УДК 519.633.9

ВСТРОЕННЫЙ ЯЗЫК ДЛЯ ИНСТРУМЕНТАЛЬНОГО КОМПЛЕКСА, ОРИЕНТИРОВАННОГО НА ИНТЕРАКТИВНУЮ РАБОТУ СО СЛОЖНЫМИ СТРУКТУРАМИ ДАННЫХ

О. Б. Арушанян[1], **Н. А. Богомолов**[1], **А. Д. Ковалев**[1]

Рассматривается язык описания данных и сценариев их обработки, используемый в рамках разрабатываемого в НИВЦ МГУ программного инструментального комплекса, предназначенного для автоматизации создания приложений, ориентированных на интерактивную работу со сложными структурами данных. Обсуждаются концепция разработки, особенности реализации, а также различные аспекты использования встроенного языка в рамках инструментального комплекса. Работа выполнена при поддержке грантов РФФИ (коды проектов 04–07–90288 и 05–07–90328).

1. Общие замечания. В состав описанного в [1] инструментального программного комплекса (далее ИК), реализованного в среде Delphi [2] на языке Object Pascal [3] и предназначенного для автоматизации создания приложений, ориентированных на интерактивную работу со сложными структурами данных, входит встроенный язык описания данных и сценариев их обработки, названный SPL (Script Programming Language).

Наличие собственного встроенного языка программирования позволяет использовать внутренние структуры данных компилятора для описания структур прикладных данных встроенной базы данных ИК. Кроме того, собственный встроенный язык программирования обеспечивает наиболее тесную интеграцию программ, написанных на встроенном языке с программами на базовом языке реализации ИК.

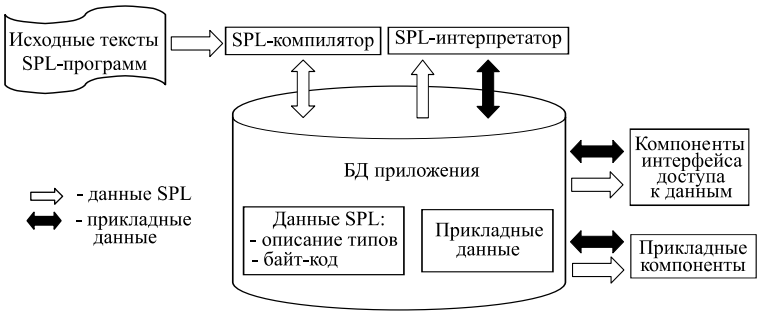

Рис. 1. Схема использования SPL

При разработке языка SPL использовался принцип "минимальной достаточности", в соответствии с которым набор выразительных средств языка был ограничен потребностями ИК. Однако получившийся в результате язык оказался в большой мере универсальным процедурным объектно-ориентированным языком программирования. Он обладает характерными для современных языков программирования средствами описания типов данных и алгоритмов, а также средствами оформления программных элементов (классы, процедуры, функции). Кроме того, реализация языка SPL такова, что позволяет легко наращивать его функциональность за счет подключения новых встроенных функций, создаваемых на базовом языке реализации ИК (Object Pascal).

Язык SPL выполняет в ИК одновременно две инструментальные функции: описание типов прикладных данных, используемых в процессе работы приложения, и описание сценариев обработки данных.

Язык SPL тесно интегрирован с другими компонентами ИК (рис. 1). Результатом работы SPL-компилятора являются структуры данных, которые хранятся в специальном разделе встроенной базы данных ИК (далее БД). Эти структуры данных содержат описания объявленных средствами SPL типов данных и глобальных переменных, а также байт-код программных элементов: процедур, функций и методов классов. Выполнение откомпилированных программных элементов осуществляется SPL-интерпретатором байт-кода.

Информация, полученная в результате работы SPL-компилятора, активно используется различными компонентами ИК в процессе работы приложения. Прежде всего, эта информация необходима СУБД ИК

[1] Научно-исследовательский вычислительный центр, Московский государственный университет им. М.В. Ломоносова, 119992, Москва; e-mail: arush@srcc.msu.su, nbogom@srcc.msu.su, kovalev@srcc.msu.su

при создании и манипулировании экземплярами объектов БД. Кроме того, она используется компонентами, обеспечивающими пользовательский интерфейс доступа к данным.

Сценарии, реализованные на языке SPL, предназначены для обеспечения гибкой настройки бизнес-логики готового (созданного средствами ИК) приложения, без его перекомпиляции. Одним из возможных применений SPL-сценариев является реализация алгоритмов обработки прикладных данных. Другим возможным применением SPL-сценариев служит создание или изменение управляющих структур данных, которые контролируют работу отдельных блоков приложения. В этом случае текст SPL-сценария играет роль пользовательского интерфейса по заданию содержимого этих управляющих структур данных.

Синтаксис SPL основан на синтаксисе языка Object Pascal [3]. Однако средства описания производных типов SPL обладают рядом специальных возможностей, обусловленных спецификой ИК (прежде всего, необходимостью определения структур прикладных данных СУБД ИК).

Программы, написанные на языке SPL, могут эффективно взаимодействовать с программами, реализованными на языке Object Pascal. Это достигается за счет того, что внутреннее представление встроенных и производных типов данных языка SPL в точности соответствует внутреннему представлению языка Object Pascal. Такое соответствие позволяет без каких-либо накладных расходов обрабатывать структуры данных, созданные с помощью SPL-программ, в программах Object Pascal и, наоборот, обращаться к полям структур данных Object Pascal из SPL-программ.

