ProcRRIIVal = RECORD (Meta.Value) p: ProcRRII END;
```

В Блэкбокс 1.6 возможности *Meta* были расширены — появилась поддержка процедур (кроме методов):

```
CONST
   …, parObj = 7;
   …
   value = 10; in = 11; out = 12; var = 13;
…
TYPE
   …
   Item = RECORD (Value)
      …
      (VAR proc: Item) NumParam (): INTEGER, NEW;
```

```
        (VAR proc: Item) GetParam (n: INTEGER; VAR par: Item), NEW;
        (VAR proc: Item) GetParamName (n: INTEGER; OUT name: Name), NEW;
        (VAR proc: Item) GetReturnType (VAR type: Item), NEW;
        …
        (VAR proc: Item) ParamCall (IN par: ARRAY OF Item; VAR dest: Item; OUT ok: BOOLEAN), NEW;
        (VAR proc: Item) ParamCallVal (IN par: ARRAY OF POINTER TO Value; VAR dest: Item; OUT ok: BOOLEAN), NEW;
        …
    END;
```

Возможность динамического вызова произвольной процедуры позволяет реализовать не только отмеченное в начале подмножество КП, но и более широкое. В этом и состоит задача:
- выявить максимально возможное с точки зрения целесообразности подмножество КП, которое имеет смысл поддержать для интерпретатора команд, скорректировав его с учётом динамики *Meta*;
- запрограммировать интерпретатор.

2. Прототип-1

Реализует выполнение присваиваний и вызовов процедур при практически полной поддержке выражений КП и объявления переменных:

- 426-й — основной модуль (парсер/выполнение/интерфейс);
- 427-й — обвязки для Meta — почти все простые операции вынесены из 426-го для сокращения его объёма;
- 311/312-й — сканер *DevCPS* и требующийся ему библиотечный *DevCPM* из штатного компилятора КП с небольшими правками;
- 421-й — инструмент генерации "обёрточных" модулей. Основная работа по переделыванию *DevBrowser.ShowInterface* выполнена Тарасовым С. И.

ipuiK426	C S D	Интерпретатор
ipuiK427	C S	Обвязка для Meta
ipuiK311	C S	Копия DevCPS
ipuiK312	C S	Копия DevCPM
ipuiK421	C S	Генератор обёрточных модулей

2.1 Синтаксис КПд-1 (д — динамический)

```
StatementSeq   = Statement {";" Statement}.
Statement =  [ VAR VarDecl
      | Designator ":=" Expr
      | Designator ["(" [ExprList] ")"]
      ].
Expr    = ["^"] SimpleExpr [Relation SimpleExpr].
SimpleExpr  = ["+" | "-"] Term {AddOp Term}.
Term    = Factor {MulOp Factor}.
Factor  = Designator | number | character | string | NIL | Set | "(" Expr ")" | "~" Factor.
Set   = "{" [Element {"," Element}] "}".
Element   = Expr [".." Expr].
Relation   = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp =  "+" | "-" | OR.
MulOp =  "*" | "/" | DIV | MOD | "&".
Designator   = Qualidentident {"." ident | "[" ExprList "]" | "^" | "(" Qualident ")" | "(" [ExprList] ")" } ["$" ].
ExprList   = Expr {"," Expr}.
Qualident =  [ident "."] ident.

VarDecl    = IdentList ":" Type.
Type =  Qualident | BasicType | [POINTER TO] ARRAY [Expr] OF BasicType.
BasicType =  BOOLEAN | BYTE | SHORTINT | INTEGER | LONGINT | SHORTREAL | REAL
      | SET | ANYPTR | CHAR | SHORTCHAR.
IdentList   = IdentDefident {"," IdentDefident}.
```

Пояснения по изменениям в синтаксисе:

Statement. Объявление переменных вынесено в качестве оператора. Пока поддерживается только *StatementSeq* это упрощает грамматику.

Expr. Интерпретатор оперирует с элементами модулей, и прочими объектами через *Meta.Item*. Значения выражений, переменных и констант — динамические записи, производные от *Meta.Value* с полями соответствующего типа.

