Сообщение о языке Компонентный Паскаль

Oberon microsystems, Inc.
Technoparkstrasse 1
CH-8005 Zurich
Switzerland

Authors
Oberon microsystems, Inc.
March 2001

Authors of Oberon-2 report
H. Mossenbock, N. Wirth
Institut fur Computersysteme, ETH Zurich
October 1993

Author of Oberon report
N. Wirth
Institut fur Computersysteme, ETH Zurich
1987

Перевод выполнен выполнен с документации системы BlackBox Component Builder, v.1.4, для спецкурса "Современное программирование. Введение в объектные и компонентные технологии" (Ф.В.Ткачев, кафедра теоретической физики, физический факультет МГУ).

Терминология перевода в основном ориентируется на книгу:
Н. Вирт "Программирование на языке Модула-2", Москва, Мир, 1987, перевод В.А. Серебрякова и В.М. Ходукина под ред. В.М. Курочкина.

В квадратных скобках даны английские термины оригинала.
В угловых скобках — примечания переводчика.

Содержание

1. Введение

Компонентный Паскаль — усовершенствованная версия языка Оберон-2, разработанная компанией Oberon microsystems. Компания Oberon microsystems благодарит Х. Мёссенбёка и Н. Вирта за дружеское разрешение использовать их Сообщение о языке Оберон-2 в качестве основы данного документа.

Компонентный Паскаль — язык общего назначения в традиции языков Паскаль, Модула-2 и Оберон. Его самые важные черты — блочная структура, модульность, раздельная компиляция, статическая связь переменных с типами (статическая типизация, static typing) и строгая проверка типов (в том числе через границы модулей), <расширенное> переопределение типов [type extension] вместе с методами, динамическая загрузка модулей, а также <автоматический> сбор мусора.

Переопределение типов делает Компонентный Паскаль объектно-ориентированным языком. Объект — переменная абстрактного типа данных [abstract data type] <прилагательное "абстрактный" употребляется здесь в неспецифическом смысле, отличном от конкретного смысла, вкладываемого в атрибут записей ABSTRACT в Компонентном Паскале>, состоящая из приватных данных [private data] (т.е. его состояния) и процедур, оперирующих с этими данными. Абстрактные типы данных определяются как записи, допускающие <расширенное> переопределение [extensible records]. В Компонентном Паскале для большей части понятий объектно-ориентированных языков используется установившийся словарь императивных языков, чтобы минимизировать число терминов для сходных понятий. <Буквальный перевод терминологии оригинала оказывается громоздким в силу разной структуры английского и русского языков, поэтому в переводе для краткости пришлось добавить термины "потомок" и "предок" для описания типов.>

Полная защита в отношении типов переменных [complete type safety] и требование динамической объектной модели делают Компонентный Паскаль компонентно-ориентированным языком.

Данное сообщение не является учебником. Его краткость намеренна. Его функция — служить справочником для программистов. Большинство недоговоренностей оставлено намеренно, либо потому что соответствующее уточнение может быть выведено из сформулированных правил языка, либо потому что уточнение потребовало бы конкретизировать определения там, где универсальная конкретизация не кажется полезной.

Приложение А определяет некоторые термины, используемые для выражения правил проверки типов в Компонентном Паскале. В тексте такие термины даны курсивом, чтобы указать на их специальный смысл (например, *одинаковый* тип).

Рекомендуется минимизировать использование процедурных типов и супер-вызовов [super calls], т.к. они считаются устаревшими средствами. Они пока сохранены, чтобы облегчить использование существующих программ на Обероне-2. Поддержка этих средств может быть уменьшена в последующих выпусках продукта. В дальнейшем тексте то, что касается этих устаревших средств, отмечено красным цветом.

2. Синтаксис

Для описания синтаксиса Компонентного Паскаля используется расширенный формализм Бэкуса-Наура (РФБН). Альтернативы разделяются символом |. Квадратные скобки [и] означают необязательность заключенного в них выражения, а фигурные скобки { и } означают его возможное повторение (0 или более раз). В случае необходимости для группирования лексем используются круглые скобки (и). Нетерминальные лексемы начинаются с большой буквы (например, Statement). Терминальные лексемы либо начинаются с маленькой буквы (например, ident), либо записаны только большими буквами (например, BEGIN), либо обозначаются цепочками литер (например, ":=").

3. Словарь и изображение

Изображение (терминальных) лексем посредством литер использует стандарт ISO 8859-1, т.е. расширение Latin-1 набора литер ASCII. Такие лексемы суть идентификаторы, числа, операции и ограничители. Следует соблюдать следующие лексические правила: Пробелы и концы строк не должны появляться внутри лексем (за исключением комментариев, а также пробелов в литерных цепочках). Они игнорируются, если они не нужны для разделения двух последовательных лексем. Большие и маленькие буквы различаются.

1. Идентификаторы [identifiers] суть последовательности букв, цифр и символов подчеркивания. Первая литера не должна быть цифрой.

```
ident = (letter | "_") {letter | "_" | digit}.
letter = "A" .. "Z" | "a" .. "z" | "А".."Ц" | "Ш".."ц" | "ш".."я".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примеры:

```
x Scan Oberon2 GetSymbol firstLetter
```

2. Числа суть целые или вещественные константы (без знака). Типом целой константы является INTEGER, если значение константы принадлежит <диапазону значений типа> INTEGER, или LONGINT в противном случае (см. 6.1). Если константа задана с суффиксом 'H' или 'L', представление является 16-ричным, в противном случае представление десятичное. Суффикс 'H' используется для записи 32-битных констант в диапазоне -2147483648 .. 2147483647. Разрешается не более 8 значащих 16-ричных цифр. Суффикс 'L' используется для записи 64-битных констант.

Вещественное число всегда содержит десятичную точку. Оно также может содержать десятичный масштабный множитель. Буква E означает "умножить на 10 в степени". Вещественное число всегда имеет тип REAL.

```
number = integer | real.
integer = digit {digit} | digit {hexDigit} ( "H" | "L" ).
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+" | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
```

Примеры:

```
1234567           INTEGER 1234567
0DH               INTEGER 13
12.3              REAL 12.3
4.567E8           REAL 456700000
0FFFF0000H        INTEGER -65536
0FFFF0000L        LONGINT 4294901760
```

3. Литеры [characters] обозначаются своим порядковым номером в 16-ричной нотации, за которым следует буква X.

```
character = digit {hexDigit} "X".
```

4. Литерные цепочки [strings] — последовательности литер, заключенные в одиночные (') или двойные (") кавычки. Открывающая кавычка должна всегда совпадать с закрывающей, и не содержаться внутри цепочки. Число литер в цепочке называется ее длиной. Цепочка длины 1 может использоваться всюду, где разрешена литерная константа, и наоборот.