Интеграция программ, реализованных на SPL и Object Pascal, осуществляется не только на уровне доступа к данным, но и на уровне программного кода. Так, имеется возможность непосредственного вызова из SPL-программ процедур и функций, реализованных на языке Object Pascal. Эти процедуры должны быть написаны по определенным правилам с учетом того, что их вызов осуществляется из SPL-интерпретатора. Синтаксис вызова таких процедур в SPL совпадает с синтаксисом вызова процедур и функций самого языка SPL. Встраивание в SPL подобных процедур, реализованных на языке Object Pascal, позволяет расширять возможности базового SPL с учетом потребностей конкретного приложения.

В ИК имеются также средства вызова SPL-процедур и SPL-функций из программ на Object Pascal. При этом вызываемым процедурам и функциям можно передать необходимые параметры.

2. Основные синтаксические правила записи программ. Язык SPL имеет следующие основные синтаксические правила записи программ:

— все используемые типы, переменные, функции и процедуры должны быть объявлены или описаны до их первого использования;

— при записи идентификаторов могут использоваться латинские и русские буквы, цифры и символ подчеркивания; прописные и строчные буквы в описании идентификаторов эквивалентны; длина идентификаторов не ограничена;

— для доступа к элементам объектов (полям записей и классов) и к методам классов используется нотация с символом-разделителем между именем объекта и именем поля; в случае записи таким разделителем является символ "." (Rec.Field); в случае класса таким разделителем является символ "!" (Obj!Field или Obj!Func(...));

— каждое предложение языка заканчивается символом ";", в одной строке текста программы может быть расположено несколько предложений языка;

— комментарии в тексте программы заключаются в фигурные скобки "{...}"; комментарий может быть введен в любом месте текста программы и может занимать как произвольную часть одной строки, так и несколько строк; еще один способ введения комментария: размещение его в конце строки после двух символов "слеш" ("//");

— операторные скобки begin...end выделяют составной оператор.

3. Структура модуля. Тексты SPL-программ размещаются в оформленных специальным образом текстовых фрагментах — модулях. Модуль является единицей компиляции. Тексты модулей могут храниться во внешних файлах, полях базы данных или динамически создаваться в процессе работы приложения.

Каждый модуль в общем случае имеет следующую структуру:

```
unit <идентификатор модуля>;
uses <список подключаемых модулей>;
<секция глобальных объявлений типов>
var
<секция глобальных объявлений переменных>
<секция реализации>
initialization
```

<секция инициализации>

Идентификатор модуля представляет собой путь к объекту БД, который создается в результате компиляции данного модуля. Таким образом, идентификатор модуля позволяет управлять размещением данных компиляции в БД.

Предложение uses содержит список перечисленных через запятую идентификаторов подключаемых модулей, результаты компиляции которых необходимы при компиляции данного модуля. Предложение uses может отсутствовать, если данный модуль содержит все необходимые для собственной компиляции объявления. При взаимных ссылках модулей друг на друга с помощью предложения uses запрещаются циклические ссылки.

Секция глобальных объявлений типов содержит объявления типов, доступ к которым возможен из других модулей.

Секция объявлений глобальных переменных начинается с ключевого слова var и служит для объявления глобальных переменных, доступ к которым возможен из других модулей.

В секции реализации размещаются описания всех процедур, функций и методов, объявленных в данном модуле классов. Секция реализации может отсутствовать, если данный модуль содержит только глобальные объявления производных типов данных и глобальных переменных.

Секция инициализации необязательна. Она содержит операторы, осуществляющие начальную настройку глобальных переменных модуля. Результат компиляции секции инициализации представляет собой процедуру без параметров с именем Init. Эта процедура автоматически вызывается один раз после завершения компиляции данного модуля. Оформление результатов компиляции секции инициализации в виде обычной процедуры позволяет в случае необходимости повторно выполнять код инициализации переменных модуля за счет явного вызова этой процедуры.

4. Переменные. Переменная представляет собой идентификатор, с которым связана область памяти, предназначенная для хранения значения переменной. Размер этой области памяти определяется типом переменной. Объявление переменных выполняется в секции объявлений, которая начинается с ключевого слова var, в следующей форме:

```
var   <идентификатор переменной> : <имя типа>;
      ...
      <идентификатор переменной> : <имя типа>
```

В теле модуля могут существовать несколько секций с объявлениями переменных. В секции объявлений глобальных переменных модуля перечислены все глобальные переменные, к которым может быть обеспечен доступ из других модулей. Они размещаются в специальном объекте данных, который создается в БД при компиляции модуля и сохраняется для использования до конца работы приложения или до момента удаления из БД объекта данных модуля.

Объявления переменных могут встречаться в описаниях процедур и функций. Перечисленные там переменные называются локальными. Память для размещения этих переменных выделяется только на время вызова соответствующих процедур или функций и автоматически освобождается после возврата из них.

