Алгоритмические языки и программирование

КОНСПЕКТ ЛЕКЦИЙ

электронная версия от 5.10

Предисловие*

Предметы АЯ и П

АЯ — изучение конкретных ЯП, сравнительный анализ ЯП, разбиение ЯП на классы по различным признакам и т. п. П — умение создавать программные продукты, использую имеющиеся средства программирования, в частности, ЯП.

Определение понятия алгоритм

Алгоритм — это конечный набор правил, позволяющий чисто механически решать любую конкретную задачу из некоторого класса однотипных задач.

Основные свойства алгоритма: 1) массовость — исходные данные могут изменяться в определенных пределах; 2) детерминированность — процесс применения правил к исходным данным определен однозначно.

Это не строгое определение. Точное определение этого понятия дается в курсе “Математическая логика”.

Способы записи алгоритма

Блок-схема — логическая схема программы. Элементы блок-схемы: ромб — условный переход, прямоугольник — блок последовательных операторов, овал — начало и конец, стрелки — переходы. Иногда некоторые элементы могут иметь другой вид.

Пример программы “Утро у моря” (рис. 1): “Умыться. Если хорошая погода, то идти купаться в море, иначе читать журнал. Позавтракать.”

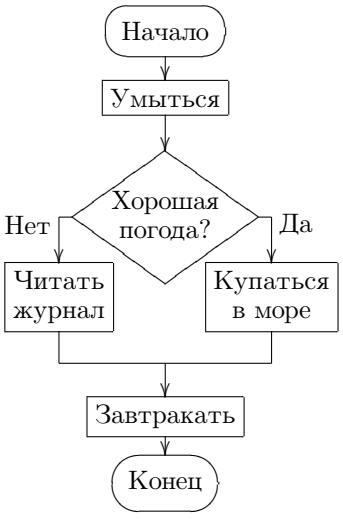


Рис. 1

Пример программы “Пятиразрядный счетчик” (рис. 2): “Шаг 1. Присвоить счетчику значение 0. Шаг 2. Отобразить значение счетчика. Шаг 3. Если значение счетчика меньше 99999, то увеличить счетчик на единицу, иначе перейти к шагу 1. Шаг 4. Перейти к шагу 2.”

Рис. 2

* Для подготовки материалов использовались системы Plain TEX, XY-pic и AMSFonts

АЯ (формальный язык для записи алгоритмов) и ЯП (формальный язык для записи программ) — часто синонимы, но иногда ЯП трактуется более широко, чем АЯ, т.к. в некоторых ЯП, например, в прологе, собственно ход вычисления (алгоритм) скрыт.

Уровни ЯП

ЯП низкого уровня характеризуются тем, что они являются зависимыми от центрального процессора вычислительной машины, т.е. использовать их с разными процессорами невозможно.

ЯП низкого уровня делятся в свою очередь на три подуровня:

1) машинный код или последовательность чисел. Программу на этом "ЯП" компьютер может исполнить непосредственно;
2) мнемокод. Он позволяет программисту использовать названия команд (мнемоники), что позволяет наглядно отделять как команды друг от друга, так и названия операций от их операндов. Требуется очень простая и маленькая программа-интерпретатор, например, MS-DOS debug, для перевода команды мнемокода в машинный код и наоборот (эти процессы называются соответственно ассемблированием и дизассемблированием);
3) ассемблер. Это уже настоящий ЯП. Для того, чтобы перевести программу, написанную на языке ассемблера, в машинный код требуется компилятор с языка ассемблер заданного процессора. Сам такой компилятор также называется ассемблером. Современные ассемблеры — это весьма сложные и большие по объему программы, сравнимые по возможностям с трансляторами с языков высокого уровня.

ЯП высокого уровня характеризуются своей независимостью от аппаратуры. Рассмотрим наиболее известные из языков высокого уровня.

Фортран (FORTRAN — Formula Translation, 1954, IBM, 1-й ЯП).

Бэйсик (BASIC — Beginner's All-purpose Symbolic Instruction Code — многоцелевой символический ЯП для начинающих, Дортмутский колледж, США, 1964).

Кобол (COBOL — COmmon Business Oriented Language — ориентированный на решение экономических задач язык, Пенсильванский университет, 1959, самый используемый ЯП с середины 60-х до конца 70-х).

Алгол-60/68 (ALGOL — Algoritmic Language, международный, математики),

ПЛ/1 (PL/1 — Programming Language 1, прототип 1964, IBM).

Си (C, начало 70-х, AT&T, UNIX, Брайн Керниган, Деннис Ритчи, США).

Паскаль (Pascal, 1971, Блез Паскаль — построил первую механическую считающую машину, Никлаус Вирт: "Программа = алгоритм + структуры данных").

Ада (Ada, 1980, Ада Лавлайс — первый в мире программист, конкурс МО США).

Эти языки высокого уровня классической структуры. Они являются прямыми потомками языков низкого уровня.

Для перечисляемых далее языков существует семантика, существенно отличающаяся от языков низкого уровня. Иногда отдельные из них называют языками сверхвысокого уровня.

Лисп (LISP — List Processing, 1958, Массачусетский технологический институт, США, 2-й ЯП).

Пролог (PROLOG — Programming in Logic, 1973, Франция).

SQL (Structured Query Language — язык структурных запросов, IBM, 1974).

Оккам (Occam, язык программирования параллельных процессов, многопроцессорных компьютеров, назван в честь английского философа XIV века Уильяма Оккама, разработан при участии Хоара, 1983).

Форт (Forth, начало 70-х, США, Чарльз Мур создал для себя и своего радиотелескопа, стековый язык).

Постскрипт (Postscript, 1982, Adobe Systems Inc., аппаратно-независимый стековый язык описания страниц, используется для управления принтерами, дисплеями и т.п., похож на Форт, программы на нем могут автоматически создаваться драйверами устройств печати).

ООЯП (Си++ (C++), прототип с 1980, синтез си и ЯП Симула-67, AT&T, Бьярне Страуструп, США, ныне стандартный индустриальный ЯП; паскаль с объектами (Borland Turbo Pascal 5.5), 1989).

ПОЯ — предназначены для решения задач в узкой области. Например, язык для инженерных расчетов APL (A Programming Language).

Перл (Perl — Practical Extraction and Report Language, 1988, Larry Wall, интерпретирующий язык, соединивший в себе средства популярных в среде Unix языков си, sed, awk и некоторых других; популярен в Internet для создания CGI-скриптов, т.е. программ выполняющихся на компьютере, содержащем Web-страницу);

Tcl/Tk (Tool Command Language/Toolkit, John Osterhout, Sun Microsystems, конец 80-х — начало 90-х, простой интерпретирующий язык с мощной поддержкой аппарата обработки сообщений, простейший язык для создания высококачественных графических приложений в оконных интерфейсах, неудобен для написания больших программ);

Ява (Java, Sun Microsystem, начало 90-х, подмножество C++, транслируется в независимый от процессора промежуточный код, выполняющийся на виртуальной ява-машине);

Яваскрипт (JavaScript, Netscape, середина 90-х, язык для включения кода в интернет-страницы);

Питон (Python, Штихтингский математический центр, Нидерланды, 1990, интерпретирующийся язык, становящийся все более популярным, по возможностям сопоставим как с Перл, так и с Tcl/Tk).

Рубин (Ruby, 1993, Matz, наиболее объектно-ориентированный язык в своем классе).

ПХП (PHP, Personal Home Page, 2-я половина 1990-х, разработан специально для Internet).

Хаскелл (Haskell, 1990, "чистый" функциональный язык, назван в честь математика XX века).

Бэйсик, паскаль и си++ — сравнительная характеристика

Бэйсик, си++ и паскаль — широко используемые ЯП.

На бэйсик и паскаль есть стандарты Международной организации по стандартизации, ISO (International Standards Organization), не отражающие современный уровень этих языков. На си в 1999, 2011 установлены стандарт ISO. На си++ стандарт ISO установлены в 1998, 2003, 2011, 2014 годах.

Бэйсик: (+) простота основ, идеален для маленьких программ, наиболее широко используется; (−) нет возможностей для создания структур данных, громоздкость синтаксиса, развивает неаккуратность у программиста.

Бэйсик в индустрии программирования используется для написания больших программ лишь в единичных случаях. Практически невозможно найти программиста-профессионала в совершенстве владеющего бэйсиком, си или паскалем, который бы для реализации сложного проекта предпочел бы бэйсик. Однако из-за своей простоты для новичков, бэйсик ныне стал основным языком программирования систем делопроизводства (компоненты системы Microsoft Office, например, Word и Excel). [Дейкстра: “Практически невозможно научить хорошему стилю программирования студентов, уже знакомых с бэйсиком: как потенциальные программисты они умственно изуродованы без надежды на выздоровление”.]

Си++: (+) индустриальный стандарт, имеет самые эффективные компиляторы среди прочих ЯП высокого уровня; (−) сложен для изучения.

Паскаль: (+) прост в целом, развивает аккуратность, практически все понятия имеют аналоги в си++; (−) реже используется для решения производственных задач.

Паскаль был разработан как учебный язык. В начале 80-х паскаль стал весьма популярной мобильной системой программирования, существовавшей практически для всех типов компьютеров (системы P-кода), но из-за некоторых своих недостатков эта система вскоре была вытеснена другими. В 1983 году фирмой Borland был создан Turbo Pascal для ОС CP/M и MS-DOS. Эта программа и последующие ее версии благодаря их уникальной дружественности к программисту стали популярнейшими средствами для разработки различного ПО. С 1987 года паскаль фирмы Borland поддерживает концепции ООП. Современный этап в развитии этих трансляторов — это система Delphi.

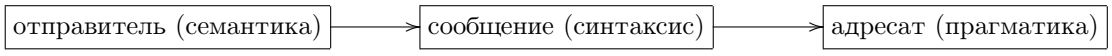
Элементы общей теории языков программирования

Примеры неудачного определения ЯП:

1) Перечисление всех ЯП — это определение не дает предсказывать новые частные случаи;
2) “Инструмент для планирования поведения исполнителя” — в этом определении все требует уточнения. Например, является ли языком программирования должностная инструкция? Кроме того, в этом определении не отражены ряд аспектов использования ЯП, например, ЯП служат еще и для обмена программами между людьми.

Наиболее удачным определением ЯП является: “ЯП — это средство для достижения взаимопонимания людей с компьютерами и между людьми по поводу управления компьютерами”.

Добиться взаимопонимания бывает очень сложно. Чтобы выделить возникающие здесь проблемы, рассмотрим следующую модель передачи сообщения.



Отправитель — это автор или генератор сообщения, адресат — это получатель, читатель или слушатель сообщения, сообщение — это текст, последовательность звуков и т. п. В скобках приведены названия наук, занимающихся правилами построения допустимых сообщений (синтаксис), правилами сопоставления таким сообщениям смысла (семантика) и правилами, регулирующими использование сообщений (прагматика).

Недоразумения могут возникнуть на любом этапе передачи сообщения. Например, недоразумение, связанное с сообщением: “Казнить нельзя помиловать” — это синтаксическое недоразумение. Недоразумение, связанное с фразой из поваренной книги: “Добавить немного перца” — это семантическое недоразумение (неясен смысл, приданный автором слову “немного”). И наконец, недоразумение, возникающее тогда, когда у автора и получателя существенно различаются представление об окружающем мире или решаемой задаче, — это прагматическое недоразумение. Пример такого недоразумения — это лекция по новой теме, когда некоторые студенты, нацеленные на восприятие информации в рамках своего текущего кругозора, воспринимают качественно новую информацию как ненужную. ISO разработан документ, содержащий, в частности, классификацию стандартных дефектов ЯП, отражающий приведенные в лекции построения.

Основные понятия ЯП

Текст программы — последовательность строк, состоящих из символов. Символы делятся на незначимые и значимые. Последние образуют алфавит языка. Они же делятся на три группы: буквы, цифры и специальные знаки. В паскале как и в бэйсике буквы — это латинские заглавные и строчные буквы. Знак подчеркивания считается буквой. В паскале и в бэйсике заглавные и строчные буквы не различаются с точки зрения значимости. Цифры — это десять цифр от 0 до 9 [в большинстве диалектов бэйсика знак “точка” приравнивается к цифре]. Специальные символы — это, например, знаки “двоеточие”, “плюс”, “косая черта” и т. п. Незначимые символы для паскаля (или бэйсика) — это, например, буквы кириллицы.

Лексема — минимальная единица языка, имеющая смысл (символ не означает ничего, кроме себя самого).

Типы лексем:

1) Служебные или зарезервированные слова (например, бэйсика — PRINT, IF, GOTO, STOP, REM; паскаля — begin, if, record, goto, in; си — if, goto, for, int, do);
2) Идентификаторы (имена) — вводятся для именования различных объектов программы (переменных, типов, процедур, функций и т. п.), они должны начинаться с буквы, состоять только из букв и цифр и не быть служебными словами;
3) Литералы — обозначают сами себя: числа, символы, строки и т. п. Примеры:

	Бэйсик	Паскаль	Си
в.числа	.01	0.01	0.01
ц.числа	10	10	10
символа	"A"	'A'	'A'
строки	"ABC"	'ABC'	"ABC";

4) Знаки операций — формируются из одного или нескольких специальных знаков (например, лексемы деления "/", сложения "+", ">=" и т. п.);
5) Разделители — бывают двух видов:
   I) необходимые в синтаксических конструкциях данного ЯП, например, знак ";" в паскале или си [в бэйсике подобных разделителей нет];
   II) используемые для разделения лексем — это комментарии, пробелы, знаки табуляции и концы строк. Лексемы, соединение которых даст новую лексему, необходимо отделять друг от друга, например, необходимо отделять идентификатор от числа, другого идентификатора или служебного слова. Прочие лексемы отделять друг от друга необязательно, но иногда полезно для придания большей наглядности программе.

Типы данных и переменные

Программа — это совокупность описаний данных (структура данных) и операторов (алгоритм). В паскале блок описаний данных обязателен и отделен от блока операторов. [В бэйсике программист может вообще не описывать данные.]

Описание данных — это прежде всего описание типов данных. Тип данных, во-первых, задает ограничения на диапазон значений объектов своего типа, а, во-вторых, определяет набор допустимых операций с объектами своего типа. Кроме того, каждый тип ассоциируется с каким-либо общим понятием.