```
DEFINITION ipuiK427;
  IMPORT Meta;
  …
  TYPE
    BoolVal = RECORD (Meta.Value)
      x: BOOLEAN
    END;
    …
    MetaItemVal = RECORD (Meta.Value)
      x: Meta.Item
    END;
  …
END ipuiK427.
```

Таким образом переменная типа *Meta.Item* является ссылкой на значение произвольного выражения, в которой может оказаться не только значение переменной или константы, но и модуль или тип. Такую ссылку даёт операция "^" в Expr.

Ссылки на модули используются вместо импорта под псевдонимом. Сравните:

```
КП: IMPORT Log := StdLog;
КПд: VAR Log: Meta.Item; Log := ^ StdLog;
```

Для ссылок, как и указателей КП поддерживается неявное разыменовывание при доступе к элементу и индексации:

```
Log^.Ln <-> Log.Ln
VAR b: Meta.Item; b := ^ Dialog.appName;
Log.Char(b^[0]) <-> Log.Char(b[0])
```

Designator. Замена QualIdent на ident отражает факт того, что значением выражения может быть и модуль.
Приведение типа можно запрограммировать через механизм присваивания — этого пока не сделано.
Операция извлечения цепочки не имеет смысла: для простоты все литерные массивы рассматриваются как контейнеры цепочек в соответствующих операциях. Для КПд этого достаточно.

Type. Типы динамически конструировать через *Meta* нельзя. Тип любого значения (*item.obj* = *Meta.varObj*) является статическим типом поля *x* соответствующего расширения *Meta.Value* — см. выше *ipuiK427*. Интерпретатор поддерживает все простые скалярные типы и одномерные массивы простых типов.

Все массивы динамические и представляются указателями. Поэтому разница между объявлением массива и указателя состоит лишь в том, что первый требует обязательного указания положительного значения длины.

```
VAR buf: ARRAY 100 OF BYTE;
VAR pc: POINTER TO ARRAY OF CHAR;
```

```
DEFINITION ipuiK427;
  …
  TYPE
    PtrByteVal = RECORD (Meta.Value)
      x: POINTER TO ARRAY OF BYTE
    END;
```

```
      PtrCharVal = RECORD (Meta.Value)
         x: POINTER TO ARRAY OF CHAR
      END;
   …
END ipuiK427.
```

Значения прочих типов требует наличия соответствующих обёрток, которые ищутся:
- во-первых, в модуле типа с приставкой "Val";
- во-вторых, в "обвязочном" модуле с оригинальным именем. "Обвязочным" считается тестовый модуль №99.

Например, для *Files.Name*, сначала *Files.NameVal*, затем *System_devFiles_99.Name*.

IdentList. IdentDef заменён на ident, т. к. экспорт без конструкции модуля смысла не имеет.

2.2 Переменные

Переменные хранятся в односвязном циклическом списке с заголовком. Список упорядочен по возрастанию имён, заголовок — поисковый барьер.

```
TYPE
   VarList* = POINTER TO RECORD
      head: Var
   END;
   VarListVal* = RECORD (Meta.Value) x*: VarList END;

   Var = POINTER TO RECORD
      next: Var;
      name: ARRAY 64 OF CHAR;
      x: ANYPTR
   END;
```

Текущий список переменных перед выполнением добавляется в себя как переменная *LOCAL*.

```
PROCEDURE Exec*;
   VAR localVal: POINTER TO VarListVal;
BEGIN
   NEW(localVal); localVal.x := NewVarList(); PutVarVal(localVal.x, "LOCAL", localVal);
   Do(localVal.x)
END Exec;
```

Так его можно сохранить в глобальной переменной и затем использовать в другом скрипте.

```
ipuiK426.Exec  (* 1 *) … pi := 3.14 … Mod1.lc := LOCAL
ipuiK426.Exec  (* 2 *) Log.Real(Mod1.lc.pi) …
```

Интерпретатор в операции доступа к элементу распознаёт *VarListVal* и выполняет поиск в списке.