5. Процедуры и функции. Процедуры и функции представляют собой программные элементы, выполнение которых может быть инициировано либо из других программных элементов, либо из программ на Object Pascal. Описание программного элемента имеет следующий синтаксис:

```
<заголовок>
<секция объявления переменных>
<секция объявления меток>
begin
  <тело программного элемента>
end
```

Секция объявления локальных переменных программного элемента может отсутствовать. Секция объявления меток начинается с ключевого слова label, за которым следует список идентификаторов меток, разделенных символом ",". Завершается список меток символом ";". Секция объявления меток программного элемента может отсутствовать. Заголовок программного элемента имеет для функций и процедур следующую форму:

```
procedure <имя процедуры>
        (<секция формальных параметров>);
```

```
function <имя функции>
        (<секция формальных параметров>) :
<имя типа возвращаемого значения>
```

Секция формальных параметров может отсутствовать в заголовке, однако обрамляющие круглые скобки должны оставаться и в случае отсутствия у программного элемента формальных параметров.

Функция после завершения вызова возвращает значение определенного типа, заданного в поле описания <имя типа возвращаемого значения>. Функция может использоваться в выражениях наряду с константами и переменными. При этом возвращаемое после вызова функции значение подставляется в соответствующее место выражения (например, a := b + Func();).

Однако функция может использоваться и как процедура вне выражения (например, Func();). В этом случае возвращаемое значение функции игнорируется. Для задания возвращаемого значения в теле функции необходимо изменить содержимое предопределенной переменной result, например:

```
function IncVar(v : integer; delta : integer) : integer;
begin
    result := v;
    result := result + delta;
end
```

Вызов программного элемента завершается после выполнения его последнего оператора. Для завершения вызова программного элемента в произвольном месте его тела необходимо выполнить оператор exit, например:

```
function DecVar(v : integer) : integer;
begin
    result := v;
    if (result LE 0) then exit;
    result := result - 1;
end
```

Программный код процедуры и функции может оперировать в процессе выполнения с тремя группами переменных: глобальные переменные, локальные переменные программного элемента и формальные параметры программного элемента. К глобальным переменным, доступным в теле программного элемента, относятся переменные, объявленные в секции глобальных переменных того модуля, в котором описан этот программный элемент, а также глобальные переменные, объявленные в модулях, перечисленных в предложении uses этого модуля.

Переменные, перечисленные в секции объявлений в описании программного элемента, являются локальными переменными данного программного элемента и доступны только в теле данного программного элемента. Они создаются в момент его вызова и удаляются по завершении вызова.

Формальные параметры программного элемента — множество переменных, фактические значения которым присваиваются в момент вызова программного элемента. Формальные параметры, как и локальные переменные, существуют только во время выполнения вызова программного элемента. Секция описания формальных параметров имеет следующую форму:

```
<имя параметра> : <имя типа>;
...
<имя параметра> : <имя типа>
```

Оператор вызова программного элемента представляет собой имя программного элемента, за которым следует заключенный в круглые скобки список разделенных символом “,” фактических значений формальных параметров. Порядок значений в этом списке должен соответствовать порядку формальных параметров в заголовке программного элемента, например:

```
function IncVar(v : integer; delta : integer) : integer;
...
v := IncVar(v, d)
```

Передача фактических значений формальным параметрам в момент вызова программного элемента осуществляется по значению. Как и для локальных переменных, для фактических параметров выделяется временная память, а затем фактические значения пересылаются в эту временную память. На этом связь

переменных, которые были источником фактических значений, с формальными параметрами программного элемента "заканчивается". Любые изменения значений формальных параметров в процессе вызова программного элемента никак не сказываются на значениях переменных, которые были источником их фактических значений. Для того чтобы в процессе работы программного модуля можно было изменить значение некоторой переменной, которая передается в программный элемент в качестве параметра, необходимо передавать программному элементу не значение этой переменной, а указатель на нее. В этом случае необходимо соответствующим образом изменить и тип формального параметра, например:

```
procedure IncVar(pv : ^integer; delta : integer) : integer;
begin
        value(pv) := value(pv) + delta;
end;
...
IncVar(Adr(v), 10)
```

В этом примере процедура IncVar может изменять значение переменной, адрес которой передается ей через формальный параметр pv.

6. Операции. Операции — это встроенные в язык действия над операндами, которые могут комбинироваться в определенную последовательность, называемую выражением. Операндами операций могут быть константы, переменные, функции и сами выражения. Большинство операций имеют два операнда, один из которых помещается до знака операции, а другой — после. Существуют и унарные операции, имеющие один операнд, который размещается после знака операции. Последовательность выполнения операций в выражениях определяется скобками и приоритетом операций.

6.1. Арифметические операции (табл. 1). При выполнении арифметических операций операнды должны иметь совпадающий тип. Для выполнения арифметических операций над операндами различных типов необходимо вначале выполнить приведение типов операндов к единому типу.

Таблица 1

Бинарные арифметические операции

Обозначение	Операция	Типы операндов	Тип результата	Пример
+	сложение	целый, действительный	целый, действительный	X+Y
-	вычитание	целый, действительный	целый, действительный	X-Y
*	умножение	целый, действительный	целый, действительный	X*Y
/	деление	действительный	действительный	X/Y

Такие операции, как целочисленное деление и остаток целочисленного деления, реализованы в виде встроенных функций.

Определена одна унарная арифметическая операция — унарный минус, которая обеспечивает изменение знака операнда и применяется для действительных и целых операндов (-X).

6.2. Операции отношения (табл. 2). Операции отношения обеспечивают сравнение двух операндов. Поскольку в SPL отсутствует логический тип, операции отношения возвращают в качестве значения "истина" число 0, а в качестве значения "ложь" — число, имеющее единицы во всех двоичных разрядах (-1).