```
string = ' " ' {char} ' " ' | " ' " {char} " ' ".
```

Примеры:

```
"Component Pascal" "Don't worry!" "x"
```

5. Операции и ограничители [operators and delimiters] суть специальные литеры, пары литер или ключевые слова, перечисленные ниже. Ключевые слова содержат только большие буквы и не могут использоваться как идентификаторы.

+	:=	ABSTRACT	EXTENSIBLE	POINTER
-	^	ARRAY	FOR	PROCEDURE
*	=	BEGIN	IF	RECORD
/	#	BY	IMPORT	REPEAT
~	<	CASE	IN	RETURN
&	>	CLOSE	IS	THEN
.	<=	CONST	LIMITED	TO
,	>=	DIV	LOOP	TYPE
;	..	DO	MOD	UNTIL
|	:	ELSE	MODULE	VAR
$		ELSIF	NIL	WHILE
(	)	EMPTY	OF	WITH
[	]	END	OR	
{	}	EXIT	OUT	

6. Комментарии [comments] могут вставляться между любой парой лексем в программе. Они представляют собой произвольные последовательности литер, открывающиеся скобкой (* и закрывающиеся скобкой *). Комментарии могут быть вложены друг в друга. Они не влияют на смысл программы.

4. Описания и правила видимости

Каждый идентификатор, встречающийся в программе, должен быть введен с помощью описания [declaration], за исключением случаев, когда это предопределенный [predeclared] идентификатор. Описания также задают некоторые перманентные свойства объекта <термин "объект" употребляется здесь в неспецифическом смысле — "то, что может быть описано [declared] в языке">, такие как является ли он константой, типом, переменной или процедурой. Тогда идентификатор используется для ссылок на связанный с ним объект.

Область видимости [scope] объекта x текстуально распространяется от точки его описания до конца блока (модуля, процедуры или записи), к которому принадлежит описание и по отношению к которому объект, таким образом, считается локальным [local]. Из этой области исключаются области видимости объектов с таким же именем, описанных в блоках, вложенных в данный. Правила видимости таковы:

1. Идентификатор может обозначать только один объект в данной области видимости (т.е. никакой идентификатор не может быть объявлен в блоке дважды);

2. На объект можно сослаться только в его области видимости;

3. Описание типа T, содержащее ссылки на другой тип T1 могут стоять в точках, где T1 еще не известен. Описание типа T1 должно следовать далее в том же блоке, в котором локализован T;

4. Идентификаторы полей записей [record fields] (см. 6.3) или методов (см. 10.2) могут встречаться только в составных именах [designators]. <В книге по Модуле-2 для перевода узкоспециального термина designator выбрано неспецифичное слово "обозначение", что не представляется удачным.>

За идентификатором, описанном в блоке модуля, в его описании может следовать метка экспорта (" * " или " - "), чтобы указать, что он экспортируется. Идентификатор x, экспортированный из модуля M, можно использовать в других модулях при условии, что они импортируют M (см гл. 11). В таких модулях идентификатор обозначается как M.x и называется квалифицированным идентификатором [qualified identifier]. Переменные и поля записей, отмеченные в своих описаниях символом " - ", доступны в импортирующих модулях только для чтения [are read-only] (в случае переменных и полей) или только для реализации [implement-only] (в случае методов).

```
Qualident = [ident "."] ident.
IdentDef = ident [" * " | " - "].
```

Следующие идентификаторы являются предопределенными; их значения описаны в указанных разделах:

ABS	(10.3)	INTEGER	(6.1)
ANYPTR	(6.1)	FALSE	(6.1)
ANYREC	(6.1)	LEN	(10.3)
ASH	(10.3)	LONG	(10.3)
ASSERT	(10.3)	LONGINT	(6.1)
BITS	(10.3)	MAX	(10.3)
BOOLEAN	(6.1)	MIN	(10.3)
BYTE	(6.1)	NEW	(10.3)
CAP	(10.3)	ODD	(10.3)
CHAR	(6.1)	ORD	(10.3)
CHR	(10.3)	REAL	(6.1)
DEC	(10.3)	SET	(6.1)
ENTIER	(10.3)	SHORT	(10.3)
EXCL	(10.3)	SHORTCHAR	(6.1)
HALT	(10.3)	SHORTINT	(6.1)
INC	(10.3)	SHORTREAL	(6.1)
INCL	(10.3)	SIZE	(10.3)
INF	(6.1)	TRUE	(6.1)

5. Описания констант

Описание константы связывает идентификатор с некоторым неизменяемым значением.

```
ConstantDeclaration = IdentDef "=" ConstExpression.
ConstExpression = Expression.
```

Константное выражение — это выражение, которое может быть вычислено при простом текстуальном просмотре без фактического выполнения программы. Его операнды суть константы (гл. 8) или предопределенные функции (10.3), которые могут быть вычислены при компиляции.

Примеры константных выражений:

```
N = 100
limit = 2*N - 1
fullSet = {MIN(SET) .. MAX(SET)}
```

6. Описания типов

Тип данных определяет множество значений, которое может принимать переменная этого типа, а также применимые к ней операции. Описание типа связывает идентификатор с типом. В случае структурированных типов (массивов и записей) оно также определяет структуру переменных этого типа. Структурированный тип не может содержать себя.

```
TypeDeclaration = IdentDef "=" Type.
Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.
```

Примеры:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node = EXTENSIBLE RECORD
    key : INTEGER;
    left, right: Tree
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
    width: INTEGER;
    subnode: Tree
END
Object = POINTER TO ABSTRACT RECORD END;
Function = PROCEDURE (x: INTEGER): INTEGER
```

6.1 Основные типы [base types]

Основные типы обозначаются предопределенными идентификаторами. Соответствующие операции определены в 8.2, а предопределенные процедуры-функции — в 10.3. Значения основных типов таковы:

1.	BOOLEAN	логические значения TRUE и FALSE
2.	SHORTCHAR	литеры набора Latin-1 (0X .. 0FFX)
3.	CHAR	литеры набора Unicode (0X .. 0FFFFX)
4.	BYTE	целые от MIN(BYTE) до MAX(BYTE)
5.	SHORTINT	целые от MIN(SHORTINT) до MAX(SHORTINT)
6.	INTEGER	целые от MIN(INTEGER) до MAX(INTEGER)
7.	LONGINT	целые от MIN(LONGINT) до MAX(LONGINT)
8.	SHORTREAL	вещественные числа от MIN(SHORTREAL) до MAX(SHORTREAL), значение INF
9.	REAL	вещественные числа от MIN(REAL) до MAX(REAL), значение INF
10.	SET	множества целых чисел из диапазона от 0 до MAX(SET)

Типы 4-7 суть целые типы, типы 8 и 9 суть вещественные типы, а вместе они называются числовыми типами. Они образуют иерархию; старший тип включает младший (включение касается значений):

```
REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE
```

Типы 2 и 3 суть литерные типы со следующей иерархией типов:

```
CHAR >= SHORTCHAR
```

6.2 Типы массивов [array types]

Массив — структура, состоящая из некоторого количества элементов, имеющих один и тот же тип, называемый типом элементов. Количество элементов массива называется его длиной. Элементы массива выбираются с помощью индексов, являющихся целыми числами из диапазоне от 0 до длина минус 1.

```
ArrayType = ARRAY [Length {"," Length}] OF Type.
Length = ConstExpression.
```

Тип вида

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

интерпретируется как сокращенная запись для

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

Массивы, описанные без длины, называются открытыми массивами. Их использование ограничено базовыми типами для указателей (см. 6.4), типами элементов открытых массивов, а также типами формальных параметров (см. 10.1).

Примеры:

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

6.3 Типы записей [record types]

Запись это структура, состоящая из фиксированного количества элементов, называемых полями, возможно, имеющих разные типы. Описание типа записей указывает имя и тип каждого поля. Область видимости идентификаторов полей распространяется от точки их описания до конца данного типа, но они также видимы внутри составных имен, обозначающих поля переменных данного типа (см. 8.1). Если тип записей экспортируется, то идентификаторы полей, которые нужно иметь видимыми вне описывающего модуля, должны быть соответствующим образом помечены. Они называются публичными полями [public fields]; непомеченные элементы называются приватными полями [private fields].

```
RecordType = RecAttributes RECORD ["("BaseType")"] FieldList {";" FieldList} END.
RecAttributes = [ABSTRACT | EXTENSIBLE | LIMITED].
BaseType = Qualident.
FieldList = [IdentList ":" Type].
IdentList = IdentDef {"," IdentDef}.
```

Использование каждого типа записей ограничено наличием или отсутствием одного из следующих атрибутов:

ABSTRACT, **EXTENSIBLE** и **LIMITED**.

