This document is in Windows 1251 cyrillic coding

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

Язык программирования Оберон-2

Х.Мёссенбёк, Н.Вирт

Институт компьютерных систем, ETH, Цюрих

Июль 1996

Перевод с английского С.Свердлова

От переводчика

Язык программирования Оберон создан автором Паскаля и Модулы-2 Никлаусом Виртом в 1987 году в ходе разработки одноименной операционной системы для однопользовательской рабочей станции Ceres. Язык и операционная система названы именем одного из спутников планеты Уран - Оберона, открытого английским астрономом Уильямом Гершелем ровно за двести лет до описываемых событий.

"Сделай так просто, как возможно, но не проще того" - это высказывание А.Эйнштейна Вирт выбрал эпиграфом к описанию языка. Удивительно простой и даже аскетичный Оберон является, вероятно, минимальным универсальным языком высокого уровня. Он проще Паскаля и Модулы-2 и в то же время обогащает их рядом новых возможностей. Важно то, что автором языка руководили не сиюминутные коммерческие и конъюнктурные соображения, а глубокий анализ реальных программистских потребностей и стремление удовлетворить их простым, понятным, эффективным и безопасным образом, не вводя по возможности новых понятий.

Являясь объектно-ориентированным языком, Оберон даже не содержит слова object. Оберон представляется идеальным языком для изучения программирования. Сочетание простоты, строгости и неизбыточности предоставляет начинающему программисту великолепную возможность, не заблудившись в дебрях, выработать хороший стиль, освоив при этом и структурное и объектно-ориентированное и модульно-компонентное программирование.

В 1992 году сотрудничество Н.Вирта с Ханспетером Мёссенбёком привело к добавлению в язык ряда новых средств. Новая версия получила название Оберон-2. Описание именно этого языка по состоянию на 1 октября 1996 года (последние изменения внесены авторами в июле 1996 года) и дается в настоящем переводе. Оберон-2 представляет собой почти правильное расширение Оберона и является фактическим стандартом языка,

который поддерживается большинством современных Оберон-систем. В Оберон-2 добавлены:

- связанные с типом процедуры;
- экспорт только для чтения;
- открытые массивы в качестве базового типа для указателей;
- оператор with с вариантами;
- оператор for.

Отдельного внимания заслуживает само описание, с которым вам предстоит познакомиться. Вирт и его соавтор достигли совершенства не только в искусстве разработки, но, несомненно, и в деле описания языков программирования. Поражают изумительная точность и краткость этого документа. Почти каждая его фраза превращается при написании компилятора в конкретные строки программного кода.

Возникшие при переводе описания Оберона-2 на русский язык терминологические вопросы решались исходя из следующих соображений: предпочтительным является буквальный перевод; недопустимо добавление терминов, отсутствующих в оригинале; должны быть учтены отечественные традиции в терминологии алголоподобных языков; предпочтительно использование терминов, привычных широкому кругу программистов, вместо узкоспециальных. Ниже приведен список терминов, перевод которых не представляется очевидным.

(direct) base type	(непосредственный) базовый тип
array compatible	совместимый массив
array type	тип массив
assignment compatible	совместимый по присваиванию
basic type	основной тип
browser	смотритель
case statement	оператор case
character	символ, знак
declaration	объявление
designator	обозначение
direct extension	непосредственное расширение
equal types	равные типы
exit statement	оператор выхода
expression compatible	совместимое выражение
for statement	оператор for
function procedure	процедура-функция
if statement	оператор if
loop statement	оператор loop
matching	совпадение
operator	операция
pointer type	тип указатель
predeclared	стандартный

private field	скрытое поле
proper procedure	собственно процедура
public field	доступное поле
qualified	уточненный
real	вещественный
record type	тип запись
repeat statement	оператор repeat
return statement	оператор возврата
same type	одинаковый тип
scale factor	порядок
scope	область действия
statement	оператор
string	строка
symbol	слово
type extension	расширение типа
type guard	охрана типа
type inclusion	поглощение типа
type tag	тег
type test	проверка типа
type-bound procedures	связанные с типом процедуры
while statement	оператор while
with statement	оператор with

