Язык программирования Оберон

Редакция 1.11.2008

Никлаус Вирт

Делай настолько просто, насколько возможно, но не проще. (А. Эйнштейн)

Содержание

1. Введение

Оберон - язык программирования общего назначения, являющийся развитием языка Модула-2. Его принципиально новой особенностью является понятие расширения типа. Это позволяет создавать новые типы данных на основе существующих и связывать их.

Этот документ не является учебником по программированию. Он преднамеренно краток. Его функция в том, чтобы служить справочным пособием для программистов, разработчиков, и писателей руководств. Кое-что осталось недосказанным сознательно, либо потому, что это следует из правил языка, либо потому, что это неоправданно ограничивает свободу разработчиков.

Этот документ описывает язык определенный в 1988/90 в редакции 2007 года.

2. Синтаксис

Язык это бесконечное множество предложений – предложений, правильно оформленных в согласии с его синтаксисом. В Обероне эти предложения называются единицами компиляции. Каждая единица это конечная последовательность символов из некоторого конечного словаря. Словарь Оберона состоит из идентификаторов, чисел, строк, операторов, разделителей и комментариев. Они называются *лексическими символами* и образуются последовательностью знаков. (Обратите внимание на разницу между символами и знаками.)

Для описания синтаксиса используются Расширенные Формы Бэкуса-Наура (EBNF). Квадратные скобки [и] означают необязательность записанного внутри них выражения, а фигурные скобки { и } означают его повторение (возможно 0 раз). Синтаксические сущности (нетерминальные символы) обозначают английскими словами выражающими их интуитивное значение. Символы словаря языка (терминальные символы) обозначают строками заключенными в кавычки или словами написанными заглавными буквами, так называемыми *зарезервированными словами*.

3. Словарь и представление

Представление символов в терминальных знаках определяются с помощью набора Latin-1. Символы - это идентификаторы, числа, строки, операторы, разделители, и комментарии. Должны быть соблюдены следующие лексические правила. Пробелы и разрывы строк не должны встречаться внутри символов (исключая комментарии, и пробелы в символьных строках). Они игнорируются если они несущественны для разделения двух следующих друг за другом символов. Прописные и строчные буквы считаются различными.

1. Идентификаторы это последовательность букв и цифр. Первый знак должен быть буквой.

ident = letter {letter | digit}.

Примеры:

x scan Oberon GetSymbol firstLetter

2. Числа это (без знаковые) целые или вещественные числа. Целые это последовательность цифр за которой может следовать суффикс-буква. Если суффикс не указывается, представление десятичное. Суффикс H указывает на шестнадцатеричное представление.

Вещественное число всегда содержит десятичную точку. Оно может также содержать десятичный порядок. Литера E читается как "умножить на десять в степени". Вещественное число имеет тип REAL, если не задан масштабный коэффициент литерой D, в этом случае его тип LONGREAL.

number = integer | real.

integer = digit {digit} | digit {hexDigit} "H".

real = digit {digit} "." {digit} [ScaleFactor].

ScaleFactor = ("E" | "D“) ["+" | "–"] digit {digit}.

hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

Примеры:

1987

100H = 256

12.3

4.567E8 = 456700000

3. Символьная константа обозначается либо одиночным символом заключенным в кавычки либо порядковым номером символа в шестнадцатеричном виде за которым следует литера X.

CharConst = """ character """ | digit {hexDigit} "X".

Примеры:

‘A’ ‘%’ 22X (но не “%”)

4. Строки представляют собой последовательности символов, заключенных в кавычки. Строка не может содержать ограничивающую кавычку. Строки заканчиваются нулевым символом. Количество символов в строке (включая нулевой символ) называется *длиной* строки. Строки могут быть присвоены и сравнены с массивами символов (см. 9.1 и 8.2.4).

string = """ {character} """ | "'" character "'".

Примеры:

"OBERON" "Don't worry!" ‘also’ “%” (но не ‘%’)

5. Операторы и разделители это специальные символы, пары символов, или зарезервированные слова, перечисленные ниже. Эти зарезервированные слова состоят исключительно из заглавных букв и не могут быть использованы в роли идентификаторов.

+	:=	ARRAY	IMPORT	THEN
-	^	BEGIN	IN	TO
*	=	BY	IS	TRUE
/	#	CASE	MOD	TYPE
~	<	CONST	MODULE	UNTIL
&	>	DIV	NIL	VAR
.	<=	DO	OF	WHILE
,	>=	ELSE	OR	
;	..	ELSIF	POINTER	
|	:	END	PROCEDURE	
(	)	FALSE	RECORD	

[	]	FOR	REPEAT
{	}	IF	RETURN

6. Комментарии могут быть вставлены между любыми двумя символами в программе. Это произвольная последовательность символов начинающаяся с (* и заканчивающаяся *). Комментарии не влияют на смысл программы. Они могут быть вложенными.