Переменные типа записей, помеченного как ABSTRACT, не могут быть размещены <также: инстанцированы [instantiated]>: не может существовать ни переменных, ни полей такого типа. Абстрактные типы <т.е. описанные с атрибутом ABSTRACT> используются только как базовые типы для других типов записей (см. ниже).

Переменные типа записей, имеющего атрибут LIMITED, могут размещаться [allocated] только внутри того модуля, где описан данный тип. Ограничение применимо как к статическому размещению посредством описания переменных (гл. 7), так и к динамическому размещению с помощью стандартной процедуры NEW (10.3).

Тип записей, помеченный как ABSTRACT или EXTENSIBLE, является расширяемым, т.е. можно определить новый тип записей как расширение такого типа.

В примере

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

T1 является (непосредственным) потомком [extension] типа T0, а T0 — (непосредственным) предком [base type] для T1 (см. Приложение A). <В оригинале термин base type используется в двух разных вариантах, поэтому выбран данный вариант перевода.> Тип-потомок [extended type] T1 состоит из полей типа-предка и полей, объявленных в T1. Все идентификаторы, описанные в типе-потомке, должны быть отличны от идентификаторов, описанных в его предке. Тип-предок абстрактного типа записей должен быть абстрактным.

Кроме того, в качестве типа-предка можно использовать указательный тип. В этом случае в качестве типа-предка описываемого типа записей используется базовый тип записей данного указателя.

Нельзя экспортировать тип записей, являющийся потомком скрытого (т.е. не экспортированного) типа записей.

Каждая запись является неявным потомком предопределенного типа ANYREC. ANYREC не содержит никаких полей и может использоваться только в описаниях указателей и параметров-переменных.

Сводка атрибутов:

атрибут	определение потомков [extension]	размещение
отсутствует	нет	да
EXTENSIBLE	да	да
ABSTRACT	да	нет
LIMITED	только в определяющем	модуле

Примеры описаний типов записей:

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

6.4 Указательные типы [pointer types]

Переменные, имеющие указательный тип P, принимают в качестве значений указатели на переменные некоторого типа T. T называется базовым типом [pointer base type] для P и должен быть типом записей или типом массивов. Указательные типы наследуют отношение потомок-предок для своих базовых типов: если тип T1 является потомком типа T, а P1 имеет тип POINTER TO T1, то P1 также является потомком типа P.

```
PointerType = POINTER TO Type.
```

Если p — переменная типа P = POINTER TO T, то вызов предопределенной процедуры NEW(p) (см. 10.3) размещает переменную типа T в свободной памяти. Если T — тип записей или тип массивов с фиксированной длиной, то размещение следует выполнять оператором NEW(p); если T — n-мерный открытый массив, то размещение следует выполнять оператором NEW(p, e0, ..., en-1), где T размещается с длинами, задаваемыми выражениями e0, ..., en-1. В обоих случаях указатель на размещенную переменную присваивается переменной p. p имеет тип P. Разыменованная [referenced <т.е. та, на которую ссылается p>] переменная p^ (читается: p-разыменованная), имеет тип T. Любой указатель может принимать значение NIL, которое не указывает ни на какую переменную вообще.

Все поля и элементы вновь размещенной записи или массива очищаются; в частности, значения все содержащиеся в них указательные и процедурные переменные устанавливаются в NIL.

Предопределенный тип ANYPTR определяется как POINTER TO ANYREC. Поэтому любой указатель на какой-либо тип записей является потомком типа ANYPTR. Процедуру NEW применять к переменным типа ANYPTR нельзя.

6.5 Процедурные типы

Переменные процедурного типа T имеют в качестве значения некоторую процедуру или NIL. Если переменной типа T присвоена процедура P, то списки формальных параметров (см. 10.1) для P и T должны соответствовать [match] (см. Приложение A). P не может быть ни предопределенной процедурой, ни методом, ни быть локальной в другой процедуре.

```
ProcedureType = PROCEDURE [FormalParameters].
```

6.6 Типы цепочек литер [string types]

Значения типов цепочек литер суть последовательности литер, оканчивающиеся нуль-литерой (0X). Длина цепочки — это количество литер в ней, исключая нуль-литеру.

Цепочки могут либо быть константами, либо храниться в массиве литерного типа. Для типов цепочек нет предопределенных идентификаторов, т.к. они для описаний не нужны.

Константные цепочки, состоящие только из литер в диапазоне 0X..0FFX, и цепочки, хранящиеся в массиве элементов типа SHORTCHAR, имеют тип Shortstring, все остальные — тип String.

7. Описания переменных

Описания переменных вводят переменные, определяя для них идентификатор и тип данных.

```
VariableDeclaration = IdentList ":" Type.
```

Переменные типов записей и указательных типов имеют как статический тип (тот тип, с которым они описаны — будем просто называть его их типом) и динамический тип (тип их значения при выполнении программы). Для указателей и параметров-переменных типа записей динамический тип может быть потомком их статического типа. Статический тип определяет, какие поля записи доступны. Динамический тип используется для вызова методов (см. 10.2).

Примеры описаний переменных (см. примеры в 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 это конструкции, описывающие вычислительные правила, в соответствии с которыми комбинируются константы и текущие значения переменных для вычисления других значений посредством применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для выражения конкретных связей между операциями и операндами.

8.1 Операнды

За исключением конструкторов множеств и буквальных [literal] констант (т.е. чисел, литер или цепочек), операнды обозначаются составными именами [designators]. Составное имя содержит идентификатор, обозначающий константу, переменную или процедуру. Этот идентификатор может быть квалифицирован [qualified] идентификатором модуля (см. гл. 4 и 11), и кроме того за ним могут следовать селекторы [selectors], если обозначаемый объект является элементом массива или полем записи.

```
Designator = Qualident {"." ident | "[" ExpressionList "]" | "^" | "(" Qualident
")" | ActualParameters} [ "$" ].
ExpressionList = Expression {"," Expression}.
ActualParameters = "(" [ExpressionList] ")".
```

Если a — составное имя, обозначающее массив, то a[e] обозначает тот элемент массива a, у которого индекс равен текущему значению выражения e. Тип значения e должен быть целым. Составное имя, имеющее вид a[e0, e1, ..., en], является сокращением для a[e0][e1]...[en]. Если r является составным именем, обозначающим запись, то r.f обозначает поле f записи r или метода f динамического типа записи r (10.2). Если a или r доступны только для чтения, то это же справедливо для a[e] и r.f.

Если p — составное имя, обозначающее указатель, то p^ обозначает переменную, на которую ссылается p. Составные имена p^.f, p^[e] и p^$ можно сокращать до p.f, p[e] и p$, т.е. селекторы записи, массива и литерной цепочки подразумевают разыменование [dereferencing]. Разыменование также подразумевается, если указатель присваивается переменной типа запись или типа массив (9.1), если указатель используется как фактический параметр, соответствующий формальному параметру типа запись или типа массив (10.1), или если указатель используется как аргумент стандартной процедуры LEN (10.3).

Охрана типа [type guard] v(T) удостоверяет, что динамический тип переменной v есть T (или потомок типа T), т.е. выполнение программы аварийно прекращается, если динамический тип переменной v не T (и не потомок T). Тогда внутри составного имени переменная v считается имеющей статический тип T. Охрана применима, если

1. v является IN или VAR параметром типа записей или v является указателем на запись, и если
2. T является потомком статического типа переменной v

Если обозначаемый объект — константа или переменная, то <в контексте выражения> составное имя ссылается на ее значение в данный момент. Если это процедура, то составное имя ссылается на эту процедуру, исключая случаи, когда за ним следует (возможно, пустой) список параметров, и в этих случаях оно подразумевает вызов процедуры и представляет получающийся результат. Фактические параметры должны соответствовать формальным как в обычных вызовах процедур (см. 10.1).