[С. Свердлов](mailto:c3c@uni-vologda.ac.ru)
c3c@uni-vologda.ac.ru
2 октября 1996 г. - 12 июня 1998 г.
Вологда

Введение

Оберон-2 - язык программирования общего назначения, продолжающий традицию языков Паскаль и Modula-2. Его основные черты - блочная структура, модульность, раздельная компиляция, статическая типизация со строгим контролем соответствия типов (в том числе межмодульным), а также расширение типов и связанные с типами процедуры.

Расширение типов делает Оберон-2 объектно-ориентированным языком. Объект - это переменная абстрактного типа, содержащая данные (состояние объекта) и процедуры, которые оперируют этими данными. Абстрактные типы данных определены как расширяемые записи. Оберон-2 перекрывает большинство терминов объектно-ориентированных языков привычным словарем языков императивных, обходясь минимумом понятий в рамках тех же концепций.

Этот документ не является учебником программирования. Он преднамеренно краток. Его назначение - служить справочником для программистов, разработчиков компиляторов и авторов руководств. 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.

В приложении А определены некоторые термины, которые используются при описании правил соответствия типов Оберона-2. В тексте эти термины выделены курсивом, чтобы подчеркнуть их специальное значение (например, *одинаковый* тип).

2. Синтаксис

Для описания синтаксиса Оберона-2 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). 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 Oberon2 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	RETURN
-	^	BEGIN	IN	THEN
*	=	BY	IS	TO
/	#	CASE	LOOP	TYPE
~	<	CONST	MOD	UNTIL
&	>	DIV	MODULE	VAR
.	<=	DO	NIL	WHILE
,	>=	ELSE	OF	WITH
;	..	ELSIF	OR	
|	:	END	POINTER	
(	)	EXIT	PROCEDURE	
[	]	FOR	RECORD	
{	}	IF	REPEAT	

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. 6.4) 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.

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	(10.3)	LEN	(10.3)
ASH	(10.3)	LONG	(10.3)
BOOLEAN	(6.1)	LONGINT	(6.1)

CAP	(10.3)	LONGREAL	(6.1)
CHAR	(6.1)	MAX	(10.3)
CHR	(10.3)	MIN	(10.3)
COPY	(10.3)	NEW	(10.3)
DEC	(10.3)	ODD	(10.3)
ENTIER	(10.3)	ORD	(10.3)
EXCL	(10.3)	REAL	(6.1)
FALSE	(6.1)	SET	(6.1)
HALT	(10.3)	SHORT	(10.3)
INC	(10.3)	SHORTINT	(6.1)
INCL	(10.3)	SIZE	(10.3)
INTEGER	(6.1)	TRUE	(6.1)

5. Объявления констант

Объявление константы связывает ее идентификатор с ее 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. 10.3), 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 | 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
Function = PROCEDURE(x: INTEGER): INTEGER
```

6.1 Основные типы

Основные типы обозначаются стандартными идентификаторами. Соответствующие операции определены в 8.2, а стандартные функции в 10.3. 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. 6.4), 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, которые должны быть видимы 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. A). 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 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 или запись. Типы указатель заимствуют 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*, вызов стандартной процедуры *NEW(p)* (см. 10.3) 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*.

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

6.5 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. A). *P* не должна быть стандартной или связанной с типом процедурой, и не может быть локальной в другой процедуре.

```
ПроцедурныйТип = PROCEDURE [ФормальныеПараметры].
```

7. Объявления переменных

Объявления переменных дают описание переменных, определяя идентификатор и тип данных для них.

