Язык программирования Alien

Концепции, общие соображения по синтаксису, семантике и использованию

Легалов А.И. (2000-2013)
Швец Д.А. (2001-2005)
Легалов И.А. (2003-2007)
Бовкун А.Я. (2008-2011)
Косов П.В. (2012-2013)

12.07.2004 – 01.08.2009 - …

1. Введение

Alien (чуждый, несвойственный) – язык, поддерживающий процедурно-параметрическое программирование в качестве основной парадигмы. Назван в честь процедурного программирования, отторгаемого поголовным большинством современных разработчиков программного обеспечения. В ходе его разработки решаются следующие задачи:

1. Создание языка, обеспечивающего эволюционную разработку программ с применением множественного полиморфизма, реализуемого за счет использования процедурно-параметрической парадигмы программирования.
2. Использование гибкого расширение пространств имен за счет использования подключаемых модулей.
3. Использование в языке средств, ориентированных только на процедурное программирование. При этом процедурно-параметрический стиль должен обеспечить необходимую гибкость при разработке программ, являясь частью процедурного подхода.
4. Создание компактного языка, схожего в этом со своими предшественниками.
5. Создание минимального языка, способного к дальнейшему расширению по различным альтернативным направлениям, что может оказаться полезным в исследовательских целях.
6. Применение языка как средства кодирования приложений, спроектированных с использованием новых методологических приемов, опирающихся на широкое использование множественного полиморфизма.
7. Использование разработанной конструктивной оболочки в других процедурных, функциональных и мультипарадигменных языках, поддерживающих эволюционную разработку последовательных и параллельных программ.

За основу Alien взят язык программирования O2M [1], который расширял своего предшественника (язык Оберон-2) [2, 3] механизмами инструментальной поддержки процедурно-параметрического программирования (ППП) [4, 5]. Основная цель разработки O2M заключалась в экспериментальной проверке методов параметрического программирования и их использовании при создании эволюционно расширяемых программ. При этом сохранилась преемственность с Обероном-2, которая могла быть нарушена только совпадением новых ключевых слов с идентификаторами ранее написанных программ.

Вместе с тем, реализация O2M показала, что, обладая возможностями писать программы с использованием трех различных парадигм (процедурной, объектно-ориентированной и процедурно-параметрической), данный язык утратил изящество, присущее прототипу:

- сохранились ОО рудименты, перекрываемые процедурно-параметрическим стилем (это привело к избыточному дублированию методов написания кода программы).
- использование ППП оказалось громоздким при добавлении как новых специализаций, так и их обработчиков во вновь создаваемых модулях [1].

Так как механизм безболезненного добавления модулей, содержащих новые специализации и обобщающие параметрические процедуры, а также обработчики специализаций для ранее написанных обобщенных процедур, является краеугольным камнем ППП, необходимо было обеспечить его более простое и прозрачное использование. При этом одной из основных задач являлась организация единого пространства имен для программных объектов, расширяемых в различных программных модулях. При этом мы стремились сохранить модульную структуру программы, а не переходить на поддержку пространств имен, используемых в языках программирования C++ [6] и C# [7].

Следует отметить и синтаксические изменения в Alien по сравнению с O2M. Вместо угловых скобок "<" и ">" признак специализации был заключен в круглые скобки "(" и ")". С одной стороны, это снизило его «узнаваемость» в описаниях. Однако меньше вопросов стало возникать в кодовой части. В новом виде его практически невозможно спутать с операциями отношения (в C++, например, подобные путаницы между шаблонами и операциями возникают постоянно), что, ко всему прочему, упростило и синтаксический разбор. Помимо этого функциональная природа признака, как отображения обобщенной переменной в специализацию, способствует использованию круглых скобок.

В отличие от подхода к формированию руководств по языкам программирования Оберон-2 (использовался перевод С.З. Свердлова [3], из которого заимствованы переводы основных терминов англоязычного оригинала) и O2M [8], данный документ написан намеренно избыточно. В нем дается не только описания языка, но и приводятся соображения, связанные с выбором тех или иных технических решений. В большинстве случаев дополнительная информация приводится в виде примечаний и обоснований. Такой подход, при необходимости, позволит достаточно просто выделить отдельный документ, посвященный только описанию языка.

В Alien отсутствует поддержка объектно-ориентированного стиля предшественников. Мы пытаемся создать язык, не содержащий избыточных понятий, перекрываемых процедурно-параметрической парадигмой. В частности, ППП позволяет использовать эволюционно расширяемые мультиметоды, которые перекрывают монометоды ООП [9, 10]. В языке отсутствует расширение записей, замененное использованием обобщений внутри записей (обобщенных записей).

Этот документ не является учебником по программированию. Скорее всего, он отражает принятые решения. Эти решения постоянно эволюционируют. Поэтому представленное описание не является завершенным. Предполагаются дальнейшие расширения языка после создания базовой версии транслятора.

2. Синтаксис

Для описания синтаксиса Alien используются Расширенные Бэкуса-Наура Формы (РБНФ). Варианты разделяются знаком "|". Квадратные скобки "[" и "]" означают необязательность записанного внутри них выражения, а фигурные скобки "{" и "}" означают его повторение (возможно 0 раз). Комбинированные скобки "{/" и "/}" означают повторение 1 или более раз. Круглых скобки "(" и ")" используются для повышения приоритета отдельных групп синтаксических выражений внутри правила. Нетерминальные символы начинаются с заглавной буквы (например, Оператор). Терминальные символы или начинаются малой буквой (например, идент), или записываются целиком заглавными буквами (например, BEGIN), или заключаются в кавычки (например, ":=").

3. Словарь и представление

Для представления терминальных символов предусматривается использование набора знаков ASCII. Слова языка - это идентификаторы, числа, строки, операции и разделители. Должны соблюдаться следующие лексические правила. Пробелы и концы строк не должны встречаться внутри слов (исключая комментарии и пробелы в символьных строках). Пробелы и концы строк игнорируются, если они не существенны для отделения двух последовательных слов. Заглавные и строчные буквы считаются различными.

Примечание. *Возможно, словарь стоило расширить до Юникода. Но в данном случае, несмотря на разнобой с кириллицей, мы считаем, что для научных исследований достаточно и байтовых символов.*

8.07.2013. Предполагаемая в настоящий момент разработка компилятора с применением Qt должна автоматически вывести набор символов на UT-8.

1. *Идентификаторы* – последовательности букв, цифр и символа подчеркивания "_". Первый символ должен быть буквой или "_".

идент = (буква | "_") {буква | цифра | "_"}.

Примеры:

```
x  Scan  Oberon2M  GetSymbol  firstLetter
Hello_World little_boy system_v2_1
```

Примечание. *По сравнению с предшественниками, добавился символ подчеркивания. Возможно, это следовало сделать еще в O2M. Но лучше поздно, чем никогда. Всего лишь восстановлен статус-кво с другими языками.*

2. *Числа* - целые или вещественные (без знака) константы. Типом целочисленной константы считается минимальный тип, которому принадлежит ее значение. Если константа начинается с символов 0x, она является шестнадцатеричной, иначе – десятичной.

Примечание. *Предполагается, что на первом этапе работ не будет большого разнообразия типов и размеров констант. Скорее всего, целочисленные константы ограничатся 32-разрядными величинами.*

Примечание. *По сравнению с предшественниками, изменился вид целочисленной шестнадцатеричной константны. В дальнейшем предполагается наличие разнообразных систем счисления. Поэтому формат их представления выбран исходя из удобства дальнейшего расширения.*

Вещественное число всегда содержит десятичную точку. Оно может также содержать десятичный порядок. Буква E означает "умножить на десять в степени". Вещественное число относится к типу REAL.

```
число          = целое | вещественное.
целое          = цифра {цифра} | "0""X"{/шестнЦифра/}.
вещественное   = цифра {цифра} "." {цифра} [Порядок] .

порядок        = "E" ["+" | "-"] цифра {цифра}.
шестнЦифра     = цифра | "A" | "B" | "C" | "D" | "E" | "F".
Цифра          = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примеры:

```
1991            INTEGER    1991
0XD             INTEGER    13
12.3            REAL       12.3
```

```
4.567E8          REAL       456700000
0.57712566E-6    REAL       0.00000057712566
```

3. *Символьные* константы состоят из одного видимого знака или управляющего символа, используемой кодовой таблицы, например ASCII, ограниченного одинарными кавычками.

символ = " ‘ " (видимый_знак | управляющий_символ |
" \ " цифра {шестнЦифра}) " ‘ "

Управляющие символы задаются с префиксом в виде обратной косой черты. С этой же чертой записываются видимые символы: пробел, обратная косая черта, апостроф и кавычка. Кавычка может записываться в символьных константах и без обратной косой черты. Пробел допускается представлять значением самого символа. Шестнадцатеричные числа используются для задания неотображаемых и любых других символов. Ниже представлены символы, записываемые специальным образом:

- перевод строки (LF) – **‘\n’**;
- горизонтальная табуляция (HT) – **‘\t’**;
- вертикальная табуляция (VT) – **‘\v’**;
- возврат на шаг (BS) – **‘\b’**;
- возврат каретки (CR) – **‘\r’**;
- перевод формата (FF) – **‘\f’**;
- обратная косая (\) - **‘\\’**;
- нулевой символ (nul) – **‘\0’**;
- пробел ( ) – **‘\s’** или **‘ ’**;
- числовое представление символа – ‘\
- апостроф (‘) – **‘\’’**;
- кавычка (“) – **‘\"’** или **‘"’**.

Примеры:

```
'A'   'b'   '1'   '\\'   '\n'   ' ' ≡ '\s'
```

4. *Строки* - последовательности видимых и управляющих символов, заключенные в кавычки (**"**). Для представления кавычки внутри строки используется ее комбинация с обратной косой чертой (**\"**). Апостроф может быть задан как один символ (‘) или в комбинации с обратной косой чертой (\’). Длинная строка может быть представлена последовательностью более коротких строк, разделителем между которыми могут являться только пробельные символы. Пробелы внутри строк могут задаваться с использованием как управляющего символа, так и своим обычным знаком. Число символов в строке называется ее *длиной*.

Примечание. *Строка длины 1 не может использоваться вместо символьной константы и наоборот (их эквивалентность существует в Обероне-2 и O2M).*

строка = {/ “ " ” {символ} “ " “ /}.

Примеры:

```
"Oberon-2"     "Don't worry!"     "x"
```

5. *Операции и разделители* – это специальные символы, пары символов или зарезервированные слова, перечисленные ниже. Зарезервированные слова состоят исключительно из заглавных букв и не могут использоваться как идентификаторы.

+	:=	IMPORT	ARRAY	REPEAT
-	^	BEGIN	IN	RETURN
*	=	BY	IS	THEN
/	#	CASE	LOCAL	TO
~	<	CONST	LOOP	TYPE
&	>	DIV	MOD	UNTIL

.	<=	DO	MODULE	VAR
,	>=	ELSE	NIL	WHILE
;	..	ELSIF	OF	WITH
|	:	END	OR	IMPORT
(	)	EXIT	POINTER	FALSE
[	]	FOR	PROCEDURE	TRUE
{	}	IF	RECORD	XOR

7. *Комментарии* реализованы с стиле C++. Допускаются многострочные и однострочные комментарии. Они могут быть написаны между любыми двумя словами программы

Многострочные комментарии – это произвольные последовательности символов, начинающиеся скобкой “/*” и оканчивающиеся “*/”. Они могут быть вложенными и не влияют на смысл программы.

комментарий = "/*" {знак} "*/".

Однострочные комментарии начинаются с “//” и заканчиваются концом строки.

комментарий = "//" {знак} конец_строки.
Возможна интерпретация что символы – это то, что используется в языке, а знаки – то, что добавляется в комментариях и строках, например, русские буквы.

4. Объявления и области действия

Каждый идентификатор, встречающийся в программе, должен быть объявлен, если это не стандартный идентификатор. Объявления задают некоторые постоянные свойства объекта, например, является ли он константой, типом, переменной или процедурой. После объявления идентификатор используется для ссылки на соответствующий объект.

Область действия объекта *x* распространяется текстуально от точки его объявления до конца блока (модуля, процедуры или записи), в котором находится объявление. Для этого блока объект является *локальным*. Это разделяет области действия одинаково именованных объектов, которые объявлены во вложенных блоках. Правила для областей действия таковы:

1. Идентификатор не может обозначать больше чем один объект внутри данной области действия (то есть один и тот же идентификатор не может быть объявлен в блоке дважды) за исключением объявлений процедур и псевдонимов процедур (отличие которых задается уникальностью сигнатуры);
2. Ссылаться на объект можно только изнутри его области действия;
3. Тип *T* вида POINTER TO *T1* (см. 6.5) может быть объявлен в точке, где *T1* еще неизвестен. Объявление *T1* должно следовать в том же блоке, в котором *T* является локальным;
4. Идентификаторы, обозначающие поля записи (см. 6.3) могут употребляться только в обозначениях записи.
5. Идентификаторы, обозначающие признаки специализаций (см. 6.4) обобщений, могут употребляться только в обозначениях специализаций.

Идентификатор, объявленный в блоке модуля, может сопровождаться при своем объявлении экспортной меткой ("*" или "-"), чтобы указать, что он экспортируется. Идентификатор *x*, экспортируемый модулем *M*, может использоваться в других модулях, если они импортируют *M* (см. гл. 11). Тогда идентификатор обозначается в этих модулях *M.x* и называется *уточненным идентификатором*. Переменные и поля записей, помеченные знаком "-" в их объявлении, предназначены *только для чтения* в модулях-импортерах.

УточнИдент = [идент "."] идент.

ИдентОпр = идент ["*" | "-"].

Следующие идентификаторы являются стандартными; их значение определено в указанных разделах:

ABS	(10.5)
ASH	(10.5)
BOOLEAN	(6.1)
CAP	(10.5)
CHAR	(6.1)
CHR	(10.5)
COPY	(10.5)
DEC	(10.5)
ENTIER	(10.5)
EXCL	(10.5)
FALSE	(6.1)
HALT	(10.5)
INC	(10.5)
INCL	(10.5)
INTEGER	(6.1)
LEN	(10.5)
LONG	(10.5)
LONGINT	(6.1)
LONGREAL	(6.1)
MAX	(10.5)
MIN	(10.5)
NEW	(10.5)
ODD	(10.5)
ORD	(10.5)
REAL	(6.1)
SET	(6.1)
SHORT	(10.5)
SHORTINT	(6.1)
SIZE	(10.5)
TRUE	(6.1)
UNSIGNED	

5. Объявления констант

Объявление константы связывает идентификатор со значением.

ОбъявлениеКонстанты = ИдентОпр "=" КонстантноеВыражение.
КонстантноеВыражение = Выражение.

Константное выражение - это выражение, которое может быть вычислено по его тексту без фактического выполнения программы. Его операнды - константы (Гл. 8) или стандартные функции (Гл. 10.5), которые могут быть вычислены во время компиляции.
Примеры объявлений констант:

```
N = 100
limit = 2*N - 1
```

6. Объявления типа

Тип данных определяет набор значений, которые могут принимать переменные этого типа, и набор применимых операций. Объявление типа связывает идентификатор с типом. В случае структурированных типов (массивы, записи, обобщения) объявление также определяет структуру переменных этого типа. Структурированный тип не может содержать сам себя.

ОбъявлениеТипа **= (ИдентОпр "=" Тип) | Расширение.**

Тип **= УточнИдент | ТипМассив | ТипЗапись | ТипОбобщение | ТипУказатель | ПроцедурныйТип.**

Примеры:

```
Table = ARRAY N OF REAL

Tree = POINTER TO Node
Node = RECORD
     key : INTEGER;
     left, right: Tree
CASE TYPE OF
END

CenterTree = POINTER TO CenterNode

CenterNode = RECORD
     width: INTEGER;
     subnode: Tree
END

Node += CenterNode

Rectangle = RECORD
     x, y: INTEGER;

END

Triangle = RECORD
     a, b, c: INTEGER;
END

Shape1 = CASE TYPE LOCAL OF Rectangle | Triangle END

Shape2 = CASE OF r: Rectangle | t: Triangle END

PShape1 = POINTER TO Shape1

Function = PROCEDURE(x: INTEGER): INTEGER
```

6.1 Основные типы

Основные типы обозначаются стандартными идентификаторами. Соответствующие операции определены в 8.2, а стандартные функции в 10.5. Предусмотрены следующие основные типы:

1. **BOOLEAN** **логические значения TRUE и FALSE**
2. **CHAR** **символы расширенного набора ASCII (0X .. 0FFX)**
3. **INTEGER** **целые в интервале от MIN(INTEGER) до MAX(INTEGER)**
4. **UNSIGNED** **Беззнаковые целые в интервале от 0 до MAX(UNSIGNED)**
5. **REAL** **вещественные числа в интервале от MIN(REAL) до MAX(REAL)**

Между типами отсутствует явное преобразование и отношение порядка. В отличие от предшественников, один тип не может поглотить другой. Вместе с тем существуют предопределенные функции взаимного приведения типов (их имена совпадают с именами типов), явно обеспечивающие допустимые преобразования.

Примечание. *Из всех целых и вещественных типов пока оставлены только два. Это связано со стремлением не загромождать язык на начальном этапе дополнительными*

наворотами, которые в дальнейшем предполагается реализовать средствами самого же языка, включая их в служебный модуль SYSTEM, который будет являться предком всех других модулей. Множественный тип показался громоздким и малоэффективным по сравнению с его реализацией посредством дополнительных библиотек. Нам видится, что проще ввести беззнаковый тип UNSIGNED, который, наряду с арифметическими и массовыми логическими операциями, может достаточно эффективно моделировать множественный тип предшественников.

6.2 Тип массив

Массив - структура, состоящая из определенного количества элементов одинакового типа, называемого *типом элементов*. Число элементов массива называется его *длиной*. Элементы массива обозначаются индексами, которые являются целыми числами от 0 до длины массива минус 1.

ТипМассив = ARRAY [Длина {"," Длина}] OF Тип.
Длина = КонстантноеВыражение.

Тип вида

```
ARRAY L0, L1, ... , Ln OF T
```

понимается как сокращение

```
ARRAY L0 OF
      ARRAY L1 OF
            ...
                  ARRAY Ln OF T
```

Массивы, объявленные без указания длины, называются *открытыми массивами*. Они могут использоваться только в качестве базового типа указателя (см. 6.5), типа элементов открытых массивов и типа формального параметра (см. 10.1). Примеры:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

6.3 Тип обобщение

Тип обобщение – структура, состоящая из произвольного числа альтернативных элементов-специализаций, которые отличаются признаками. Специализация может быть записью или другим обобщением, следовательно, ее тип задается уточненным идентификатором. Допускается отсутствие типа специализации (вместо него задается значение **NIL**). Каждый элемент обобщения задает одну из альтернатив, фиксируемую при создании переменной. Обобщения похожи на вариантные записи языков программирования Паскаль или Модула-2. Однако они обладают рядом только им присущих особенностей и используются в другом контексте.

ТипОбобщение = CASE [TYPE] [LOCAL] OF [СписокСпециализаций]
[ELSE Специализация] END .
СписокСпециализаций = ((СписокПризнаков ":" (Тип | NIL)) | УточнИдент)
{ "|" ((СписокПризнаков ":" (Тип | NIL))
| УточнИдент) } .
СписокПризнаков = идент { "," идент } .
Специализация = (идент ":" (УточнИдент | NIL)) | УточнИдент.

Допускается идентификация по типу или по ключу. При этом все специализации одного обобщения должны задаваться одинаковым способом. Идентификация по типу используется при наличии ключевого слова TYPE. В этом случае каждая специализация обобщения отличается значением типа, который также рассматривается и как обозначение признака.

Пример обобщения, идентифицируемого типом:

```
Shape1 = CASE TYPE LOCAL OF Rectangle | Triangle END
```

В примере используется обобщение прямоугольника и треугольника как одной из двух геометрических фигур. Подобная запись может рассматриваться как совпадение имени признака с именем типа, что в целом и позволяет сократить ее написание.

Специализация по ключу позволяет идентифицировать обобщения, имеющие одинаковый тип. Ключ задается идентификатором, уникальным внутри обобщения. Идентификация по ключу формируется при отсутствии в описании обобщения ключевого слова TYPE.

Пример обобщения, идентифицируемого ключом:

```
Shape2 = CASE OF r: Rectangle | t: Triangle END
```

После ключевого слова ELSE может следовать тип специализации, устанавливаемый по умолчанию в случае, когда при описании переменной ее специализация явно не указана. Если часть ELSE отсутствует, то описание переменной обязано содержать признак специализации.

Пример использования ключевого слова ELSE:

```
Circle = RECORD   r: INTEGER  END

Shape3 = CASE OF r: Rectangle | t: Triangle ELSE c: Circle END
```

При явном задании признаков вместо типа можно ставить значение NIL, что позволяет строить пространства признаков, не привязанные к обрабатываемым данным, но допускающие их эволюционное расширение и использование в полиморфных операциях.

Пример обобщения, использующего только признаки:

```
WeekDay = CASE LOCAL OF
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday: NIL
END
```

Обобщение может быть локальным или расширяемым. Локальное обобщение создается в одном модуле и не может быть изменено в дальнейшем. Для этого в описании обобщения необходимо указать ключевое слово LOCAL. Расширяемое обобщение допускает добавление новых специализаций в других модулях, являющихся потомками модуля, определяющего обобщение. Это позволяет эволюционно наращивать альтернативы при расширении существующего проекта. Расширение допускается как для обобщений по типу, так и по ключу, способ идентификации в расширении должен соответствовать обобщению. Оно задается следующим синтаксисом:

Расширение = УточнИдент "+=" СписокСпециализаций .

Примеры использования расширений:

```
Shape2 += l: Line | c: MCircle.Circle
```

В примере предполагается, что к существующему обобщению Shape2 добавляются две специализации, построенные на основе записей Line и Circle. Причем, запись Circle находится в импортируемом модуле MCircle, а запись Line располагается в текущем модуле. Определение обобщения Shape2 размещено в одном из предков текущего модуля. Допускается многократное поэтапное расширение уже расширенных обобщений. Возможно независимое расширение обобщений в разных несвязанных модулях с последующей сборкой воедино на более поздних этапах формирования окончательной программы.

6.4 Тип запись

Тип запись - структура, состоящая из фиксированного числа элементов, которые могут быть различных типов и называются *полями*. В этом случае она называется фиксированной записью. После полей может размещаться одно обобщение, что определяет обобщенную

запись. Объявление типа запись определяет имя, тип каждого поля, признаки обобщения. Область действия идентификаторов полей простирается от точки их объявления до конца объявления типа запись, но они также видимы внутри обозначений, ссылающихся на элементы переменных-записей (см. 8.1). Если тип запись экспортируется, то идентификаторы полей, видимые вне модуля, в котором объявлены, должны быть помечены. Они называются *доступными полями*; непомеченные элементы называются *скрытыми полями*.

ТипЗапись = RECORD [СписокПолей {";" СписокПолей}]
(ТипОбобщение | [CASE ИмяОбобщения] END).
СписокПолей = СписокИдент ":" Тип.
СписокИдент = ИдентОпр {"," ИдентОпр}.

Наличие обобщения в записи позволяет использовать его для последующего расширения. Допустимость внутри записи только одного обобщения позволяет использовать признак специализации для характеристики всей записи. В примере

```
T0 = RECORD x: INTEGER; CASE OF END;
T0 += y: REAL
```

T0 – обобщенная запись, которая содержит пустое обобщение. В следующей строке к записи добавляется специализация ***y*** вещественного типа. В результате возможно существование двух альтернативных объектов: записи, состоящей только из поля ***x***, и записи, которая помимо поля ***x*** содержит вещественное поле с признаком ***y***. Обращение к этому полю состоит из имени переменной – записи с указанием после точки вместо имени поля круглых скобок.

Пример:

```
VAR v(y): T0
   ...
v(y) = 3.14;
```

Последняя запись является избыточной, так как признак специализированной переменной является неизменным. Поэтому, можно использовать и альтернативное обозначение:

```
v() = 3.14;
```

Примечание. *Возможно, что для обращения к специализации внутри записи следует подобрать более подходящее обозначение.*

Примечание. Должны ли по умолчанию закрывать признак специализации? Также по умолчанию специализации обобщения или обобщенной записи экспортируются.

ИмяОбобщения = УточнИдент.

Примеры объявлений фиксированной записи:

```
RECORD
      day, month, year: INTEGER
END

RECORD
      name, firstname: ARRAY 32 OF CHAR;
      age: INTEGER;
      salary: REAL
END
```

Примеры обобщенных записей:

```
Figure = RECORD
     x, y : REAL;
CASE TYPE OF
     Circle | Rectangle
END;
ColorDecorator = RECORD
     color : UNSIGNED;
```

```
CASE TYPE OF
     Decorator
END;
```

Примечание. *В отличие от предшественников, из записи убрано расширение типа. Несмотря на удобство, данный механизм построения новых абстрактных типов аналогичен по свойствам использованию обобщения внутри записи. В Alien делается попытка решить все задачи программирования с использованием только процедурного и расширяющего его процедурно-параметрического стилей. Считается, что в техническом аспекте этими подходами перекрываются все возможности ООП. Религиозные, философские и идеологические догмы, о том, что ООП естественнее описывает окружающий нас мир, игнорируются.*

6.5 Тип указатель

Переменные-указатели типа *P* принимают в качестве значений указатели на переменные некоторого типа *T*. *T* называется базовым типом указателя типа *P* и должен быть типом массив, запись или обобщение.

Тип указатель заимствует отношение расширения своих базовых типов: если тип *T1* - расширение *T* и *P1* - это тип POINTER TO *T1*, то *P1* также является расширением *P*.

Запрещены указатели на специализации и специализированные записи.

ТипУказатель =POINTER TO Тип.

Если *p* - переменная типа *P* = POINTER TO *T*, вызов стандартной процедуры NEW(p) (см. 10.5) размещает переменную типа *T* в свободной памяти. Если *T* – основной тип, фиксированная запись или тип массив с фиксированной длиной, размещение должно быть выполнено вызовом *NEW(p)*; если тип *T* - n-мерный открытый массив, размещение должно быть выполнено вызовом *NEW(p, e0, ..., en-1)*, чтобы *T* был размещен с длинами, заданными выражениями *e0, ..., en-1*. В любом случае указатель на размещенную переменную присваивается *p*. Переменная *p* имеет тип *P*. Переменная *p^* (*динамическая переменная*), на которую ссылается *p,* имеет тип *T*.

Если *T* – тип обобщения, то переменная *p* типа *P* = POINTER TO *T* может указывать только на *T*. Однако тип такой переменной определяется не только *T*. Он может уточняться специализациями, определяемыми значениями признаков. Размещение обобщенной переменной в свободной памяти осуществляется с помощью процедуры *NEW(p(t))*, в которой признак *t* определяет создаваемую специализацию. В том случае, если указатель на обобщение *p* типа *T* имеет специализацию по умолчанию с признаком *d*, ее размещение в памяти может быть организовано с использованием процедуры *NEW(p())* или *NEW(p(d))*.

При динамическом размещении обобщенной записи типа *T* с использованием переменной *p* типа *P* возможны следующие варианты.

1. Явное задание признака специализации *t* позволяет создать специализированную запись процедурой *NEW(p(t))*.
2. При наличии специализации по умолчанию с признаком *d*, ее размещение в памяти может быть организовано с использованием процедуры *NEW(p())*, или *NEW(p(d))*.
3. Во всех иных случаях вызов процедуры *NEW(p())* ведет к созданию записи, которая содержит только фиксированные поля.

Наряду с указателями на обобщения и обобщенные записи в языке имеются указатели на специализации. В этом случае при объявлении переменной-указателя в типе явно указывается признак специализации. При наличии специализации по умолчанию можно указать только круглые скобки *“()”* без обязательного задания признака по умолчанию. Использование

указателей на специализации в процедуре *NEW* позволяет разместить в памяти нужную специализацию без дополнительного задания признаков.

Любая переменная-указатель может принимать значение NIL, которое не указывает ни на какую переменную вообще.

6.6 Процедурные типы

Переменные процедурного типа *T*, имеют значением процедуру (или NIL). Если процедура *P* присваивается переменной типа *T*, списки формальных параметров (см. Гл. 10.1) *P* и *T* должны совпадать (см. Прил. А). *P* не должна быть стандартной процедурой, процедурой, связанной с типом, обобщающей процедурой или обработчиком специализации. Она не может быть локальной в другой процедуре.

Примечание. *При необходимости, сигнатуру любой из процедур можно привести к процедурному типу, используя псевдонимы процедур.*

ПроцедурныйТип = PROCEDURE [ФормальныеПараметры].

Для подключения через указатели запрещенных вариантов процедур можно воспользоваться псевдонимами процедур, обеспечивающими совместимость сигнатур.

7. Объявления переменных

Объявления переменных дают их описание, определяя идентификатор и тип данных.

ОбъявлПерем = СписокИдент ":" Тип | ОбъявлОбобщПерем.
ОбъявлОбобщПерем = СписокИдент ":" УточнИдент "(" [Признак] ")" | идент"(" [Признак] ")" {“,” идент"(" [Признак] ")" } ":" УточнИдент.
Признак = УточнИдент .

Все переменные типа запись, обобщение и указатель имеют статический тип (тип, с которым они объявлены, называемый просто их типом), который может уточняться в случае наличия обобщения. Для указателей и параметров-переменных типа обобщенная запись и обобщение уточненный тип является специализацией их статического типа. Статический тип определяет, какие поля записи доступны. Уточненный тип используется для вызова обобщающих параметрических процедур (см. 10.3). При объявлении обобщенных переменных после указания типа обобщения указывается признак, задающий специализацию обобщения. Признак задается его идентификатором, который, при использовании в объявлении идентификации по типам, совпадает с именем типа.

Обобщенные переменные могут быть объявлены двумя способами:

- в первом случае задаются переменные только одной специализации;
- второй вариант используется для одновременного задания нескольких специализаций.

Не допускается одновременное использование признаков специализаций после идентификаторов переменных и типа. Для признака по умолчанию обязательно присутствие круглых скобок. При отсутствии этих скобок в случае записи создается фиксированная запись. В случае обобщения порождается сообщение об ошибке (невозможно создать пустую переменную, которая не содержит специализации)

Примеры объявлений переменных (со ссылками на примеры из Гл. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
```

```
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
          name: ARRAY 32 OF CHAR;
          count: INTEGER
     END
t, c: Tree
rect : Rectangle
sh11, sh12 : Shape1(Rectangle)
sh21 : Shape2(t)
sh31(t), sh32(), sh33(c) : Shape3
psh: PShape1
```