Если a — составное имя, обозначающее массив литер, то a$ обозначает строку, оканчивающуюся нуль-литерой, содержащуюся в a. Если a не содержит литеру 0X, то использование a$ приводит к ошибке при выполнении программы. Селектор $ применяется неявно, если a используется как операнд конкатенации (8.2.4), операции отношения (8.2.5) или одной из предопределенных процедур LONG и SHORT (10.3).

Примеры составных имен (см. примеры в гл. 7):

```
i (INTEGER)
a[i] (REAL)
w[3].name[i] (CHAR)
t.left.right (Tree)
t(CenterTree).subnode (Tree)
w[i].name$ (String)
```

8.2 Операции [operators]

В выражениях синтаксически различаются операторы четырех классов с разными приоритетами [precedences] (т.е. силой связывания [binding strengths]). Операция ~ имеет наивысший приоритет, за ней следуют мультипликативные операции, аддитивные операции и, наконец, отношения. Операции с одинаковым приоритетом связывают операнды слева направо. Например, x-y-z обозначает (x-y)-z.

```
Expression = SimpleExpression [Relation SimpleExpression].
SimpleExpression = ["+" | "-"] Term {AddOperator Term}.
Term = Factor {MulOperator Factor}.
Factor = Designator | number | character | string | NIL | Set | "(" Expression
")" | "~" Factor.
Set = "{" [Element {"," Element}] "}".
Element = Expression [".." Expression].
Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOperator = "+" | "-" | OR.
MulOperator = "*" | "/" | DIV | MOD | "&".
```

Имеющиеся операции перечислены в нижеследующих таблицах. Некоторые операции применимы к операндам различных типов, обозначая различные операции. В этих случаях фактическая операция определяется типами операндов. Операнды должны быть совместимы по выражению [expression compatible] по отношению к операции (см. Приложение A).

8.2.1 Логические операции

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN. Второй операнд логического ИЛИ вычисляется только если результат первого равен FALSE. Второй операнд логического И вычисляется, только если результат первого равен TRUE.

OR	логическое ИЛИ	p OR q	"если p, то TRUE, иначе q"
&	логическое И	p & q	"если p, то q, иначе FALSE"
~	отрицание	~ p	"не p"

8.2.2 Арифметические операции

+	сумма
-	разность
*	произведение
/	вещественное частное
DIV	целое частное
MOD	остаток

Операции +, -, * и / применимы к операндам числовых типов. Типом результата будет REAL в случае операции деления (/) или если один из операндов имеет тип REAL. В противном случае типом результата будет SHORTREAL, если тип одного из операндов — SHORTREAL, LONGINT если тип одного из операндов — LONGINT, или INTEGER в любом другом случае. Если результат вещественной операции слишком велик, чтобы представлять вещественное число, то он заменяется на предопределенное значение INF с тем же знаком, что и исходный результат. Заметим, что это правило применимо и к 1.0/0.0, но не к 0.0/0.0, т.к. последнее выражение не имеет определенного результата вообще и приводит к ошибке при выполнении программы. Если используются одноместные операции, - обозначает перемену знака, а + обозначает тождественную операцию. Операции DIV и MOD применимы только к целым операндам. Они связаны следующими формулами:

```
x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y или  0 >= (x MOD y) > y
Заметим, что: x DIV y = ENTIER(x / y)
```

Примеры:

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

Заметим, что: `(-5) DIV 3 = -2`, но `(-5) DIV 3 = -(5 DIV 3) = -1`

8.2.3 Операции над множествами

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

Операции над множествами применимы к операндам типа SET и дают результат типа SET. Одноместный минус обозначает дополнение множества x, т.е. -x обозначает множество целых от 0 до MAX(SET), которые не являются элементами множества x. Операции над множествами не являются ассоциативными ((a+b)-c # a+(b-c)).

Конструктор множества определяет значение множества перечислением его элементов между фигурными скобками. Элементы должны быть целыми в диапазоне 0..MAX(SET). Диапазон a..b обозначает все целые i такие, что i >= a и i <= b.

8.2.4 Операции над цепочками

\+ конкатенация

Операция конкатенации применяется к операндам типов цепочек. Получающаяся цепочка состоит из литер первого операнда, за которыми следуют литеры второго операнда. Если оба операнда имеют тип Shortstring, то тип результата будет Shortstring, в противном случае тип результата будет String.

8.2.5 Отношения

=	равно
#	неравно
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
IN	принадлежность множеству
IS	проверка типа

Отношения дают результат типа BOOLEAN. Отношения =, #, <, <=, > и >= применимы к числовым типам, литерным типам и типам цепочек литер. Отношения = и # применимы также к BOOLEAN и SET, а также к указательным и процедурным типам (включая значение NIL). x IN s означает "x является элементом s". x должен быть целым в диапазоне 0..MAX(SET), а s иметь тип SET. v IS T означает "динамический тип переменной v равен T (или потомку типа T)" и называется проверкой типа. Она применима, если

1. v является IN или VAR параметром типа записей или указателем на тип записей, и если
2. T является потомком статического типа v

Примеры выражений (см. примеры в 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 [statements]

Операторы обозначают действия. Есть элементарные и структурированные операторы. Элементарные операторы не содержат частей, которые сами являются операторами. Это: присваивание, вызов процедуры, оператор возврата RETURN и оператор выхода EXIT. Структурированные операторы состоят из частей, которые сами являются операторами. Они используются для выражения последовательного, условного, выборочного и повторяющегося выполнения. Оператор может быть пустым, и в этом случае он обозначает отсутствие действия. Пустой оператор разрешен, чтобы ослабить правила пунктуации в операторных последовательностях.

```
Statement = [ Assignment | ProcedureCall | IfStatement | CaseStatement |
WhileStatement | RepeatStatement |
ForStatement | LoopStatement | WithStatement |
EXIT | RETURN [Expression] ].
```

9.1 Присваивания [assignments]

Присваивание заменяет текущее значение переменной новым, определяемым неким выражением. Выражение должно быть совместимым по присваиванию [assignment compatible] с переменной (см. Приложение A). Операция присваивания записывается как ":=" и произносится становится равной.

```
Assignment = Designator ":=" Expression.
```

Если выражение e типа Te присваивается переменной v типа Tv, происходит следующее:

1. если Tv и Te имеют тип запись, динамические и статические типы e и v все равны и присваиваются все поля этого типа;
2. если Tv и Te имеют указательные типы, то динамическим типом v становится динамический тип e;
3. если Tv — массив литерного типа, а e — цепочка литер длины m < LEN(v), то v[i] становится равным ei для i = 0..m-1, а v[m] становится равным 0X. Если m >= LEN(v), то в процессе выполнения генерируется ошибка.

Примеры присваиваний (см. примеры в 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 [procedure calls]

Вызов процедуры активизирует процедуру. Он может содержать список фактических параметров, которые замещают соответствующие формальные параметры, определенные в описании процедуры (см. гл. 10). Соответствие устанавливается по положению параметров в списках фактических и формальных параметров. Имеется два типа параметров: параметры-переменные и параметры-значения.

Если формальный параметр является параметром-переменной, то соответствующий фактический параметр должен быть составным именем некоторой переменной. Если он обозначает элемент структурированной переменной, то соответствующие селекторы вычисляются при подстановке фактического параметра вместо формального, т.е. до выполнения процедуры. Если формальный параметр — параметр-значение, то соответствующий фактический параметр должен быть выражением. Это выражение вычисляется до активизации процедуры, а получающееся значение присваивается формальному параметру (см. также 10.1).

```
ProcedureCall = Designator [ActualParameters].
```

Примеры:

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

9.3 Операторные последовательности

Операторная последовательность обозначает последовательность действий, указанных отдельными операторами, разделенными точками с запятой.