При выполнении операций отношения операнды должны иметь совпадающий тип. Для сравнения операндов различных типов необходимо вначале выполнить приведение типов операндов к единому типу. Для сравнения указателей и экземпляров классов необходимо привести их к целому типу, например:

```
if (Integer(P1) NE Integer(P2)) then exit
```

6.3. Логические поразрядные операции (табл. 3). Поскольку в SPL отсутствует логический тип, логические поразрядные операции можно использовать как логические операции. При этом в качестве константного значения "истина" необходимо использовать число 0, а в качестве константного значения "ложь" необходимо использовать число, имеющее единицы во всех двоичных разрядах (-1).

Таблица 2

Операции отношения

Обозначение	Операция	Типы операндов	Тип результата	Пример
EQ	равно	целый, действительный, строка	логический (целый)	X EQ Y
NE	не равно	целый, действительный, строка	логический (целый)	X NE Y
LT	меньше	целый, действительный	логический (целый)	X LT Y
GT	больше	целый, действительный	логический (целый)	X GT Y
LE	меньше или равно	целый, действительный	логический (целый)	X LE Y
GE	больше или равно	целый, действительный	логический (целый)	X GE Y

Таблица 3

Логические поразрядные операции

Обозначение	Операция	Типы операндов	Тип результата	Пример
not	поразрядное отрицание	целый	целый	not X
and	поразрядное И	целый	целый	X and Y
or	поразрядное ИЛИ	целый	целый	X or Y
xor	поразрядное исключающее ИЛИ	целый	целый	X xor Y

7. Операторы.

7.1. Оператор присваивания. Оператор присваивания имеет следующий синтаксис:

<переменная> := <выражение>,

где <переменная> — переменная любого типа, а <выражение> — любое допустимое выражение, тип которого совпадает с типом переменной, указанной в левой части. В случае несовпадения типов левой и правой частей оператора необходимо привести тип выражения в правой части к типу переменной, указанной в левой части. Для записи значения по адресу, заданному указателем на переменную, необходимо в левой части оператора присваивания поместить вызов встроенной функции value, которая обеспечивает преобразование значения указателя в значение переменной, например:

```
var
P : ^integer;
A : integer;
...
value(P) := a,
```

где P — указатель на переменную.

7.2. Оператор передачи управления goto. Оператор goto позволяет изменить последовательность выполнения операторов процедуры или функции посредством передачи управления в произвольную точку программного элемента, помеченную меткой. Метки задаются в секции объявления меток программного элемента (см. раздел 5).

Точка, в которую может передать управление оператор goto, помечается идентификатором метки, за которым следует символ ":". Управление передается на оператор, располагающийся в теле программного элемента непосредственно за меткой. При этом оператор перехода на выбранную метку может размещаться как до, так и после помеченного места, например:

```
goto <идентификатор метки>;
...
```

<идентификатор метки>:
<оператор на который осуществляется переход>;
...
goto <идентификатор метки>

7.3. Условный оператор выбора if. Оператор if предназначен для обеспечения условного выполнения фрагментов программного кода. Оператор имеет две формы:

if (<условие>) then <оператор1>

или

if (<условие>) then <оператор1> else <оператор2>

Ключевое слово then является необязательным, а скобки после if — обязательными. Если условие истинно (выражение в скобках равно целому числу 0), то и в той и в другой форме оператора if будет выполняться <оператор1>. Если же условие ложно (выражение в скобках не равно целому числу 0), то <оператор1> выполняться не будет (во второй форме оператора if в этом случае будет выполнен <оператор2>).

При использовании вложенных конструкций if компилятор считает, что очередной else относится к последнему оператору if. Если надо изменить порядок проверки условий во вложенном операторе if, то необходимо использовать в этой конструкции составные операторы, например:

if (<условие1>) then begin if (<условие2>) then <оператор1> end
else <оператор2>

7.4. Оператор цикла while. Оператор while предназначен для обеспечения циклического выполнения оператора (в том числе и составного), называемого телом цикла, при выполнении определенного условия. Оператор while имеет форму

while (<условие>) do <оператор>

Ключевое слово do является необязательным, а скобки после while — обязательными. Оператор работает следующим образом. Сначала вычисляется выражение <условие>. Если результатом вычисления этого выражения является "истина" (целое число, равное 0), то выполняется оператор тела цикла, после чего вновь вычисляется выражение, определяющее условие. Такое повторение тела цикла и проверки условия продолжаются до тех пор, пока выражение, вычисляющее условие, не получит значение "ложь" (целое число, не равное 0). После этого управление передается оператору, следующему за оператором while.

Поскольку вычисление и проверка условия выполняются в начале оператора while, то тело цикла может не выполниться ни разу.

7.5. Операторы break, continue, exit. Прерывание выполнения цикла осуществляется с помощью оператора break. При выполнении оператора break управление сразу же (без проверки условия завершения) передается оператору, следующему за оператором цикла.

Оператор continue прерывает выполнение текущей итерации цикла. При выполнении оператора continue управление передается на начало очередной итерации цикла.

Оператор exit позволяет прервать выполнение процедуры или функции. Выполнение оператора exit вызывает немедленное завершение выполнения последовательности операторов процедуры или функции, в том числе и операторов цикла.

8. Типы данных в языке SPL.

8.1. Встроенные и производные типы. Средства описания типов данных в SPL позволяют определять структуры данных, типичные для большинства современных языков программирования (C, Pascal). Типы данных в SPL можно разделить на две категории: предопределенные (встроенные) и производные (определяемые пользователем). Классификация типов языка SPL приведена на рис. 2.