```
PROCEDURE OpenStruct (c: Context; IN name: Meta.Name; VAR x: M1.Value);
   VAR s: Meta.Item; res: M1.Value; p: ANYPTR; varL: BOOLEAN;
      …
BEGIN
   IF x.x.typ IN {Meta.anyPtrTyp, Meta.ptrTyp} THEN
      p := x.x.PtrVal();
      varL := (p # NIL) & (p IS VarList);
      IF varL  THEN
         p := Value(p(VarList), name);
         IF p = NIL THEN
            Error(c, 0)
         ELSE
            M1.SetItem(p, res.x); res.const := FALSE;
            x := res; LogValue(c, res, name)
         END
      END
   ELSE
   …
```

Неявное объявление. Поддерживается неявное объявление переменных при присваивании и формировании списка фактических параметров при вызове процедур (создаются переменные для OUT-параметров). Тип объявляемой переменной соответствует типу значения присваиваемого выражения или типу формального параметра.

С практической точки зрения этот подход видится естественным для динамики. На практике нареканий пока не вызывал, но однозначно утверждать о его необходимости нельзя.

Реализация проста. *GetIdent* при первом упоминании несуществующего идентификатора, который не является именем модуля, помещает имя в список переменных. Значение формируется после вычисления соответствующего выражения или перед вызовом процедуры.

2.3 Вызов методов

Методы *Meta* не поддерживает, поэтому для них нужно писать простые процедуры-обёртки.

```
TYPE Object = RECORD END;
PROCEDURE (VAR o: Object) Proc (parList);
->
PROCEDURE ObjectProc (VAR o: Object, parList);
```

Конструкция вызова метода интерпретируется как вызов процедуры-обёртки из "обёрточного" модуля. Если *item* связан с переменной и доступ к элементу *Lookup*(…) неуспешен, а искомое имя начинается с заглавной буквы (имя процедуры по соглашениям об оформлении исходников), то в "обёрточном" модуле выполняется поиск "обёрточной" процедуры. При успехе *item* сохраняется и вставляется как первый параметр перед её вызовом. Реализация отмечена.

```
        …
            s.Lookup(name, res.x); res.const := res.x.vis < Meta.exported;
            IF res.x.Valid() THEN
                x := res; LogValue(c, res, name)
            ELSE
                CheckMethod(c, x, name); IF x.this = NIL THEN Error(c, 0) END
            END
        END
    END
END OpenStruct;
```

2.4 Константы

Meta не поддерживает константы и их существование обеспечивают одноимённые переменные в "обёрточном" модуле.

```
MODULE X;
    CONST x* = 1;
END X;
->
MODULE Обёртка(X);
    VAR x-: INTEGER;
BEGIN
    x := 1
END Обёртка(X).
```

Операция доступа к элементу *OpenStruct*, в случае его необнаружения у модуля, пытается искать константу в "обёрточном". Реализация отмечена.

```
        IF s.obj = Meta.modObj THEN
            s.Lookup(name, res.x); res.const := res.x.vis < Meta.exported;
            IF res.x.Valid() THEN
                x := res; LogValue(c, res, name)
            ELSE
                CheckConst(s, name, res.x);
                IF res.x.Valid() & (res.x.obj = Meta.varObj) THEN
                    res.const := TRUE;
```

```
            x := res; LogValue(c, res, name + "(C)")
          ELSE
            Error(c, 0)
          END
        END
      …
```

2.5 Протокол выполнения

Протоколирование выделено в исходнике. В протокол распечатывается интерпретируемый текст и возникающие в процессе вычислений значения.

2.6 Выдача ошибок

Для пометки и показа ошибок используются штатные средства компилятора КП — *DevCPM.err* и *DevCPM.InsertMarks*.

Для некоторых ситуаций подходящих кодов ошибок КП нет, выбирались наиболее подходящие по смыслу. Например, длина массива в КП — целочисленное константное выражение (ошибки "константа не целочисленная" (42) и "недопустимое значение константы" (63)), а в КПд это обычное выражение.

3. Прототип-2. Тезисы

1) Переписывать "с чистого листа".

2) Расширить КПд-2.
2.1) Ввести управляющие конструкции — множественный выбор, цикл Дейкстры и REPEAT.
2.2) Реализовать приведение типа.