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

9.4 Условный оператор IF

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

Оператор IF задает условное выполнение охраняемых [guarded] операторных последовательностей. Логическое выражение, предшествующее операторной последовательности, называется его охраной [guard]. Охраны вычисляются в том порядке, в котором они встречаются в тексте, до тех пор, пока одна из них не даст значение TRUE, после чего выполняется соответствующая операторная последовательность. Если ни одна охрана не будет удовлетворена, выполняется операторная последовательность, следующая за лексемой ELSE, если таковая имеется.

Пример:

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

9.5 Оператор выбора CASE

Оператор CASE указывает выбор и выполнение некоторой операторной последовательности в зависимости от значения некоторого выражения. Сначала вычисляется выражение выбора, затем выполняется та операторная последовательность, чей список меток выбора содержит полученное значение. Выражение выбора должно иметь целый или литерный тип, который включает значения всех меток выбора. Метки выбора — константы, и никакое значение не может встречаться более одного раза. Если значение выражения не встречается в качестве метки, то выполняется операторная последовательность, следующая за лексемой ELSE, если таковая имеется, в противном случае программа аварийно останавливается.

```
CaseStatement = CASE Expression OF Case {"|" Case}[ELSE StatementSequence] END.
Case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
```

Пример:

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

9.6 Цикл с условием продолжения WHILE

Оператор цикла WHILE описывает выполнение операторной последова-тельности, повторяющееся, пока некое логическое выражение (его охрана) дает TRUE. Охрана проверяется заново перед каждым вычислением операторной последовательности.

```
WhileStatement = WHILE Expression DO StatementSequence 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 Цикл с условием окончания REPEAT

Оператор цикла REPEAT описывает выполнение операторной последовательности, повторяющееся до тех пор, пока некое логическое выражение не даст TRUE. Последовательность выполняется хотя бы один раз.

```
RepeatStatement = REPEAT StatementSequence UNTIL Expression.
```

9.8 Цикл с шагом FOR

Оператор цикла FOR описывает выполнение операторной последовательности, повторяющееся для последовательности значений, присваиваемых целой переменной, называемой управляющей переменной цикла.

```
ForStatement =
FOR ident ":=" Expression TO Expression [BY ConstExpression] DO StatementSequence
END.

Оператор

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

эквивалентен следующему

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 имеет одинаковый тип с переменной v. step должно быть ненулевым константным выражением. Если выражение step не указано, оно берется 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 Безусловный цикл LOOP

Оператор LOOP описывает повторяющееся выполнение операторной последовательности. Оно прекращается выполнением оператора выхода EXIT внутри этой последовательности (см. 9.10).

```
LoopStatement = LOOP StatementSequence END.
```

Пример:

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

Циклы LOOP полезны для выражения повторяющихся вычислений с несколькими точками выхода или в тех случаях, когда условие выхода находится в середине повторяющейся операторной последовательности.

9.10 Операторы возврата RETURN и выхода EXIT

Оператор возврата RETURN указывает прекращение процедуры. Он обозначается лексемой RETURN, за которой следует выражение, если речь идет о процедуре-функции. Тип выражения должен быть совместим по присваиванию (см. Приложение A) с типом результата, указанным в заголовке процедуры (см. гл. 10).

Процедуры-функции требуют наличия оператора возврата, указывающего значение-результат. В обычных процедурах [proper procedures], оператор возврата неявно подразумевается в конце тела процедуры. Поэтому любой явный оператор возврата появляется как дополнительная (вероятно, исключительная) точка выхода.

Оператор выхода обозначается лексемой EXIT. Он означает, что выполнение охватывающего оператора LOOP должно быть прекращено, а выполнение программы должно быть продолжено с оператора, следующего за этим оператором LOOP. Оператор EXIT контекстуально, хотя и не синтаксически, связан с содержащим его оператором LOOP.

9.11 Операторы конкретизации типа WITH

Оператор WITH выполняет операторную последовательность в зависимости от результата проверки типа и применяет охрану типа к каждому вхождению проверяемой переменной внутри операторной последовательности.

```
WithStatement = WITH [ Guard DO StatementSequence ]
    {"|" [ Guard DO StatementSequence} ]
    [ELSE StatementSequence] END.
Guard = Qualident ":" Qualident.
```

Если v — параметр-переменная типа записей или указательная переменная, и если ее статический тип T0, то значение оператора

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

таково: если динамическим типом переменной v оказался T1, то выполняется операторная последовательность S1, в которой v рассматривается как если бы ее статическим типом был T1; в противном случае если динамическим типом v оказался T2, то выполняется S2, в которой v рассматривается как если бы ее статическим типом был T2; в противном случае выполняется S3. T1 и T2 должны быть потомками типа T0. Если ни одна проверка типа не дала положительного результата и если отсутствует ELSE (вместе с соответствующей операторной последовательностью), то программа аварийно останавливается.

Пример:

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

10. Описания процедур

Описание процедуры состоит из заголовка процедуры [procedure heading] и тела процедуры [procedure body]. Заголовок задает идентификатор процедуры и ее формальные параметры. Для методов он также задает принимающий параметр [receiver parameter] и атрибуты (см. 10.2). Тело содержит объявления и операторы. Идентификатор процедуры повторяется в конце ее описания.

Есть два вида процедур: обычные процедуры [proper procedures] и процедуры-функции [function procedures]. Процедура-функция активизируется составным именем функции, выступающим в качестве части выражения, и дает результат, являющийся операндом выражения. Обычные процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если список ее формальных параметров задает некоторый тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет ее результат.

Все константы, переменные, типы и процедуры, описанные внутри тела процедуры, являются локальными по отношению к этой процедуре. Поскольку процедуры тоже могут описываться как локальные объекты, описания процедур могут быть вложены друг в друга. Вызов процедуры внутри ее описания означает рекурсивную активизацию.

Локальным переменным указательных или процедурных типов присваивается значение NIL перед выполнением тела процедуры.

Объекты, описанные в контексте, окружающем процедуру, видимы также и в тех частях процедуры, где они не перекрыты локально определенным объектом с точно тем же именем.

```
ProcedureDeclaration = ProcedureHeading ";" [ ProcedureBody ident ].
ProcedureHeading = PROCEDURE [Receiver] IdentDef [FormalParameters]
MethAttributes.
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
DeclarationSequence = {CONST {ConstantDeclaration ";"} |
    TYPE {TypeDeclaration ";"} |
    VAR {VariableDeclaration ";"} }
    {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
ForwardDeclaration = PROCEDURE " ^ " [Receiver] IdentDef [FormalParameters]
MethAttributes.
```

Если в описании процедуры указан принимающий параметр [receiver parameter], то процедура считается методом, связанным с типом принимающего параметра (см. 10.2). Упреждающее описание [forward declaration] служит для того, чтобы разрешить ссылки на процедуру, чье фактическое описание содержится в последующем тексте. Списки формальных параметров упреждающего и фактического описаний должны соответствовать [match] (см. Приложение А), а имена соответствующих параметров должны быть равными. <На практике для задания упреждающего описания достаточно скопировать заголовок процедуры и вставить в копии литеру "^" после лексемы PROCEDURE.>

10.1 Формальные параметры

Формальные параметры суть идентификаторы, описанные в списке формальных параметров процедуры. Они соответствуют фактическим параметрам, указанным в вызове процедуры. Соответствие между формальными и фактическими параметрами устанавливается при вызове процедуры. Есть два вида параметров: параметры-значения [value parameters] и параметры-переменные [variable parameters], последние отмечены в списке формальных параметров одним из описателей VAR, IN или OUT. Параметры-значения представляют собой локальные переменные, которым в качестве начального значения присваивается значение соответствующего фактического параметра. Параметры-переменные соответствуют фактическим параметрам, являющимся переменным, и они представляют [stand for] эти переменные. Параметры-переменные могут использоваться только для ввода данных (IN), только для вывода данных (OUT) или для ввода и вывода (VAR). Описатель IN можно использовать только для параметров, являющихся массивами или записями. Внутри процедуры IN-параметры доступны только для чтения. Как и локальные переменные, OUT-параметры указательных или процедурных типов инициализируются в NIL. Значения других выходных параметров должны рассматриваться как неопределенные до первого присваивания в процедуре.