4. Объявления и правила области видимости

Каждый идентификатор, встречающийся в программе должны быть объявлен заранее, если он не является предопределённым идентификатором. Объявления также служат для задания определенных постоянных свойств объекта, например, является ли он константой, типом, переменной или процедурой.

Позднее идентификатор используется для ссылки на соответствующий объект. Это возможно только в тех частях программы, которые находятся внутри *области* объявления. Никакой идентификатор не может указывать более чем на один объект внутри данной области. Область распространяется текстуально от места объявления до конца блока (процедуры или модуля) к которому объявление принадлежит и по отношению к которому объект является локальным. Правило видимости имеет следующие поправки:

1. Если тип *P* определяется как POINTER TO T (см. 6.4), то идентификатор *T* может быть объявлен по тексту ниже объявления *P*, но он должен находиться в той же области.

2. Идентификаторы полей объявленной записи (см. 6.3) действительны только в обозначениях записи.

При объявлении идентификатора в блоке модуля за ним может следовать метка экспорта (*), что указывает, что он будет *экспортироваться* из объявления модуля. В этом случае, идентификатор может использоваться в других модулях, если они импортируют объявления модуля. Тогда у идентификатора будет идентификатор-префикс обозначающий модуль (см. Гл. 11). Префикс и идентификатор разделены точкой и вместе называются *уточнённый идентификатор*.

qualident = [ident "."] ident.

identdef = ident ["*"].

Следующие идентификаторы предопределены; их значение описано в разделе 6.1 (типы) или 10.2 (процедуры):

ABS	ASR	ASSERT	BOOLEAN	CHAR
CHR	DEC	FLOOR	FLT	INC
INTEGER	LEN	LSL	LSR	LONGREAL
NEW	ODD	ORD	PACK	REAL
SET	UNPK			

5. Объявления констант

Объявление константы ассоциирует идентификатор с константным значением.

ConstantDeclaration = identdef "=" ConstExpression.

ConstExpression = expression.

Константное выражение может быть вычислено при транслировании текста программы без реального её выполнения. Его операнды константы (см. Гл. 8). Примеры объявления констант:

N = 100

limit = 2*N -1

all = {0 .. WordSize-1}

6. Объявления типов

Тип данных определяет множество значений, которое могут принимать переменные этого типа, и применимые операции. Объявление типа используется для связи идентификатора с типом. Типы определяют структуру переменных этого типа и, как следствие, операции, которые применимы к их компонентам. Существуют две различные структуры, а именно: массивы и записи, которые отличаются селектором.

TypeDeclaration = identdef "=" StrucType.

StrucType = ArrayType | RecordType | PointerType | ProcedureType.

type = qualident | StrucType.

Примеры:

```
Table       =   ARRAY N OF REAL
Tree        =   POINTER TO Node
Node        =   RECORD
                    key: INTEGER;
                    left, right: Tree
                END
CenterNode  =   RECORD (Node)
                    name: ARRAY 32 OF CHAR;
                    subnode: Tree
                END
Function    =   PROCEDURE (x: INTEGER): INTEGER
```

6.1. Основные типы

Основные типы обозначаются предопределёнными идентификаторами. Соответствующие операции определены в 8.2, а предопределенные процедуры-функции в 10.2. Предусмотрены следующие основные типы:

1. BOOLEAN логические значения TRUE и FALSE.
2. CHAR символы расширенного набора Latin-1.
3. INTEGER целые в интервале от -231 and +231-1.
4. REAL вещественные числа (IEEE Standard, 32 bits).
5. LONGREAL длинные вещественные числа (IEEE Standard, 64 bits).
6. SET множества целых чисел из диапазона от 0 до 31.

6.2. Тип-массив

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

ArrayType = ARRAY length {"," length} OF type.

length = ConstExpression.

Объявление вида

ARRAY N0, N1, ... , Nk OF T

понимается как сокращение объявления

```
ARRAY N0 OF
ARRAY N1 OF
...
ARRAY Nk OF T
```

Примеры типа-массив:

```
ARRAY N OF INTEGER
ARRAY 10, 20 OF REAL
```

6.3. Тип-запись

Запись - это структура, состоящая из фиксированного количества элементов возможно различных типов. Запись определяет для каждого элемента, называемого *полем*, его тип и идентификатор, который обозначает это поле. Область видимости этих идентификаторов полей - само определение записи, но они также видимы внутри составных имен (см. 8.1), обозначающих поля переменных данного типа.

