Язык программирования O2M

Легалов А.И., Швец Д.А.

Февраль 2004

Введение

O2M - язык программирования, расширяющий своего предшественника (Оберон-2) [1] механизмами инструментальной поддержки процедурно-параметрического программирования (ППП) [2, 3]. Основная цель разработки – экспериментальная проверка методов параметризации и их использование при создании эволюционно расширяемых программ. Сделана попытка сохранить преемственность, которая может быть нарушена совпадением новых ключевых слов с идентификаторами ранее написанных программ.

*Примечание. Первоначально рабочим названием языка было «Оберон-2М». Его сокращение до «O2M» вызвано стремлением не нарушать торговые марки и прочие бантики. Помимо этого наличие буквы «М» после цифры «2» смотрелось амбициозно, хотя с самого начала она не означала «модификацию» языка Оберон-2. Это – римская цифра, переводимая на «арабский» как 1000. 2М = 2*1000 = 2000 – год начала работ над процедурно-параметрическими проектами.*

При формировании данного документа многое взято из описания языка программирования Оберон-2 в переводе С.З. Свердлова [4]. Заимствованы из него и переводы основных терминов англоязычного оригинала.

В O2M сохраняется поддержка объектно-ориентированного стиля предшественника: объект состоит из переменных абстрактного типа, содержащих данные, и связанных с ним процедур. Абстрактные типы данных заданы как расширяемые записи. Наряду с этим в O2M реализованы дополнительные абстракции: обобщения и обобщающие процедуры, которые поддерживают процедурно-параметрическую парадигму. Этот стиль позволяет непосредственно использовать множественный полиморфизм, частным случаем которого является объектно-ориентированный полиморфизм [5, 6].

Как и первоисточник, этот документ не является учебником по программированию. Он преднамеренно краток. Если о чем-то не сказано, то обычно сознательно: или потому, что это следует из других правил языка, или потому, что потребовалось бы определять то, что фиксировать для общего случая представляется неразумным. Предполагается, что более подробные объяснения и толкования будут сделаны в отдельных публикациях.

В приложении А определены некоторые термины, которые используются при описании правил соответствия типов O2M. В тексте эти термины выделены курсивом, чтобы подчеркнуть их специальное значение (например, *одинаковый* тип). В приложении В приведен синтаксис языка O2M.

2. Синтаксис

Для описания синтаксиса O2M 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). 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.

1. *Идентификаторы* - последовательности букв и цифр. Первый символ должен быть буквой.

идент = буква {буква | цифра}.

Примеры:

```
x  Scan  Oberon2M  GetSymbol  firstLetter
```

2. *Числа* - целые или вещественные (без знака) константы. Типом целочисленной константы считается минимальный тип, которому принадлежит ее значение (см. 6.1). Если константа заканчивается буквой H, она является шестнадцатеричной, иначе - десятичной.

Вещественное число всегда содержит десятичную точку. Оно может также содержать десятичный порядок. Буква E (или D) означает "умножить на десять в степени". Вещественное число относится к типу REAL кроме случая, когда у него есть порядок, содержащий букву D. В этом случае оно относится к типу LONGREAL.

число = целое | вещественное.
целое = цифра {цифра} | цифра {шестнЦифра} "H".
вещественное = цифра {цифра} "." {цифра} [Порядок].
порядок = ("E" | "D") ["+" | "-"] 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
0DH              SHORTINT   13
12.3             REAL       12.3
4.567E8          REAL       456700000
0.57712566D-6    LONGREAL   0.00000057712566
```

3. *Символьные* константы обозначаются порядковым номером символа в шестнадцатеричной записи, оканчивающейся буквой X.

символ = цифра {шестнЦифра} "X".

4. *Строки* - последовательности символов, заключенные в одиночные (') или двойные (") кавычки. Открывающая кавычка должна быть такой же, что и закрывающая и не должна встречаться внутри строки. Число символов в строке называется ее *длиной*. Строка длины 1 может использоваться везде, где допустима символьная константа и наоборот.

строка = ' " ' {символ} ' " ' | " ' " {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	
(	)	EXIT	POINTER	
[	]	FOR	PROCEDURE	
{	}	IF	RECORD	

6. *Комментарии* могут быть вставлены между любыми двумя словами программы. Это произвольные последовательности символов, начинающиеся скобкой (* и оканчивающиеся *). Комментарии могут быть вложенными. Они не влияют на смысл программы.

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);
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), или процедуры, связанные с типом, (см. 10.2) 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)	LEN	(10.5)
ASH	(10.5)	LONG	(10.5)
BOOLEAN	(6.1)	LONGINT	(6.1)
CAP	(10.5)	LONGREAL	(6.1)
CHAR	(6.1)	MAX	(10.5)
CHR	(10.5)	MIN	(10.5)
COPY	(10.5)	NEW	(10.5)
DEC	(10.5)	ODD	(10.5)
ENTIER	(10.5)	ORD	(10.5)
EXCL	(10.5)	REAL	(6.1)
FALSE	(6.1)	SET	(6.1)
HALT	(10.5)	SHORT	(10.5)
INC	(10.5)	SHORTINT	(6.1)
INCL	(10.5)	SIZE	(10.5)
INTEGER	(6.1)	TRUE	(6.1)

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
fullSet = {MIN(SET) .. MAX(SET)}
```