Область видимости формального параметра распространяется от его описания до конца процедурного блока, в котором он описан. Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызываться с помощью составного имени, у которого есть пустой список фактических параметров. Тип результата процедуры не может быть ни записью, ни массивом.

```
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection = [VAR | IN | OUT] ident {"," ident} ":" Type.
```

```
Пусть f — формальный параметр, и пусть a — соответствующий фактический параметр.
Если f — открытый массив, то a должен быть совместим по массивам [array
compatible] с f, и длины f берутся из a. В противном случае a должен быть
совместим по параметрам [parameter compatible] с f (см. Приложение A).
```

```
Примеры описаний процедур:
```

```
PROCEDURE ReadInt (OUT 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 (IN 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; (* assume x > 0 *)
BEGIN
    y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
    RETURN y
END Log2

PROCEDURE Modify (VAR n: Node);
BEGIN
    INC(n.key)
END Modify
```

10.2 Методы

Процедуры, описанные глобально <т.е. на уровне модуля, а не в процедурах, вложенных м модуль>, могут быть связаны с каким-либо типом записей, описанным в том же модуле. Такие процедуры называют методами [methods], связанными с данным типом записей. Связь выражается посредством указания типа принимающего параметра в заголовке описания процедуры. Получающий параметр может быть VAR или IN параметром типа T или параметром-значением типа POINTER TO T, где T — тип записей. Метод связан с типом T и считается в нем локальным.

```
ProcedureHeading = PROCEDURE [Receiver] IdentDef [FormalParameters]
MethAttributes.
Receiver = "(" [VAR | IN] ident ":" ident ")".
MethAttributes = ["," NEW] ["," (ABSTRACT | EMPTY | EXTENSIBLE)].
```

Если метод M связан с типом T0, он также неявно связан с любым потомком T1 типа T0. Однако если метод M' (с тем же именем, что и у M) описан как связанный с T1, он становится связан с T1 вместо

M. M' считается переопределением M для T1. Списки формальных параметров M и M' должны соответствовать [match], кроме случаев, когда M — процедура-функция, возвращающая указательный тип. В последнем случае тип результата функции M' должен быть расширением типа результата M (ковариантность) (см. Приложение A). Если M и T1 экспортируются (см. гл. 4), то M' тоже должен экспортироваться.

Если M не экспортируется, то M' тоже не должен экспортироваться. Если M и M' экспортируются, их метки экспорта должны быть одинаковыми.

Для ограничения и документирования предполагаемого использования метода используются следующие атрибуты:

NEW, **ABSTRACT**, **EMPTY** и **EXTENSIBLE**

— соответственно: новый, абстрактный, пустой и переопределяемый.

Необходимо использовать атрибут NEW для всех вновь вводимых методов, и его нельзя использовать для переопределяющих методов. Этот атрибут помогает обнаружить несогласованности в определениях методов, связанных с типом и его потомками; такие несогласованности могут возникнуть при переименованиях методов.

Описания абстрактных и пустых методов состоят только из заголовка процедуры. Абстрактные методы не могут вызываться. Тип записей, содержащий абстрактный метод, должен быть абстрактным. Метод, переопределяемый абстрактным методом, должен быть абстрактным. Абстрактный метод экспортируемого типа запись должен экспортироваться. Вызов пустого метода не имеет никакого эффекта. Пустые методы не могут быть процедурами-функциями и не могут иметь OUT параметров. Тип записей, содержащий новые пустые методы, должен быть переопределяемым или абстрактным. Метод, переопределенный пустым методом, должен быть пустым или абстрактным. Абстрактные или пустые методы некоторого типа обычно переопределяются (реализуются) в его потомках. Они не должны вызываться супер-вызовами. Конкретный (не абстрактный) тип записей, являющийся потомком абстрактного типа, должен реализовать все абстрактные методы своего типа-предка.

Конкретные методы (имеющие процедурное тело) должны быть либо переопределяемыми, либо конечными [final] (т.е. описанными без атрибутов). Конечный метод не может переопределяться в потомках своего типа. Тип записей, содержащий переопределяемые методы должен быть переопределяемым или абстрактным.

Если v — составное имя, а M — метод, то v.M обозначает метод M, связанный с динамическим типом v. Заметим, что такой метод может отличаться от одноименного метода, связанного со статическим типом переменной v. v передается принимающему параметру метода M' по правилам передачи параметров, указанным в 10.1.

Если r — принимающий параметр, описанный как имеющий тип T, то r.M^ означает метод M, связанный с типом-предком типа T (супер-вызов). В упреждающем и фактическом описаниях метода принимающие параметры должны быть соответствующих типов. Списки формальных параметров обоих описаний должны соответствовать (Приложение A), а имена соответствующих параметров должны быть равны.

Методы, помеченные символом " - ", считаются экспортированными только для реализации. Такой метод может быть переопределен в любом импортирующем модуле, но может вызываться только в модуле, содержащем описание метода. (В настоящее время компилятор допускает супер-вызовы методов, экспортированных только для реализации, вне их определяющего модуля. Это допускается временно для облегчения переноса программ.)

Примеры:

```
PROCEDURE (t: Tree) Insert (node: Tree), NEW, EXTENSIBLE;
    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); (* redefinition *)
    BEGIN
        WriteInt(node(CenterTree).width);
        t.Insert^ (node) (* calls the Insert method of Tree *)
    END Insert

    PROCEDURE (obj: Object) Draw (w: Window), NEW, ABSTRACT

    PROCEDURE (obj: Object) Notify (e: Event), NEW, EMPTY
```

10.3 Предопределенные [predeclared] процедуры

Следующая таблица дает список предопределенных процедур. Некоторые процедуры являются обобщенными [generic], т.е. они применимы к разным типам операндов. v представляет переменную, x и y — выражения, а T — тип. Первая подходящая строка дает правильный тип результата.

Процедуры-функции

Имя	Тип аргументов	Тип результата	Функция
ABS(x)	<= INTEGER вещественный тип, LONGINT	INTEGER тип x	абсолютное значение
ASH(x, y)	x: <= INTEGER x: LONGINT y: целый тип	INTEGER LONGINT	арифметический сдвиг (x * 2^y)
BITS(x)	INTEGER	SET	{i | ODD(x DIV 2^i)}
CAP(x)	литерный тип	тип x	x — буква из набора Latin-1: соответствующая большая буква
CHR(x)	целый тип	CHAR	литера с порядковым номером x
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превосходящее x
LEN(v, x)	v: массив; x: целая константа	INTEGER	длина v в измерении x (первое измерение = 0)
LEN(v)	тип массивов String <цепочка>	INTEGER INTEGER	эквивалентно LEN(v, 0) длина цепочки (не считая 0X)
LONG(x)	BYTE SHORTINT INTEGER SHORTREAL SHORTCHAR Shortstring	SHORTINT INTEGER LONGINT REAL CHAR String	тождество
MAX(T)	T = основной тип T = SET	T INTEGER	максимальное значение для типа T максимальный элемент множества
MAX(x, y)	<= INTEGER целый тип	INTEGER LONGINT	большее значение из x и y

	<= SHORTREAL	SHORTREAL	
	числовой тип	REAL	
	SHORTCHAR	SHORTCHAR	
	литерный тип	CHAR	
MIN(T)	T = основной тип	T	минимальное значение для типа T
	T = SET	INTEGER	0
MIN(x, y)	<= INTEGER	INTEGER	меньшее значение из x и y
	целый тип	LONGINT	
	<= SHORTREAL	SHORTREAL	
	числовой тип	REAL	
	SHORTCHAR	SHORTCHAR	
	литерный тип	CHAR	
ODD(x)	целый тип	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	порядковый номер литеры x
	SHORTCHAR	SHORTINT	порядковый номер литеры x
	SET	INTEGER	(SUM i: i IN x: 2^i)
SHORT(x)	LONGINT	INTEGER	тождество
	INTEGER	SHORTINT	тождество
	SHORTINT	BYTE	тождество
	REAL	SHORTREAL	тождество (возможно усечение)
	CHAR	SHORTCHAR	проекция
	String	Shortstring	проекция
SIZE(T)	любой тип	INTEGER	количество байт, требуемое для T

SIZE не может применяться в константных выражениях, т.к. его значение зависит от фактической реализации компилятора.