RecordType = RECORD ["(" BaseType ")"] [FieldListSequence] END.

BaseType = qualident.

FieldListSequence = FieldList {";" FieldList}.

FieldList = IdentList ":" type.

IdentList = identdef {"," identdef}.

Если тип-запись экспортируется, поля которые должны быть видимы вне объявления модуля должны быть отмечены. Они называются *общедоступными полями*, не отмеченные - *приватными полями*.

Типы-запись - расширяемы, т.е. тип-запись может быть определена как расширение другого тип-записи. Например представленный выше тип *CenterNode* расширяет (напрямую) тип *Node*, который является его (прямым) базовым типом. Точнее, *CenterNode* расширяет *Node*, добавляя поля *name* и *subnode*.

Определение: тип *T* расширяет тип *T0*, если он и есть *T0*, или если он непосредственно расширяет расширение *T0*. И наоборот, тип *T0* - базовый тип для *T*, если он и есть *T* или является прямым базовым типом базового типа *T*.

Пример типа-записи:

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

6.4. Тип-указатель

Переменные типа-указатель *P0* принимают в качестве значений указатели на переменные некоторого типа *T0*. Этот тип должен быть записью. Тип-указатель *P0*, как говорится *связан с типом T0*, и *T0* это *базовый тип указателя P0*. Типы-указатель наследуют отношения расширения базового типа, при их наличии. Если тип *T* - расширение *T0* и *P* - тип-указатель связанный с *T*, тогда *P* также является расширением *P0*.

PointerType = POINTER TO type.

Если *p* - переменная типа P = POINTER TO T, то вызов встроенной функции NEW(p) производит следующее (см. 10.2): в свободной памяти выделяется место для переменной типа *T*, и указатель на неё присваивается *p*. Этот указатель *p* типа *P* ссылается на переменную *p^* типа *T*. Ошибка выделения памяти приводит к присваиванию *p* значения *NIL*. Любой переменной-указателю может быть присвоено значение *NIL*, которое не указывает ни на какую переменную вообще.

6.5. Процедурные типы

Переменные процедурного типа *T* принимают в качестве значений процедуру или NIL. Если процедура *P* связана с процедурной переменной типа *T*, типы формальных параметров процедуры *P* должны совпадать с типами соответствующих параметров типа *T*. То же касается и типа результата, для процедур-функций (см. 10.1). Тип *P* не может быть определен локально, и не может принимать стандартные процедуры.

ProcedureType = PROCEDURE [FormalParameters].

7. Описание переменных

Описание переменных позволяет вводить переменные и связывать их с идентификаторами, которые должны быть уникальными в своём контексте. Создавать переменные можно только для фиксированных типов.

VariableDeclaration = IdentList ":" type.

Переменные, идентификаторы которых перечислены в списке, имеют один общий тип. Пример описания переменных (ссылка на примеры из Гл. 6):

```
i, j, k:  INTEGER
x, y:     REAL
p, q:     BOOLEAN
s         SET
a:        ARRAY 100 OF REAL
```

```
w:      ARRAY 16 OF
            RECORD ch: CHAR;
                   count: INTEGER
            END
t:      Tree
```

8. Выражения

Выражение - это конструкция, содержащая правила вычисления значений с использованием констант и текущих значений переменных, объединённых с получением значений операторов и функций. Выражение состоит из операндов и операторов. Выражение состоит из операндов и операторов. Скобки могут использоваться для выражения связи между операндами и операторами.

8.1. Операнды

За исключением множеств и литерных констант, таких, как числа и символьные строки, операнды определяются описателями. Описатель может быть идентификатором константы, переменной или процедуры. Этот идентификатор может быть уточнён идентификатором модуля (см. Гл. 4 и 11) и может следовать за селектором, если описываемый объект - часть структуры.

Если *A* определена как массив, то *A[E]* обозначает, элемент массива *A*, чей индекс - текущее значение выражения *E*. *E* должно иметь тип INTEGER. Определение вида *A[E1, E2..., En]* обозначает *A[E1][E2]...[En]*. Если *p* определена как переменная указатель, тогда *p^* обозначает переменную, на которую ссылается *p*. Если *r* определена как запись, то *r.f* обозначает поле *f* в записи *r*. Если *p* определена как указатель, *p.f* обозначает поле *f* записи *p^*, то есть точка подразумевает разыменование т.о. *p.f* означает *p^.f*.