Пример, тип "целое число". Из-за ограниченности памяти, которую можно отвести для хранения данных, представляющих целое число, данные этого типа будут иметь ограничения на диапазон возможных значений. Имеет место очевидная зависимость, чем больше памяти будет отводиться для хранения целого числа, тем большим будет диапазон его возможный значений. Поэтому для представления целых чисел в большинстве ЯП имеется несколько типов. Причем для всех этих типов определен один и тот же набор операций: сложение, умножение, вычитание и т. п.

Набор операций, применимых к тому или иному типу, очевидно зависит от понятия, ассоциируемого с ним. Например, к строкам символов можно применять операцию сложения, но нельзя — вычитания, числа можно умножать, складывать и т. п.

Основная идея в концепции типов данных — это заставить компьютер работать с сущностями, максимально приближенными к тем, которые есть в реальности. Так, например, тип "целое число" является простейшим, им можно описывать многие реальные количественные отношения. Строковый тип позволяет работать с реальной текстовой информацией. Тип массив — с информацией, заданной в виде таблиц (матриц). Объектный тип позволяет адекватно отобразить в машинной форме практически любую сущность. Например, возможен объектный тип "самолет", для которого заданы операции "взлет", "посадка", "заправка", "техобслуживание" и т. п., т. е. к такому типу применимы очень сложные операции, аналога которых на машинном, родном языке компьютера в принципе быть не может. На машинном языке можно работать лишь с двоичными числами, применяя к ним операции, по сложности не превосходящие арифметические.

Иногда описание типа, как, например, в Бэйсике, может происходить неявно, согласно виду идентификатора переменной. Практика показала, что такой подход приводит к большему числу ошибок при создании программы.

Переменная — это именованный экземпляр объекта выбранного типа или конкретная реализация общего понятия, связанного с этим типом. Переменная кроме имени характеризуется адресом и размером области памяти компьютера, которую она занимает.

Некоторые идентификаторы в большинстве ЯП резервируются для обозначения стандартных величин и операций. Их можно переопределять и использовать в другом качестве, но тогда теряется их стандартное назначение. Можно, например, переопределить имена для процедур вывода на экран, используя их как имена переменных, но как тогда после этого выводить результаты расчетов?!

В современном программировании работа с типами данных имеет первостепенное значение.

[Далее будет изучаться язык паскаль с объектами (Borland Pascal 7.0, Free Pascal), иногда, с эквивалентными примерами на бэйсике. Настоятельно рекомендуется все примеры программ, приводимых на лекциях, самостоятельно набрать и выполнить на компьютере.]

Разделители

Программа состоит из операторов и деклараций, которые должны быть отделены друг от друга знаком ";".

Комментарии должны быть заключены между парными знаками фигурных скобок или знаками круглая скобка и звездочка. Не допускается вложенность однотипных и пересечения разнотипных комментариев.

Комментарии, пробелы, знаки табуляции и концы строк в любом количестве можно ставить между двумя любыми лексемами.

Пример. В конструкции [верной и для Бэйсика]

```
i f a > = b b * 6 . 1 then goto 10
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
0 1 2 3 4 5 6 7 8 9 10
```

разделитель необходим в позициях 1 и 10. В позициях 0, 3, 5, 8 и 9 разделители недопустимы — они разорвут лексемы. В прочих позициях разделители опциональны. Таким образом, правильны следующие две конструкции:

1) `if(*условие*)a>=bb*6.1 then(*действие*)goto 10`
2)
```
if a >= bb*6.1 then
    goto(*метка*)10.
```

Иерархия типов

I. Скалярные: дискретные (базовые (целые, перечислимые, логические, символьные) и ограниченные) и вещественные;
II. Составные: регулярные, строки, множества, комбинированные, файловые;
III. Указатели;
IV. Объектные (связаны с комбинированным типом).

[В Бэйсике строгое использование концепции типов — это опциональное, искусственно надстроенное расширение.]

Ко всем типам данных применимы только две операции:

@ — получение адреса: если v — имя переменной, то @v — ее адрес

sizeof(u) — стандартная функция получения размера области памяти для объектов заданного типа: если u — имя переменной некоторого типа или тип, то sizeof(u) — это размер памяти в байтах, необходимой для хранения величины заданного типа.

Ко всем скалярным типам применимы операции сравнения: `=, <>, <, >, <=, >=`. К данным типа указателей тоже можно применять операции сравнения, но только две: `=` и `<>`. Нет операций общих для данных составного или объектного типа.

Дискретный тип характеризуется тем, что все его значения можно занумеровать. Ко всем дискретным типам применимы следующие операции (стандартные функции):

pred(d) (pred — это сокращение от слова predecessor — предшественник) — получить предшествующее значение, например, pred(5)=4, а pred('B')='A', применение этой функции к минимальной величине заданного типа даст неопределенный результат;

succ(d) (succ — это сокращение от слова successor — преемник) — получить следующее значение, например, succ(5)=6, а succ('B')='C', применение этой функции к максимальной величине заданного типа даст неопределенный результат;

low(d) — возвращает значение минимальной величины заданного дискретного типа, в качестве аргумента может выступать тип или переменная;

high(d) — возвращает значение максимальной величины заданного дискретного типа, в качестве аргумента может выступать тип или переменная;

ord(d) (ord — это сокращение от слова order — порядок) — возвращает порядковый номер величины-аргумента в соответствующем ей дискретном типе.

Кроме того, к переменным дискретного типа можно применять следующие процедуры:

inc(d) (inc — это сокращение от слова increment — увеличение) — увеличение значения, например, если v1=1, а v2='C', то после inc(v1) и inc(v2) v1=2 и v2='D';

dec(d) (dec — это сокращение от слова decrement — уменьшение) — уменьшение значения, например, если v1=1, а v2='C', то после dec(v1) и dec(v2) v1=0 и v2='B'.

Целые типы

	идентификатор типа	размер, байт(n)		диапазон значений	Бэйсик
беззнаковые	byte	1	$0 \dots 2^{8n}-1 =$	$0 \dots 255$	
	word	2		$0 \dots 65535$	
	longword	4		$0 \dots 4'294'967'295$	
	qword	8		$0 \dots 18'446'744'073'709'551'615$	
знаковые	shortint	1	$-2^{8n-1} \dots 2^{8n-1}-1 =$	$-128 \dots 127$	
	integer	2		$-32768 \dots 32767$	INTEGER(%)
	longint	4		$-2147483648 \dots 2147483647$	LONG(&)

Следовательно, sizeof(byte)=1, sizeof(longint)=4, high(byte)=255, low(integer)=-32768, ord(1)=1.

К данным целого типа применимы следующие операции:

+ — сложение, например, 1+3 будет 4;

- — вычитание, например, 3-1 будет 2;
* — умножение, например, 2*2 будет 4;
div (division) — деление, например, 6 div 2 будет 3, а 7 div 3 будет 2, т.е. остаток отбрасывается;
mod (modulo) — остаток от деления, например, 6 mod 2 будет 0, а 7 mod 3 будет 1;
shr (shr — это сокращение от Shift Right — сдвиг вправо) — сдвиг двоичного представления числа заданное число раз вправо, например, 1 shr 2 будет 0, а 61 shr 1 будет 30 ($61_{10} = 111101_2$, 111101_2 shr $1 = 11110_2 = 30_{10}$), кроме того, для неотрицательных чисел можно пользоваться формулой x shr n = x div 2^n;
shl (shl — это сокращение от Shift Left — сдвиг влево) — сдвиг двоичного представления числа заданное число раз влево, например, 1 shl 2 будет 4, а 3 shl 2 будет 12 ($3_{10} = 11_2$, 11_2 shl $2 = 1100_2 = 12_{10}$), кроме того, для беззнаковых типов можно пользоваться следующей формулой x shl n = x*2^n mod $2^{\text{sizeof(x)}*8}$;
and — поразрядное логическое "И", например, 61 and 35 = 33, т.к.

$$\begin{array}{rcl} 61_{10} & = & 111101_2 \\ 35_{10} & = & 100011_2 \\ & \text{and} & \text{———} \\ & & 100001_2 \quad = \quad 33_{10}; \end{array}$$

or — поразрядное логическое "ИЛИ", например, 61 or 35 = 63, т.к.

$$\begin{array}{rcl} 61_{10} & = & 111101_2 \\ 35_{10} & = & 100011_2 \\ & \text{or} & \text{———} \\ & & 111111_2 \quad = \quad 63_{10}; \end{array}$$

xor (eXclusive OR) — поразрядное логическое "исключающее ИЛИ", например, 61 xor 35 = 30, т.к.

$$\begin{array}{rcl} 61_{10} & = & 111101_2 \\ 35_{10} & = & 100011_2 \\ & \text{xor} & \text{———} \\ & & 011110_2 \quad = \quad 30_{10}; \end{array}$$

not — поразрядное логическое отрицание, результат этой операции зависит от того, к какому типу данных она применяется: данные могут иметь длину только 8, 16 или 32 бита, — например, not 61 = 194, но если v переменная типа word со значением 61, то not v = 65474, т.к.

$$\begin{array}{rcl} 61_{10} & = & 00111101_2 \\ & \text{not} & \text{————————} \\ & & 11000010_2 \quad = \quad 194_{10} \end{array} \qquad \begin{array}{rcl} 61_{10} & = & 0000000000111101_2 \\ & \text{not} & \text{————————————————} \\ & & 1111111111000010_2 \quad = \quad 65474_{10}. \end{array}$$

Стандартные функции:
sqr(i) (sqr — сокращение от слова square — квадрат) — возведение в квадрат, например, sqr(2)=4 и sqr(4)=16;
abs(i) (absolute value) — модуль числа, например, abs(-1)=1, abs(2)=2, abs(0)=0;
odd(i) (нечетность) — возвращает логическое значение "истина" (true), если аргумент нечетен, и "ложь" (false), если — четен, например, odd(11)=odd(3)=true, odd(2)=odd(224)=false;
random(i) (случайный) — выбирает случайное число из диапазона от 0 до уменьшенного на единицу значения аргумента (аргумент должен быть типа word), например, random(5) может оказаться равным любому целому числу от 0 до 4;
chr(i) (character — символ) — эту функцию будем рассматривать при рассмотрении символьного типа.

Литералы целого типа могут быть 10-ми и 16-ми: 16-е числа должны начинаться с $, например, 100=$64, $1f=31, 46=$2E.

Перечислимые типы

Они позволяют реализовать абстракцию данных. Например, задано, что переменная BaseDirections может принимать только четыре значения: North, South, East, West. Можно описать необходимый для такой переменной тип, явно перечисляя все ее возможные значения. Такое перечисление должно быть в круглых скобках, элементы перечисления должны быть отделены друг от друга запятыми: (North, South, East, West). В качестве элементов этого перечисления можно использовать только идентификаторы! Для описанного ранее типа верно, что ord(South)=1 и pred(East)=South.

Стандартных операций только для перечислимого типа нет.

Перечислимый тип — это мощное средство для создания типов, соответствующих реальным понятиям.

[В Бэйсике нет понятия перечислимого типа.]

Логический тип

Можно рассматривать как стандартный перечислимый тип: (false, true). Таким образом, переменные этого типа могут принимать лишь два означенных значения. Значению false соответствует, очевидно, "ложь", а true

— “истина”. Главное отличие логического типа от перечислимого в том, что к нему можно применять ряд операций типа “И”, “ИЛИ” и других. Логический тип описывается идентификатором boolean.

К любым данным логического типа применимы следующие операции:

and — логическое “И”;

or — логическое “ИЛИ”;

xor — логическое “исключающее ИЛИ”;

not — отрицание.

Литералы логического типа — это только величины true и false.

[В Бэйсике нет отдельного логического типа, но нулевое значение имеет также логическое значение “ложь”, а любое ненулевое — “истина”.]

Символьный тип

Величина этого типа может иметь значение любого символа: цифры, буквы, знака препинания, невидимого управляющего символа (например, символа табуляции или перехода на новую строку) и т. п. Символьный тип описывается идентификатором char (character — символ), sizeof(char)=1.

Литералы-символы задаются двумя способами. Заключением в знаки одинарных кавычек (апострофов) требуемого символа, сам знак-кавычка в литерале должен быть записан дважды внутри кавычек, например, ’A’, ’1’, ’’’’, ’*’. Этот способ позволяет записать только те символы, которые можно увидеть на экране дисплея. Второй способ более общий, но менее удобный. Он основан на том, что каждый символ имеет свой номер в дискретном типе (от 0 до 255): нужно перед номером заданного символа ставить знак #. Примеры: #65=#$41=’A’, #49=#$31=’1’, #13=#$0d — управляющий знак возврата каретки (CR — Carriage Return).

Специальных операций для символьного типа данных нет, но с символьным типом часто используются функции ord и chr. Функция chr является обратной ord относительно символьного типа: по аргументу, целому числу, она возвращает символ с номером, равным этому аргументу. Например, верно, что chr(68)=’D’ и chr(50)=’2’.

[В Бэйсике символьный и строковый типы (STRING($)) неразличимы.]

Ограниченные типы

Получаются из других дискретных наложением дополнительных ограничений на диапазон значений объектов этого типа. Наборы операций переходят от базового к ограниченному типу без изменений.

Примеры ограниченных типов:

1..100 (* ограничения на тип byte *)

-1..70000 (* ограничения на тип longint *)

’a’..’z’ (* ограничения на тип char, переменные этого типа могут принимать значения только строчных латинских букв *)

South..West (* ограничение на перечислимый тип, определенный ранее *)

Ограничения реализуются лишь на уровне дополнительного контроля, который полезен на стадии отладки, но в готовом продукте, как правило, отключается, т. к. этот контроль приводит к существенному замедлению работы программы. Использование ограниченных типов не приводит к экономии памяти.

[В Бэйсике нет ограниченных типов]

Вещественные типы

	идентификатор типа	размер, байт	диапазон порядка	точность, 10-х цифр	Бэйсик
основной	real	6	-39..38	11-12	
дополнительные	single	4	-45..36	7-8	SINGLE(!)
(сопроцессорные)	double	8	-324..308	15-16	DOUBLE(#)
	extended	10	-4951..4932	19-20	
	comp	8	-63..63	19-20	

Литералы вещественного типа задаются мантиссой и, если необходимо, порядком. Порядок (целое число) нужно указывать после буквы e или E. Если в записи числа отсутствует порядок, то необходимо ставить 10-ю точку в мантиссе. Например, 1.5, 10.18e22, -4., 5.001e-17, 87.45E+12.