```
ОбъявлениеПеременных = СписокИдент ":" Тип.
```

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. Статический тип определяет какие поля записи доступны. Динамический тип используется для вызова связанных с типом процедур (см. 10.2).

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
```

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* (или расширение *T*), то есть выполнение программы прерывается, если динамический тип *v* - не *T* (или расширение *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*

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)

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*)" и называется *проверкой типа*. Проверка типа применима, если

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

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 Присваивания

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

```
Присваивание = Обозначение ":=" Выражение.
```

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

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

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
```

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).

```
ВызовПроцедуры = Обозначение [ФактическиеПараметры].
```

Примеры:

```
WriteInt(i*2+1)      (* см. 10.1 *)
INC(w[k].count)
t.Insert("John")     (* см. 11 *)
```

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 (в оригинале - guard. *Прим. перев*.) Условия проверяются последовательно одно за другим, пока очередное не окажется равным TRUE, после чего выполняется связанная с этим условием последовательность операторов. Если ни одно условие не удовлетворено, выполняется последовательность операторов, записанная после слова ELSE, если оно 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
```

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 также определяется параметр-приемник. 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].
```

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

Если объявление процедуры содержит параметр-приемник, процедура рассматривается как связанная с типом (см. 10.2). *Опережающее объявление* служит чтобы разрешить ссылки на процедуру, чье фактическое объявление появляется 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 вместо формальных 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.

```
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*. Для параметров-значений *a* должен быть *совместим по присваиванию* с *f* (см. Прил. A).

Если *Tf* - открытый массив, то a должен быть *совместимым массивом* для *f* (см. Прил. A). 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. A). 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 Стандартные процедуры

Следующая таблица содержит список стандартных процедур. Некоторые процедуры - обобщенные, то есть они применимы к операндам нескольких типов. 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*2n)
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 фиксированной длины	размещает v ^
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.

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).

Последовательность операторов после символа BEGIN выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, тот циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы. Эти процедуры служат *командами* (см. Приложение D1).

```
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*.

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*.

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:

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 SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	BOOLEAN SET NIL, тип указатель T0 или T1 процедурный тип T, NIL	BOOLEAN BOOLEAN BOOLEAN BOOLEAN
IN	*целый*	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 Оберона-2