8. Выражения

Выражения – конструкции, задающие правила вычисления по значениям констант и текущим значениям переменных других значений путем применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для группировки операций и операндов.

8.1 Операнды

За исключением литералов (чисел, символьных констант или строк), операнды представлены обозначениями. Обозначение содержит идентификатор константы, переменной или процедуры. Этот идентификатор может быть уточнен именем модуля (см. Гл. 4 и 11) и может сопровождаться селекторами, если обозначенный объект – элемент структуры.

Обозначение = УточнИдент { "." идент | "[" СписокВыражений "]" | "^" | "(" [УточнИдент] ")" }.
СписокВыражений = Выражение {"," Выражение}.

Если *a* - обозначение массива, *a[e]* означает элемент *a*, чей индекс - текущее значение выражения *e*. Тип *e* должен быть целым типом. Обозначение вида *a[e0, e1, ..., en]* применимо вместо *a[e0] [e1] ... [en]*. Если *r* обозначает запись, то *r.f* означает поле *f* записи *r*. Если *p* обозначает указатель, *p^* означает переменную, на которую ссылается *p*. Обозначения *p^.f* и *p^[e]* могут быть сокращены до *p.f* и *p[e]*, то есть запись и индекс массива подразумевают разыменование. Если *a* или *r* доступны только для чтения, то *a[e]* и *r.f* также предназначены только для чтения.

Охрана типа v(t) требует, чтобы специализацией *v* была *t*, то есть выполнение программы прерывается, если тип специализации *v* - не *t*. В пределах такого обозначения *v* воспринимается как имеющая статический тип *T(t)*. Охрана применима, если

1. *v* - параметр-переменная типа запись, или *v* - указатель, и если
2. *t* – специализация статического типа *T*.

Если обозначенный объект - константа или переменная, то обозначение ссылается на их текущее значение. Если он - процедура, то обозначение ссылается на эту процедуру, если только обозначение не сопровождается (возможно, пустым) списком параметров. В последнем случае подразумевается активация процедуры и подстановка значения результата, полученного при ее исполнении. Фактические параметры должны соответствовать формальным параметрам, как и при вызовах собственно процедуры (см. 10.1).

Примеры обозначений (со ссылками на примеры из Гл. 7):

i	(INTEGER)
a[i]	(REAL)
w[3].name[i]	(CHAR)
t.left.right	(Tree)
t(CenterNode).subnode	(Tree)
sh11	(Shape1<Rectangle>)

8.2 Операции

В выражениях синтаксически различаются четыре класса операций с разными приоритетами (порядком выполнения). Операция ~ имеет самый высокий приоритет, далее следуют операции типа умножения, операции типа сложения и отношения. Операции одного приоритета выполняются слева направо. Например, x-y-z означает (x-y)-z.

Выражение = **ПростоеВыражение [Отношение ПростоеВыражение].**
ПростоеВыражение = **["+" | "-"] Слагаемое {ОперацияСложения Слагаемое}.**
Слагаемое = **Множитель {ОперацияУмножения Множитель}.**
Множитель = **Обозначение [ФактическиеПараметры] | число | символ | строка | NIL | Множество | "(" Выражение ")" | "~" Множитель.**
Множество = **"{" [Элемент {"," Элемент}] "}".**
Элемент = **Выражение [".." Выражение].**
ФактическиеПараметры = **"{" [СписокВыражений] "}" ["(" [СписокВыражений] ")"] | "(" [СписокВыражений] ")".**
Отношение = **"=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.**
ОперацияСложения = **"+" | "-" | OR | XOR.**
ОперацияУмножения = **"*" | "/" | DIV | MOD | "&".**

Предусмотренные операции перечислены в следующих таблицах. Некоторые операции применимы к операндам различных типов, обозначая разные действия. В этих случаях фактическая операция определяется типом операндов. Операнды должны быть совместимыми выражениями для данной операции (см. Прил. А).

8.2.1 Логические операции

OR	логическая дизъюнкция	p OR q	"если p, то TRUE, иначе q"
XOR	исключающее ИЛИ	p XOR q	(~p & q) OR (p & ~q)
&	логическая конъюнкция	p & q	"если p то q, иначе FALSE"
~	Отрицание	~p	"не p"

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN. Они также поразрядно применимы к операндам типа UNSIGNED и дают результат типа UNSIGNED.

8.2.2 Арифметические операции

+	сумма – разность
*	Произведение
/	вещественное деление
DIV	деление нацело
MOD	Остаток

Операции +, -, *, и / применимы к операндам одинаковых числовых типов. Тип их результата – совпадает с типом операндов. В отличие от предшественников, в языке отсутствует поглощение числовых типов. Перед выполнением операции с операндами разных типов необходимо один из них привести к типу другого путем вызова предопределенной процедуры приведения. Операция деления (/), всегда в качестве результата возвращает вещественный тип. При использовании в качестве одноместной операции "-" обозначает перемену знака, а "+" - тождественную операцию. Операции DIV и MOD применимы только к целочисленным и беззнаковым операндам. Они связаны следующими формулами, определенными для любого *x* и положительного делителя *y*:

x = (x DIV y) * y + (x MOD y)
0 < = (x MOD y) < y

Примеры:

x	y	x DIV y	x MOD y
5	3	1	2
-5	3	-2	1

8.2.3 Дополнительные операции над беззнаковыми целыми

Примечание. *Следующий абзац убирается из текущей реализации. Возможно, подобная реализация не будет выполнена и в дальнейшем. Надо внимательно обдумать необходимость подобного расширения языка.*

Использование беззнаковых целых чисел позволяет легко имитировать множества, используемые в предшественниках. Основные функции над множествами: объединение, разность, пересечение и др. легко реализуются с помощью поразрядных логических операций. Для выделения отдельных разрядов из беззнаковых чисел вводятся операции с индексами, задаваемыми в виде отдельных чисел, диапазонов и списков индексов, разделенных запятыми. Если v – беззнаковая переменная, то ее разряд [i] доступен посредством операции v[i], где i находится в диапазоне от 0 до N-1, а N – номер старшего разряда числа. Операция v[i..j, k..l] обеспечивает одновременный доступ к диапазонам разрядов.

Примечание. *Необходимо более детально проработать этот параграф. Пока на него не хочется убивать время.*

8.2.4 Отношения

=	Равно
#	не равно
<	Меньше
<=	меньшее или равно
>	Больше
>=	больше или равно
IN	принадлежность разряду беззнакового типа
IS	проверка специализации

Отношения дают результат типа BOOLEAN. Отношения =, #, <, <=, >, и >= применимы к числовым типам, типу CHAR, строкам и символьным массивам, содержащим в конце 0X. Отношения = и # кроме того применимы к типу BOOLEAN, а также к указателям и процедурным типам (включая значение NIL). *x IN s* означает "*x* является элементом *s*". *x s* должны быть целого беззнакового типа., а *s* - типа UNSIGNED. *v IS t* означает "обобщенная переменная *v* специализирована признаком *t*" и называется *проверкой специализации*. Проверка специализации применима, если

1. *v* - параметр-переменная типа обобщенная запись, или обобщение, или *v* – указатель на обобщение, или *v* – параметр-переменная типа обобщение и если
2. *T* – тип обобщенной величины с признаком *t*.

Примеры выражений (со ссылками на примеры из Гл. 7):

1991	INTEGER
i DIV 3	INTEGER
~p OR q	BOOLEAN
(i+j) * (i-j)	INTEGER
REAL(i) + x	REAL
a[i+j] * a[i-j]	REAL
(0<=i) & (i<100)	BOOLEAN
t.key = 0	BOOLEAN
k IN u	BOOLEAN
w[i].name <= "John"	BOOLEAN
t IS CenterTree	BOOLEAN
sh11 IS Rectangle	BOOLEAN

9. Операторы

Операторы обозначают действия. Есть простые и структурные операторы. Простые операторы не содержат в себе никаких частей, которые являются самостоятельными операторами. Простые операторы - присваивание, вызов процедуры, операторы возврата и выхода. Структурные операторы состоят из частей, которые являются самостоятельными операторами. Они используются, чтобы выразить последовательное и условное, выборочное и повторное исполнение. Оператор также может быть пустым, в этом случае он не обозначает никакого действия. Пустой оператор добавлен, чтобы упростить правила пунктуации в последовательности операторов.