К любым данным вещественного типа применимы следующие операции:

+ — сложение;

- — вычитание;

/ — деление;

* — умножение;

trunc(r) (trunc — сокращение от слова truncate — урезать) — возвращает целое число, равное целой части аргумента, например, trunc(4.2)=4, trunc(5.7)=5;

round(r) (округлять) — возвращает целое число, ближайшее к аргументу, например, round(4.2)=4, round(5.7)=6, round(11.5)=12, кроме того, верно, что round(x)=trunc(x+0.5);

abs(r) — модуль числа, например, abs(-1.0)=1.0, abs(2.1)=2.1;

sqr(r) — возведение в квадрат, например sqr(2.0)=4.0 и sqr(2.5)=6.25;

sqrt(r) (sqrt — сокращение от слов square root — квадратный корень) — извлечение квадратного корня, например sqrt(4.0)=2.0 и sqrt(0.16)=0.4;
int(r) — возвращает целую часть числа, не меняя тип, например, int(4.2)=4.0, int(5.7)=5.0;
frac(r) (frac — это сокращение от слова fraction — дробь) — возвращает дробную часть числа, например, frac(4.2)=0.2, frac(5.7)=0.7;
sin(r) — синус;
cos(r) — косинус;
arctan(r) — арктангенс;
ln(r) — натуральный логарифм;
exp(r) — экспоненциальная функция;
random — выбирает случайное число большее или равное 0 и меньшее 1, например, верно, что random(n)= trunc(n*random).

Составные типы

Они составлены композицией элементов других типов.

Регулярные типы (массивы)

Они называются так потому, что составлены из элементов других типов регулярным образом. Для выбранного типа можно построить регулярный тип, считая последний занумерованным набором из элементов исходного типа. Регулярный тип описывается при помощи служебного слова array.

Примеры:

array[0..10] of array[0..10] of char (* массив из 11 элементов типа массив из 11 символов - матрица 11x11 *)
array[0..10,0..10]of char (* это описание того же самого типа *)
array[0..500]of integer (* массив из 501 целого числа*)

Диапазоны в квадратных скобках задают возможные значения индексов элементов массива. Количество индексов — это размерность массива. В первых двух примерах массивы двумерные, а в третьем — одномерный.

Следующие две функции предназначены для работы с данными типа массив:
low(a) — возвращает наименьшее значение первого индекса массива a (a — это тип или переменная);
high(a) — возвращает наибольшее значение первого индекса массива a.

[Аналогичные типы можно использовать и в Бэйсике]

Строки

Строки — это (упакованные) массивы символов. Строковый тип соответствует регулярному типу array[0..n]of char, где n — максимальная длина строки (n<256) возможная для данного типа. Элемент такого массива с индексом 0 хранит длину строки, а сама строка хранится, начиная с индекса 1. Строковый тип описывается служебным словом string, например, тип string[n] соответствует приведенному ранее. Для строкового типа существует множество стандартных операций. Последнее отличает строки от типа одномерный массив символов.

Литералы-строки — это просто идущие подряд литералы-символы, но смежные кавычки нужно удалять. Пример: 'AC'#15#12'A''B'='A'+'C'+#15+#12+'A'+''''+'B'.

К любым данным строкового типа применимы операции сравнения: =, <, >, <=, >=, <>. Строки сравниваются так: сначала происходит посимвольное сравнение, а затем сравниваются длины. Например, верно, что 'ABC' < 'XY' и 'AB' < 'ABC'. Такое сравнение называют словарным или лексикографическим.

К строкам применимы следующие стандартные операции:
+ — склейка строк, например, 'ABC'+'CDE'='ABCDE';
concat(s1,..,sn) (concatenate — сцеплять) — другой способ склейки строк: concat(s1,..,sn) возвращает s1+...+sn;
length(s) — возвращает длину строки s, например, length('ABCDEF')=6;
pos(s1,s2) (position — позиция) — поиск строки s1 в строке s2: если поиск успешен, то эта функция возвращает позицию s1 в s2, если же поиск неуспешен, то функция возвращает 0, — например, pos('AB','RABABR')=2, pos('A','BC')=0;
copy(s,p,l) — выделение подстроки длины l с позиции p строки s, например, значением функции copy('UVWXYZ',2,4) будет строка 'VWXY';
low(s) — для любой строки или строкового типа s возвращает 0;
high(s) — для строки или строкового типа s возвращает максимальную возможную длину для строк данного типа.

Следующие стандартные процедуры предназначены для работы со строками:
insert(s1,s2,p) — вставка строки s1 в строку s2 с позиции p, например, если s='ABC', то после insert('YZ',s,2) s будет равно 'AYZBC';
delete(s,p,l) — удаление подстроки длины l с позиции p из строки s, например, если s='ABCDE', то после delete(s,3,2) s будет равна 'ABE';
val(s,n,p) (value — значение) — преобразование строки s в число n, например, val('123',n,p) установит n=123. Если s соответствует число, то p=0, иначе p установится равным позиции, с которой преобразование строки в число стало невозможным, например, вызов val('12xy',n,p) установит p=3. Параметр n может быть любого числового типа, а параметр p должен быть типа integer;

str(i[:w[:d]],s) (string — строка) — преобразование числа i (любого числового типа) в строку s. Параметры w и d служат для форматирования: w — это количество позиций для числа, а d — количество цифр после десятичной точки, например, после вызова str(10,s) установится s='10', после str(10:4,s) — s=' 10', после str(sqrt(2):6:4,s) — s='1.4142'.

Строки частично совместимы с символьным типом: тип символ можно рассматривать как строку с константной длиной, равной 1. В строковых операциях можно, где это осмысленно (+, concat, length, insert — s1, pos), использовать данные типа символ.

[В Бэйсике связь строк и массивов скрыта.]

Множества

Этот тип предназначен для работы с данными, представимыми в виде множеств. Множество — это чрезвычайно общее понятие. Поэтому в компьютере можно представлять лишь множества, на которые наложены ряд ограничений. Множества могут быть только конечными (до 256 элементов) и должны состоять из однотипных элементов. Тип множества описывается при помощи служебного слова set.

Примеры:

```
set of (David, Helena, Diana, Ann, Robert) (* множество имен *)
set of 'A'..'Z'       (* множество из заглавных латинских букв *)
set of char           (* множество символов *)
```

Литералы-множества записываются перечислением своих элементов в квадратных скобках. Элементы должны отделяться друг от друга запятой. Кроме того, в таком списке можно использовать диапазоны, первый элемент которых должен отделяться от второго двумя точкам. Примеры: [0..9] — множество всех чисел от 0 до 9, ['0'..'9','A'..'F','a'..'f'] — множество шестнадцатеричных символов-цифр.

К данным множественного типа применимы операции сравнения: = и <>. К ним также применимы операции включения <= ($\subset$) и >= ($\supset$), например, если A и B множества, то A<=B означает A подмножество B ($A \subset B$) и, аналогично, A>=B означает, что B подмножество A ($B \subset A$). Например, верно, что [1,2]<=[0..5], и неверно, что [2,5,8]<=[3..4].

К множествам применимы еще следующие операции:

\+ — объединение множеств ($\cup$);

− — вычитание множеств ($\setminus$);

* — пересечение множеств ($\cap$);

in — проверка на вхождение элемента в множество, например, если s — множество, состоящее из чисел 1, 4 и 5, то значение выражения 2 in s — false, а 5 in s — true.

Примеры: [1..3]+[1,4,5]=[1..5], [1..3]-[1,4,5]=[2,3], [1..3]*[1,4,5]=[1], 3 in [1..8] = true.

[В Бэйсике и в Си++ нет возможности непосредственно работать с множествами.]

Файлы

Файловый тип описывается при помощи служебного слова file. Файл — это последовательность элементов одного типа, не содержащего в своем описании слова file. Имеется непосредственный доступ к только одному элементу из этой последовательности и процедуры перехода к некоторым ее элементам. Файлы отличаются от одномерных массивов следующими характеристиками:

1) количество элементов в файле заранее не ограничивается;
2) к произвольному элементу файла нет быстрого доступа;
3) наборы операций к файлам и массивам совершенно различные;
4) информация может сохраняться в файлах после выключении компьютера, т. е. файлы существуют независимо от программы.

Примеры:

```
file of integer  (* файл из целых чисел *)
file of array[2..4]of byte  (* файл из массивов, состоящих из 3-х
                            элементов целого типа *)
{file of array[1..2]of file of byte (* так нельзя *)}
```

[Подробнее файлы будут рассмотрены позже.]

Комбинированный тип (запись)

Этот тип позволяет совмещать в одном элементе данных не только однотипные величины, как в массивах, но и разнотипные. Последнее очень удобно, т. к. реальные данные имеют, как правило, именно такую структуру. Например, любой документ имеет ряд разнотипных реквизитов: даты, источник, адресат, отметка о выполнении, содержание и т. п. Другой пример, анкетные данные, которые содержат: ФИО (строка), дату рождения (массив из трех элементов целого типа), номер паспорта (целое число), годовой доход (вещественное число) и т. д. Базы данных, как правило, представляют собой файл записей. Область для хранения каждой отдельной величины записи называется полем. Все поля в записи имеют свои идентификаторы. Доступ к этим величинам производится по этим идентификаторам. Тип записи описывается при помощи служебного слова record.

Пример.

```
record name:string[32]; age:byte end (* эта запись состоит из двух полей, ФИО и возраста *)
```

Особым случаем записей являются записи с вариантами. Они используются для экономии памяти и преобразования типов (последнее — только для опытных программистов). Например, в анкетных данных на пенсионера нужно указывать размер пенсии, а на непенсионера — стаж. Таким образом получается, что поля (варианты) “стаж” и “размер пенсии” никогда не нужны вместе. Если резервировать во всех записях место как для хранения длительности стажа, так и для размера пенсии, то получится бессмысленный перерасход памяти. Запись с вариантами позволит избежать этого перерасхода, используя для хранения обоих полей одно и то же место в памяти: памяти выделяется ровно столько, сколько необходимо большему из всех совмещаемых полей.

Пример описания типа, решающего описанную проблему:

```
record
  name: string[63];
  id: (worker, pensioner);
  case byte of
    1: (length_of_service: byte);
    2: (pension: real)
end
```

Поле id необходимо для того, чтобы знать, какой из вариантов использовать. Чтобы подчеркнуть особую роль поля id, этот же тип можно описать и так:

```
record
  name: string[63];
  case id: (worker, pensioner) of
    worker: (length_of_service: byte);
    pensioner: (pension: real)
end
```

Во втором случае поле id имеет специальное название, дискриминант.

Отсутствие подобного id поля в записи с вариантами не является синтаксической ошибкой, но такая запись является примером описания типа, который невозможно осмысленно использовать, т.е. ошибкой семантической.

Для записей нет специальных операций.

Подробнее особенности использования комбинированных типов будут рассмотрены позже. [В современных версиях языка Бэйсик поддерживаются комбинированные типы, но без записей с вариантами.]

Объектные типы

Они являются дальнейшим развитием идеи комбинированного типа. В Си++, например, типы объекта и записи практически означают одно и то же. Объектные типы описывается при помощи служебного слова object. В программировании сильна тенденция к использованию при создании программ исключительно объектных типов.

[Подробно эти типы планируется рассмотреть при изучении объектно-ориентированного программирования (ООП) в 4-м семестре. Даже в новейших версиях языка Бэйсик использование объектных типов возможно только неявным образом.]

Типы указателей

Они предназначены для описания сущностей, подобных книжным предметным или именным указателям, т.е. это тип для данных, хранящих адреса участков памяти. Типы указателей описываются при помощи специального символа, ^ (стрелка вверх). Например, `^integer (* тип указателя на данные целого типа *)`. Данные этого типа — это адрес участка памяти из двух байт, интерпретируемых как целое число.

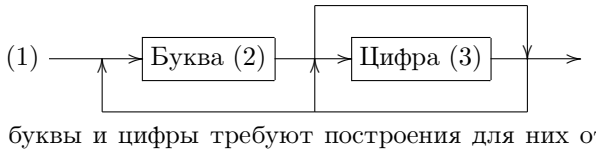
Особым случаем типов-указателей можно считать процедурные типы.

Подробнее особенности использования типов-указателей будут рассмотрены позже. [В языке Бэйсик указатели не поддерживаются.]

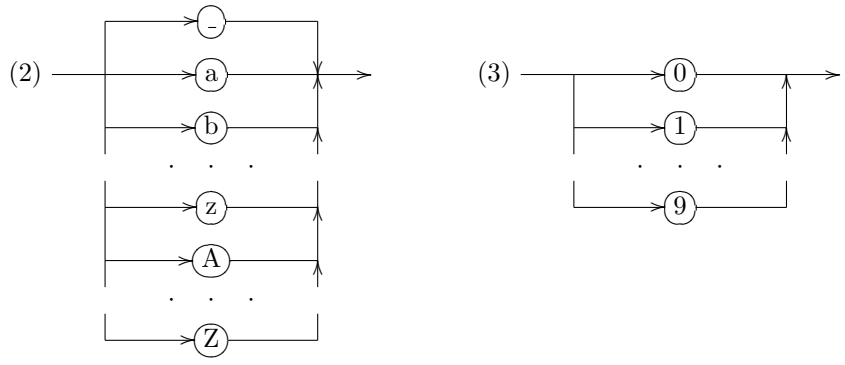
Синтаксические диаграммы

Конструкция из стрелочек, соединяющих элементы структуры языка, с выделенными начальной и конечной точками. Любой путь из начальной в конечную точку соответствует структуре, которая является синтаксически правильной для рассматриваемого языка. В прямоугольник будут заключаться элементы структуры языка, требующие дальнейшего уточнения, а в овал – конечные или терминальные элементы (их можно видеть в тексте программы). Синтаксические диаграммы, приводимые в данном курсе, будут, кроме того, нумероваться (для ссылок).

Синтаксическая диаграмма идентификатора в си, паскале или бэйсике



В этой диаграмме понятия буквы и цифры требуют построения для них отдельных синтаксических диаграмм, которые уже уникальны для языка паскаль.



Следующие строки символов будут являться правильными идентификаторами в языке паскаль: _0001, ABC, z12AV_12, aAbB, false, byte. А следующие — неправильными: 7abc, AAA$, VVV.S1, set, Begin.