```
Модуль          =MODULE идент ";" [СписокИмпорта] ПослОбъявл [BEGIN
                  ПослОператоров] END идент ".".
СписокИмпорта   =IMPORT [идент ":="] идент {"," [идент ":="] идент} ";".
ПослОбъявл      ={ CONST {ОбъявлКонст ";" } | TYPE {ОбъявлТипа ";" }
                  | VAR {ОбъявлПерем ";" }} {ОбъявлПроц ";"
                  | ОпережающееОбъяв";"}.
ОбъявлКонст     =ИдентОпр "=" КонстВыраж.
ОбъявлТипа      =ИдентОпр "=" Тип.
ОбъявлПерем     =СписокИдент ":" Тип.
ОбъявлПроц      =PROCEDURE [Приемник] ИдентОпр [ФормальныеПарам]";"
                  ПослОбъявл [BEGIN ПослОператоров] END идент.
ОпережающееОбъяв =PROCEDURE "^" [Приемник] ИдентОпр [ФормальныеПарам].
ФормальныеПарам ="(" [СекцияФП {";" СекцияФП}] ")" [":" УточнИдент].
СекцияФП        =[VAR] идент {"," идент} ":" Тип.
Приемник        ="(" [VAR] идент ":" идент ")".
Тип             =УточнИдент | ARRAY [КонстВыраж {"," КонстВыраж}] OF Тип
                  | RECORD ["("УточнИдент")"] СписокПолей {";" СписокПолей} END
                  | POINTER TO Тип | PROCEDURE [ФормальныеПарам].
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       =Обозначение ["(" [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 [ "*" | "-" ].
```

Приложение C: Модуль SYSTEM

Модуль SYSTEM содержит некоторые типы и процедуры, которые необходимы для реализации операций *низкого уровня*, специфичных для данного компьютера и/или реализации. Они включают, например, средства для доступа к устройствам, которые управляются компьютером, и средства, позволяющие обойти правила совместимости типов, наложенные определением языка. Настоятельно рекомендуется ограничить использование этих средств специфическими модулями (модулями *низкого уровня*). Такие модули непременно являются непереносимыми, но легко распознаются по идентификатору SYSTEM, появляющемуся в их списке импорта. Следующие спецификации действительны для реализации Оберон-2 на компьютере Ceres.

Модуль SYSTEM экспортирует тип BYTE со следующими характеристиками: переменным типа BYTE можно присваивать значения переменных типа CHAR или SHORTINT. Если формальный параметр-переменная имеет тип ARRAY OF BYTE, то соответствующий фактический параметр может иметь любой тип.
Другой тип, экспортируемый модулем SYSTEM, - тип PTR. Переменным типа PTR могут быть присвоены значения переменных-указателей любого типа. Если формальный параметр-переменная имеет тип PTR, фактический параметр может быть указателем любого типа.
Процедуры, содержащиеся в модуле SYSTEM, перечислены в таблицах. Большинство их соответствует одиночным командам и компилируются непосредственно в машинный код. О деталях читатель может справиться в описании процессора. В таблице *v* обозначает переменную, *x, y, a,* и *n* - выражения, а *T* - тип.

Процедуры-функции

Название	Типы аргументов	Тип результата	Функция
ADR(v)	любой	LONGINT	адрес переменной v
BIT(a,n)	a: LONGINT; n:целый	BOOLEAN	n-й бит Память[a]
CC(n)	целая константа	BOOLEAN	условие n (0 <= n <= 15)
LSH(x,n)	x: целый, CHAR, BYTE; n: целый	совпадает с типом x	логический сдвиг
ROT(x,n)	x: целый, CHAR, BYTE; n: целый	совпадает с типом x	циклический сдвиг
VAL(T,x)	T, x: любого типа	T	x интерпретируется как значение типа T

Собственно процедуры

Название	Типы аргументов	Функция
GET(a, v)	a: LONGINT; v: любой основной тип, указатель, процедурный тип	v := Память[a]
PUT(a, x)	a: LONGINT; x: любой основной тип, указатель, процедурный тип	Память[a] := x
GETREG(n, v)	n: целая константа; v: любой основной тип, указатель, процедурный тип	v := Регистр n
PUTREG(n, x)	n: целая константа; x: любой основной тип, указатель, процедурный тип	Регистр n := x

MOVE(a0,a1,n)	a0, a1: LONGINT; n: целый	Память[a1..a1+n-1] := Память[a0..a0+n-1]
NEW(v, n)	v: любой указатель; n: целый	размещает блок памяти размером n байт; присваивает его адрес переменной v

Приложение D: Среда Оберон

Программы на Обероне-2 обычно выполняются в среде, которая обеспечивает *активацию команд*, *сбор мусора*, *динамическую загрузку модулей* и определенные *структуры данных времени выполнения*. Не являясь частью языка, эта среда способствует увеличению мощности Оберона-2 и до некоторой степени подразумевается при определении языка. В приложении D описаны существенные особенности типичной Оберон-среды и даны советы по реализации. Подробности можно найти в [1], [2], и [3].

D1. Команды

Команда - это любая процедура *P*, которая экспортируется модулем *M* и не имеет параметров. Она обозначается *M.P* и может быть активирована под таким именем из оболочки операционной системы. В Обероне пользователь вызывает команды вместо программ или модулей. Это дает лучшую структуру управления и предоставляет модули с несколькими точками входа. Когда вызывается команда *M.P*, модуль *M* динамически загружается, если он уже не был в памяти (см. D2) и выполняется процедура *P*. Когда *P* завершается, *M* остается загруженным. Все глобальные переменные и структуры данных, которые могут быть достигнуты через глобальные переменные-указатели в *M*, сохраняют значения. Когда *P* (или другая команда *M*) вызывается снова, она может продолжать использовать эти значения.

Следующий модуль демонстрирует использование команд. Он реализует абстрактную структуру данных *Counter*, которая содержит переменную-счетчик и обеспечивает команды для увеличения и печати его значения.