К предопределенным типам относятся различные виды целых (1, 2, 4, 8 байтов — byte, int1, uint2, short, integer, hex, int64) и действительных чисел (4, 8 байтов — float, double), цвет (color), строка (string), объект БД ИК (TN_UDBase), дата (TDate), откомпилированный код типа (TypeCode) и др. В состав предопределенных типов данных включен ряд структур, которые могли бы быть реализованы и как производные типы. К ним относятся пары и четверки чисел, описывающие соответственно положение точки и прямоугольной области. Точка и прямоугольник могут иметь целые четырехбайтовые координаты (TPoint, TRect), а также два типа действительных координат — четырех- и восьмибайтовые (TFPoint, TFRect, TDPoint, TDRect).

В SPL отсутствует логический тип. Для хранения логического значения может быть использован любой целый тип, при этом значение нуль интерпретируется как "истина", а значение, неравное нулю, как "ложь".

Средства описания производных типов данных позволяют определять следующие производные типы: перечисления, множества, записи, динамические массивы, процедурные типы.

Объявление производных типов осуществляется с помощью ключевого слова type, за которым следует собственно объявление в форме

type <имя типа> = <описание типа>

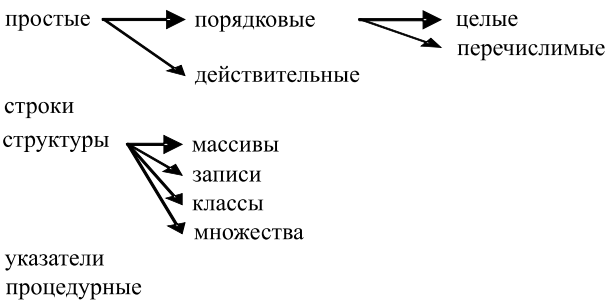

Рис. 2. Классификация типов

Объявление типа может быть использовано для создания псевдонима типа. В этом случае вместо описания типа необходимо указать имя ранее определенного или встроенного типа:

type <имя нового типа> = <имя типа>

SPL поддерживает операцию преобразования типа переменной или выражения в другой тип. Преобразование (приведение) типов осуществляется явным образом с помощью следующей конструкции:

<имя типа> (<выражение>)

В SPL введен абстрактный встроенный тип undef, означающий любой тип. Абстрактный тип undef используется в объявлениях процедур и функций для указания того, что фактический параметр и возвращаемое значение функции могут быть различного типа, а также при объявлении массивов и указателей в том случае, если они предназначены для хранения массивов различного типа и соответственно указателей на переменные различного типа. Возможность в качестве результата работы процедур и функций получить переменную типа arrayof undef или ^undef (указатель на undef) обусловило необходимость включения в состав типов, встроенных в SPL, типа данных, соответствующего значению кода типа — TypeCode. Специальные встроенные функции позволяют получить значение кода типа переменных. Возможность манипулирования объектами, тип которых не определен в момент компиляции программы, особенно необходима при написании SPL-программ, работающих с полями объектов БД.

8.2. Строки. Строка представляет собой последовательность символов произвольной длины (до двух Гбайт). Конкатенация строк осуществляется операцией "+". Таким образом, для конструирования строковых переменных можно использовать выражения вида

$$\mathrm{NewStr} := \mathrm{Str1} + \ldots + \mathrm{StrN},$$

где Str1, ... , StrN могут быть не только переменными, но и литеральными константами, а также функциями, возвращающими строковые значения. Встроенная функция SLength возвращает длину строки. Для получения новой строки, являющейся фрагментом другой строки, служит встроенная функция SubString, которой в качестве параметра передается исходная строка, индекс первого символа и количество символов в искомой подстроке. Индекс начального символа строки равен 0.

В SPL реализован ряд функций, обеспечивающих преобразование в текстовую строку значений переменных и выражений. Встроенная функция ToString обеспечивает неформатное преобразование значения переменной любого, в том числе и производного, типа в текстовую строку. Встроенная функция format выполняет форматное преобразование переменной или выражения простого типа (целое, действительное, строка). В качестве параметров функции format передается форматная строка и преобразуемое значение.

8.3. Массивы. Объявление массива осуществляется с помощью ключевого слова arrayof. Массив в SPL — это одномерный динамический массив элементов заданного типа. Объявление динамического массива можно осуществить как явным объявлением типа переменной:

var <имя переменной> : arrayof <имя типа элемента>,

так и предварительным объявлением нового типа данных, а затем использованием этого типа при объявлении переменной:

type <имя производного типа> = arrayof <имя типа элемента>;
var <имя переменной> : <имя производного типа>

Доступ к элементам динамического массива с именем A осуществляется с помощью выражения A[i], где i — индекс, являющийся целым числом, начинающимся с 0. Для поддержки работы с динамическими массивами в SPL имеется ряд встроенных функций.

Массив в SPL фактически представляет собой указатель на некоторый объект, который управляет памятью, выделяемой для размещения элементов массива, хранит информацию о количестве элементов и их типе. При объявлении массива память для хранения его элементов не отводится. До использования массива необходимо задать его размер с помощью встроенной функции SetLength. Эта функция и осуществляет конструирование экземпляра массива.

Определить длину массива можно с помощью встроенной функции ALength. Встроенная функция SubArray позволяет получить новый массив, содержащий копию указанных элементов исходного массива. Встроенные функции InsElements и DelElements могут изменить длину уже существующего массива, вставляя в указанное место массива заданное число элементов или удаляя указанный фрагмент массива.