Знание всех синтаксических диаграмм некоторого языка — это знание грамматики этого языка. Тот, кто знает синтаксические диаграммы никогда не сделает ошибки в своей программе и, кроме того, сможет находить ошибки в чужих программах.

Алфавит

Буквы, цифры и 23 специальных символа: `^*()-+='<>/:;.,[]{}<пробел>@#$` (три последних символа добавлены к алфавиту фирмой Borland).

Служебные слова

ОСНОВНЫЕ

and	array	begin	case	const	div	do	downto	else
end	file	for	function	goto	if	in	label	mod
nil	not	of	or	procedure	record	repeat	set	shl
shr	string	then	to	type	until	var	while	with
xor	[всего 37]							

МАЛОИСПОЛЬЗУЕМЫЕ

program	packed

ДЛЯ РАБОТЫ С ОБЪЕКТНЫМИ ТИПАМИ

constructor	destructor	inherited	object

ДЛЯ ПОДДЕРЖКИ МОДУЛЬНОСТИ

implementation		interface	unit	uses

ДЛЯ РАБОТЫ НА МАШИННОМ ЯЗЫКЕ

asm	inline

[Слова первых двух групп нужно знать к концу этого семестра, следующих двух — к концу следующего.]

Лексемы операций, состоящие из более чем одного символа

`:=` — присвоить
`<>` — неравенство
`<=` — меньше или равно
`>=` — больше или равно
`(.` — аналог знака [
`.)` — аналог знака]
`..` — диапазон

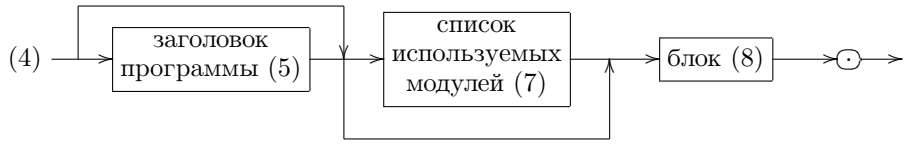
Общая структура программы

Паскаль-программа состоит из двух частей: описания последовательности действий, которые необходимо выполнить, и описания данных, с которыми оперируют действия. Действия представляются операторами языка, данные вводятся посредством деклараций.

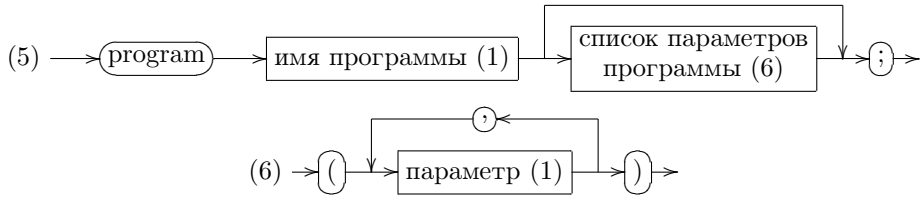
Описания данных должны предшествовать действиям с ними!

Некоторые из строимых далее синтаксических диаграмм не учитывают некоторых расширений паскаля фирмы Borland.

Синтаксическая диаграмма любой паскаль-программы

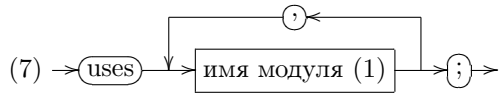


Из этой диаграммы видно, что заголовок программы и список используемых модулей можно опускать.

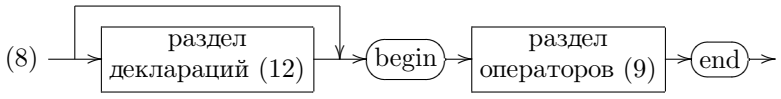


Пример заголовка программы: `program hello(input,output);`.

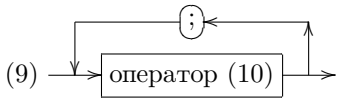
Параметры программы игнорируются*. Получается, что наличие или отсутствие заголовка программы почти никак не влияет на ее функционирование.



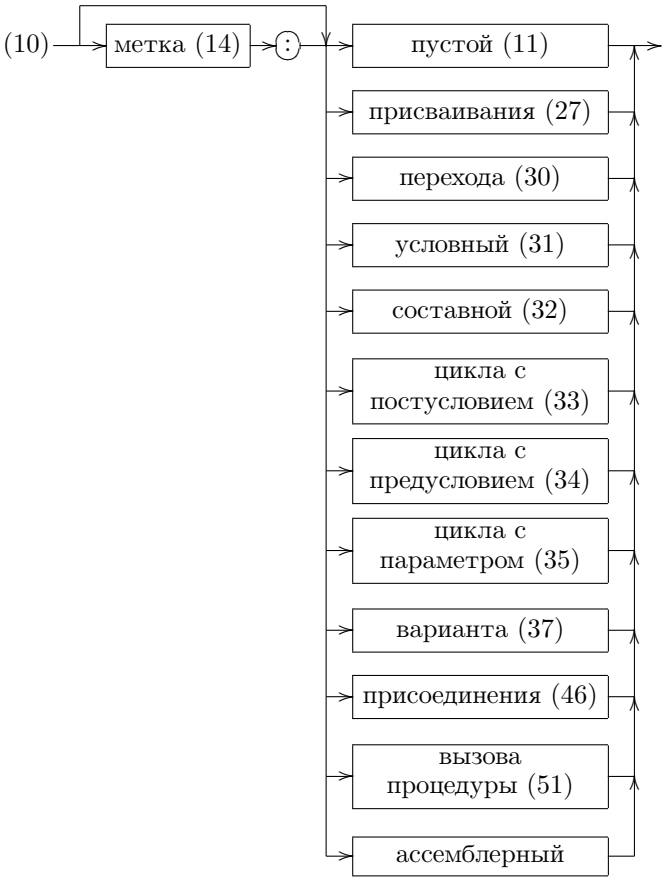
Пример списка используемых модулей: `uses dos,crt;`. [Подробно модульное программирование планируется рассматривать в следующем семестре.]



Следовательно, раздела деклараций может не быть вообще.

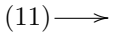


Операторы



Таким образом, метку можно поставить перед любым оператором.

Пустой оператор



Этот оператор не производит никакого действия.

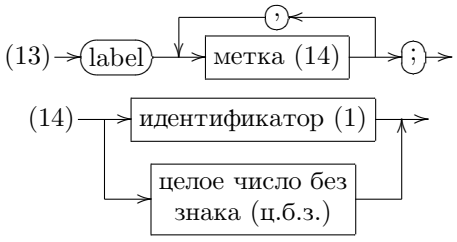
Из приведенного материала следует, что самая короткая программа на паскале имеет следующий вид: `begin end.` Эта программа состоит только из одного пустого оператора. Программа, состоящая из только двух пустых операторов, имеет вид: `begin;end.`

---

* их использовали прежние варианты языка

Раздел деклараций программы

Описание меток

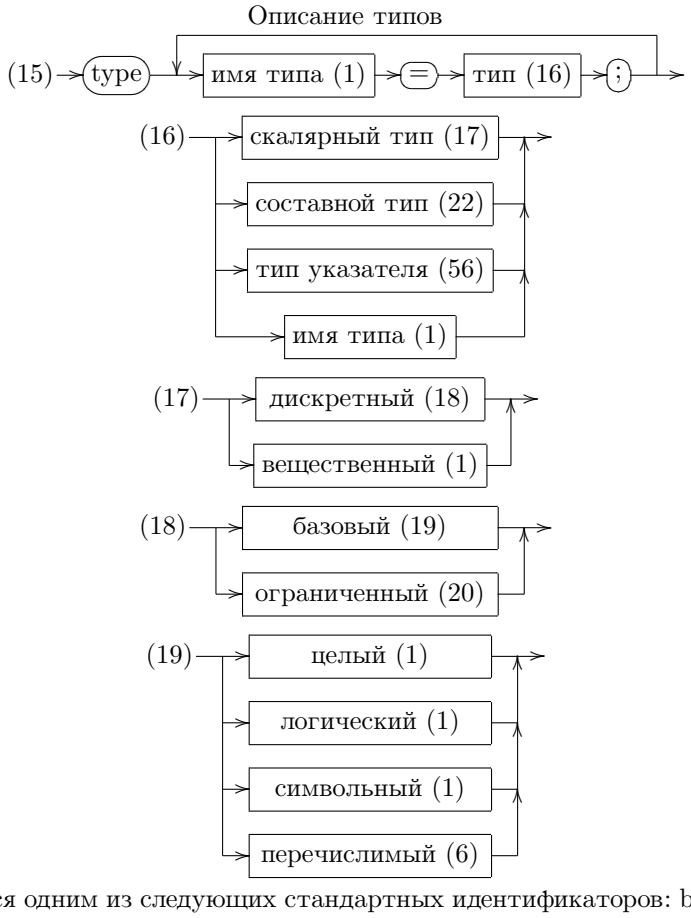


Целое число для метки допустимо только из диапазона от 0 до 9999.

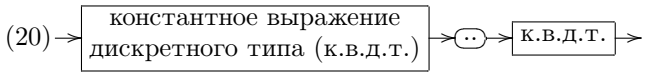
Пример 1:

```
Label 1, MyLabel, L800, _12M;
```

Описание типов



Целый тип определяется одним из следующих стандартных идентификаторов: byte, shortint, word, integer и longint. Логический — boolean. Символьный — char. Вещественный — real, single, double, extended и comp. Кроме того, можно описывать новые имена для таких типов. Например, можно описать тип my_char декларацией `type my_char=char;` и после этого использовать идентификатор my_char для именования символьного типа, т. е. my_char станет синонимом слова char. Если после такой декларации переопределить значение слова char декларацией `type char=integer;`, то доступ к символьному типу станет возможен только через идентификатор my_char.



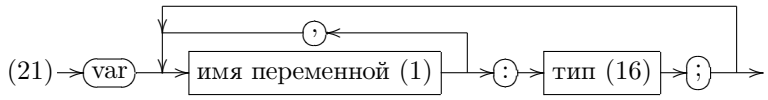
Первое константное выражение дискретного типа должно быть не больше второго.

В константные выражения не допускается вхождения величин, неизвестных в момент старта программы: значений переменных, вызовов функций, определенных в программе, и т. п. В константных выражениях допускается использование только простейших стандартных функций.

Продолжение примера 1.

```
Type
  WeekDays = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
  MySymbols = 'A'..'Z'; (* только 26 заглавных латинских букв *)
  MyDays = Sunday..Friday;
  Money = Real;
  OldInt = Integer; (* после этих деклараций все программные объекты типа *)
  Integer = LongInt;(* integer изменят свой тип на longint *)
```

Описание переменных

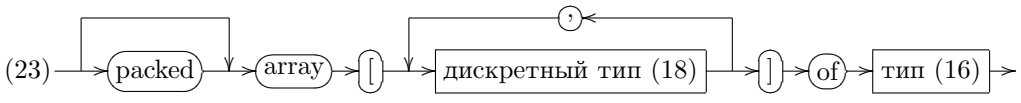


Продолжение примера 1.

```
Var
  Index: 2..81;  (* переменная Index может хранить числа от 2 до 81 *)
  MyArray: Array[2..81]of Char; (* массив MyArray может хранить 80
    символов *)
  Counter: Byte; (* переменная Counter может хранить числа от 0 до 255 *)
  i, j: Integer; (* эти переменные могут хранить числа с 9-ю знаками *)
  Day: WeekDays;
  Dollars: Money;
```

Составные типы

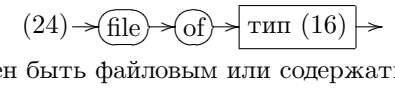
Массивы



Продолжение примера 1.

```
Type
  BigArray = Array[1..20000]of byte;
  SmallArray = Array[4..4]of boolean;
  Enums = (First, Second, Third, Fourth, Fifth, Sixth, Seventh);
Var
  Array1: Array[Byte]of Byte;
  Array2: Array[Char, 2..7]of Integer;
  Array3: Array['A'..'C', Boolean, Enums, Enums]of
    Array[Second..Sixth]of SmallArray;
  Array4: BigArray;
  Array5: Array[false..true]of Enums;
```

Файлы

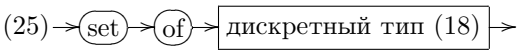


Тип в определении файла не должен быть файловым или содержать в себе компоненты файлового типа.

Продолжение примера 1.

```
Type
  MyFile = File of Byte;
Var
  File1: File of SmallArray;
  File2: MyFile;
  Files: Array[1..7]of File of Char;
```

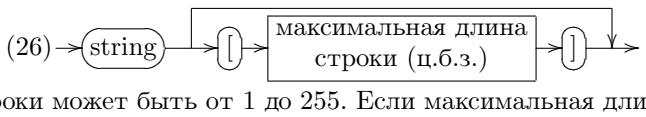
Множества



Продолжение примера 1.

```
Type
  MySet = Set of Byte;
Var
  Set1: Set of 1..12;
  Set2: MySet;
  Set3: Set of Enums;
  Sets: Array[7..12]of MySet;
```

Строки

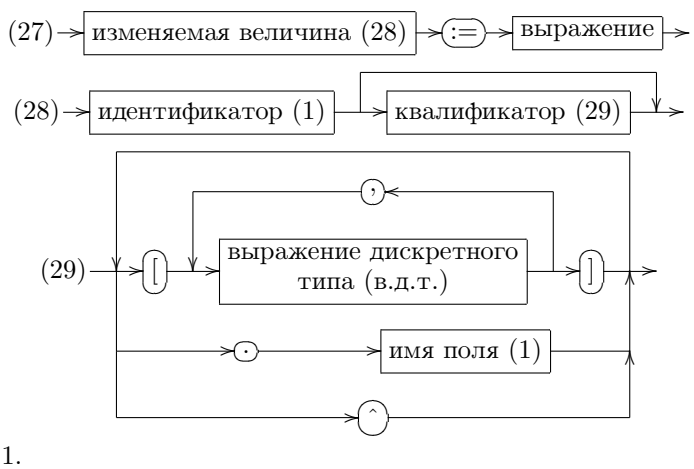


Максимальная длина строки может быть от 1 до 255. Если максимальная длина строки не приведена явно, то она считается равной 255.