Оператор = Присваивание | ВызовПроцедуры | ОператорIf | ОператорCase
| ОператорWhile | ОператорRepeat | ОператорFor | ОператорLoop
| ОператорWith | ОператорGoto | ОператорПустой | EXIT
| RETURN [Выражение]].

ПослОператоров = Оператор {«;» Оператор}.

9.1 Присваивание

Присваивание заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть совместимо по присваиванию с переменной (см. Приложение. A). Знаком операции присваивания является ":=", который читается "присвоить".

Присваивание = Обозначение ":=" Выражение.

Если выражение *e* типа *Te* присваивается переменной *v* типа *Tv*, имеет место следующее:

1. Если *Tv* и *Te* - записи, то в присваивании участвуют только те поля *Te*, которые также имеются в *Tv* (*проецирование*); динамический тип *v* и статический тип *v* должны быть одинаковы, и не изменяются присваиванием;
2. Если *Tv* и *Te* - типы указатель, динамическим типом *v* становится динамический тип *e*;
3. Если *Tv* это ARRAY *n* OF CHAR, а *e* - строка длины $m < n$, *v[i]* присваиваются значения *ei* для $i = 0 .. m-1$, а *v[m]* получает значение 0X;
4. Если *Tv* и *Te* – специализации обобщения, то присваивание выполняется в случае одинаковых специализаций;

5. Если *Tv* - специализация обобщения и *Te* - тип, входящий в обобщение, или *Te* – специализация обобщения и *Tv* - тип, входящий в обобщение, то присваивание выполняется по правилам для соответствующих типов.

Примеры присваиваний (со ссылками на примеры из Гл. 7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2                       /* см. 10.1 */
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c
sh11 := sh12
rect := sh11
sh12 := rect
```

9.2 Вызовы процедур

Вызов процедуры активирует процедуру. Он может содержать список фактических параметров, которые заменяют соответствующие формальные параметры, определенные в объявлении процедуры (см. Гл. 10). Соответствие устанавливается в порядке следования параметров в списках фактических и формальных параметров. Имеются два вида параметров: параметры-переменные и параметры-значения.

Если формальный параметр - параметр-переменная, соответствующий фактический параметр должен быть обозначением переменной. Если фактический параметр обозначает элемент структурной переменной, селекторы компонент вычисляются, когда происходит замена формальных параметров фактическими, то есть перед выполнением процедуры. Если формальный параметр - параметр-значение, соответствующий фактический параметр должен быть выражением. Это выражение вычисляется перед вызовом процедуры, а полученное в результате значение присваивается формальному параметру (см. также 10.1).

При вызове обобщенной процедуры дополнительно указываются обобщенные параметры. Список обобщенных параметров указывается между обозначением процедуры и списком формальных параметров

ВызовПроцедуры = Обозначение [ФактическиеПараметры].

Примеры:

```
WriteInt(i*2+1)   // см. 10.1
INC(w[k].count)
t.Insert("John")  // см. 11
Perimeter{sh12};
```

9.3 Последовательность операторов

Последовательность операторов, разделенных точкой с запятой, означает поочередное выполнение действий, заданных составляющими операторами.

ПоследовательностьОператоров = Оператор {";" Оператор}.

ПослПомечОператоров = [Метка] Оператор {";" [Метка] Оператор}.

Метка = идент «:».

9.4 Оператор If

ОператорIf =
IF Выражение THEN ПоследовательностьОператоров
{ELSIF Выражение THEN ПоследовательностьОператоров}
[ELSE ПоследовательностьОператоров]
END.

Операторы if задают условное выполнение входящих в них последовательностей операторов. Логическое выражение, предшествующие последовательности операторов, называется условием. Условия проверяются последовательно одно за другим, пока результат проверки не окажется равным TRUE, после чего выполняется связанная с этим условием последовательность операторов. Если ни одно из условий не удовлетворено, выполняется последовательность операторов, записанная после слова ELSE, если она имеется.

Пример:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = '\'') OR (ch = '"') THEN ReadString
ELSE SpecialCharacter
END
```

9.5 Оператор Case

Операторы Case определяют выбор и выполнение последовательности операторов по значению выражения. Сначала вычисляется выбирающее выражение, а затем выполняется та последовательность операторов, чей список меток варианта содержит полученное значение. Выбирающее выражение должно быть такого целого типа, который поглощает типы всех меток вариантов, или и выбирающее выражение и метки вариантов должны иметь тип CHAR. Метки варианта - константы, и ни одно из их значений не должно употребляться больше одного раза. Если значение выражения не совпадает с меткой ни одного из вариантов, выбирается последовательность операторов после слова ELSE, если оно есть, иначе программа прерывается.

ОператорCase = CASE Выражение OF Вариант {" | " Вариант}
[ELSE ПоследовательностьОператоров] END.
Вариант = [СписокМетокВарианта ":"
ПоследовательностьОператоров].
СписокМетокВарианта = МеткиВарианта {"," МеткиВарианта }.
МеткиВарианта = КонстантноеВыражение [".." КонстантноеВыражение].

Пример:

```
CASE ch OF
   'A' .. 'Z': ReadIdentifier
   | '0' .. '9': ReadNumber
   | '\'', '"' : ReadString
ELSE
  SpecialCharacter
END
```

9.6 Оператор While

Операторы while задают повторное выполнение последовательности операторов, пока логическое выражение (условие) остается равным TRUE. Условие проверяется перед каждым выполнением последовательности операторов.

ОператорWhile = WHILE Выражение DO ПоследовательностьОператоров END.

Примеры:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

9.7 Оператор Repeat

Оператор repeat определяет повторное выполнение последовательности операторов пока условие, заданное логическим выражением, не удовлетворено. Последовательность операторов выполняется по крайней мере один раз.

ОператорRepeat = REPEAT ПоследовательностьОператоров UNTIL Выражение.

9.8 Оператор For

Оператор for определяет повторное выполнение последовательности операторов фиксированное число раз для прогрессии значений целочисленной переменной, называемой *управляющей переменной* оператора for.

ОператорFor = FOR идент ":=" Выражение TO Выражение
[BY КонстантноеВыражение] DO
ПоследовательностьОператоров END.

Оператор

```
FOR v := beg TO end BY step DO statements END
```

эквивалентен

```
temp := end; v := beg;
IF step > 0 THEN
      WHILE v <= temp DO statements; v := v + step END
ELSE
      WHILE v >= temp DO statements; v := v + step END
END
```

temp и *v* имеют одинаковый тип. Шаг (*step*) должен быть отличным от нуля константным выражением. Если шаг не указан, он предполагается равным 1.

Примеры:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

9.9 Оператор Loop

Оператор loop определяет повторное выполнение последовательности операторов. Он завершается после выполнения оператора выхода внутри этой последовательности (см. 9.10).

ОператорLoop = LOOP ПоследовательностьОператоров END.

Пример:

```
LOOP
      ReadInt(i);
      IF i < 0 THEN EXIT END;
      WriteInt(i)
END
```

Операторы loop полезны, чтобы выразить повторения с несколькими точками выхода, или в случаях, когда условие выхода находится в середине повторяемой последовательности операторов.

***Примечание**. Нужен ли этот оператор? Ведь то же самое можно сделать с помощью Repeat.*

Или while true. Цикл подобного рода можно и не использовать. Пусть пока будет. Реализовать несложно. Не будем отклоняться от Вирта. Надо в них внимательнее посмотреть, где допустимо использование exit

9.10 Операторы возврата и выхода

Оператор возврата выполняет завершение процедуры. Он обозначается словом RETURN, за которым следует выражение, если процедура является процедурой-функцией. Тип выражения должен быть совместим по присваиванию (см. Приложение A) с типом результата, определенным в заголовке процедуры (см. Гл. 10).

Процедуры-функции должны быть завершены оператором возврата, задающим значение результата. В собственно процедурах оператор возврата подразумевается в конце тела процедуры. Любой явный оператор появляется, следовательно, как дополнительная (вероятно, для исключительной ситуации) точка завершения.

Оператор выхода обозначается словом EXIT. Он определяет завершение охватывающего оператора loop и продолжение выполнения программы с оператора, следующего за оператором loop. Оператор выхода связан с содержащим его оператором loop контекстуально, а не синтаксически.

9.11 Операторы With

Операторы with выполняют последовательность операторов в зависимости от результата проверки типа и применяют охрану типа к каждому вхождению проверяемой переменной внутри этой последовательности операторов.

ОператорWith =WITH Охрана DO ПоследовательностьОператоров {"|" Охрана DO
ПоследовательностьОператоров} [ELSE
ПоследовательностьОператоров] END.
Охрана =УточнИдент ":" Признак.

Если *v* - параметр-переменная типа запись или переменная-указатель, и если ее статический тип *T0*, оператор

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

имеет следующий смысл. При типе специализации **v – T1** выполняется последовательность операторов **S1** в которой **v** воспринимается так, будто она имеет статический тип **T1**; иначе, если тип специализации **v – T2**, выполняется **S2**, где **v** воспринимается как имеющая статический

тип **T2**; иначе выполняется **S3**. **T1** и **T2** должны быть специализациями **T0**. Если ни одна проверка типа не удовлетворена, а ELSE отсутствует, программа прерывается. Специализация задается указанием признака. Пример:

```
WITH psh: Rectangle DO
    psh.x := 30; psh.y := 10
| psh: Triangle DO
    psh.a := 20; psh.b := 10; psh.c := 15
END
```

10. Объявления процедур

Объявление процедуры состоит из **заголовка процедуры** и **тела процедуры**. Заголовок определяет имя процедуры и **формальные параметры**. Для обобщенных и специализированных процедур в объявлении определяется список обобщаемых параметров. Тело содержит объявления и операторы. Имя процедуры повторяется в конце объявления процедуры.