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 и записи) объявление также определяет структуру переменных этого типа. Структурированный тип не может содержать сам себя.

ОбъявлениеТипа =ИдентОпр "=" Тип.
Тип =УточнИдент | ТипМассив | ТипЗапись | ТипОбобщение | ТипУказатель | ПроцедурныйТип.

Примеры:

```
Table = ARRAY N OF REAL

Tree = POINTER TO Node
Node = RECORD
     key : INTEGER;
```

```
     left, right: Tree
END

CenterTree = POINTER TO CenterNode

CenterNode = RECORD (Node)
     width: INTEGER;
     subnode: Tree
END

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. **SHORTINT** **целые в интервале от MIN(SHORTINT) до MAX(SHORTINT)**
4. **INTEGER** **целые в интервале от MIN(INTEGER) до MAX(INTEGER)**
5. **LONGINT** **целые в интервале от MIN(LONGINT) до MAX(LONGINT)**
6. **REAL** **вещественные числа в интервале от MIN(REAL) до MAX(REAL)**
7. **LONGREAL** **вещественные числа от MIN(LONGREAL) до MAX(LONGREAL)**
8. **SET** **множество из целых от 0 до MAX(SET)**

Типы от 3 до 5 - *целые типы*, типы 6 и 7 - *вещественные типы*, а вместе они называются *числовыми типами*. Эти типы образуют иерархию; больший тип *поглощает* меньший тип:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

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 запись

Тип запись - структура, состоящая из фиксированного числа элементов, которые могут быть различных типов и называются *полями*. Объявление типа запись определяет имя и тип каждого поля. 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 ["(" БазовыйТип ")"] СписокПолей {";" СписокПолей} END.
БазовыйТип =УточнИдент.
СписокПолей =[СписокИдент ":" Тип].

Тип запись может быть объявлен как расширение другого типа запись. В примере

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 - (непосредственное) *расширение* *T0*, а *T0* - (непосредственный) *базовый тип* *T1* (см. Прил. А). Расширенный тип *T1* состоит из полей своего базового типа и полей, которые объявлены в *T1*. Все идентификаторы, объявленные в расширенной записи, должны быть отличны от идентификаторов, объявленных в записи (записях) ее базового типа.

Примеры объявлений типа запись:

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

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

6.4 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 ":" (УточнИдент | NIL))

| УточнИдент) { "|" ((СписокПризнаков ":" (УточнИдент | NIL))

| Тип 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 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, что позволяет эволюционно наращивать число альтернатив 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:

```
Figures.Shape2 += c: Circle
```

Здесь Figures – имя модуля, содержащего первоначальное определение обобщения. Допускается многократное расширение уже расширенных обобщений. Возможно независимое расширение обобщений в разных модулях с последующей сборкой воедино на более поздних этапах.

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*.

ТипУказатель = POINTER TO Тип.

Если *p* - переменная типа *P* = POINTER TO *T*, и тип *T* не является обобщением, вызов стандартной процедуры *NEW(p)* (см. 10.5) размещает переменную типа *T* в свободной памяти. Если *T* - тип 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 динамическим тип такой переменной не является *T*. Он может принимать значения только типов-специализаций, определяемых значениями признаков. Указатель на специализацию обобщения не допускается. Размещение обобщенной переменной в свободной памяти осуществляется с помощью процедуры *NEW(p<t>)*, в которой признак *t* определяет создаваемую специализацию.

Примечание. Возможно, что дальнейшее развитие языка приведет к появлению указателей на специализации. На это «намекают» операция IS и оператор WITH.

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.

ПроцедурныйТип = PROCEDURE [ФормальныеПараметры].

7. Объявления переменных

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

ОбъявлениеПеременных = СписокИдент ":" Тип | ОбъявлОбобщПерем.

ОбъявлОбобщПерем = СписокИдент ":" УточнИдент "<" [Признак] ">"

| идент"<" [Признак] ">" {“,” идент"<" [Признак] ">" } ":" УточнИдент.

Признак = УточнИдент .

Переменные типа запись и указатель имеют как статический тип (тип, с которым они объявлены - называемый просто их типом), так и динамический тип (тип их значения во время выполнения). Для указателей и параметров-переменных типа запись динамический тип может быть расширением их статического типа. Для указателей и параметров-переменных типа

обобщение динамический тип может быть специализацией их статического типа. Статический тип определяет, какие поля записи доступны. Динамический тип используется для вызова связанных с типом процедур (см. 10.2) или для вызова обобщающих параметрических процедур (см. 10.3). При объявлении обобщенных переменных после указания типа обобщения указывается признак, позволяющий определить 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 угловых скобок.

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 конструкторов множества и 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* или процедуру *f*, связанную с динамическим типом *r* (Гл. 10.2). 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 динамическим типом *v* был *T* (расширение *T*, или специализация *T*), то есть выполнение программы прерывается, если динамический тип *v* - не *T* (расширение *T*, или специализация *T*). В пределах такого обозначения *v* воспринимается как имеющая статический тип *T*. Охрана применима, если

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

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(CenterTree).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] ")".**
Отношение **="=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.**
ОперацияСложения **="+" | "-" | OR.**
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"
&	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.

8.2.2 Арифметические операции

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

Операции +, -, *, и / применимы к операндам числовых типов. Тип их результата - тип того операнда, который поглощает тип другого операнда, кроме деления (/), чей результат - наименьший вещественный тип, который поглощает типы обоих операндов. При использовании в качестве одноместной операции "-" обозначает перемену знака, а "+" - тождественную операцию. Операции DIV и MOD применимы только к целочисленным 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 Операции над множествами

+	объединение
-	разность (x - y = x * (-y))
*	пересечение
/	симметрическая разность множеств (x / y = (x-y) + (y-x))

Эти операции применимы к операндам типа SET и дают результат типа SET. Одноместный "минус" обозначает дополнение *x*, то есть *-x* это множество целых между 0 и MAX(SET), которые не являются элементами *x*. Операции с множествами не ассоциативны *((a+b)-c # a+(b-c))*. Конструктор множества задает значение множества списком элементов, заключенным в фигурные скобки. Элементы должны быть целыми в диапазоне 0..MAX(SET). Диапазон *a..b* обозначает все целые числа в интервале [a, b].

8.2.4 Отношения

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

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 типам BOOLEAN и SET, а также к указателям и процедурным типам (включая значение NIL). *x IN s* означает "*x* является элементом *s*". *x* должен быть целого типа, а *s* - типа SET. *v IS T* означает "динамический тип *v* есть *T* (или расширение *T*, или специализация *T*)" и называется *проверкой типа*. Проверка типа применима, если

1. *v* - параметр-переменная типа запись, или *v* - указатель, или *v* – параметр-переменная типа обобщение и если
2. *T* - расширение статического типа *v*, или
3. *T* – специализация обобщения типа *v*.

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

1991	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
w[i].name <= "John"	BOOLEAN
t IS CenterTree	BOOLEAN
sh11 IS Shape1<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
| EXIT | RETURN [Выражение]].

9.1 Присваивание

Присваивание заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть совместимо по присваиванию с переменной (см. Приложение. А). 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. Допускается два способа оформления вызова обобщенной процедуры:

1. В префиксном стиле, при котором список обобщенных параметров указывается перед обозначением процедуры и отделяется от него точкой
2. В инфиксном стиле, когда список обобщенных параметров указывается между обозначением процедуры и списком формальных параметров

Не допускается одновременное использование префиксной и инфиксной форм в одном вызове обобщенной процедуры.

ВызовПроцедуры = [ОбобщенныеПараметры "."] Обозначение
[ФактическиеПараметры] | Обозначение [ОбобщенныеПараметры]
[ФактическиеПараметры].

Примеры:

```
WriteInt(i*2+1)   (* см. 10.1 *)
INC(w[k].count)
t.Insert("John")  (* см. 11 *)
{sh11}.Perimeter;
Perimeter{sh12};
```

9.3 Последовательность операторов

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

ПоследовательностьОператоров = Оператор {";" Оператор}.

9.4 Операторы 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, будем называть 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 Операторы 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 Операторы 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 Операторы 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 Операторы 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 Операторы 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.

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 ":" УточнИдент.

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: если динамический тип *v* - *T1*, то выполняется последовательность операторов *S1* в которой *v* воспринимается так, будто она имеет статический тип *T1*; иначе, если динамический тип *v* - *T2*, выполняется *S2*, где *v* воспринимается как имеющая статический тип *T2*; иначе выполняется *S3*. *T1* и *T2* должны быть расширениями *T0*. Если ни одна проверка типа не удовлетворена, а ELSE отсутствует, программа прерывается.

Пример:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

Если *v* - параметр-переменная типа обобщение или переменная-указатель на обобщение, и если ее обобщающий тип тип *T0*, то этот же оператор имеет следующий смысл: если тип специализации *v* - *T1*, то 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 обобщающего типа, уточненного признаком специализации.

Пример:

```
WITH psh: PShape1<Rectangle> DO psh.x := 30; psh.y := 10
|    psh: PShape1<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 связанных с типом процедур в объявлении определяется параметр-приемник. 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.

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

Если объявление процедуры содержит параметр-приемник, процедура рассматривается как связанная с типом (см. 10.2). 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 *Tf* - тип формального параметра *f* (не открытого массива) и *Ta* - тип соответствующего фактического параметра *a*. Для параметров-переменных *Ta* и *Tf* должны быть одинаковыми типами или *Tf* должен быть типом запись, а *Ta* - расширением *Tf*, или *Tf* должен быть обобщением, а *Ta* - специализацией. Для параметров-значений *a* должен быть *совместим по присваиванию* с *f* (см. Прил. А).

Если *Tf* - открытый массив, то а должен быть *совместимым массивом* для *f* (см. Прил. А). 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 Процедуры, связанные с типом

Глобально объявленные процедуры могут быть ассоциированы с типом запись, объявленным в том же самом модуле. В этом случае говорится, что процедуры *связаны* с типом запись. Связь выражается типом *приемника* в заголовке объявления процедуры. Приемник может быть или параметром-переменной типа *T*, если *T* - тип запись, или параметром-значением типа POINTER TO *T* (где *T* - тип запись). Процедура, связанная с типом *T*, рассматривается как локальная для него.

ЗаголовокПроцедуры =PROCEDURE [Приемник] ИдентОпр [ФормальныеПараметры].
Приемник ="(" [VAR] имя ":" имя ")".

Если процедура *P* связана с типом *T0*, она неявно также связана с любым типом *T1*, который является расширением *T0*. Однако процедура *P'* (с тем же самым именем что и *P*) может быть явно связана с *T1*, перекрывая в этом случае связывание с *P*. *P'* рассматривается как

переопределение P для *T1*. Формальные параметры *P* и *P'* должны *совпадать* (см. Прил. А). Если *P* и *T1* экспортируются (см. Главу 4), *P'* также должна экспортироваться.

Если *v* - обозначение, а *P* - связанная процедура, то *v.P* обозначает процедуру *P*, связанную с динамическим типом *v*. Заметим, что это может быть процедура, отличная от той, что связана со статическим типом *v*. *v* передается приемнику процедуры *P* согласно правилам передачи параметров, определенным в Главе 10.1.

Если *r* - параметр-приемник, объявленный с типом *T, r.P^* обозначает (переопределенную) процедуру *P*, связанную с базовым для *T* типом. В опережающем объявлении связанной процедуры и в фактическом объявлении процедуры параметр-приемник должен иметь *одинаковый* тип. Списки формальных параметров в обоих объявлениях должны быть идентичны.