```
MODULE Counter;
   IMPORT Texts, Oberon;

   VAR
      counter: LONGINT;
      w: Texts.Writer;

PROCEDURE Add*; (* получает числовой аргумент из командной строки *)
   VAR s: Texts.Scanner;
BEGIN
   Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos);
   Texts.Scan(s);
   IF s.class = Texts.Int THEN INC(counter, s.i) END
```

```
END Add;

PROCEDURE Write*;
BEGIN
   Texts.WriteInt(w, counter, 5); Texts.WriteLn(w);
   Texts.Append(Oberon.Log, w.buf)
END Write;

BEGIN counter := 0; Texts.OpenWriter(w)
END Counter.
```

Пользователь может выполнить следующие две команды:

Counter.Add n	Добавляет значение n к переменной counter
Counter.Write	Выводит текущее значение counter на экран

Так как команды не содержат параметров, они должны получать свои аргументы из операционной системы. Вообще команды вольны брать параметры отовсюду (например из текста после команды, из текущего выбранного фрагмента или из отмеченного окна просмотра). Команда *Add* использует сканер (тип данных, обеспечиваемый Оберон-системой) чтобы читать значение, которое следует за нею в командной строке.
Когда *Counter.Add* вызывается впервые, модуль *Counter* загружается и выполняется его тело. Каждое обращение *Counter.Add n* увеличивает переменную *counter* на *n.* Каждое обращение *Counter.Write* выводит текущее значение *counter* на экран.

Поскольку модуль остается загруженным после выполнения его команд, должен существовать явный способ выгрузить его (например, когда пользователь хочет заменить загруженную версию перекомпилированной версией). Оберон-система содержит команду, позволяющую это сделать.

D2. Динамическая загрузка модулей

Загруженный модуль может вызывать команду незагруженного модуля, задавая ее имя как строку. Специфицированный модуль при этом динамически загружается и выполняется заданная команда. Динамическая загрузка позволяет пользователю запустить программу как небольшой набор базисных модулей и расширять ее, добавляя последующие модули во время выполнения по мере необходимости.

Модуль *M0* может вызвать динамическую загрузку модуля *M1* без того, чтобы импортировать его. *M1* может, конечно, импортировать и использовать *M0*, но *M0* не должен знать о существовании *M1*. *M1* может быть модулем, который спроектирован и реализован намного позже *M0*.

D3. Сбор мусора

В Обероне-2 стандартная процедура NEW используется, чтобы распределить блоки данных в свободной памяти. Нет, однако, никакого способа явно освободить распределенный блок. Взамен Оберон-среда использует *сборщик мусора* чтобы найти блоки, которые больше не

используются и сделать их снова доступными для распределения. Блок считается используемым только если он может быть достигнут через глобальную переменную-указатель по цепочке указателей. Разрыв этой цепочки (например, установкой указателя в NIL) делает блок утилизируемым.

Сборщик мусора освобождает программиста от нетривиальной задачи правильного освобождения структур данных и таким образом помогает избегать ошибок. Возникает, однако, необходимость иметь информацию о динамических данных во время выполнения (см. D5).

D4. Смотритель

Интерфейс модуля (объявления экспортируемых объектов) извлекается из модуля так называемым *смотрителем*, который является отдельным инструментом среды Оберон. Например, смотритель производит следующий интерфейс модуля *Trees* из Гл. 11.