Простые процедуры

Имя	Типы аргументов	Функция
ASSERT(x)	x: логическое выражение	остановить программу, если не x
ASSERT(x, n)	x: логическое выражение; n: целая константа	остановить программу, если не 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, 0 <=x <= MAX(SET)	v := v - {x}
HALT(n)	целая константа	остановить программу
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, 0 <=x <= MAX(SET)	v := v + {x}
NEW(v)	указатель на запись или	разместить v ^
	фиксированный массив	

NEW(v, x0, ..., xn)	v: указатель на открытый массив; xi: целый тип	разместить v ^ с длинами x0.. xn

В ASSERT(x, n) и HALT(n) интерпретация n определяется конкретной реализацей.

10.4 Финализация

Предопределенный метод с именем FINALIZE связан с каждым типом записей как если бы он был описан как связанный с типом ANYREC:

```
PROCEDURE (a: ANYPTR) FINALIZE-, NEW, EMPTY;
```

Метод FINALIZE может быть реализован для любого указательного типа. Этот метод вызывается в не определенное время после того, как объект этого типа (или соответствующего базового типа) стал недоступен через другие указатели (перестал быть глобально доступным [globally anchored]) и до того, как память, выделенная под объект, утилизуется системой.

Не рекомендуется вновь делать этот объект глобально доступным внутри этого метода, и этот метод не вызывается повторно, если объект снова станет недоступным. Порядок финализации нескольких недоступных объектов не определен.

11. Модули

Модуль это набор описаний констант, типов, переменных и процедур вместе с некоторой операторной последовательностью для присваивания начальных значений переменным <а также операторной последовательностью финализации>. Модуль представляет собой текст, компилируемый как целое.

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

Список импорта указывает имена импортируемых модулей. Если модуль A импортируется модулем M, и A экспортирует идентификатор x, то внутри M на x ссылаются как A.x. Если A импортируется как B := A, то на объект x следует ссылаться как B.x. Благодаря этому возможно использовать краткие псевдонимы для имен модулей в квалифицированных идентификаторах. Модуль не может импортировать сам себя. Идентификаторы, которые нужно экспортировать (т.е. которые должны быть видимы в модулях-клиентах), должны быть надлежащим образом помечены в описаниях (см. главу 4).

Операторная последовательность, следующая за BEGIN, выполняется, когда модуль добавляется к системе (загружается), что выполняется после загрузки импортируемых модулей. Следовательно, циклический импорт модулей невозможен. Отдельные экспортированные процедуры могут быть активизированы системой, и такие процедуры служат командами.

Содержимое переменных, описанных в модуле, очищается перед выполнением тела модуля. Это подразумевает, что все переменные указательных или процедурных типов инициализируются в NIL.

Операторная последовательность, следующая за CLOSE, выполняется, когда модуль удаляется из системы <т.е. выгружается из оперативной памяти>.

Пример:

```
MODULE Trees; (* exports: Tree, Node, Insert, Search, Write, Init *)
    IMPORT StdLog;
    TYPE
        Tree* = POINTER TO Node;
        Node* = RECORD (* exports read-only: Node.name *)
            name-: POINTER TO ARRAY OF CHAR;
            left, right: Tree
        END;
```

```
    PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR), NEW;
        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$) + 1); p.name^ := name$;
        IF name < father.name^ THEN father.left := p ELSE father.right := p END
    END Insert;

    PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree, NEW;
        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*, NEW;
    BEGIN
        IF t.left # NIL THEN t.left.Write END;
        StdLog.String(t.name); StdLog.Ln;
        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
    StdLog.String("Trees loaded"); StdLog.Ln
CLOSE
    StdLog.String("Trees removed"); StdLog.Ln
END Trees.
```

Приложение А: Определения терминов

Литерные типы	SHORTCHAR, CHAR
Целые типы	BYTE, SHORTINT, INTEGER, LONGINT
Вещественные типы	SHORTREAL, REAL
Числовые типы	целые и вещественные типы
Типы цепочек литер	Shortstring, String
Основные типы	BOOLEAN, SET, литерные и числовые типы

Одинаковые типы [Same types]

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 описан в описании типа вида Ta = Tb, или
3. a и b появляются в одном списке идентификаторов в описании переменных, полей записи или формальных параметров.

Эквивалентные типы [Equal types]

Два типа Ta и Tb эквивалентны, если

1. Ta и Tb имеют одинаковый тип, или
2. Ta и Tb суть типы открытых массивов с эквивалентными типами элементов, или
3. Ta и Tb суть процедурные типы, чьи списки формальных параметров соответствуют.
4. Ta и Tb суть указательные типы с эквивалентными базовыми типами.

Соответствие списков формальных параметров [Matching formal parameter lists]

Два списка формальных параметров соответствуют, если

1. они имеют одинаковое число параметров, и
2. они имеют либо эквивалентные типы результатов функции, либо не имеют никаких, и
3. параметры в соответствующих позициях имеют эквивалентные типы, и
4. параметры в соответствующих позициях суть оба либо параметры-значения, либо IN, OUT или VAR параметры.

Включение типов [Type inclusion]

Числовые и литерные типы включают (значения) меньших типов того же класса в соответствии со следующими иерархиями:

```
REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE
CHAR >= SHORTCHAR
```

Типы-потомки и типы-предки [Type extension]

Если дано описание типа Tb = RECORD (Ta) ... END, то Tb является непосредственным потомком типа Ta, а Ta — непосредственным предком типа Tb. Какой-либо тип Tb является потомком [extension] типа Ta (Ta является предком [base type] для Tb), если

1. Ta и Tb суть одинаковые типы, или
2. Tb — непосредственный потомок некоторого потомка типа Ta, или
3. Ta имеет тип ANYREC.

Если Pa = POINTER TO Ta и Pb = POINTER TO Tb, то Pb — потомок типа Pa (Pa — предок для Pb), если Tb является потомком типа Ta.

Совместимость по присваиванию [Assignment compatible]

Выражение e типа Te является совместимым по присваиванию с переменной v типа Tv, если выполняется одно из следующих условий:

1. Te и Tv эквивалентны и не являются ни типами открытых массивов, ни типами записей, допускающими потомков <т.е. описанными с атрибутами ABSTRACT или EXTENSIBLE>;
2. Te и Tv — числовые или литерные типы, и Tv включает Te;
3. Te и Tv — указательные типы, и Te — потомок типа Tv;
4. Tv — указательный или процедурный тип, и e есть NIL;
5. Tv — числовой тип, а e — константное выражение, чье значение содержится в Tv;
6. Tv — массив CHAR, Te — String или Shortstring, и LEN(e) < LEN(v);
7. Tv — массив SHORTCHAR, Te — Shortstring, и LEN(e) < LEN(v);
8. Tv — процедурный тип, а e — имя процедуры, чьи формальные параметры соответствуют параметрам Tv.