Продолжение примера 1.

```
Type
  MyString = String[31];
Var
  String1: MyString;
  String2: String;
  String3: String[2];
  Strings1: Array[char]of String[63];
  Strings2: File of String[63];
```

Оператор присваивания

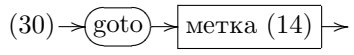


Продолжение примера 1.

```
Begin
  i := -1;
  1: j := 120000000 + i;
  Inc(Counter);
  Dollars := 100000.70;
  i := -i;
  Day := Wednesday;
  L800: Array1[Counter + 2] := 0;
  Array3['B', false, First, Third][Fifth][4] := true;
  (* следующая строка означает то же, что и предыдущая *)
  Array3['B', false, First, Third, Fifth, 4] := true;
  Array4[5] := 6;
  Array5[true] := Second;
```

```
  String3 := 'Test'; (* после этой операции значением String3 будет 'Te' *)
  String1 := 'Test';
  String1[1] := 'W'; (* после этой операции значением String1 будет 'West' *)
  _12M: Strings1['V'] := 'String ';
  Strings1['V',7] := 's'; (* после этой операции значением элемента массива
    Strings1 с индексом 'V' станет 'Strings' *)
```

Оператор перехода



Метка и оператор перехода на нее должны находиться в одном блоке.

Продолжение примера 1.

```
  goto 1;
  goto L800; (* этот и следующий операторы никогда не будут выполнены *)
  goto _12M
End.
```

Приведенная программа-пример никогда не остановится: переменная Counter будет циклически менять значения от 0 до 255, переменная j периодически менять значение 120000000 в 119999999 и обратно и т.п. [Остановить выполнение этой программы можно нажатием сочетания клавиш Ctrl и Break.]

В концепции программирования, утвердившейся с начала 70-х годов и получившей название структурного программирования, использование оператора goto считается недопустимым кроме редких случаев, например, для выхода из вложенных циклов. Частое использование goto в программе делает ее трудной для отладки, исправлений и модификаций. Вместо него нужно использовать операторы циклов, условный, варианта и некоторые другие средства. [Этот оператор идентичен оператору GOTO Бэйсика.]

Условный оператор

Пример программы:

```
var
  i: byte;
begin
  i := 2;
  if i > 3 then i := 5 else i := 0;
  (* после выполнения последнего оператора в i будет храниться значение 0 *)
  if i <> 0 then i := 1;
  (* после выполнения последнего оператора значение i не изменится *)
  if false then if i > 0 then i := 0 else i := 1 else i := 5
  (* последний оператор присвоит i значение 5 *)
end.
```

Если в условный оператор входят другие условные операторы, то нужно пользоваться правилом: “Каждый else всегда относится к ближайшему слева свободному, не образовавшему еще пары с else, if”.

Последний оператор можно для большей наглядности записать следующим образом:

```
  if false then (* в этой записи четко видно, к *)
    if i > 0 then (* какому if относится каждый else *)
      i := 0
    else i := 1
  else i := 5.
```

Концепцию структурного программирования также составляют способы наглядной записи программ. Нужно использовать отступы, показывая вложенность-подчиненность одной синтаксической конструкции другой. Хорошим стилем считается запись не более одного оператора в строку.

Составной оператор

На Бэйсике условный оператор может выглядеть так:

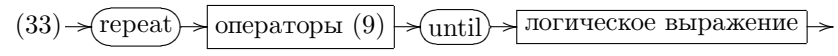
```
IF A > 8 THEN
  B = 17
  V = 22
ELSE
  B = 22
  V = 17
END IF
```

— здесь стоит по два оператора для каждой ветви условия. А условный оператор в паскале позволяет в каждой ветви условия находиться только одному оператору. Но в паскале есть составной оператор, позволяющий группировать несколько операторов в один.

Используя составной оператор, можно написать на паскале аналог приведенной конструкции языка бэйсик:

```
if a > 8 then begin
  b := 17;
  v := 22
end else begin
  b := 22;
  v := 17
end.
```

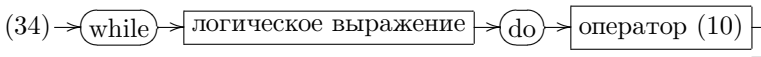
Оператор цикла с постусловием



Этот оператор выполняется так: сначала выполняются операторы до служебного слова until, затем вычисляется логическое выражение. Если оно истинно, то происходит выход из цикла, иначе происходит переход к первому оператору, следующему после слова repeat, и цикл продолжает выполняться. Необходимо отметить, цикл будет выполнен не менее одного раза.

Например, оператор `repeat array7[i]:=0; i:=i+1 until i>=100` заполнит компоненты массива array7 с индексами от начального значения i до 99 нулями.

Оператор цикла с предусловием

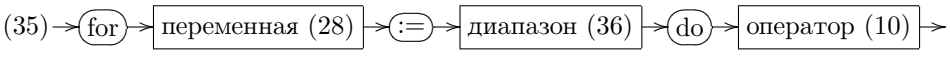


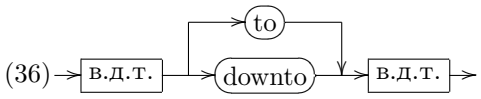
Этот оператор выполняется так: сначала вычисляется логическое выражение. Если оно истинно, то выполняется оператор и цикл продолжает выполняться, иначе происходит выход из цикла. Таким образом, цикл может не выполняться ни разу.

Например, оператор `while i<100 do begin array7[i]:=0; inc(i) end` функционально аналогичен оператору из предыдущего примера, но здесь в случае, когда начальное значения i больше 99 возможно ошибочное присваивание не выполнится.

Оператор цикла с параметром

Этот оператор полезен в случаях, когда требуется, чтобы цикл выполнялся заданное количество раз.





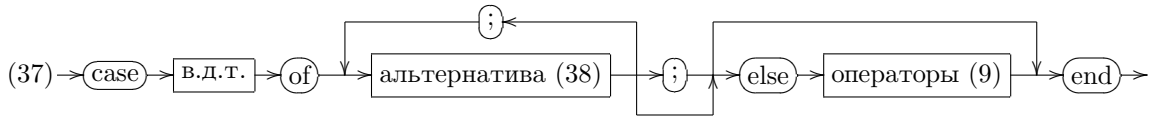
Примеры:

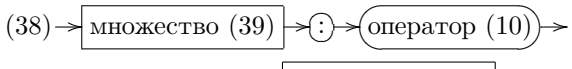
```
for i := 1 to 99 do array7[i] := random(8);
for i := 99 downto 1 do array7[i] := random(8) (* оба этих оператора
  заполняют компоненты массива array7 с индексами от 1 до 99 случайными
  числами от 0 до 7 *)
```

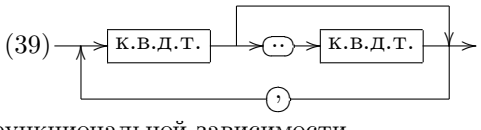
Если диапазон изменений переменной пуст, например, 22 to 4 или 5 downto 6, то цикл не выполняется ни разу.

Шаг изменений переменной фиксирован и равен 1 (данные любого дискретного типа можно занумеровать).

Оператор варианта







Пример. Вычисление сложной функциональной зависимости

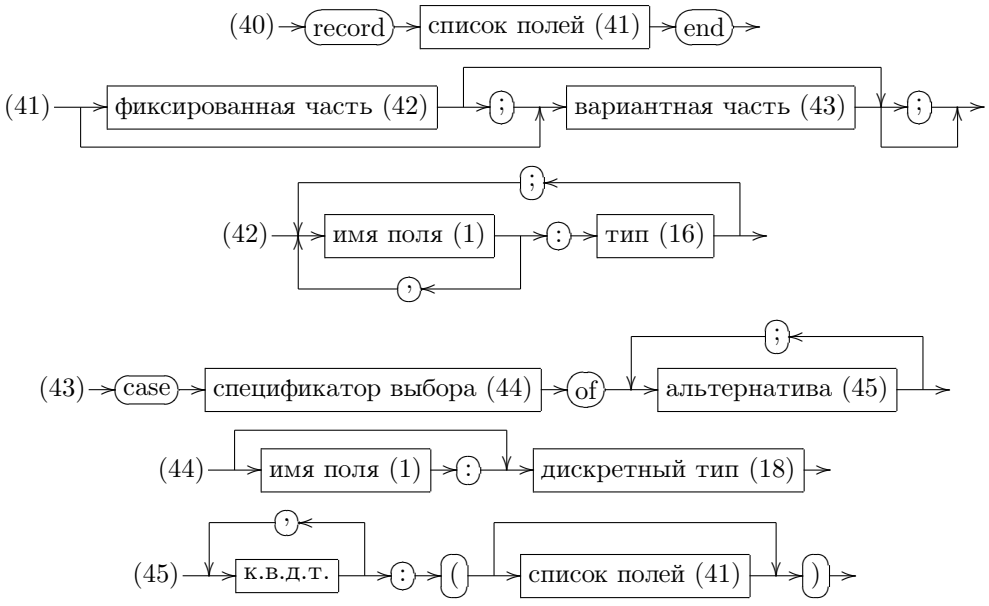
$$f(x) = \begin{cases} 1, & \text{при } x = 0; \\ 10, & \text{при } x \in \{1, \ldots, 10, 12\}; \\ 100, & \text{при } x \in \{11, 14, \ldots, 22, 44, \ldots, 55\}; \\ 1000, & \text{при } x \in \{66, \ldots, 90, 25, 27\}; \\ 10000, & \text{иначе.} \end{cases}$$

```
case x of
  0: f := 1;
  1..10, 12: f := 10;
  11, 14..22, 44..55: f := 100;
  66..90, 25, 27: f := 1000
else
  f := 10000
end
```

Вместо case всегда можно использовать вложенные if, но case делает код более ясным и компактным.

```
if x = 0 then
  f := 1
else if x in [1..10, 12] then
  f := 10
else if x in [11, 14..22, 44..55] then
  f := 100
else if x in [66..90, 25, 27] then
  f := 1000
else
  f := 10000
```

Комбинированный тип



Пример. Пусть требуется ввести в компьютер некоторые факты, которые описываются сопроводительной записью и некоторой специфической информацией: датой, вещественным числом или комбинацией двух целых чисел. Для представления таких данных наиболее естественно использование типа массив или файл записей с вариантами.

```
type
  type_for_fact = record
    id: string[64];
    case kind: (D, N, I) of (* kind - дискриминант, в стандарте паскаля
        типом дискриминанта может быть только идентификатор *)
      D: (date: string[10]);
      N: (quantity: real);
      I: (weight: word; length: word)
  end;
var
  facts: array[1..4] of type_for_fact; (* список фактов *)
  k: integer;
begin
  (* доступ к полям записи происходит с использованием символа точка *)
  facts[1].id := 'Битва при Ватерлоо';
  facts[1].date := '18-6-1815';
  facts[1].kind := D;
  facts[2].id := 'Длина экватора Земли в км';
```

```
  facts[2].quantity := 40075.696;
  facts[2].kind := N;
  facts[3].id := 'Вес (в кг) и длина (в см) немецкого истребителя Bf-109E';
  facts[3].weight := 2500;
  facts[3].length := 865;
  facts[3].kind := I;
  facts[4].id := 'Дата рождения Блеза Паскаля';
  facts[4].date := '19-6-1623';
  facts[4].kind := D;
  for k := 1 to 4 do begin
    writeln('Факт ', k);
    writeln(facts[k].id);
    case facts[k].kind of
      D: writeln(facts[k].date);
      N: writeln(facts[k].quantity:1:4);
      I: writeln(facts[k].weight, ', ', facts[k].length)
    end;
    writeln
  end
end.
```

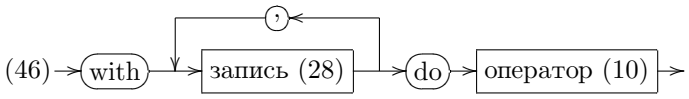
Тип type_of_facts можно было также описать и так:

```
  type_for_fact = record
    id: string[64];
    kind: (D, N, I);
    case byte of
      0: (date: string[8]);
      1: (quantity: real);
      2: (weight, length: word)
  end
```

Контроль за правильностью использования того или иного варианта целиком ложится на программиста. Например, в приведенной программе бессмысленно интересоваться значением facts[3].date или facts[1].length, т.е. перед тем как обращаться к тому или иному полю-варианту в программе должен проверяться дискриминант или соответствующее ему по смыслу поле. Если при заполнении данных дискриминант или соответствующее ему по смыслу поле будет введено неверно, то вся информация из вариантной части записи становится недоступной для формальной обработки.

Оператор присоединения

Он предназначен исключительно для работы с данными комбинированного типа.



Смысл этого оператора — сократить запись программных конструкций, содержащих данные комбинированного типа.

Пример.

```
var
  person: record name: string[31]; age: byte; weight: real end;
begin
  person.name := 'John Smith'; person.age := 32; person.weight := 70.5;
  (* те же действия, но с оператором присоединения *)
  with person do begin name := 'John Smith'; age := 32; weight := 70.5 end
end.
```

Если полей в записи много, то использование этого оператора сократит запись алгоритма. В одном операторе присоединения могут объединяться несколько записей.