Охрана типа *v(T)* требует, чтобы у *v* был тип *T*, то есть выполнение программы прерывается, если тип *v* не *T*. Охрана применима, если

1. *T* является расширением описанного типа *T0*, и если

2. *v* - параметр-переменная типа запись, или *v* - указатель.

```
designator = qualident {selector}.
selector = "." ident | "[" ExpList "]" | “^” | "(" qualident ")".
ExpList = expression {"," expression}.
```

Если обозначенный объект является переменной, то обозначение ссылается на текущее значение переменной. Если объект - процедура, обозначение без списка параметров ссылается на эту процедуру. Если за ним следует (возможно, пустой) список параметров, обозначение подразумевает активацию процедуры и обозначает возвращаемый результат её исполнения. Фактические параметры (и их типы) должны соответствовать формальным параметрам, указанным при декларации процедуры (см. Гл. 10).

Примеры обозначений (см. примеры из Гл. 7):

```
i                       (INTEGER)
a[i]                    (REAL)
w[3].ch                 (CHAR)
t.key                   (INTEGER)
t.left.right            (Tree)
t(CenterNode).subnode   (Tree)
```

8.2. Операции

В синтаксисе выражений различается четыре класса операций с различными приоритетами (порядком выполнения). Операция ~ имеет наивысший приоритет, затем идут мультипликативные операции, аддитивные операции, и отношения. Операции одинакового приоритета выполняются слева направо. Например, *x-y-z* обозначает *(x-y)-z*.

```
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["+"|"-"] term {AddOperator term}.
AddOperator = "+" | "-" | “OR”.
term = factor {MulOperator factor}.
```

```
MulOperator = "*" | "/" | DIV | MOD | "&" .
factor = number | CharConst | string | NIL | TRUE | FALSE | set | designator [ActualParameters]
        | "(" expression ")" | "~" factor.
set = "{" [element {"," element}] "}".
element = expression [".." expression].
ActualParameters = "(" [ExpList] ")".
```

Доступные операции приведены в следующей таблице. В некоторых случаях, несколько различных операций обозначаются одним и тем же символом операции. В этих случаях, фактическая операция определяется типом операндов.

8.2.1. Логические операции

символ	результат
OR	логическая дизъюнкция
&	логическая конъюнкция
~	отрицание

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN.

P OR q	означает	"если p, то TRUE, иначе q"
p & q	означает	"если p то q, иначе FALSE"
~p	означает	"не p"

8.2.2. Арифметические операции

символ	результат
+	сумма
-	разность
*	произведение
/	частное
DIV	целое частное
MOD	остаток от деления

Операции +, -, *, и / применимы к операндам числовых типов. Оба операнда должны быть одинакового типа, который также является типом результата. При использовании в качестве унарной операции "-" обозначает инвертирование знака, а "+" - тождественную операцию.

Операции DIV и MOD применимы только к операндам типа INTEGER. Пусть q = x DIV y, а r = x MOD y. Тогда частное *q* и остаток *r* определены уравнением

x = q*y + r 0 ≤ r < y

8.2.3 Операции над множествами

символ	результат
+	объединение
-	разность
*	пересечение
/	симметрическая разность множеств

Когда используется с одиночным операндом множества знак "минус", то это означает дополнение множества.

8.2.4. Отношения

символ	результат
=	равно
#	неравно
<	меньше
<=	меньше или равно

>	больше
>=	больше или равно
IN	принадлежность множеству
IS	проверка типа

Отношения дают результат типа BOOLEAN. Отношения <, <=, >, >= применяются к числовым типам, CHAR, и символьным массивам (строкам). Отношения = и # также применяются к типам BOOLEAN и SET, и к указателям и процедурам. Отношения <= и >= обозначают включение, когда применяются к множествам.

x IN s означает "x является элементом s". x должен быть типа INTEGER, a s - типа SET.

v IS T означает "тип v есть T" и называется проверкой типа. Он применим если

1. T является расширением типа T0 с которым объявлена v, и если

2. v это изменяемый параметр типа запись, или v это указатель.

Предположим, например, что T это расширение T0 и что v - объявленный указатель типа T0, тогда тест v IS T определяет, является ли фактический тип переменной (не только T0, но также и) T. Значение NIL IS T неопределенно.

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

1987	(INTEGER)
i DIV 3	(INTEGER)
~p OR q	(BOOLEAN)
(i+j) * (i-j)	(INTEGER)
s - {8, 9, 13}	(SET)
i + x	(REAL)
a[i+j] * a[i-j]	(REAL)
(0<=i) & (i<100)	(BOOLEAN)
t.key = 0	(BOOLEAN)
k IN {i .. j-1}	(BOOLEAN)
t IS CenterNode	(BOOLEAN)

9. Операторы

Операторы обозначают действия. Есть простые и структурные операторы. Простые операторы не состоят ни из каких частей, и являются самостоятельными операторами. Это присваивание и вызов процедуры. Структурные операторы состоят из частей, которые являются самостоятельными операторами. Они используются, чтобы выразить последовательное, условное, выборочное и повторное выполнение. Оператор может также быть пустым, тогда он не обозначает никакого действия. Пустой оператор добавлен, чтобы упростить правила пунктуации в последовательности операторов.

statement = [assignment | ProcedureCall | IfStatement | CaseStatement |

WhileStatement | RepeatStatement | ForStatement].