Имеются два вида процедур: *собственно процедуры* и *процедуры-функции*. Последние активизируются обозначением функции как часть выражения и возвращают результат, который является операндом выражения. Собственно процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если ее формальные параметры задают тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет результат.

Все константы, переменные, типы и процедуры, объявленные внутри тела процедуры, *локальны* в процедуре. Поскольку процедуры тоже могут быть объявлены как локальные объекты, объявления процедур могут быть вложенными. Вызов процедуры изнутри ее объявления подразумевает рекурсивную активацию.

Объекты, объявленные в окружении процедуры, также видимы в тех частях процедуры, в которых они не перекрыты локально объявленным объектом с тем же самым именем.

ОбъявлПроцедуры = ЗаголовокПроцедуры ";" ТелоПроцедуры идент .
ЗаголовокПроцедуры = PROCEDURE ИдентОпр [ОбобщенныеПарам] [ФормальныеПараметры] .
ТелоПроцедуры = ПослОбъявлений [BEGIN ПослПомечОператоров] END .
ПослОбъявлений = {CONST {ОбъявлениеКонстант ";"} | TYPE {ОбъявлениеТипов ";"} | VAR {ОбъявлениеПеременных ";"}} {ОбъявлПроцедуры ";" | ОпережающееОбъявление";"} .
ОпережающееОбъявление = PROCEDURE "^" ИдентОпр [ОбобщенныеПарам] [ФормальныеПараметры] .

Если объявление процедуры содержит список обобщенных параметров, процедура рассматривается как обобщающая. Если объявление процедуры содержит список параметров-специализаций, процедура рассматривается как обработчик специализации. **Опережающее объявление** служит для того, чтобы разрешить ссылки на процедуру, чье фактическое объявление появится в тексте позже. Списки формальных параметров опережающего объявления и фактического объявления должны быть идентичны.

10.1 Формальные параметры

Формальные параметры - идентификаторы, объявленные в списке формальных параметров процедуры. Им соответствуют фактические параметры, которые задаются при вызове процедуры. Подстановка фактических параметров происходит при вызове процедуры. Имеются два вида параметров: *параметры-значения* и *параметры-переменные*, обозначаемые в

списке формальных параметров отсутствием или наличием ключевого слова VAR. Параметры-значения это локальные переменные, которым в качестве начального присваивается значение соответствующего фактического параметра. Параметры-переменные соответствуют фактическим параметрам, которые являются переменными, и означают эти переменные. Область действия формального параметра простирается от его объявления до конца блока процедуры, в котором он объявлен. Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызываться обозначением функции, чей список фактических параметров также пуст. Тип результата процедуры не может быть ни записью, ни массивом, ни обобщением.

ФормальныеПараметры = "(" [СекцияФП {";" СекцияФП }] ")"
[":" УточненныйИдент] .
СекцияФП = [VAR] идент {"," идент} ":" Тип .

Пусть $\mathbf{T_f}$ – тип формального параметра **f** (не открытого массива) и $\mathbf{T_a}$ – тип соответствующего фактического параметра **a**. Для параметров-переменных $\mathbf{T_a}$ и $\mathbf{T_f}$ должны быть одинаковыми типами или $\mathbf{T_f}$ должен быть обобщением, а $\mathbf{T_a}$ – специализацией. Для параметров-значений **a** должен быть **совместим по присваиванию** с **f**.

Если $\mathbf{T_f}$ – открытый массив, то **a** должен быть **совместимым массивом** для **f**. Длина **f** становится равной длине **a**.

Примеры объявлений процедур:

```
PROCEDURE ReadInt (VAR x: INTEGER);
      VAR i: INTEGER; ch: CHAR;
BEGIN i := 0; Read(ch);
      WHILE ("0" <= ch) & (ch <= "9") DO
            i := 10*i + (ORD(ch)-ORD("0")); Read(ch)
      END;
      x := i
END ReadInt

PROCEDURE WriteInt (x: INTEGER); // 0 <= x <100000
      VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN i := 0;
      REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i) UNTIL x = 0;
      REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt

PROCEDURE WriteString (s: ARRAY OF CHAR);
      VAR i: INTEGER;
BEGIN i := 0;
      WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString;

PROCEDURE log2 (x: INTEGER): INTEGER;
      VAR y: INTEGER; // предполагается x > 0
BEGIN
      y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
      RETURN y
END log2
```

10.2 Обобщающие параметрические процедуры

Основной задачей обобщающей параметрической процедуры является предоставление единой сигнатуры обработчикам параметрических специализаций, каждый из которых используется для обработки одной комбинации специализаций, составляющих параметрический список.

ОбобщающаяПроцедура = PROCEDURE ИдентОпр ОбобщенныеПарам
[ФормальныеПараметры] (":=" "0"
| ";" ПослОбъявлений
[BEGIN ПослПомечОператоров] END идент
).

Обобщенные параметры – идентификаторы, объявленные в списке обобщенных параметров. Им соответствуют обобщающие переменные, которые задаются при вызове обобщающей процедуры. Подстановка параметров при вызове процедуры и область видимости аналогичны формальным параметрам. Списки обобщенных параметров используются только с обобщающими процедурами и обработчиками параметрических специализаций. Указание пустого списка обобщенных параметров не допускается.

ОбобщенныеПарам ="{" ОбобщеннаяСекцияФП {";" ОбобщеннаяСекцияФП }"}".
ОбобщеннаяСекцияФП =[VAR] идент {"," идент } ":" УточнИдент.

Обобщенный параметр может быть или параметром-переменной типа *T*, если *T* - обобщение, или параметром-значением типа POINTER TO *T* (где *T* - обобщение). Обобщающая параметрическая процедура может быть уточнена обработчиком специализаций внутри этого же модуля или в любом из внешних модулей, импортирующем модуль с ее объявлением. Обобщающая параметрическая процедура может быть объявлена в этом же модуле или в любом модуле, импортирующем параметрические обобщения, используемые в списке обобщенных параметров.

Примеры:

```
PROCEDURE P {VAR s: Shape1}: INTEGER := 0

PROCEDURE P2 { ps: pShape2};
   BEGIN SendExeption('Incorrect parameter')
END P2
```

10.3 Обработчики параметрических специализаций

Обработчики параметрических специализаций обеспечивают реализацию кода для различных комбинаций параметрических обобщений. Комбинация, на которую "настроен" конкретный обработчик, задается значениями признаков при описании его сигнатуры в соответствии со следующими синтаксическими правилами:

ОбработчикСпециализации = PROCEDURE идент
СписокСпецПарам [ФормальныеПараметры] ТелоПроцедуры идент .
СписокСпецПарам = "{" ГруппаСпецПарам { ";" ГруппаСпецПарам } "}" .
ГруппаСпецПарам = [VAR] идент { "," идент } ":"
ОбобщающийТип "(" [Признак] ")"
| [VAR] идент "(" [Признак] ")"{ "," идент "(" [Признак] ")"}
":" ОбобщающийТип.
ОбобщающийТип = УточнИдент.

Каждый элемент списка специализированных параметров должен задавать конкретное значение признака в виде идентификатора, соответствующего методу описания признака в обобщении (имя типа или имя признака). Идентификатор признака может отсутствовать, если описывается специализация, обрабатываемая по умолчанию. Способы задания специализаций и их типы должны поэлементно соответствовать параметрам обобщающей процедуры. В обязательном теле процедуры кодируется конкретный метод обработки. Как и при описании переменных, может использоваться любой из двух способов задания специализаций, более удобный в рассматриваемом контексте.

Обработчики специализаций размещаются в тех же модулях, где находятся их обобщающие процедуры или в специальных подключаемых модулях, область видимости которых перекрывается с областью видимости родительских модулей (к которым осуществляется подключение).

Примеры:

```
/* Вычисление периметра прямоугольника */
PROCEDURE P {VAR r: Shape1(Rectangle)}: INTEGER;
      BEGIN RETURN r.x + r.y END P;

/* Вычисление периметра треугольника */
PROCEDURE P {VAR t(Triangle): Shape1}: INTEGER;
      BEGIN RETURN t.a + t.b + t.c END P;
```

10.4 Стандартные процедуры

Следующая таблица содержит список стандартных процедур. Некоторые процедуры - обобщенные, то есть они применимы к полиморфным операндам. Буква *v* обозначает переменную, *x* и *n* - выражения, *T* - тип.