Если тип элементов массива объявлен как неопределенный (arrayof undef), то может возникнуть необходимость определить фактический тип элементов в процессе выполнения программы. Это можно осуществить с помощью функций ATypeCode или ATypeName, которые возвращают соответственно либо значение типа TypeCode, либо строку с именем типа элемента.

Если тип элемента массива явно задан в программе при объявлении переменной, то функция SetLength для этой переменной создает экземпляр массива с элементами соответствующего типа. Для создания экземпляра массива, тип элемента которого определяется в процессе выполнения программы, служит встроенная функция-конструктор массива NewArray. Этой функции в качестве параметра передается строка с именем или с кодом типа элементов массива (TypeCode).

Язык SPL позволяет работать и с многомерными массивами. Многомерный массив в SPL моделируется одномерным массивом, элементом которого служит другой массив. Для объявления многомерного массива необходимо завести новый тип данных, который является массивом, а затем использовать этот тип данных для объявления одномерного массива, элементом которого будет динамический массив. Обращение к элементам двумерного массива с именем MA осуществляется с помощью выражения MA[i][j], где i — индекс по массиву MA и j — индекс по массиву, являющемуся элементом MA.

8.4. Записи. Запись представляет собой структуру данных, состоящую из набора полей, имеющих определенные типы данных и уникальные для этой записи имена. Записи могут использоваться как переменные в SPL-программах и как содержимое объектов-контейнеров прикладных данных в БД ИК. В SPL реализованы записи двух типов — обычные и вариантные. В обычных записях поля размещаются в памяти последовательно друг за другом. Объявление этого типа записи осуществляется в следующей форме:

```
type <имя типа записи> = record
        <квалификатор 1> <имя поля 1> : <имя типа поля 1>;
        ...
        <квалификатор N> <имя поля N> : <имя типа поля N>;
end
```

В вариантных записях поля “перекрываются”, размещаясь в начале памяти, занимаемой записью. Объявление этого типа записи осуществляется в следующей форме:

```
type <имя типа записи> = vrecord
        <квалификатор 1> <имя поля 1> : <имя типа поля 1>;
        ...
        <квалификатор N> <имя поля N> : <имя типа поля N>;
end
```

Использование определения типа записи для описания структуры прикладных данных объектов БД обусловливает необходимость такого элемента описания полей записи, как квалификатор. Квалификатор задает некоторые свойства поля записи, которые необходимы, прежде всего, для прикладных полей объектов БД, и является необязательным элементом описания. Квалификатор может принимать следующие значения: variant, runtime, obsolete. Квалификатор variant позволяет объявить внутри обычной записи группу подряд идущих вариантных полей. Все поля такой группы имеют одинаковое смещение от начала записи. Квалификатор runtime позволяет объявить поля, значения которых не сохраняются в файле БД при завершении сеанса работы приложения. Квалификатор obsolete облегчает работу по реструктуризации существующих БД. Он позволяет присвоить полю признак “устаревшего” и не использовать это поле

при актуализации базы данных, сериализованной во внешнем файле с применением "устаревшей" схемы данных.

Доступ к отдельным полям переменной, в которой хранится запись, осуществляется с помощью следующего выражения: <имя переменной>.<имя поля>.

8.5. Указатели. Указатель предназначен для хранения адреса памяти, в которой хранится экземпляр данных. Объявление указателя можно осуществить посредством как явного объявления типа переменной (var <имя переменной> : ^<имя типа данных>), так и предварительного объявления нового типа данных, а затем использования этого типа при объявлении переменной:

```
type <имя производного типа> : ^<имя типа данных>;
var <имя переменной> : <имя производного типа>
```

Указатели широко используются в SPL-программах, работающих с полями объектов БД. Встроенные функции UDP и UDRP обеспечивают доступ к полю объекта БД. Эти функции возвращают типизированный указатель на поле объекта БД с использованием строки, описывающей путь к полю БД.

Доступ к данным по указателю осуществляется с помощью встроенной функции value(P), где P — указатель на переменную или поле БД. Конструкция value(P) может использоваться и в левой части оператора присваивания. Если же указатель содержит адрес записи, то доступ к отдельным полям записи осуществляется с помощью следующего выражения:

```
<имя переменной указателя>!<имя поля>
```

Для получения адреса данных, хранящихся в некоторой переменной, используется встроенная функция adr, возвращающая адрес переменной, переданной ей в качестве параметра:

```
var
I : integer;
PI : ^integer;
...
PI := adr(I);
...
```

В переменной-указателе сохраняется только адрес памяти и отсутствует информация о типе данных, на которые "смотрит" указатель. Однако реализация SPL-интерпретатора предполагает, что в стек вызова процедур и функций должны попадать не только данные, но и информация об их типе. Такая реализация SPL-интерпретатора позволяет осуществлять динамический контроль соответствия типов, а также реализовывать встроенные функции, специфика выполнения которых зависит от типа данных, переданных им в качестве параметров (примером такой функции может служить встроенная функция sizeof, возвращающая размер памяти, необходимой для размещения переменной заданного типа). Для переменных-указателей, тип которых явно задан в определении переменных, дополнительных действий не требуется.