```
StatementSeq   =  Statement {";" Statement}.
Statement =  [ VAR VarDecl
      | Designator ":=" Expr
      | Designator ["(" [ExprList] ")"]
      | IF Expr THEN StatementSeq
        {ELSIF Expr THEN StatementSeq}
        [ELSE StatementSeq] END
      | WHILE Expr DO StatementSeq {ELSIF Expr DO StatementSeq} END
      | REPEAT StatementSeq UNTIL Expr
      ].

VarDecl   =  IdentList ":" Type.
Type =  Qualident | [POINTER TO] ARRAY [Expr] OF Type.
IdentList   =  ident {"," ident}.

Expr    =  ["^"] SimpleExpr [Relation SimpleExpr].
SimpleExpr   =  ["+" | "-"] Term {AddOp Term}.
Term    =  Factor {MulOp Factor}.
Factor   =  Designator | number | character | string | NIL | Set | "(" Expr ")" | "~" Factor.
Set   =  "{" [Element {"," Element}] "}".
Element   =  Expr [".." Expr].
Relation   =  "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp =  "+" | "-" | OR.
MulOp =  "*" | "/" | DIV | MOD | "&".
Designator   =  ident {"." ident | "[" ExprList "]" | "^" | "(" Qualident ")" | "(" [ExprList] ")" }.
ExprList   =  Expr {"," Expr}.
Qualident =  [ident "."] ident.
```

2.3) Введение процедур/функций желательно, но требует отдельного продумывания (формирование контекстов). Открыт вопрос о типизации аргументов. Оставить на потом…

3) Основной пункт: упрощение и обобщение реализации за счёт разделения этапов разбора и выполнения. Выполнять на стэковой машине, оперирующей *Meta.Item*.

3.1) Номенклатура операций машины будет шире операций КП. Учитывая динамику процесса можно и нужно максимум действий вынести на операции. Туда должно уйти фактически всё.

3.1.1) Унарные:
- отрицание "~" Factor;
- смена знака "-" Term;
- ссылка "^" SimpleExpr …;
- разыменовывание Designator "^";
- множество-1 "{" Element "}".

3.1.2) Бинарные:
- отношения "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS;
- аддитивные "+" | "-" | OR;
- мультипликативные "*" | "/" | DIV | MOD | "&";
- получение элемента Designator "." ident;
- множество-2 "{" Expr ".." Expr "}";
- присваивание Designator ":=" Expr.

3.1.3) n-арные
- "вызов" Designator "(" ExprList ")" -> пар1 … парN N Процедура "вызов"
- "индекс" Designator "[" ExprList "]" -> пар1 … парN N Массив "индекс"
- "Объявление" переменных IdentList ":" Type:
  - "скаляры" (имя1, …, имяN, тип) -> имя1 ... имяN N тип "скаляры";
  - "векторы" (имя1, …, имяN, типЭлемента, размер) -> имя1 ... имяN N типЭлемента "векторы".

Под "скаляром" условно понимается одно значения типа Qualident. Типы элементов вектора — простые. Простые типы вводятся как предопределённые (по аналогии с LOCAL) ссылки на тип в спецмодуле:

```
MODULE BasicTypes;
   IMPORT Meta;
   TYPE
      Integer* = INTEGER;
      …
   VAR
      metaInteger-: Meta.Item;
      …
BEGIN
   Meta.LookupPath("BasicTypes.Integer", metaInteger); ASSERT(metaInteger.Valid());
   …
END BasicTypes.
```

4) Реализацию списков переменных *ipuiK426.VarList* вынести в отдельный библиотечный модуль, реализующий ассоциативный массив ANYPTR-в.