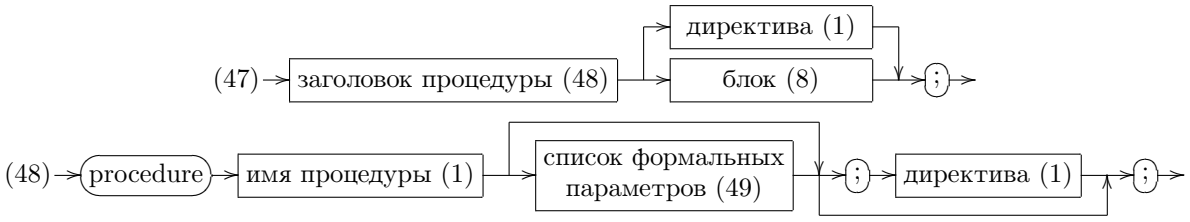
Подпрограммы

Подпрограммы в паскале, как и в современном бэйсике, могут быть двух видов: 1) процедуры; 2) функции. Использование подпрограмм имеет два основных назначения:

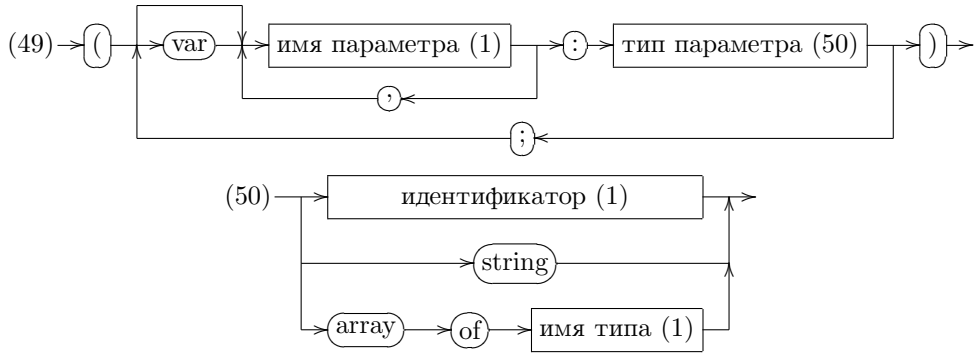
1) Экономия памяти. Если в алгоритме есть группы идентичных повторяющихся действий, то их можно оформить в виде подпрограммы и использовать затем ее имя всякий раз вместо каждой из этих групп. Чем больше такая группа действий и чем большее число раз она повторяется, тем большим будет выигрыш от использования подпрограммы, заменяющей ее;

2) Упрощение работы программиста. Если имеется последовательность операций, выполняющих описываемое одним понятием задание, то естественно дать этой последовательности имя этого понятия. Затем, реализовав эту последовательность в виде подпрограммы с выбранным именем, программист может думать об этой последовательности как об одном целом, не вникая в детали реализации этой подпрограммы. Это позволяет программисту путем последовательного деления всего алгоритма на именованные части думать о реализации программы в целом, не зацикливаясь на мелких деталях.

Синтаксис описания процедуры



Стандартные директивы-идентификаторы рассматриваются далее.



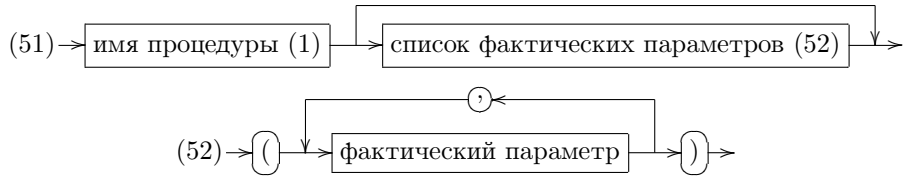
Таким образом, заголовок процедуры procedure with_bad_header(a:set of char); — неправилен, т.к. для задания такого типа параметра нельзя использовать приведенный синтаксис. Нужно было сначала описать нужный тип идентификатором, а затем использовать этот идентификатор в качестве имени типа в заголовке процедуры.

Пример правильно описанной процедуры:

```
Procedure ShowMessage(s: String; n: Byte);
(* выводит сообщение s n раз на экран *)
  Var i: Byte;
  begin
    For i := 1 to n do
      Writeln(s)
  end;
```

Добавление описания процедуры в раздел деклараций позволяет затем при помощи оператора вызова процедуры использовать ее в разделе операторов.

Оператор вызова процедуры



Фактический параметр, соответствующий формальному, следующему после служебного слова var, может быть только переменной. Другими фактическими параметрами могут быть произвольные выражения. В обоих случаях типы выражений (переменная наряду с константой — это простейшие выражения) должны совпадать с соответствующими типами, заданными при описании формальных параметров.

Продолжение предыдущего примера, превращающего его в законченную программу.

```
begin
  ShowMessage('Hello World', 1+1)
  (* Если параметр s или n были бы описаны после var, то подобный вызов
     процедуры привел бы к ошибке при компиляции программы *)
end.
```

Параметры-ссылки и параметры-значения

Параметры, следующие после служебного слова var в заголовке подпрограммы, — это параметры-ссылки. Прочие — это параметры-значения.

Параметры-значения — это отдельные переменные, которым присваиваются значения соответствующих фактических параметров. Параметры-ссылки — это переменные, имеющие тот же адрес в памяти, что и соответствующие им фактические параметры. Отсюда следует, что изменение значения формального параметра-ссылки в подпрограмме приведет к такому же изменению фактического параметра. Фактическим параметром, соответствующим формальному параметру-ссылке, в паскале может быть только переменная. При передаче по ссылке происходит отождествление формального и фактического параметра.

Пример.

```
procedure factorial(n: byte; var result: longint);
(* эта процедура вычисляет n!, возвращая результат в result *)
  begin
    result := 1;
    while n > 1 do begin
      result := result*n;
      dec(n)
    end
  end;
var
  answer: longint;
begin
  factorial(7, answer);
  writeln(answer) (* на экране дисплея будет напечатано 5040 *)
end.
```

Когда лучше использовать параметры-ссылки, а когда параметры-значения? При передаче параметра по значению происходит копирование фактического параметра в переменную, формальный параметр. Если тип параметра соответствует данным, занимающим много места в памяти, например, массиву или записи, то при вызове подпрограммы нужно копировать много данных, что может существенно замедлить процесс вычислений. Кроме того, в этом случае подпрограмма должна при каждом своем вызове выделять много места для хранения значения этого большого параметра. Подпрограммы могут вызывать сами себя, а в случае большого параметра-значения, каждый такой вызов будет требовать много памяти, что позволит делать такие вызовы только очень ограниченное число раз. Итак для передачи больших данных, как правило, лучше пользоваться параметрами-ссылками, т.к. при их использовании требуется копировать лишь адрес данных, размер которого мал (2, 4 или 8 байт) и не зависит от размера самих данных. Использование параметра-ссылки нежелательно тогда, когда нужно гарантировать неизменность фактического параметра после вызова подпрограммы, т.к. при передачи параметра по ссылке все изменения формального параметра приведут к таким же изменениям фактического.

Вывод. Параметры-ссылки целесообразно использовать в следующих двух случаях:

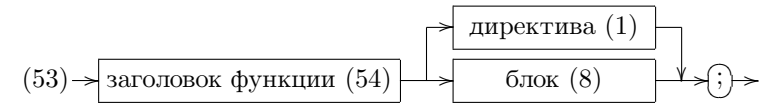
1) при передаче параметров большого размера;
2) в случае, когда требуется, чтобы значение возвращалось через параметр.

Параметры-значения обычно используют в следующих трех случаях:

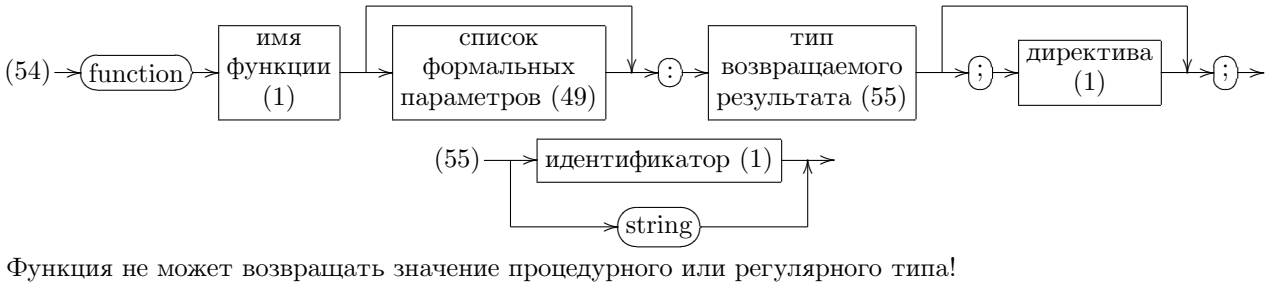
1) при передаче параметров маленького размера;
2) в случае, когда требуется обеспечить неизменность фактического параметра;
3) в случае, когда требуется передавать формальному параметру значение непеременной.

В последнем примере вместо процедуры лучше было бы использовать функцию. Функция отличается от процедуры тем, что она возвращает значение и поэтому может быть использована в выражениях.

Синтаксис описания функции



Директивы у функций такие же как и у процедур. В блок функции должен входить оператор присваивания, в левой части которого — имя описываемой функции. Если подобный оператор не описан или не выполнялся при вызове функции, то результат выполнения функции будет неопределен.



Функция не может возвращать значение процедурного или регулярного типа!

Пример.

```
function factorial(n: byte): longint;
```

```
  var temp:longint;
  begin
    temp := 1;
    while n > 1 do begin
      temp := temp*n;
      dec(n)
    end;
    factorial := temp
  end;
begin
  writeln(factorial(5) - 5)
end. (* на экране дисплея будет напечатано 115, т.к. 5! - 5 = 115 *)
```

Рекурсия и предварительное описание подпрограмм

Сделать функцию из последнего примера более компактной поможет механизм рекурсии. Если подпрограмма обращается к себе самой, то она называется рекурсивной. Рекурсия бывает двух видов: прямая и косвенная. Если подпрограмма обращается к себе самой непосредственно, то это прямая рекурсия. Если же такое обращение происходит через какую-то другую подпрограмму, то это косвенная рекурсия.

Итак, используя рекурсию, окончательно программу для вычисления факториала можно записать так:

```
function factorial(n: byte): longint;
  begin
    factorial := 1;
    if n > 1 then
      factorial := factorial(n - 1)*n;  (* n!=n(n-1)! *)
  end;
begin
  writeln(factorial(7) - 7) (* на экране дисплея будет напечатано 5033 *)
end.
```

Тривиальным примером использования косвенной рекурсии служит следующая программа для приближенного расчета квадратного корня из 2. Квадратный корень из 2 можно рассчитать при помощи двух целочисленных функций

$$a(n) = \begin{cases} 1, & \text{при } n = 1 \\ a(n-1) + b(n-1), & \text{при } n > 1 \end{cases} \quad \text{и} \quad b(n) = \begin{cases} 1, & \text{при } n = 1 \\ 2 * a(n-1) + b(n-1), & \text{при } n > 1 \end{cases}.$$

$$\lim_{n\to\infty} \frac{b(n)}{a(n)} = \sqrt{2}.$$

```
Function A(n: byte): word;
  Begin
    if n = 1 then
      A := 1
    else
      A := A(n - 1) + B(n - 1)
  End;
Function B(n:byte): word;
  Begin
   if n = 1 then
     B := 1
   else
     B := 2*A(n - 1) + B(n - 1)
  End;
Var
  n: byte;
Begin
  n := 10; (* чем больше n, тем выше точность *)
  writeln(B(n)/A(n):10:7) (* печатает корень из 2 с точностью 7 знаков *)
End.
```

В последнем примере есть ошибка. Дело в том, что, как это уже отмечалось ранее, использованию имени должно предшествовать его описание. Из этого правила есть только одно исключение, связанное со ссылочным типом данных. В рассматриваемой же программе в описании функции A есть вызов еще неописанной функции B... Если же первой описать функцию B, то получится вызов неописанной функции A. Выходом из этой ситуации

является предварительное описание B, которое состоит только из заголовка функции. Итак, чтобы приведенная программа заработала нужно добавить к ее началу строчку

```
Function B(n: byte): word; forward; ,
```

где forward — это директива, сообщающая компилятору о том, что предшествующий ей заголовок приведен только для предварительного описания имени. Точно также используется forward и для процедур.

Внешние подпрограммы

Кроме директивы forward с заголовком подпрограммы может быть использована директива external, которая означает, что машинный код для этой подпрограммы уже готов и находится в указываемом в специальной директиве компилятору файле. Эта директива позволяет добавлять к программе на паскале подпрограммы, написанные на других ЯП, например, си или ассемблере.

Стандартные процедуры для работы в циклах

Существует множество готовых подпрограмм для выполнения самых разных требований программиста. Например, подпрограмма для печати данных на экране дисплея, writeln.

Часто необходимо организовать выход из цикла до его завершения по основному условию. Конечно можно воспользоваться оператором goto. Но использование безусловного перехода следует по возможности избегать. Стандартная процедура break при своем вызове в цикле передает управление на оператор, следующий за этим оператором цикла. Кроме того, существует еще стандартная процедура continue, которая при своем вызове в цикле передает управление на начало (новый виток) этого цикла.

Пример.

```
i := 1;
(* 1: *)
while i < 100 do begin
  inc(i);
  if array7[i] < 0 then continue (* goto 1 *);
  a := a + array7[i];
  if a > 1000 then break (* goto 2 *)
end;
(* 2: *)
writeln(a)
```

Выход из подпрограммы

Стандартная процедура exit позволяет выйти из любого места подпрограммы. Если ее вызвать в разделе операторов программы, то этот вызов приведет к окончанию выполнения всей программы.

Пример.

```
procedure test_exit(n: byte);
  begin
    if n = 0 then
      exit (* goto 1 *);
    writeln(n)
(* 1: *)
  end;
```

Побочный эффект подпрограмм

Если подпрограмма изменяет значения переменных, не объявленных в ней, то она называется подпрограммой с побочным эффектом. Использование побочного эффекта может вызвать трудноустранимые ошибки. Не следует использовать такие подпрограммы. Вместо побочного эффекта, как правило, лучше использовать параметры-ссылки.

Выход из программы

Вызов стандартной процедуры halt приводит к окончанию выполнения программы.

Совместимость типов

В паскале принята концепция именной совместимости типов: два типа считаются совместимыми, если они имеют одинаковые имена или если их имена различаются, но получены прямым переименованием. Совместимость типов необходима в операторе присваивания (левая и правая части должны быть совместимых типов) и при вызова подпрограммы, в последнем случае тип фактического параметра должен быть совместим с типом соответствующего формального.

Пример.

```
type
  a = array[1..5]of byte;
  b = array[1..5]of byte;
  c = a;
```

```
var
  u: array[1..5]of byte;
  v: array[1..5]of byte;
  w, x: a;
  y: b;
  z: c;
begin
{  x := y; (* ошибка! *)}
  x := z; (* правильно *)
  x := w; (* правильно *)
{  x := v; (* ошибка! *)}
{  u := v  (* ошибка! *)}
end.
```

Видимость имен

Глобальная — имя видимо (доступно) в любом месте программы, может перекрываться локальным именем. Локальная — имя видимо и, следовательно, доступно только в пределах некоторой конструкции. Локальные имена определяются в разделах деклараций подпрограмм, а глобальные в разделе деклараций самой программы.