9.1. Присваивания

Присваивание служит, чтобы заменить текущее значение переменной новым значением, определяемым выражением. Оператор присваивания пишется ":=" и произноситься как присвоить.

assignment = designator ":=" expression.

Тип выражения должен быть таким же как и у переменной. Существуют следующие исключения:

1. Константа NIL может быть присвоена любым переменным типа указатель или процедура.

2. У массивов должен быть тот же самый тип элементов и длина целевого массива не должна быть меньше чем длина исходного массива.

3. Строки могут быть присвоены любому массиву символов, если строка не длиннее, чем массив.

4. В случае записей, тип источника должен быть расширением целевого типа.

Примеры присваиваний (см. примеры из Гл. 7):

```
i := 0
p := i = j
x := FLT(i + 1)
k := (i + j) DIV 2
F := log2
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].ch := "A"
```

9.2. Вызовы процедур

Вызов процедуры служит, чтобы активировать процедуру. Вызов процедуры может содержать список фактических параметров, которые заменяют соответствующие формальные параметры, определенные при описании процедуры (см. Гл. 10). Соответствие устанавливается в порядке следования параметров в списках фактических и формальных параметров. Существуйте три вида параметров: константы, параметры-переменные и параметры-значения.

В случае переменных-параметров фактический параметр должен быть указателем, указывающим на переменную. Если он указывает на элемент структурной переменной, селекторы компонент вычисляются, когда происходит замена формальных параметров фактическими, то есть перед выполнением процедуры. Если параметр - константа или параметр-значение, соответствующий фактический параметр должен быть выражением. Это выражение вычисляется до активации процедуры, и полученное значение присваивается формальному параметру, который теперь представляет собой локальную переменную (см. также 10.1.).

```
ProcedureCall = designator [ActualParameters].
```

Примеры вызовов процедур:

```
ReadInt(i)          (см. Гл. 10)
WriteInt(2*j + 1, 6)
INC(w[k].count)
```

9.3. Последовательность операторов

Последовательность операторов определяет последовательность действий, определенных составных операторов, которые разделены точками с запятой.

```
StatementSequence = statement {";" statement}.
```

9.4. Операторы If

```
IfStatement =    IF expression THEN StatementSequence
{ELSIF expression THEN StatementSequence}
[ELSE StatementSequence]
END.
```

If операторы определяют условное выполнение защищенных операторов. Логическое выражение, предшествующее последовательности операторов, называют его *охраной*. Охрана вычисляется последовательно, пока очередная не станет равна TRUE, после чего выполняется связанная с ней последовательность операторов. Если никакая охрана не удовлетворена, выполняется последовательность операторов после символа ELSE если тот есть.

Примеры:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF ch = 22X THEN ReadString
END
```

9.5. Операторы Case

Операторы case определяют выбор и выполнение последовательности операторов согласно значению выражения. Сначала вычисляется выражение, затем выполняется последовательность операторов, чей список меток выбора содержит полученное значение. Выражение должно иметь тип INTEGER и все метки

должны быть неотрицательными целыми числами.

CaseStatement = CASE expression OF case {"|" case} END.

case = [CaseLabelList ":" StatementSequence].

CaseLabelList = LabelRange {"," LabelRange}.

LabelRange = label [“..” label].

label = integer | ident.

Примеры:

```
CASE k OF
        0: x := x + y
        | 1: x := x - y
        | 2: x := x * y
        | 3: x := x / y
END
```

9.6 Операторы While

While операторы определяют повторение. Если какое-либо из логических выражений (охрана) даёт TRUE, то выполняется соответствующая последовательность операторов. Вычисление выражения и выполнение операторов повторяются, пока ни одно из логических выражений не будет давать TRUE.

WhileStatement = WHILE expression DO StatementSequence

{ELSIF expression DO StatementSequence} END.

Примеры:

```
WHILE j > 0 DO
        j := j DIV 2; i := i+1
END
WHILE (t # NIL) & (t.key # i) DO
        t := t.left
END
WHILE m > n DO m := m – n
ELSIF n > m DO n := n – m
END
```

9.7. Операторы Repeat

Оператор repeat определяет повторное выполнение последовательности операторов, пока не будет удовлетворено условие. Последовательность операторов выполняется, по крайней мере, единожды.

RepeatStatement = REPEAT StatementSequence UNTIL expression.

9.8. Операторы For

Оператор for определяет повторное выполнение последовательности операторов заданное количество раз, для прогрессии значений целочисленной переменной, называемой управляющей переменной оператора for.

ForStatement =

FOR ident “:=“ expression TO expression [BY ConstExpression] DO

StatementSequence END .

Оператор for

FOR v := beg TO end BY inc DO S END

при, *inc* > 0, эквивалентен

```
v := beg; lim := end;
WHILE v <= lim DO S; v := v + inc END
```

а при *inc* < 0 эквивалентен

v := beg; lim := end;

WHILE v >= lim DO S; v := v + inc END

Тип *v*, *beg* и *end* должен быть INTEGER, и *inc* должен быть целым числом (константное выражение). Если step не определен, то он предполагается равным 1.

10. Объявления процедур

Объявление процедуры состоит из заголовка процедуры и тела процедуры. Заголовок определяет идентификатор процедуры, формальные параметры, и тип результата (если он есть). Тело содержит объявления и операторы. Идентификатор процедуры повторяется в конце описания процедуры.

Есть два вида процедур, а именно, собственно процедуры и процедуры-функции. Последние активизируются обозначением функции как часть выражения и возвращают результат, который является операндом выражения. Собственно процедуры активизируются вызовом процедуры. Процедуру-функцию отличает наличие в объявлении после списка параметров, типа её результата. Её тело должно завершаться символом RETURN и выражением, которое определяет результат процедуры-функции.

Все константы, переменные, типы, и процедуры, объявленные в пределах тела процедуры, локальны для процедуры. Значения локальных переменных неопределенны после входа в процедуру. Так как процедуры могут быть тоже объявлены как локальные объекты, описания процедур могут быть вложенными.

В дополнение к их формальным параметрам и локально объявленным объектам, объекты, объявленные в окружении процедуры, также видимы в процедуре (за исключением переменных и тех объектов, у которых такое же имя, как и у объекта объявленного локально).

Использование идентификатора процедуры в вызове в пределах его объявления подразумевает рекурсивную активацию процедуры.

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.

ProcedureHeading = PROCEDURE identdef [FormalParameters.

ProcedureBody = DeclarationSequence [BEGIN StatementSequence]

[RETURN expression] END.

DeclarationSequence = [CONST {ConstantDeclaration ";"}]

[TYPE {TypeDeclaration ";"}] [VAR {VariableDeclaration ";"}]

{ProcedureDeclaration ";"}.

10.1. Формальные параметры

Формальные параметры это идентификаторы, которые обозначают фактические параметры, указанные в вызове процедуры. Соответствие между формальными и фактическими параметрами устанавливается, при вызове процедуры. Есть три вида параметров, а именно, *параметр-значение*, *параметр-константа*, и *параметр-переменная*. Параметр-переменная соответствует фактическому параметру, который является переменной, и он обозначает эту переменную. Параметр-константа соответствует фактическому параметру, который является выражением, и подставляет его значение, которое не может быть изменено присваиванием. Параметр-значение представляет локальную переменную, которой присвоено значение фактического выражения. Вид параметра обозначен в списке формальных параметров: Параметр-переменная обозначен символом VAR, Параметр-константа символом CONST, и параметр передаваемый по значению отсутствием префикса.

Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызваться обозначением функции, фактический список параметров которой также пуст.

Формальные параметры локальны для процедуры, то есть их область видимости - текст программы, который описывает процедуру.

FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].

FPSection = [CONST | VAR] ident {"," ident} ":" FormalType.

FormalType = {ARRAY OF} qualident.

Тип каждого формального параметра определён в списке параметров. Для параметров-переменных он должен быть идентичен типу соответствующего фактического параметра, кроме случая записи, где он должен быть основной типом соответствующего фактического параметра. Параметры, передаваемые по значению не могут быть массивом или записью.

Если тип формального параметра определен как

```
ARRAY OF T
```

тогда параметр называется *открытым массивом*, и соответствующий фактический параметр может иметь произвольную длину.

Если формальный параметр определяет тип процедуры, то соответствующий фактический параметр должен быть или процедурой, объявленной глобально, или переменной (или параметр) того типа процедуры. Это не может быть предопределённая процедура. Тип результата процедуры не может быть ни записью, ни массивом.

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 < 10^5 *)
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 log2(x: INTEGER): INTEGER;
VAR y: INTEGER; (*assume x>0*)
BEGIN y := 0;
        WHILE x > 1 DO x := x DIV 2; INC(y) END ;
        RETURN y
END log2
```

10.2. Предопределённые процедуры

В следующей таблице приведены предопределённые процедуры. Некоторые из них обобщённые процедуры, то есть они применяются к нескольким типам операндов. Буква v означает переменную, x и n - выражения, T - тип.