При использовании переменных-указателей, имеющих неопределенный тип (var P = ^undef), может появиться необходимость конструирования типизированного указателя для передачи в качестве параметров некоторой функции. Эту задачу выполняет встроенная функция TPtr, которой в качестве параметра передается сам указатель и код типа или строка с именем типа. Встроенная функция PType, наоборот, позволяет определить тип указателя, переданного ей в качестве параметра, и сохранить значение кода типа или строку с именем типа данных, которые идентифицированы данным указателем.

8.6. Перечислимые типы. Перечислимые типы определяют упорядоченное множество поименованных элементов. Переменная перечислимого типа может принимать значение, соответствующее одному из возможных элементов множества, заданного в объявлении перечислимого типа. Объявление перечислимого типа можно осуществить в следующей форме:

```
type <имя перечислимого типа> =
        (<идентификатор элемента 1> = <название элемента 1>,
        ...
        <идентификатор элемента N> = <название элемента N>) :
<имя простого типа>
```

Необязательный элемент описания "название элемента" позволяет задать в описании типа название, которое может быть использовано компонентами ИК, обеспечивающими визуальное представление и редактирование экземпляров данных этого типа.

Переменная перечислимого типа занимает в памяти целое число байтов, достаточных для представления значений данного перечислимого типа. Необязательный элемент описания "имя простого типа" используется для изменения размера памяти, выделяемой для хранения переменных данного типа. Если задан элемент описания "имя простого типа" и размер переменной этого простого типа превышает размер, необходимый для размещения значений перечислимого типа, то в качестве реального размера перечислимого типа берется размер этого простого типа. В приведенном ниже примере переменная перечислимого типа Enum1, количество возможных значений которой равно трем, будет занимать в памяти один байт, а переменная перечислимого типа Enum2, количество возможных значений которой тоже равно трем, будет занимать в памяти четыре байта:

```
type Enum1 = (en1V1, en1V2, en1V3);
type Enum2 = (en2V1, en2V2, en2V3) : integer
```

Для работы с переменными перечислимого типа в SPL реализован ряд встроенных функций. Например, функция ord, которой в качестве параметра передается переменная перечислимого типа, возвращает порядковый номер значения переменной в множестве элементов, заданных в объявлении типа.

8.7. Множества. Множество в SPL, как и перечислимый тип, определяет множество поименованных элементов. В отличие от перечислимого типа, значению экземпляра множества соответствует подмножество того множества элементов, которое было задано в объявлении типа. Объявление множества осуществляется с помощью ключевого слова setof в следующей форме:

```
type <имя производного типа> = setof
        (<идентификатор элемента 1> = <название элемента 1>,
        ...
        <идентификатор элемента N> = <название элемента N>) :
<имя простого типа>
```

Необязательный элемент описания "название элемента" позволяет задать в описании типа названия, которые могут быть использованы компонентами ИК, обеспечивающими визуальное представление и редактирование экземпляров данных этого типа.

В объявлении множества можно вместо явного перечисления элементов множества сослаться на перечислимый тип, который в этом случае и будет определять множество элементов, соответствующих объявляемому типу:

```
type <имя производного типа> = setof
        <имя перечислимого типа> :
<имя простого типа>
```

В SPL множество моделируется битовой шкалой, в которой каждому элементу множества соответствует один бит шкалы. Множество занимает в памяти целое число байтов, достаточное для размещения соответствующей битовой шкалы. Необязательный элемент описания "имя простого типа" используется для изменения размера памяти, выделяемой для хранения переменных данного типа. Если задан элемент описания "имя простого типа" и размер переменной этого простого типа превышает размер, необходимый для размещения значений множества, то в качестве реального размера берется размер этого простого типа. В приведенном ниже примере тип Set1 определяет множество, количество элементов в котором равно пяти. Экземпляр такого множества будет занимать в памяти один байт. Тип Set2 определяет множество, количество элементов в котором тоже равно пяти, однако экземпляр такого множества будет занимать в памяти четыре байта:

```
type Set1= setof (en1V1, en1V2, en1V3, en1V4, en1V5);
type Set2= setof (en2V1, en2V2, en2V3, en2V4, en2V5) : integer
```

Объявление литеральной константы, определяющей подмножество некоторого множества, имеет следующий синтаксис: в прямоугольные скобки заключается последовательность идентификаторов элементов множества, разделенных запятой.

Для работы с переменными типа множество в SPL реализован ряд встроенных функций, обеспечивающих необходимые операции над множествами (объединение, исключение, пересечение, сравнение и др.).

8.8. Классы. В SPL обеспечена ограниченная поддержка объектно-ориентированной парадигмы программирования. Из трех основополагающих принципов ООП (инкапсуляция, наследование и полиморфизм) в SPL пока реализована только инкапсуляция. SPL-классы представляют собой записи, допол-

ненные конструкторами и методами. Целью введения в SPL классов было обеспечение удобной работы с СУБД ИК. Реализация классов в SPL такова, что экземпляр SPL-класса является экземпляром одного из классов Object Pascal, выполняющих в ИК функцию объектов-контейнеров прикладных данных БД [1]. Любой объект БД, как и объект-контейнер прикладных данных, в свою очередь, является объектом встроенного SPL-типа TN_UDBase. Такая реализация классов в SPL отражает особенности объектной модели СУБД ИК, состоящие в том, что объект-контейнер обеспечивает хранение в БД набора прикладных данных, состав которого не зависит от объявления этого класса в Object Pascal и определяется на этапе выполнения программы в процессе создания экземпляра класса. Фактически, структура этого набора прикладных данных определяется в объявлении SPL-класса. Таким образом, экземпляр SPL-класса является, с одной стороны, объектом производного SPL-типа, а с другой стороны — объектом встроенного SPL-типа TN_UDBase.