Пример.

```
Var TestVar: Integer;
Procedure OutVal;
  Var TestVar: integer;
  Procedure OutVal2;
    Var TestVar: integer;
    Begin
      TestVar := 8;
      Writeln(TestVar);
    End;
  Begin
    TestVar := 7;
    OutVal2;
    Writeln(TestVar)
  End;
BEGIN
  TestVar := 5;
  OutVal;
  Writeln(TestVar)
END. (* программа напечатает 8, 7, 5 *)
```

Т.о. имена вложенных конструкций перекрывают глобальные имена.

Открытые массивы

В паскале в отличие от бэйсика нельзя менять размер массива в процессе выполнения программы, кроме того, требование именной совместимости типов позволяет подпрограммам обрабатывать массивы только фиксированного для каждой заданной подпрограммы размера. Последнее ограничение может быть снято использованием для формальных параметров-массивов типа открытый массив. Фактическому параметру, одномерному массиву, передаваемому как открытой массив, соответствует одномерный формальной параметр-массив с диапазоном индексов от нуля до количества элементов в исходном массиве минус 1.

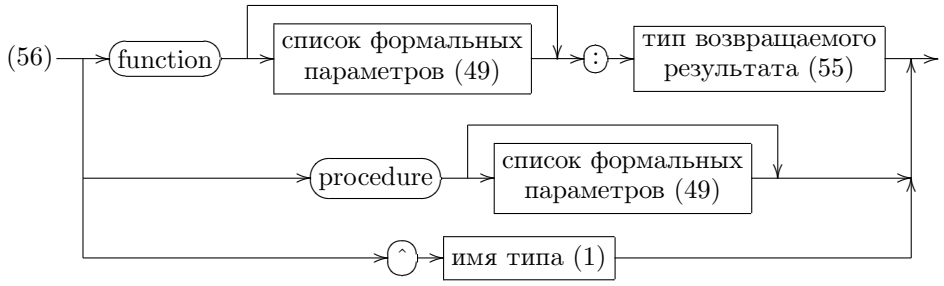
Пример.

```
function average(var a: array of byte): real;
(* вычисление среднего всех чисел в произвольном одномерном массиве *)
  var
    i: byte;
    sum: word;
  begin
    sum := 0;
    for i := low(a) to high(a) do
      sum := sum + a[i]; (* low(a)=0, high(a)=<кол-во элементов в a>-1 *)
    average := sum/(high(a) - low(a) + 1)
  end;
var
  i: shortint;
  a1: array[1..8]of byte;
  a2: array[-4..12]of byte;
```

```
begin
  randomize;
  for i := low(a1) to high(a1) do
    a1[i] := random(128);
  for i := low(a2) to high(a2) do
    a2[i] := random(129);
  writeln(average(a1):7:2, average(a2):7:2)
end.
```

Тип указателя

Можно создавать тип указателя на любой тип. Тип подпрограммы (процедурный тип) — это особый указатель.



Пример программы, печатающей слово ok.

```
var
  i1: integer;
  pi1: ^integer; (* значок ^ можно использовать подряд только один раз *)
  (* pi1 может указывать на любую переменную типа integer *)
begin
  i1 := 777;
  pi1 := @i1; (* операция @ возвращает адрес аргумента *)
  if pi1^ = i1 then writeln('ok.')
  (* операция ^ возвращает указываемый объект *)
end.
```

Служебное слово nil — это “пустой” указатель, который теоретически должен не указывать ни на какое место в памяти. Оно является единственной константой-литералом типа указателей. Необходимо иметь в виду, что практически переменная-указатель любое свое значение считает адресом и поэтому, хотя конструкция `nil^` невозможна, можно использовать ее эквивалент. Например, для переменной pi1 из предыдущего примера можно написать `pi1 := nil; pi1^ := 700`. Но последствия от использования подобных операторов очень опасны и могут разрушить систему. Выявление использования недопустимых адресов представляет наибольшую проблему при работе с указателями, т.к. эти ошибки трудно обнаружить и исправить.

Пример использования процедурного типа.

```
procedure print(s: string);
  begin
    writeln(s)
  end;
function add(a, b: byte): byte;
  begin add := a + b end;
function sub(a, b: byte): byte;
  begin add := a - b end;
var
  p: procedure(s: string);
  f: function(x, y: byte): byte;
begin
  p := print;
  p('I am testing unknown type of data...');
  f := add;
  writeln(f(6, 1)); (* будет напечатано 7 *)
  f := sub;
  writeln(f(6, 1)); (* будет напечатано 5 *)
end.
```

Преобразование типов

Преобразование типов может быть реализовано тремя способами:

1) неявным или автоматическим — такие преобразования производятся между стандартными типами данных: целое число, когда нужно, автоматически преобразуется в вещественное, а символ — в строку;
2) явным, именами типов, — имя типа можно синтаксически использовать как имя одноаргументной функции. Такая “функция”, когда это возможно, преобразует свой аргумент к своему имени-типу, например, byte(false) = 0;
3) функциями — существуют стандартные функции, задача которых привести так или иначе свой аргумент к типу возвращаемого значения, кроме того, программист может создавать подобные функции и сам.

Явным преобразованием типов можно пользоваться лишь с дискретными типами или типами-указателями. Стандартные функции-преобразователи — это trunc, round, chr и ord. Процедуры val и str также предназначены для преобразования типов — они возвращают результат через параметры-ссылки.

Константы и инициализация данных

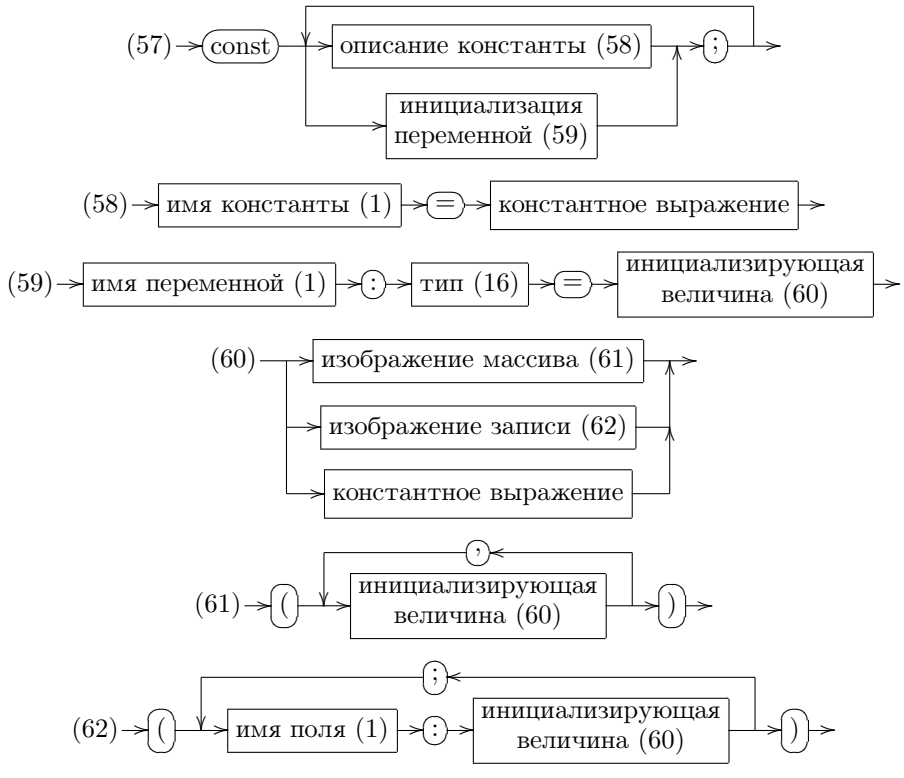
Константы бывают двух видов:

1) неименованные (литералы);
2) именованные, которые позволяют использовать идентификатор вместо литерала, например, удобнее использовать имя pi вместо самой записи числа π.

У констант нет адреса.

Именованные константы могут иметь тип. Использование таких констант с типом ничем не отличается от использования переменных. Смысл таких “констант” в том, чтобы присвоить переменным начальные значения при старте программы более компактным, чем оператор присваивания методом. По умолчанию переменным устанавливаются неопределённые значения.

Синтаксические диаграммы декларации именованных констант следующие:



Пример.

```
const
  pi = 3.141592654;
  e = 2.718281828;
  N = 3;
  copyright: string[4] = 'MATI';
  year: word = 1984;
  money: array[0..N]of longint = (3000, 2000, 1000, 5000);
  person: record name: string[8]; age: byte end = (name: 'Pole'; age: 11);
  data: array[1..2, 2..4]of byte=((1, 2, 3), (2, 3, 4));
begin
  year := 1982;
{  N := 3; (* ошибка! *)}
  writeln(year + data[1][3] + data[2,2] + person.age) (* 1982+2+2+11=1997 *)
end.
```

Динамические, статические и временные (автоматические) данные

Временные переменные существуют только тогда, когда выполняются определенные части программы; статические переменные существуют всегда, хотя на некоторых участках программы они могут быть и недоступны ("невидимы").

Все переменные подпрограмм — временные, они реально существуют только во время исполнения подпрограммы, в которой они описаны. При выходе из этой подпрограммы они уничтожаются (буквально, т.к. после выхода из подпрограммы, то место в памяти, которое они занимали, начинает использоваться для других нужд). Все переменные, описанные в первом блоке деклараций программы (глобальные), — статические.

Динамические как и временные данные не существуют в момент старта программы. Создание и уничтожение динамических данных в отличие от временных происходит не автоматически, а совершается специальными средствами в исполняемой программе.

Стандартная процедура new создает динамическую переменную, а стандартная процедура dispose уничтожает динамическую переменную.

Процедура new имеет один параметр-ссылку типа указатель. Ее выполнение заключается в выделении памяти для хранения данного, тип которого соответствует аргументу, и в присвоении аргументу значения указателя на эту память.

Процедура dispose имеет один параметр-ссылку типа указатель. Ее выполнение заключается в освобождении памяти, на которую указывает аргумент. После выполнения dispose значение ее аргумента считается неопределенным.

Рассмотрим пример.

```
var
  p: ^integer;
begin
  new(p); (* получили новую переменную! *)
  new(p); (* получили еще одну новую переменную! *)
  p^ := 7;
  writeln(p^); (* будет напечатано 7 *)
  dispose(p) (* освободить память теперь можно только от одной, созданной
                второй по счету переменной, т.к. адрес первой - утрачен и
                память, ею занимаемая, превратилась в мусор *)
end.
```

Важнейшее достоинство динамических данных — это то, что для них доступна вся оставшаяся свободная память компьютера.

Рассмотрим следующий пример. Нужно разработать программу для вычисления определителя матрицы произвольного размера. Если память выделить статически, то возможны два плохих случая:

1) заданный статически массив меньше матрицы, определитель которой нужно вычислить, хотя у компьютера достаточно памяти для помещения туда данных из этой матрицы;
2) требуется вычислить определитель матрицы очень маленького размера, но у компьютера не хватает памяти для выделения под заданный статически массив.

Использование динамически распределяемой памяти позволяет избежать таких плохих случаев при работе с данными любого типа.

Кроме того, использование динамического распределения данных совместно с комбинированным или объектным типом позволяет создавать сложные структуры данных: деревья, сети, очереди и т.п. [Основные структуры данных планируется рассмотреть подробно в следующем семестре.]

Работа с типизированными файлами

Рассмотрим стандартные операции для работы с файлами:

assign(f,s) — ставит в соответствие файловой переменной f строку s, содержащую имя файла. Эту процедуру нужно вызывать перед началом работы с любым файлом;

reset(f) — процедура, открывает файл f для чтения и записи и устанавливает позицию доступа на начальный элемент файла (в 0);

rewrite(f) — процедура, создает и открывает файл f. Если файл существовал ранее, то все его прежнее содержимое уничтожается;

close(f) — процедура, закрывает открытый файл f. Если открытый файл не закрыть перед выходом из программы и если файл изменялся, то может произойти потеря части информации файла;

read(f,v1,...,vn) — процедура, последовательно читает из файла f данные для соответствующих ему по типу однотипных переменных v1, ..., vn. Позиция доступа сдвигается на число прочитанных элементов;

write(f,v1,...,vn) — процедура, последовательно пишет в файл f значения соответствующих ему по типу однотипных переменных v1, ..., vn. Позиция доступа сдвигается на число записанных элементов;

seek(f,n) — процедура, средство прямого доступа, устанавливает позицию доступа в файле f в n;

filepos(f) — функция, возвращает текущую позицию доступа к файлу f;

filesize(f) — функция, возвращает размер открытого файла f в элементах данных файла;

eof(f) (eof — это аббревиатура от слов end of file — конец файла) — функция, возвращает true, если позиция доступа больше или равна размера файла, т.е. если достигнут конец файла, и false в противном случае;
erase(f) — процедура, уничтожает закрытый файл f;
rename(f,s) — процедура, переименовывает закрытый файл f в имя из строки s.

Пример.

```
Program DataBase;
  Type
    Person = Record Name: String; Weight: Byte End;
  Var
    f: File of Person;
    r: Person;
    Count: Word;
Begin
  (* заполнение БД *)
  Assign(f, '1st.dbs');
  Writeln('Для окончания ввода БД введите пустую строку');
  Rewrite(f);
  Repeat
    Write('Введите ФИО: ');
    Readln(r.Name);
    If r.Name = '' then break;
    Write('Введите вес: ');
    Readln(r.Weight);
    Write(f, r)
  Until False;
  Close(f);
  (* распечатка БД *)
  Reset(f);
  Count := 0;
  Repeat
    Read(f, r);
    Inc(Count);
    Writeln(r.Name, ' ', r.Weight)
  Until EOF(f);
  Writeln('Всего записей: ', Count);
  Close(f)
End.
```

Текстовые файлы

В паскале могут быть файлы аналогичные поддерживаемым в бэйсике: текстовые и бестиповые бинарные. Последние в этом курсе не рассматриваются. Текстовые файлы являются самыми распространенными и удобными для непосредственного понимания человеком.

Текстовые файлы описываются стандартным идентификатором text, например, декларация var f:text; описывает текстовый файл f. Тип text отличается от типа file of char двумя особенностями: 1) наборы операций, применимые к этим типам, не совпадают, хотя и пересекаются; 2) данные типа file of char — это поток единообразных символов, в текстовых же файлах некоторые символы и символьные комбинации, называемые управляющими, имеют особый смысл, например, последовательность из двух символов с номерами 13 и 10 является маркером конца строки в операционных системах CP/M, DOS и Microsoft Windows, любой текстовый файл разбивается такими маркерами на отдельные строки (такой маркер в Linux — это код 10).