Процедуры-функции:

Имя	Тип аргумента	Тип результата	Действие
ABS(x)	INTEGER или REAL	тип x	абсолютное значение
	SET	INTEGER	количество элементов
ODD(x)	INTEGER	BOOLEAN	x MOD 2 = 1
LEN(v)	v: ARRAY		длина v
LSL(x, n)	x: INTEGER	тип x	логический сдвиг влево, $x * 2^n$
LSR(x, n)	x: INTEGER	тип x	беззнаковый сдвиг вправо, $x * 2^{-n}$
ASR(x, n)	x: INTEGER	тип x	сдвиг вправо со знаком, $x * 2^{-n}$

Процедуры преобразования типа:

Имя	Тип аргумента	Тип результата	Действие
FLOOR(x)	REAL	INTEGER	x с удаленной дробной частью
FLT(x)	INTEGER	REAL	просто преобразование типа
ORD(x)	CHAR, BOOLEAN	INTEGER	порядковый номер x
CHR(x)	INTEGER	CHAR	литера с порядковым номером x

Собственно процедуры:

Имя	Тип аргумента	Действие
INC(v)	INTEGER	v := v + 1
INC(v, n)	INTEGER	v := v + n
DEC(v)	INTEGER	v := v - 1
DEC(v, n)	INTEGER	v := v - n
NEW(v)	основной тип	разместить v^
ASSERT(b)	BOOLEAN	остановить программу, если ~b
ASSERT(b, n)	BOOLEAN, INTEGER	
PACK(x, y)	REAL, INTEGER	упаковать x и y в x
UNPK(x, y)	REAL, INTEGER	распаковать x в x и y

Процедуры INC и DEC могут иметь явный инкремент или декремент. Он должен быть константой. Второй параметр n у ASSERT это значение, передаваемое системе как параметр аварийного прекращения работы.

Параметр y PACK представляет экспоненту x. PACK(x, y) это эквивалентно x := x * 2y. UNPK - операция обратная PACK. Результат x нормализован, то есть 1.0 <= x < 2.0.

11. Модули

Модуль - это набор объявлений констант, типов, переменных, процедур, и последовательности операторов с целью присвоения переменным начальных значений. Модуль обычно представляет собой текст, который компилируется как единое целое.

```
module = MODULE ident ";" [ImportList ";"] DeclarationSequence
        [BEGIN StatementSequence] END ident ".".
ImportList = IMPORT import {"," import} ";".
Import = ident [":=" ident].
```

Список импорта определяет имена импортируемых модулей. Если идентификатор x экспортируется из модуля M, и если M перечислен в списке импорта модуля, то x упоминается как M.x. Если в списке импорта используется форма "M1 := M", то экспортируемый объект x, объявленный в модуле M, упоминается как M1.x.

Идентификаторы, которые должны быть видимыми в клиентских модулях, то есть которые должны быть экспортированы, должны быть отмечены звездочкой (метка экспорта) при объявлении. Переменные не могут быть экспортированы, за исключением скалярных типов в режиме только для чтения.

Последовательность операторов после символа BEGIN, выполняется, когда модуль добавляется к системе (загружается). Отдельные процедуры (без параметров) могут быть активированы из системы и эти процедуры служат командами.

Пример:

```
MODULE Out; (*exported procedures: Write, WriteInt, WriteLn*)
IMPORT Texts, Oberon;
VAR W: Texts.Writer;
PROCEDURE Write*(ch: CHAR);
BEGIN
        Texts.Write(W, ch)
END ;
```

```
PROCEDURE WriteInt*(x, n: LONGINT);
VAR i: INTEGER; a: ARRAY 16 OF CHAR;
BEGIN i := 0;
        IF x < 0 THEN Texts.Write(W, "-"); x := -x END ;
        REPEAT a[i] := CHR(x MOD 10 + ORD("0")); x := x DIV 10; INC(i) UNTIL x = 0;
        REPEAT Texts.Write(W, " "); DEC(n) UNTIL n <= i;
        REPEAT DEC(i); Texts.Write(W, a[i]) UNTIL i = 0
END WriteInt;
PROCEDURE WriteLn*;
BEGIN
        Texts.WriteLn(W);
        Texts.Append(Oberon.Log, W.buf)
END WriteLn;
BEGIN
        Texts.OpenWriter(W)
END Out.
```