SPL-класс, как и любой производный тип, должен объявляться до его использования в качестве переменной или параметра процедуры. Объявление класса осуществляется с помощью ключевого слова class в следующей форме:

```
type <имя SPL-класса> = class (Instance=<имя Pascal-класса>)
    <поля и методы класса>
end
```

Необязательный элемент описания Instance позволяет указать имя класса Object Pascal, экземпляр которого необходимо создать при конструировании экземпляра SPL-класса. Если элемент описания Instance не задан, то при создании экземпляра SPL-класса будет создан экземпляр базового класса Object Pascal, являющегося контейнером прикладных данных СУБД ИК — TK_UDRArray [1]. Синтаксис объявления полей SPL-класса ничем не отличается от синтаксиса объявления полей записи:

```
type <имя SPL-класса> = class
    <квалификатор 1> <имя поля 1> : <имя типа поля 1>;
    ...
    <квалификатор N> <имя поля N> : <имя типа поля N>;
    <методы класса>
end
```

Синтаксис объявления методов SPL-класса полностью соответствует синтаксису заголовка обычных процедур или функций:

```
type <имя SPL-класса> = class
    <поля класса>
    ...
    procedure <имя процедуры> (<параметры процедуры>);
    ...
    function <имя функции> (<параметры функции>) : <тип результата>;
    ...
end
```

Реализация методов класса осуществляется в следующей форме. Перед именем метода располагается имя класса, а в качестве разделителя между именем класса и именем метода используется символ “!”. Обращение к полю класса в коде метода осуществляется с помощью ключевого слова self, а имя поля отделяется от self символом “!”:

```
type MyClass = class
    ...
    V1 : integer;
    ...
    procedure Proc1 (Par1 : integer);
    ...
end;

MyClass!procedure Proc1 (Par1 : integer);
begin
...
    self!V1 := Par1;
```

```
...
end
```

Конструкторы класса — это специальные методы-функции, имеющие тип результата TN_UDBase, создающие экземпляр класса и инициализирующие в случае необходимости его поля. Объявление конструктора по форме соответствует объявлению процедуры (поскольку тип возвращаемого результата не указывается), но вместо ключевого слова procedure это объявление предваряется ключевым словом constructor:

```
type <имя SPL-класса> = class
    <поля класса>
    ...
    constructor <имя конструктора >(<список параметров конструктора>);
    ...
    <методы класса>
end
```

Для обращения в коде конструктора к полю класса, как и в обычном методе, необходимо использовать ключевое слово self с разделителем "!", а обращение к экземпляру класса как к объекту БД, имеющему тип TN_UDBase, можно осуществить с помощью ключевого слова result.

9. О реализации SPL. В ИК реализованы интерактивные средства поддержки разработки SPL-программ, включающие редактор и отладчик программного кода. Эти средства обеспечивают:

— одновременное редактирование в многооконном режиме нескольких SPL-модулей;

— компиляцию открытых в редакторе модулей;

— отображение ошибок компиляции в окне редактора модуля;

— запуск SPL-программ в отладочном режиме;

— установку в программном коде контрольных точек;

— различные отладочные режимы выполнения SPL-программ по шагам (без захода, с заходом и с выходом из процедур и функций);

— ведение протокола трассировки выполнения SPL-программ с отображением протокола в специальном окне и с записью его во внешний файл,

— вывод в протокол трассировки значений переменных.

SPL-компилятор и SPL-интерпретатор реализованы на языке Object Pascal в Delphi 7.0. Ниже приведены результаты измерений скорости работы SPL-компилятора и SPL-интерпретатора, которые выполнялись на компьютере P4 2.67 GHz.

SPL-компилятор обеспечил скорость трансляции 14500 строк в секунду. Для сравнения, скорость работы компилятора Object Pascal в Delphi 7.0 на этом же компьютере составила 16000 – 18000 строк в секунду. Измерения времени работы SPL-интерпретатора байт-кода проводились на программе, выполняющей 10000 циклов по 100 операторов сложения. Время выполнения программы SPL-интерпретатором составило примерно 850 миллисекунд. Для сравнения, время выполнения той же программы, реализованной на JavaScript, в Netscape 7.1 составило около 500 миллисекунд, а в Internet Explorer 6.0 — 350 миллисекунд.

10. Заключение. Язык SPL был успешно использован в рамках ИК для обработки статистических данных и при создании ряда приложений, ориентированных на подготовку публикаций в Интернет разнородной территориально-привязанной информации, включая представления числовых данных на картах и схемах. Примеры сайтов, подготовленных с помощью этих приложений, можно найти в Интернет по адресу http://www.srcc.msu.su/ivis/.

СПИСОК ЛИТЕРАТУРЫ

1. *Арушанян О.Б., Богомолов Н.А., Ковалев А.Д., Волченскова Н.И.* Об одном подходе к автоматизации создания приложений, ориентированных на работу со сложными структурами данных // Вычислительные методы и программирование. 2005. **6**, № 1. 115–123.
2. *Архангельский А.Я.* Object Pascal в Delphi. М.: БИНОМ, 2002.
3. *Кэнту М.* Delphi 7: Для профессионалов. СПб.: Питер, 2004.

Поступила в редакцию
22.12.2005