```
DEFINITION Trees;
   TYPE
      Tree = POINTER TO Node;
      Node = RECORD
         name: POINTER TO ARRAY OF CHAR;
         PROCEDURE (t: Tree) Insert (name: ARRAY OF CHAR);
         PROCEDURE (t: Tree) Search (name: ARRAY OF CHAR): Tree;
         PROCEDURE (t: Tree) Write;
      END;
   PROCEDURE Init (VAR t: Tree);
END Trees.
```

Для типа запись смотритель также собирает все процедуры, связанные с этим типом, и показывает их заголовки в объявлении типа запись.

D5. Структуры данных времени выполнения

Некоторая информация о записях должна быть доступна во время выполнения: Динамический тип записей необходим для проверки и охраны типа. Таблица с адресами процедур, связанных с записью, необходима для их вызова. Наконец, сборщик мусора нуждается в информации о расположении указателей в динамически распределенных записях. Вся эта информация сохраняется в так называемых *дескрипторах типа*. Один дескриптор необходим во время выполнения для каждого типа записи. Ниже показана возможная реализация дескрипторов типа.

Динамический тип записи соответствует адресу дескриптора типа. Для динамически распределенных записей этот адрес сохраняется в так называемом *теге типа*, который предшествует фактическим данным записи и является невидимым для программиста. Если *t* - переменная типа *CenterTree* (см. пример в Гл. 6), рисунок D5.1 показывает одну из возможных реализаций структур данных времени выполнения.

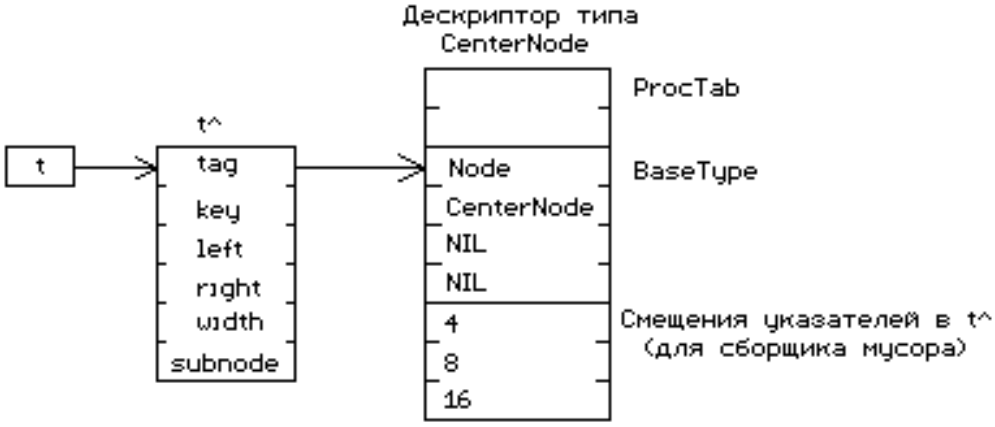

Рис. D5.1 переменная *t* типа *CenterTree*, запись *t^*, на которую она указывает, и дескриптор типа

Поскольку и таблица адресов процедур и таблица смещений указателей должны иметь фиксированное смещение относительно адреса дескриптора типа, и поскольку они могут расти, когда тип расширяется и добавляются новые процедуры и указатели, то таблицы размещены в противоположных концах дескриптора типа и растут в разных направлениях. Связанная с типом процедура *t.P* вызывается как *t^.tag^.ProcTab[IndexP]*. Индекс таблицы процедур для каждой связанной с типом процедуры известен во время компиляции. Проверка типа *v IS T* транслируется в *v^.tag^.BaseTypes [ExtensionLevelT] = TypeDescrAdrT*. И уровень расширения типа запись (*ExtensionLevelT*), и адрес описателя типа (*TypeDescrAdrT*) известны во время компиляции. Например, уровень расширения *Node* - 0 (этот тип не имеет базового типа), а уровень расширения *CenterNode* равен 1.

[1] N.Wirth, J.Gutknecht: The Oberon System. Software Practice and Experience 19, 9, Sept. 1989
[2] M.Reiser: The Oberon System. User Guide and Programming Manual. Addison-Wesley, 1991
[3] C.Pfister, B.Heeb, J.Templ: Oberon Technical Notes. Report 156, ETH Zurich, March 1991