Процедуры-функции

Название	Тип аргумента	Тип результата	Функция
ABS(x)	числовой тип	совпадает с типом x	абсолютное значение
ASH(x, n)	x, n: INTEGER	INTEGER	арифметический сдвиг ($x*2^n$)
CAP(x)	CHAR	CHAR	x - буква: соответствующая заглавная буква
CHR(x)	целый тип	CHAR	символ с порядковым номером x
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превосходящее x
LEN(v, n)	v: массив; n: целая константа	LONGINT	длина v в измерении n (первое измерение = 0)
LEN(v)	v: массив	LONGINT	равносильно LEN(v, 0)
LONG(x)	SHORTINT INTEGER REAL	INTEGER LONGINT LONGREAL	тождество
MAX(T)	T = основной тип T = SET	T INTEGER	наибольшее значение типа T наибольший элемент множества
MIN(T)	T = основной тип T = SET	T INTEGER	наименьшее значение типа T 0
ODD(x)	целый тип	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	порядковый номер x
SHORT(x)	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	тождество тождество тождество (возможно усечение)
SIZE(T)	любой тип	целый тип	число байт, занимаемых T

Собственно процедуры

Название	Типы аргументов	Функция
ASSERT(x)	x: логическое выражение	прерывает выполнение программы, если не x
ASSERT(x, n)	x: логическое выражение; n:целая	прерывает выполнение

	константа	программы, если не x
COPY(x, v)	x: символьный массив, строка; v: символьный массив	v := x
DEC(v)	целый тип	v := v – 1
DEC(v, n)	v, n: целый тип	v := v – n
EXCL(v, x)	v: SET; x: целый тип	v := v - {x}
HALT(n)	целая константа	прерывает выполнение программы
INC(v)	целый тип	v := v + 1
INC(v, n)	v, n: целый тип	v := v + n
INCL(v, x)	v: SET; x: целый тип	v := v + {x}
NEW(v)	указатель на запись или массив фиксированной длины	размещает v ^
NEW(v(t))	указатель на обобщение, уточненное признаком	размещает v(t) ^
NEW(v, x0, ..., xn)	v: указатель на открытый массив; xi: целый тип	размещает v^ с длинами x0.. xn

COPY разрешает присваивание строки или символьного массива, содержащего ограничитель 0X, другому символьному массиву. В случае необходимости, присвоенное значение усекается до длины получателя минус один. Получатель всегда будет содержать 0X как ограничитель. В ASSERT(x, n) и HALT(n), интерпретация n зависит от реализации основной системы. NEW(v(t)) создает динамическую переменную в свободной памяти, являющейся специализацией, тип которой определяется признаком *t*.

11. Модули

Модуль - совокупность объявлений констант, типов, переменных и процедур вместе с последовательностью операторов, предназначенных для присваивания начальных значений переменным. Модуль представляет собой текст, который является единицей компиляции.

Модуль = **MODULE идент ["(" идент ["*"|"+"] ")"] ";"**
[СписокИмпорта] ПоследовательностьОбъявлений
[BEGIN ПослПомечОператоров]
[FINALIZE ПослПомечОператоров]
END идент ".".
СписокИмпорта = **IMPORT Импорт {"," Импорт} ";".**
Импорт = **[идент ":="] идент.**
ПослПомечОператоров = **[Метка] Оператор {";" [Метка] Оператор}.**

В круглых скобках после имени модуля указывается модуль-родитель. Указание модуля-родителя позволяет напрямую подключить его пространство имен к создаваемому дочернему модулю. Знак «*» или «+» за именем родителя определяют права доступа к интерфейсу родительского модуля из модулей, импортирующих подключаемый модуль.

Список импорта определяет имена импортируемых модулей. Если модуль *A* импортируется модулем *M*, и *A* экспортирует идентификатор *x*, то *x* упоминается внутри *M* как *A.x*. Если *A* импортируется как *B:=A*, объект *x* должен вызываться как *B.x*. Это позволяет использовать короткие имена-псевдонимы в уточненных идентификаторах. Модуль не должен импортировать себя. Идентификаторы, которые экспортируются (то есть должны быть видимы в модулях-импортерах) нужно отметить экспортной меткой в их объявлении (см. Главу 4).

Использование модулей во многом определяется исполнительной системой, хотя в целом, ее архитектура не влияет на использование языка. В системах программирования разработанных Виртом [1], последовательность операторов модуля после символа BEGIN

выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, что циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы. Эти процедуры служат *командами*.

Последовательности операторов различных модулей, расположенные после ключевого слова **BEGIN**, начинают выполняться после загрузки исполняемого модуля в том порядке, который определен иерархией импорта и подключения (для подключаемых модулей). Последовательности операторов после ключевого слова **FINALIZE** начинают выполняться перед выгрузкой модуля. При этом выполнение завершающих действий происходит в последовательности, обратной той, в которой происходила загрузка. Существует модуль, являющийся главным в иерархии. Именно он запускает процесс, определяющий целевое назначение программы. Исполняемый файл является цельной и законченной программой. Использование процедур отдельных модулей в качестве команд не реализовано.

Примечание. Следует отметить, что реализованная схема не является единственно возможной и впоследствии может быть изменена в соответствии с тенденциями, определяющими развитие исполнительной системы.

Примечание. *Разработанная система программирования на языке O2M реализована по традиционному принципу. Необходимые модули собираются в единый проект, и являются отдельными единицами компиляции. Компилятор преобразует исходные тексты модулей в файлы программы на языке программирования **C++**. Помимо этого создается файл проекта для трансляции с этого языка. После компиляции и компоновки проекта средствами C++ создается единый исполняемый файл. Последовательности операторов различных модулей, расположенные после ключевого слова **BEGIN**, начинают выполняться после загрузки исполняемого модуля в том порядке, который определен иерархией импорта. Существует модуль, являющийся главным в иерархии. Именно он запускает процесс, определяющий целевое назначение программы. Исполняемый файл является цельной и законченной программой. Использование процедур отдельных модулей в качестве команд не реализовано. Следует отметить, что реализованная схема не является единственно возможной и впоследствии может быть изменена в соответствии с тенденциями, определяющими развитие исполнительной системы.*

Пример модуля:

```
MODULE Trees;              // экспорт: Tree, Node, Insert, Search, Write, Init
   IMPORT Texts, Oberon; // экспорт только для чтения: Node.name

   TYPE
         Tree* = POINTER TO Node;
         Node* = RECORD
               name-: POINTER TO ARRAY OF CHAR;
               left, right: Tree
         END;

   VAR w: Texts.Writer;

   PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
         VAR p, father: Tree;
   BEGIN p := t;
         REPEAT father := p;
               IF name = p.name^ THEN RETURN END;
               IF name < p.name^ THEN p := p.left ELSE p := p.right END
         UNTIL p = NIL;
         NEW(p); p.left := NIL; p.right := NIL;
         NEW(p.name, LEN(name)+1); COPY(name, p.name^);
         IF name < father.name^
         THEN father.left := p
```

```
        ELSE father.right := p END
    END Insert;

    PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
        VAR p: Tree;
    BEGIN p := t;
        WHILE (p # NIL) & (name # p.name^) DO
            IF name < p.name^ THEN p := p.left ELSE p := p.right END
        END;
        RETURN p
    END Search;

    PROCEDURE (t: Tree) Write*;
    BEGIN
        IF t.left # NIL THEN t.left.Write END;
        Texts.WriteString(w, t.name^); Texts.WriteLn(w);
        Texts.Append(Oberon.Log, w.buf);
        IF t.right # NIL THEN t.right.Write END
    END Write;

    PROCEDURE Init* (t: Tree);
    BEGIN NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
    END Init;

BEGIN Texts.OpenWriter(w)
END Trees.
```

Приложение A. Определение терминов

Целые типы INTEGER, UNSIGNED
Вещественные типы REAL
Числовые типы Целый тип, вещественный тип
Одинаковые типы

Две переменные *a* и *b* с типами *Ta* и *Tb* имеют *одинаковый* тип, если

1. *Ta* и *Tb* оба обозначены одним и тем же идентификатором типа, или
2. *Ta* объявлен равным *Tb* в объявлении типа вида *Ta* = *Tb*, или
3. *a* и *b* появляются в одном и том же списке идентификаторов переменных, полей записи или объявлении формальных параметров и не являются открытыми массивами.

Равные типы

Два типа *Ta*, и *Tb равны*, если

1. *Ta* и *Tb* - одинаковые типы, или
2. *Ta* и *Tb* - типы открытый массив с *равными* типами элементов, или
3. *Ta* и *Tb* - процедурные типы, чьи списки формальных параметров совпадают.

Поглощение типов

Поглощение типов при обработке переменных отсутствует. Используется явное преобразование одного типа в другой. Допускается автоматическое приведение числовых констант к типам операндов в операциях, при присваивании и подстановке фактических параметров. При этом целые константы могут приводится к целому, беззнаковому и действительному типам. Неявное преобразование действительных констант к целому типу не допускается.

Обобщение типов (обобщающий тип)

В объявлении типа *Ts = CASE ... OF ... Sb ...END, Sb - специализация*, а *Ts - обобщающий тип для Sb*, или просто *обобщение Sb*. Специализация Sb задается как *tb:Tb* где *tb* – признак специализации, а *Tb* – ее тип. Тип специализации может совпадать по имени с признаком (тогда, вместо *Tb:Tb*, следует писать только *Tb*), или задаваться идентификатором, локальным обобщению или значением *NIL*.

В объявлении *Ts(tb)*, *Ts(tb) - параметризация Ts признаком tb*. Тип-обобщение *Ts* может быть *параметризован* признаком *tb*, если тип *Ts* - обобщение специализации *Sb*. Обобщенные переменные имеют тип обобщения, параметризованного одним из типов, входящих в обобщение.

Если *Ps = POINTER TO Ts,* то *Ps - указатель на обобщение*. Пусть *VAR p: Ps*, тогда вызов *NEW(p(tb)),* создает динамическую переменную *p*, которая *указывает на параметризацию Ts типом Tb*. В данном случае *p* является динамической обобщенной переменной.

Совместимость по присваиванию

Выражение *e* типа *Te совместимо по присваиванию* с переменной *v* типа *Tv*, если выполнено одно из следующих условий:

1. *Te* и *Tv* - одинаковые типы;
2. *Te* и *Tv* - числовые типы и *Tv* поглощает константное выражение *Te*;
3. *Tv* - тип указатель или процедурный тип, а *e* - NIL;
4. *Tv* - ARRAY *n* OF CHAR, *e* - строковая константа из *m* символов и *m* < *n*;
5. *Tv* - процедурный тип, а *e* - имя процедуры, чьи формальные параметры совпадают с параметрами *Tv*;
6. *Tv* –обобщение, параметризированное некоторым признаком, а *Te* – тип, соответствующий заданному признаку;
7. *Tv* - указатель на обобщение, и *e* - динамическая специализация этого обобщения.