11.1 Модуль SYSTEM

Необязательный модуль SYSTEM содержит определения, которые необходимы для низкоуровневых операций программы, обращающихся непосредственно к ресурсам, данного компьютера и/или реализации. Они включают например средства для того, чтобы получить доступ к устройствам, которыми управляет компьютер, и возможно средства, чтобы нарушить правила совместимости типов данных, наложенных определением языка. Строго рекомендуется ограничить их использование определенными низкоуровневыми модулями, такие модули, по сути, непереносимы и не "безопасны с точки зрения типов". Однако, они легко распознаются по идентификатору SYSTEM, находящемся в их списке импорта. Обычно используются следующие определения. Однако, отдельные реализации могут включать в модуль SYSTEM дополнительные определения, которые специфичны для определенного, произвольного компьютера. Ниже *v* означает переменную, *x*, *a*, и *n* выражения.

Процедуры-функции:

Имя	Тип аргумента	Тип результата	Действие
ADR(v)	любой	INTEGER	адрес переменной v
SIZE(T)	любой тип	INTEGER	размер в байтах
BIT(a, n)	a, n: INTEGER	BOOLEAN	бит n у mem[a]

Собственно процедуры:

Имя	Тип аргумента	Действие
GET(a, v)	a: INTEGER; v: любой основной тип	v := mem[a]
PUT(a, x)	a: INTEGER; x: любой основной тип	mem[a] := x

Приложение

Синтаксис Оберона

```
letter = "A" | "B" | … | "Z" | "a" | "b" … | "z".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
hexdigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
ident = letter {letter | digit}.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | “D”) ["+" | "-"] digit {digit}.
number = integer | real.
```

```
charConst = "'" character "'" | digit {hexDigit} "X".
string = “"”{character} “"” | "'" {character} "'".
qualident = [ident "."] ident.
identdef = ident ["*"].
ConstantDeclaration = identdef "=" ConstExpression.
ConstExpression = expression.
TypeDeclaration = identdef "=" StrucType.
StrucType = ArrayType | RecordType | PointerType | ProcedureType.
type = qualident | StrucType.
ArrayType = "ARRAY" length {"," length} "OF" type.
length = ConstExpression.
RecordType = "RECORD" ["(" BaseType ")"] [FieldListSequence] "END".
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList = IdentList ":" type.
IdentList = identdef {"," identdef}.
PointerType = "POINTER" "TO" type.
ProcedureType = "PROCEDURE" [FormalParameters].
VariableDeclaration = IdentList ":" type.
designator = qualident {selector}.
selector = "." ident | "[" ExpList "]" | "^" | "(" qualident ")".
ExpList = expression {"," expression}.
factor = number | CharConst | string | "NIL" | "TRUE" | "FALSE" |
        set | designator [ActualParameters] | "(" expression ")" | "~" factor.
ActualParameters = "(" [ExpList] ")" .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | "DIV" | "MOD" | "&".
SimpleExpression = ["+" | "-"] term {AddOperator term}.
AddOperator = "+" | "-" | "OR".
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS".
set = "{" [element {"," element}] "}".
element = expression [".." expression].
statement = [assignment | ProcedureCall | IfStatement | CaseStatement |
        WhileStatement | RepeatStatement | ForStatement].
assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {";" statement}.
IfStatement = "IF" expression "THEN" StatementSequence
        {"ELSIF" expression "THEN" StatementSequence}
        ["ELSE" StatementSequence] "END".
CaseStatement = "CASE" expression "OF" case {"|" case} "END".
Case = CaseLabelList ":" StatementSequence.
```

```
CaseLabelList = LabelRange {"," LabelRange}.
LabelRange = label [“..” label].
label = integer | ident.
WhileStatement = "WHILE" expression "DO" StatementSequence
{“ELSIF” expression "DO" StatementSequence} "END".
RepeatStatement = "REPEAT" StatementSequence "UNTIL" expression.
ForStatement = "FOR" ident ":=" expression "TO" expression ["BY" ConstExpression] "DO"
        StatementSequence "END".
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = "PROCEDURE" identdef [FormalParameters].
ProcedureBody = DeclarationSequence ["BEGIN" StatementSequence] ["RETURN" expression] "END".
DeclarationSequence = ["CONST" {ConstDeclaration ";"}]
        ["TYPE" {TypeDeclaration ";"}]
        ["VAR" {VariableDeclaration ";"}]
        {ProcedureDeclaration ";"}.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection = ["CONST" | "VAR"] ident {"," ident} ":" FormalType.
FormalType = ["ARRAY" "OF"] qualident.
module = "MODULE" ident ";" [ImportList] DeclarationSequence
        ["BEGIN" StatementSequence] "END" ident "." .
ImportList = "IMPORT" import {"," import} “;”.
import = ident [":=" ident].
```