Совместимость по массивам [Array compatible]

Фактический параметр a типа Ta является совместимым по массивам с формальным параметром f типа Tf, если

1. Tf и Ta — эквивалентны, или
2. Tf — открытый массив, Ta — произвольный массив, и типы их элементов совместимы по массивам, или
3. Tf — открытый массив CHAR, и Ta имеет тип String, или
4. Tf — открытый массив SHORTCHAR, и Ta имеет тип Shortstring.

Совместимость по параметрам [Parameter compatible]

Фактический параметр a типа Ta является совместимым по параметрам с формальным параметром f типа Tf, если

1. Tf и Ta — эквивалентны, или
2. f — параметр-значение, а Ta — совместим по присваиванию с Tf, или
3. f — IN или VAR параметр, Tf и Ta — типы записей, и Ta — потомок типа Tf.

Совместимость по выражению [Expression compatible]

Для некоторой операции типы ее операндов совместимы по выражению, если они подчиняются правилам, суммированным в следующей таблице. Первая подходящая строка дает правильный тип результата. Тип T1 должен быть расширением типа T0:

операция	первый операнд	второй операнд	тип результата
+ - * DIV MOD	<= INTEGER целый тип	<= INTEGER целый тип	INTEGER LONGINT
/	целый тип	целый тип	REAL
+ - * /	<= SHORTREAL числовой тип	<= SHORTREAL числовой тип	SHORTREAL REAL

	SET	SET	SET
+	Shortstring	Shortstring	Shortstring
	тип цепочек	тип цепочек	String
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	числовой тип	числовой тип	BOOLEAN
	литерный тип	литерный тип	BOOLEAN
	тип цепочек	тип цепочек	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, указательный тип	NIL, указательный тип	
	T0 или T1	T0 или T1	BOOLEAN
	процедурный тип T, NIL	процедурный тип T, NIL	BOOLEAN
IN	целый тип	SET	BOOLEAN
IS	T0	тип T1	BOOLEAN

Константные выражения вычисляются при компиляции с максимальной точностью (LONGINT для целых типов, REAL для вещественных типов) и результат трактуется как буквальный <т.е. заданный непосредственно своим изображением> численный параметр с тем же значением.

Приложение B: Синтаксис Компонентного Паскаля

```
Module = MODULE ident ";" [ImportList] DeclSeq [BEGIN StatementSeq] [CLOSE
StatementSeq] END ident ".".
ImportList = IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".
DeclSeq = { CONST {ConstDecl ";" } | TYPE {TypeDecl ";"} | VAR {VarDecl ";"}}
{ProcDecl ";" | ForwardDecl ";"}.
ConstDecl = IdentDef "=" ConstExpr.
TypeDecl = IdentDef "=" Type.
VarDecl = IdentList ":" Type.
ProcDecl = PROCEDURE [Receiver] IdentDef [FormalPars] MethAttributes
    [";" DeclSeq [BEGIN StatementSeq] END ident].
MethAttributes = ["," NEW] ["," (ABSTRACT | EMPTY | EXTENSIBLE)].
ForwardDecl = PROCEDURE "^" [Receiver] IdentDef [FormalPars] MethAttributes.
FormalPars = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection = [VAR | IN | OUT] ident {"," ident} ":" Type.
Receiver = "(" [VAR | IN] ident ":" ident ")".
Type = Qualident | ARRAY [ConstExpr {"," ConstExpr}] OF Type
    | [ABSTRACT | EXTENSIBLE | LIMITED]
    RECORD ["("Qualident")"] FieldList {";" FieldList} END
    | POINTER TO Type
    | PROCEDURE [FormalPars].
FieldList = [IdentList ":" Type].
StatementSeq = Statement {";" Statement}.
Statement = [ Designator ":=" Expr | Designator ["(" [ExprList] ")"]
    | IF Expr THEN StatementSeq
      {ELSIF Expr THEN StatementSeq}
      [ELSE StatementSeq] END
    | CASE Expr OF Case {"|" Case}
      [ELSE StatementSeq] END
    | WHILE Expr DO StatementSeq END
    | REPEAT StatementSeq UNTIL Expr
    | FOR ident ":=" Expr TO Expr [BY ConstExpr] DO StatementSeq END
    | LOOP StatementSeq END
    | WITH [ Guard DO StatementSeq ]
      {"|" [ Guard DO StatementSeq ] }
      [ELSE StatementSeq] END
```

```
    | EXIT
    | RETURN [Expr]
].
Case = [CaseLabels {"," CaseLabels} ":" StatementSeq].
CaseLabels = ConstExpr [".." ConstExpr].
Guard = Qualident ":" Qualident.
ConstExpr = Expr.
Expr = SimpleExpr [Relation SimpleExpr].
SimpleExpr = ["+" | "-"] Term {AddOp Term}.
Term = Factor {MulOp Factor}.
Factor = Designator | number | character | string | NIL | Set | "(" Expr ")" | "
~ " Factor.
Set = "{" [Element {"," Element}] "}".
Element = Expr [".." Expr].
Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp = "+" | "-" | OR.
MulOp = " * " | "/" | DIV | MOD | "&".
Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " | "(" Qualident ")"
| "(" [ExprList] ")"} [ "$" ].
ExprList = Expr {"," Expr}.
IdentList = IdentDef {"," IdentDef}.
Qualident = [ident "."] ident.
IdentDef = ident [" * " | "-"].
```

Приложение C: Диапазоны значений [domains] основных типов

Тип	Диапазон значений
BOOLEAN	FALSE, TRUE
SHORTCHAR	0X .. 0FFX
CHAR	0X .. 0FFFFX
BYTE	-128 .. 127
SHORTINT	-32768 .. 32767
INTEGER	-2147483648 .. 2147483647
LONGINT	-9223372036854775808 .. 9223372036854775807
SHORTREAL	-3.4E38 .. 3.4E38, INF (32-битный формат IEEE)
REAL	-1.8E308 .. 1.8E308, INF (64-битный формат IEEE)
SET	подмножество из 0 .. 31

Приложение D: Обязательные требования к среде выполнения

Определение Компонентного Паскаля опирается на три фундаментальных предположения.

1) Во время исполнения программ доступна информация, позволяющая проверять динамический тип объекта. Это нужно для реализации проверок типов и охраны типов.

2) Отсутствует процедура DISPOSE <освобождение памяти под более не используемые объекты>. Память <занимаемая объектом> не может быть освобождена по явной инструкции программиста, поскольку это создало бы проблемы безопасности, связанные с потерей памяти [memory leaks] <память, занимаемая объектами, недоступными ни через какие указатели, которую программист забыл освободить> и с висячими указателями [dangling

pointers] <доступные указатели, продолжающие указывать на области памяти, по ошибке преждевременно освобожденные>. За исключением таких встроенных систем, где не используется динамическое управление памятью, или где ее можно разместить только однажды и никогда не нужно освобождать, требуется автоматический сбор мусора.

3) Модули и по крайней мере их экспортированные процедуры (команды) и экспортированные типы должны быть доступны динамически. Если необходимо, это может вызывать загрузку модулей. Программный интерфейс, используемый для загрузки модулей или для доступа к указанной мета-информации, не определяется языком, но компилятор должен сохранять эту информацию при генерации кода. За исключением полностью слинкованных приложений, в которых при исполнения не нужно загружать никакие модули, для модулей требуется динамический загрузчик [linking loader]. Встроенные системы являются важными примерами приложений, которые могут быть полностью слинкованы.

Реализация, которая не удовлетворяет этим требованиям к компилятору и среде выполнения, не считается удовлетворяющей определению Компонентного Паскаля.