Совместимость массивов

Фактический параметр *a* типа *Ta* является совместимым массивом для формального параметра *f* типа *Tf* если

1. *Tf* и *Ta* - одинаковые типы или
2. *Tf* - открытый массив, *Ta* - любой массив, а типы их элементов - *совместимые массивы* или
3. *f* - параметр-значение типа ARRAY OF CHAR, а фактический параметр *a* - строка.

Совместимость выражений

Для данной операции операнды являются *совместимыми выражениями*, если их типы соответствуют следующей таблице (в который указан также тип результата выражения). Символьные массивы, которые сравниваются, должны содержать в качестве ограничителя 0X. Тип T1 должен быть расширением типа T0, или обобщением, параметризованным типом T0:

операция	*Первый операнд*	*второй операнд*	*тип результата*
+ - *	числовой	числовой	число того же типа, что и операнды
/	числовой	числовой	вещественный тип
DIV MOD	целый	целый	целый тип
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN

= # < <= > >=	числовой CHAR символьный массив, строка	числовой CHAR символьный массив, строка	BOOLEAN BOOLEAN BOOLEAN
= #	BOOLEAN SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	BOOLEAN SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	BOOLEAN BOOLEAN BOOLEAN BOOLEAN
IN	целый	UNSIGNED	BOOLEAN
IS	тип T0	тип T1	BOOLEAN

Совпадение списков формальных параметров

Два списка формальных параметров *совпадают* если

1. они имеют одинаковое количество параметров, и
2. они имеют или одинаковый тип результата функции или не имеют никакого, и
3. параметры в соответствующих позициях имеют равные типы, и
4. параметры в соответствующих позициях - оба или параметры-значения или параметры-переменные.

Приложение Б. Синтаксис Alien

Модуль	=	**MODULE идент ["(" идент ["*"|"+"] ")"] ";" [СписокИмпорта] ПослОбъявлений [BEGIN ПослПомечОператоров [FINALIZE ПослПомечОператоров]] END идент ".".**
СписокИмпорта	=	**IMPORT Импорт {"," Импорт} ";".**
Импорт	=	**[идент ":="] идент.**
ПослОбъявлений	=	**{ CONST { ОбъявлениеКонстанты ";" } | TYPE { ОбъявлениеТипа ";" } | VAR {ОбъявлПерем ";" }} | {ОбъявлПроцедуры ";" | ОпережающееОбъявление";"}.**
ОбъявлениеКонстанты	=	**ИдентОпр "=" КонстантноеВыражение.**
КонстантноеВыражение	=	**Выражение.**
ОбъявлениеТипа	=	**(ИдентОпр "=" Тип) | Расширение.**
Тип	=	**УточнИдент | ТипМассив | ТипЗапись | ТипУказатель | ПроцедурныйТип | ТипОбобщение.**
ТипМассив	=	**ARRAY [КонстантноеВыражение {"," КонстантноеВыражение }] OF Тип**
ТипОбобщение	=	**CASE [TYPE] [LOCAL] OF [СписокСпециализаций] [ELSE Специализация] END .**
СписокСпециализаций	=	**((СписокПризнаков ":" (Тип | NIL)) | УточнИдент) { "|" ((СписокПризнаков ":" (Тип | NIL)) | УточнИдент) } .**
Специализация	=	**(идент ":" (УточнИдент | NIL)) | УточнИдент.**
СписокПризнаков	=	**идент { "," идент } .**
ТипЗапись	=	**RECORD [СписокПолей {";" СписокПолей}]**

		(ТипОбобщение | [CASE ИмяОбобщения] END).
ИмяОбобщения	=	**УточнИдент.**
СписокПолей	=	**СписокИдент ":" Тип.**
ТипУказатель	=	**POINTER TO Тип**
ПроцедурныйТип	=	**PROCEDURE [ФормальныеПарам]**
Расширение	=	**УточнИдент "+=" СписокСпециализаций .**
ОбъявлПерем	=	**(СписокИдент ":" Тип) | ОбъявлОбобщПерем.**
ОбъявлОбобщПерем	=	**СписокИдент ":" УточнИдент "(" [Признак] ")" | идент"(" [Признак] ")" {“,” идент"(" [Признак] ")" } ":" УточнИдент.**
ОбъявлПроцедуры	=	**ЗаголовокПроцедуры ";" ТелоПроцедуры идент.**
Признак	=	**УточнИдент .**
ЗаголовокПроцедуры	=	PROCEDURE ИдентОпр [ОбобщенныеПарам] [ФормальныеПараметры].
ТелоПроцедуры	=	ПослОбъявлений [BEGIN ПослПомечОператоров] END.
ОбобщающаяПроцедура	=	PROCEDURE ИдентОпр ОбобщенныеПарам [ФормальныеПараметры] (":=" "0" | ";" ПослОбъявлений [BEGIN ПослПомечОператоров] END идент).
ОбработчикСпециализации	=	PROCEDURE идент СписокСпецПарам [ФормальныеПараметры] ТелоПроцедуры идент.
СписокСпецПараметров	=	"{" ГруппаСпецПараметров { ";" ГруппаСпецПараметров } "}" .
ГруппаСпецПараметров	=	[VAR] идент { "," идент } ":" ОбобщающийТип "(" [Признак] ")" | [VAR] идент "(" [Признак] ")"{ "," идент "(" [Признак] ")"} ":" ОбобщающийТип.
ОпережающееОбъявление	=	PROCEDURE "^" ИдентОпр [ОбобщенныеПарам] [ФормальныеПараметры].
ФормальныеПарам	=	"(" [СекцияФП {";" СекцияФП}] ")" [":" УточнИдент].
СекцияФП	=	[VAR] идент {"," идент} ":" Тип.
ОбобщенныеПарам	=	"{" ОбобщеннаяСекцияФП {";" ОбобщеннаяСекцияФП } "}".
ОбобщеннаяСекцияФП	=	[VAR] идент {"," идент } ":" УточнИдент.
СписокПолей	=	[СписокИдент ":" Тип].
ОбобщающийТип	=	УточнИдент.
ПослПомечОператоров	=	[Метка] Оператор {";" [Метка] Оператор}.
ПослОператоров	=	Оператор {«;» Оператор}.
Оператор	=	[Обозначение «:=» Выражение | Обозначение [ФактическиеПараметры] | IF Выражение THEN ПослОператоров {ELSIF Выражение THEN ПослОператоров} [ELSE ПослОператоров] END | CASE Выраж OF Вариант {«|» Вариант} [ELSE ПослОператоров] END | WHILE Выраж DO ПослОператоров END | REPEAT ПослОператоров UNTIL Выраж | FOR идент «:=» Выраж TO Выраж [BY КонстантноеВыражение] DO ПослОператоров END | LOOP ПослОператоров END | WITH Охрана DO ПослОператоров {«|» Охрана DO ПослОператоров}

		[ELSE ПослОператоров] END | EXIT | RETURN [Выраж]].
Вариант	=	[МеткиВарианта {«,» МеткиВарианта} «:» ПослОператоров].
МеткиВарианта	=	КонстантноеВыражение [«..» КонстантноеВыражение].
Охрана	=	УточнИдент «:» УточнИдент.
Выражение	=	ПростоеВыражение [Отношение ПростоеВыражение].
ПростоеВыражение	=	[«+» | «-«] Слагаемое {ОперСлож Слагаемое}.
Слагаемое	=	Множитель {ОперУмн Множитель}.
Множитель	=	Обозначение [ФактическиеПараметры] | число | символ | строка | NIL | Множество | "(" Выражение ")" | "~" Множитель.
ФактическиеПараметры	=	"{" [СписокВыражений] "}" ["(" [СписокВыражений] ")"] | "(" [СписокВыражений] ")".
Множество	=	«{« [Элемент {«,» Элемент}] «}».
Элемент	=	Выражение [«..» Выражение].
Отношение	=	«=» | «#» | «<» | «<=» | «>» | «>=» | IN | IS.
ОперСлож	=	«+» | «-« | OR | XOR.
ОперУмн	=	«*» | «/» | DIV | MOD | «&».
Обозначение	=	УточнИдент {«.» идент | «[«СписокВыражений «]» | «^» | «(« [УточнИдент] «)»}.
СписокВыражений	=	Выражение {«,» Выражение}.
СписокИдент	=	ИдентОпр {«,» ИдентОпр}.
УточнИдент	=	[идент «.»] идент.
ИдентОпр	=	идент [«*» | «-»].
Метка	=	идент «:».

Литература

1. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования. – Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.
2. The Programming Language Oberon-2 H.Moessenboeck, N.Wirth. Institut fur Computersysteme, ETH Zurich July 1996.
3. Мёссенбёк Х., Вирт Н. Язык программирования Оберон-2 / Перевод Свердлова С.З. (материал размещен по адресу: http://www.uni-vologda.ac.ru/oberon/o2rus.htm).
4. Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с.
5. Легалов А.И. Процедурно-параметрическое программирование (материал размещен по адресу: http://www.softcraft.ru/paradigm/ppp/ppp01.shtml).
6. Страуструп, Б. Язык программирования C++. Третье издание.: Пер. с англ. / Б. Страуструп – СПб.; М.: "Невский диалект" – "Издательство БИНОМ", 1999. – 991 с.
7. Гуннерсон, Э. Введение в C#. Библиотека программиста. / Э. Гуннерсон – СПб: Питер, 2001. – 304 с.
8. Легалов А.И., Швец Д.А. Язык программирования O2M . (материал размещен по адресу: http://www.softcraft.ru/ppp/o2m/o2mref.shtml).
9. Легалов А.И. Мультиметоды и парадигмы. – Открытые системы, № 5 (май) 2002, с. 33-37.

10. Легалов А.И. Эволюция мультиметодов при процедурном подходе (материал размещен по адресу: http://www.softcraft.ru/coding/evp/evp.shtml).