Примеры:

```
PROCEDURE (t: Tree) Insert (node: Tree);
      VAR p, father: Tree;
BEGIN p := t;
      REPEAT father := p;
            IF node.key = p.key THEN RETURN END;
            IF node.key < p.key THEN p := p.left ELSE p := p.right END
      UNTIL p = NIL;
      IF node.key < father.key THEN father.left := node
      ELSE father.right := node END;
      node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree); (*переопределение*)
BEGIN
      WriteInt(node(CenterTree).width);
      t.Insert^ (node) (* вызывает процедуру Insert, связанную с Tree *)
END Insert;
```

10.3 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 ОбобщенныеПараметры [ФормальныеПараметры] (":=" "0" | ";" ПоследовательностьОбъявлений [BEGIN ПоследовательностьОператоров] 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
```

10.4 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.

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;
```

10.5 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: целый тип	LONGINT	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 константа	прерывает выполнение 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	размещает v ^

	фиксированной длины	
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 ";" [СписокИмпорта] ПоследовательностьОбъявлений [BEGIN ПоследовательностьОператоров] END идент ".".
СписокИмпорта =IMPORT Импорт {"," Импорт} ";".
Импорт =[идент ":="] идент.

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*.

*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 SHORTINT, INTEGER, LONGINT
Вещественные типы REAL, LONGREAL
Числовые типы Целые типы, вещественные типы

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

Числовые типы *поглощают* (значения) меньших числовых типов согласно следующей иерархии:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

Расширение типов (базовый тип)

В объявлении типа *Tb = RECORD (Ta) ... END, Tb - непосредственное расширение Ta*, а *Ta - непосредственный базовый тип Tb*. Тип *Tb* есть расширение типа *Ta* (*Ta* есть базовый тип *Tb*) если

1. *Ta* и *Tb* - *одинаковые* типы, или
2. *Tb - непосредственное расширение* типа, являющегося *расширением Ta*

Если *Pa = POINTER TO Ta* и *Pb = POINTER TO Tb*, то *Pb* есть *расширение Pa* (*Pa* есть *базовый тип Pb*), если *Tb* есть *расширение Ta*.

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* *Te*;
3. *Te* и *Tv* - типы запись, *Te* есть *расширение* *Tv*, а *v* имеет динамический тип *Tv*;
4. *Te* и *Tv* - типы указатель и *Te* - *расширение* *Tv*;
5. *Tv* - тип указатель или процедурный тип, а *e* - NIL;
6. *Tv* - ARRAY *n* OF CHAR, *e* - строковая константа из *m* символов и $m < n$;
7. *Tv* - процедурный тип, а *e* - имя процедуры, чьи формальные параметры *совпадают* с параметрами *Tv*;
8. *Tv* –обобщение, параметризированное некоторым признаком, а Te – тип, соответствующий заданному признаку;
9. *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*	наименьший *числовой* тип, поглощающий оба операнда
/	*числовой*	*числовой*	наименьший *вещественный* тип, поглощающий оба операнда
+ - * /	SET	SET	SET
DIV MOD	*целый*	*целый*	наименьший *целый* тип, поглощающий оба операнда
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	*числовой* CHAR символьный массив, строка	*числовой* CHAR символьный массив, строка	BOOLEAN BOOLEAN BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN

	SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	BOOLEAN BOOLEAN BOOLEAN
IN	*целый*	SET	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 B: Синтаксис O2M

```
Модуль                  =MODULE Идент ";" [СписокИмпорта] ПослОбъявлений [BEGIN
                          ПослОператоров] END идент ".".
СписокИмпорта           =IMPORT [Идент ":="] Идент {"," [Идент ":="] Идент} ";".
ПослОбъявлений          ={ CONST {ОбъявлКонстанты ";" }
                          | TYPE {ОбъявлТипа ";" }
                          | VAR {ОбъявлПерем ";" }} {ОбъявлПроцедуры ";"
                          | ОпережающееОбъявл";"}.
ОбъявлКонстанты         =ИдентОпр "=" КонстВыражение.
ОбъявлТипа              =ИдентОпр "=" Тип.
ОбъявлПерем             =(СписокИдент ":" Тип) | ОбъявлОбобщПерем.
ОбъявлениеПроцедуры     =ЗаголовокПроцедуры ";" ТелоПроцедуры Идент.
ЗаголовокПроцедуры      =PROCEDURE [Приемник] ИдентОпр [ФормальныеПараметры]
                          | ОбобщеннаяПроц
                          | СпециализирПроц.
ТелоПроцедуры           =ПоследовательностьОбъявлений BEGIN
                          [ПоследовательностьОператоров] END.
ОбобщеннаяПроц          =PROCEDURE ИдентОпр ОбобщенныеПарам [ФормальныеПарам]
                          (":=" "0" | ";" ПоследовательностьОбъявлений [BEGIN
                          ПоследовательностьОператоров] END Идент).
ОбработчикСпециализац   =PROCEDURE ИдентОпр СписокСпециализаций
                          [ФормальныеПараметры] ТелоПроцедуры Идент .

ОпережающееОбъяв        =PROCEDURE "^" [Приемник] ИдентОпр [ФормальныеПарам] |
                          PROCEDURE "^" ИдентОпр [ОбобщенныеПарам]
                          [ФормальныеПарам].
ФормальныеПарам         ="(" [СекцияФП {";" СекцияФП}] ")" [":" УточнИдент].
СекцияФП                =[VAR] идент {"," идент} ":" Тип.
Приемник                ="(" [VAR] идент ":" идент ")".
ОбобщенныеПарам         ="{" [ОбобщеннаяСекцияФП {";" ОбобщеннаяСекцияФП }] "}".
ОбобщеннаяСекцияФП      = [VAR] Идент {"," Идент } ":" УточнИдент.
ОбъявлОбобщПерем        =СписокИдент ":" УточнИдент "<" Признак ">".
ПроцОбобщПарам          ="{" [УточнИдент {"," УточнИдент }] "}".
СписокСпециализаций     ="{" ГруппаСпециализаций { ";" ГруппаСпециализаций } "}" .
ГруппаСпециализаций     =[VAR] Специализация { "," Специализация }.
```

```
Специализация    =Идент ":" ОбобщающийТип "<" [идент Признак] ">".
Тип              =УточнИдент | ARRAY [КонстВыраж {"," КонстВыраж}] OF Тип
                   | RECORD ["("УточнИдент")"] СписокПолей {";" СписокПолей}
                   END | POINTER TO Тип | PROCEDURE [ФормальныеПарам]
                   | ТипОбобщение.
ТипОбобщение     =CASE [TYPE] [LOCAL] OF [СписокСпециализаций] [ ELSE
                   Специализация ] END .
СписокСпециализаций =Специализация { "|" Специализация } .
Специализация    =(СписокПризнаков ":" (УточнИдент | NIL) ) | УточнИдент.
СписокПризнаков  =Идент [ "," Идент ] .
СписокПолей      =[СписокИдент ":" Тип].
ПослОператоров   =Оператор {";" Оператор}.
Оператор         =[ Обозначение ":=" Выраж
                   | Обозначение ["(" [СписокВыраж] ")"]
                   | IF Выраж THEN ПослОператоров {ELSIF Выраж 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 КонстВыраж] DO
                   ПослОператоров END
                   | LOOP ПослОператоров END
                   | WITH Охрана DO ПослОператоров {"|" Охрана DO
                   ПослОператоров}
                   [ELSE ПослОператоров] END
                   | EXIT
                   | RETURN [Выраж] ].
Вариант          =[МеткиВарианта {"," МеткиВарианта} ":" ПослОператоров].
МеткиВарианта    =КонстВыраж [".." КонстВыраж].
Охрана           =УточнИдент ":" УточнИдент.
КонстВыраж       =Выраж.
Выраж            =ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж     =["+" | "-"] Слагаемое {ОперСлож Слагаемое}.
Слагаемое        =Множитель {ОперУмн Множитель}.
Множитель        =[ПроцОбобщПарам "."] Обозначение [ПроцОбобщПарам] ["("
                   [СписокВыраж] ")"] | число | символ | строка | NIL | Множество
                   | "(" Выраж ")" | " ~ " Множитель.
Множество        ="{" [Элемент {"," Элемент}] "}".
Элемент          =Выраж [".." Выраж].
Отношение        ="=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
ОперСлож         ="+" | "-" | OR.
ОперУмн          ="*" | "/" | DIV | MOD | "&".
Обозначение      =УточнИдент {"." идент | "[" СписокВыраж "]" | "^" | "(" УточнИдент
                   ")"}.
СписокВыраж      =Выраж {"," Выраж}.
СписокИдент      =ИдентОпр {"," ИдентОпр}.
УточнИдент       =[идент "."] идент.
ИдентОпр         =идент [ "*" | "-" ]
```

Литература

1. The Programming Language Oberon-2 H.Moessenboeck, N.Wirth. Institut fur Computersysteme, ETH Zurich July 1996.

2. 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.

3. Легалов А.И. Процедурно-параметрическое программирование (материал размещен по адресу: http://www.softcraft.ru/paradigm/ppp/ppp01.shtml).

4. 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).

5. Легалов А.И. Мультиметоды и парадигмы. – Открытые системы, № 5 (май) 2002, с. 33-37.

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