Следующие операции работают с текстовыми файлами таким же образом, как и с типизированными: assign, close, erase, eof, rename.

Следующие операции работают с текстовыми файлами с особенностями:
reset(f) — открывает файл f только для чтения;
rewrite(f) — открывает файл f только для записи;
read(f,v1,...,vn) — вводит данные в переменные v1, ..., vn из строк файла f, автоматически производя преобразование из строкового типа в тип, соответствующий каждой из переменных. Данные в строке должны разделяться пробелами, табуляциями или концами строк, но данные в переменную строкового типа считываются целиком до конца текущей строки, т.е. до маркера конца строки;
write(f,e1,...,en) — выводит в файл f значения выражений. Каждое из e1, ..., en — это выражение вместе с опциональным спецификатором формата. В общем случае оно имеет вид E[:N[:D]], где E — это собственно выражение, N — количество позиций для вывода значения выражения, D — количество знаков после десятичной точки при выводе значения выражения вещественного типа.

Кроме того, есть еще операции, предназначенные исключительно для работы с текстовыми файлами:

readln(f,v1,...,vn) — то же, что и read, но эта процедура считывает данные строго построчно. Все данные входной строки, которым не хватит переменных игнорируются;

writeln(f,e1,...,en) — процедура, делает то же, что и write, но после вывода e1, ..., en дополнительно выводит маркер конца строки;

eoln(f) — функция, возвращает true, если текущая позиция в файле указывает на маркер конца строки, и false в противном случае.

Такие операции как seek, filepos и filesize к текстовым файлам не применимы.

Если при работе с операциями, применимыми к тестовым файлам, опускать файловую переменую, то это будет означать использование зарезервированных текстовых файлов для ввода и вывода данных, имеющих имена соответственно input и output. Файл input связан со стандартным вводом с клавиатуры, а output — со стандартным выводом на экран.

Пример.

```
(* показ содержимого файла /etc/profile на дисплее *)
var
  f: text;
  s: string[80];
begin
  assign(f, '/etc/profile');
  reset(f);
  repeat
    readln(f, s);
    writeln(s)
  until eof(f);
  close(f)
end.
```

Пример.

```
(* распечатка таблицы значений синусов чисел от 0 до 22 *)
var i: byte;
begin
  writeln('n':2, 'sin(n)':7);
  for i:=0 to 22 do
     writeln(i:2, sin(i):7:3)
end.
```

Введение в объектные типы данных

Синтаксически объекты практически идентичны записям, но они могут содержать в себе подпрограммы, средства защиты от доступа к некоторым своим компонентам и другие особенности.

Пример.

```
type
  point = object
  private  (* имена компонент объекта, следующих после этого слова,
           доступны только разработчику объекта - это невидимые компоненты *)
    x, y: integer;
  public   (* имена компонент объекта, следующих после этого слова,
      предназначены любому пользователю объекта - это видимые компоненты *)
    visible: boolean;
    procedure toggle;
    procedure move(dx, dy: integer);
    function get_x: integer;
    function get_y: integer;
end;
```

Видимые переменные объектов называют свойствами, а подпрограммы — методами.

Смысл разделения объекта на видимую и невидимую части в том, чтобы сделать недоступной ту часть объекта, которая реализует его машинное представление и которую легко повредить, испортив тем самым весь объект. Устройство, например, телевизора имеет подобную структуру: ручки регулировки, индикаторы и перемычки, использование которых требует специальных знаний, скрыты от пользователя телевизора внутри корпуса — это аналог невидимой части объекта, а ручки регулировки яркости и громкости, индикатор выбранной программы и т.п. поставлены на удобном для пользователя месте — это аналоги свойств и методов.

1) Стек (stack) — структура данных, доступ к элементам которой организуется по алгоритму “первым пришел — последним ушел” или “последним пришел — первым ушел”. По-английски этот алгоритм называется LIFO (last in — first out) или FILO (first in — last out). Последний пришедший в стек элемент называется вершиной стека.

Пример стека — это запаянная с одного конца трубка с шариками, повернутая дном вверх.

К стеку можно применять две операции: добавить элемент и взять элемент. Недопустимо брать элемент из пустого стека. Идеальный стек имеет неограниченную емкость, но память компьютера ограничена, поэтому недопустимо добавлять элемент к стеку, в котором уже нет свободного места. Попытка занесения данных в заполненный стек должна приводить к ошибке переполнения (overflow), а попытка взять данные из пустого стека должна приводить к ошибке потери значения (underflow).

В абсолютном большинстве компьютеров есть хотя бы один стек, реализуемый на аппаратном уровне. Аппаратный стек имеет фиксированную длину и иногда циклическую организацию. Последнее означает, что операции добавить и взять элемент всегда допустимы. Такой стек можно представить в виде бус с указателем на одну из бусин. При добавлении информации на бусину, указываемую указателем, записывается эта информация, а указатель сдвигается на следующую бусину. При считывании информации указатель сдвигается на предыдущую бусину и с нее считывается написанная на ней информация.

Стек очень удобен для организации исполнения подпрограмм и прерываний. При вызове подпрограммы в стек помещаются адрес следующего за оператором, вызвавшим подпрограмму, оператора (он называется точкой возврата) и, как правило, параметры подпрограммы. При выполнении последнего оператора подпрограммы из стека берется адрес точки возврата и происходит передача управления на этот адрес. Вычислительной машине нужен всего один стек для организации вызовов всех подпрограмм исполняемой программы и, кроме того, всех прерываний.

Прерывание (interrupt) — это специальная системная подпрограмма, выполнение которой происходит незаметно и независимо от программ, выполняющихся в этот же момент на компьютере. Пример: чтение книги и звонок по телефону.

Другой пример: была нажата клавиша на клавиатуре — это на IBM PC совместимых компьютерах приводит к вызову прерывания, которое заносит код нажатой клавиши в специальный буфер, из которого его может взять выполняемая программа. Без использования прерываний системе нужно было бы периодически проверять не нажата ли какая-нибудь клавиша: если проверять слишком часто, то это приведет к существенному замедлению работы программ, а если редко, то возрастет риск вообще пропустить нажатие клавиши.

Прерывания тесно связаны с аппаратурой и изучаются в курсе “ЭВМ, микропроцессорные средства, организация вычислительных систем”.

В программах стек обычно реализуется при помощи массива и индекса-указателя на вершину или при помощи списка и указателя на его элемент-вершину.

Пример реализации стека из 100 элементов, используя массив.

```
const
  stack_pointer: 1..100 = 100;
var
  s_array: array[1..100]of word;
procedure push(datum: word);
  begin
    s_array[stack_pointer] := datum;
    dec(stack_pointer)
  end;
function pop: word;
  begin
    inc(stack_pointer);
    pop := s_array[stack_pointer]
  end;
```

2) Очередь (queue) — структура данных, доступ к элементам которой организуется по алгоритму “первым пришел — первым ушел” или “последним пришел — последним ушел”. По-английски этот алгоритм называется FIFO (first in — first out) или LILO (last in — last out). Первый пришедший в очередь элемент называется ее началом, а последний — ее концом.

Пример очереди — трубка с шариками с движением в одну фиксированную сторону.

К очереди можно применять две операции: добавить элемент и взять элемент. Попытки взять элемент из пустой очереди или записать элемент в недопускающую увеличения очередь являются особыми случаями — о них нужно сигнализировать. Идеальная очередь должна быть неограниченной длины, но на практике, максимальная длина очереди является ее важнейшей характеристикой (для стека, например, его емкость важна лишь для опытных программистов).

Очередь полезна для организации взаимодействия асинхронных процессов, где она используется как буфер. Если не использовать буфер, то либо снизится скорость работы взаимодействующих процессов, либо будет происходить потеря данных. Пример: буфер клавиатуры из примера про стек (такой буфер является непременным атрибутом любой ОС). Другой пример: буфер FIFO последовательного порта IBM PC совместимого компьютера. Этот буфер при работе с модемом позволяет прикладным программам выполняться быстрее.

В следующих примерах рассмотрим как очередь может ускорить работу взаимодействующих программ или избежать потерь при передаче данных.

Пусть имеется программа А, которая передает байты данных в моменты времени 1, 2, 3, 4 и 9, и программа Б, которая обращается за байтом данных в моменты времени 4, 5, 6, 7 и 8. Программа А заносит данные в конец очереди, а Б берет их с ее начала. Если не использовать очереди, то программа А должна будет при передаче 1-го байта ждать до тех пор, пока Б не будет готова его забрать, что приведет ее к простою на 3 периода времени. Если А не будет ждать, то произойдет потеря 3 байт передаваемой информации. Программа Б будет простаивать один период, ожидая 5-й байт, и этот простой очередь устранить не поможет. Таким образом очередь может ускорить работу программы-передатчика данных.

Очередь может также ускорить и работу программы-приемника, если передатчик передает данные гораздо медленнее скорости работы приемника. Здесь ускорение происходит вследствие того, что в прерывании, получающем данные от приемника, длительность команд передачи байта данных мала по сравнению с длительностью выполнения всего прерывания. Если передавать данные по байту, то это вызовет частый неоправданный простой программы-приемника на время выполнения команд прерывания, невыполняющих передачу данных. Если укрупнять размер данных для передачи за одно прерывание, то это приведет к уменьшению среднего времени передачи одного байта и соответственно к ускорению работы программы-приемника. Очереди, как правило, используют именно для ускорения работы программы-приемника, в частности, при работе с модемом.

В программах очередь обычно реализуется при помощи массива или списка и двух указателей на ее начало и конец.

Пример реализации очереди из 16 элементов, используя массив.

```
const
  queue_start: 0..15 = 0;
  queue_end: 0..15 = 0;
var
  q_array: array[0..15]of word;
procedure put(datum: word);
  begin (* не сигнализирует о переполнении! *)
    q_array[queue_end] := datum;
    queue_end := (queue_end + 1) mod 16
  end;
function get:word;
  begin (* не сигнализирует о пустоте очереди! *)
    get := q_array[queue_start];
    queue_start := (queue_start + 1) and 15
  end;
```

3) Дек (DEQUE — Double Ended Queue — двухконцовая очередь) — структура данных, состоящая из двух соединенных стеков. Следовательно, у дека — две вершины (1-я и 2-я).

К деку можно применять 4-е операции: добавить элемент в 1-й стек, добавить элемент во 2-й стек, взять элемент из 1-го стека, взять элемент из 2-го стека. Вследствие того, что оба стека соединены, элемент, помещенный в один стек можно достать из другого. В идеале оба стека должны иметь неограниченную емкость. Недопустимо пытаться достать элемент из любого из стеков, если дек пуст. Недопустимо также добавлять элемент в дек, не имеющий свободного места.

Дек удобно использовать там, где нужны два стека. Здесь преимущество дека проявятся в случае, когда один из стеков переполнится, а в другом будет еще достаточно свободного места. Кроме того, дек можно использовать как двунаправленную очередь.

Дек как и стек или очереди реализуется обычно при помощи массива или списка.

4) Список однонаправленный (Singly-Linked List, рис. 3) — структура данных, состоящая из односторонней последовательности элементов. При работе с таким списком обычно используют, как минимум, два указателя: на первый и на текущий элементы.

К однонаправленным спискам можно применять множество операций, но лишь 4-е из них являются фундаментальными: добавить элемент в текущую позицию, удалить элемент с текущей позиции, получить доступ к следующему элементу, получить доступ к первому элементу.

Однонаправленные списки используются для хранения последовательности данных. В вычислительных системах однонаправленные списки используются очень часто, т. к. списки являются фундаментальной структурой данных.

Однонаправленные списки, как правило, реализуются, используя динамические переменные, при помощи указателей.

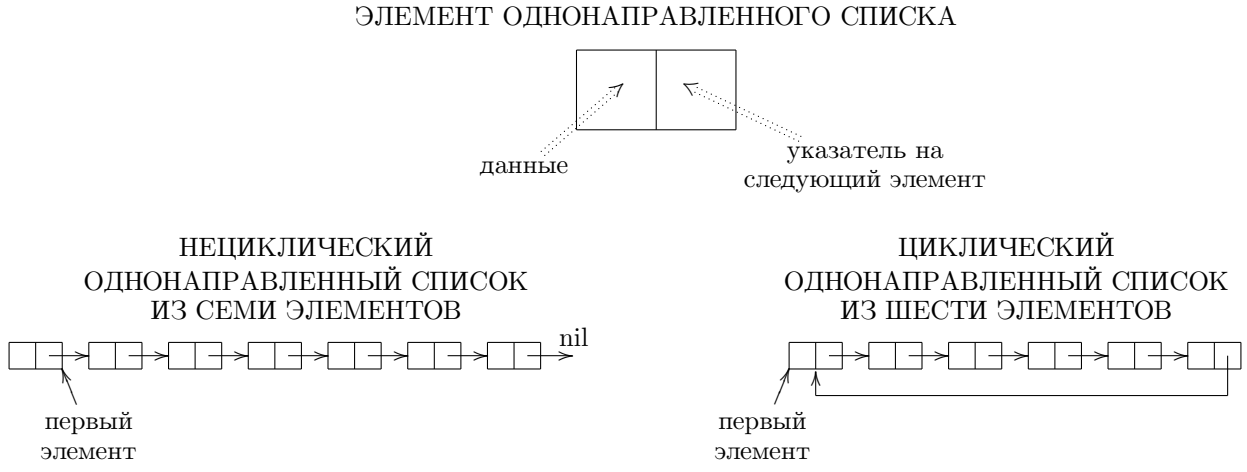


Рис. 3

Пример реализации нециклического однонаправленного списка.

```
type
  PointerToElementType = ^ElementType;
  ElementType = record
    info: word; (* данное *)
    next: PointerToElementType (* указатель на следующий элемент *)
  end;
const
  PCurrent: PointerToElementType = nil; (* указатель на текущий элемент -
    при старте программы список пуст *)
var
  PFirst: PointerToElementType; (* указатель на первый элемент *)
procedure AddElement(datum: word);
  var
    PTemp: PointerToElementType; (* вспомогательный указатель *)
  begin
    if PCurrent = nil then begin
      new(PFirst);
      PCurrent := PFirst;
      PCurrent^.next = nil
    end else begin
      new(PTemp);
      PTemp^.next := PCurrent^.next;
      PCurrent^.next := PTemp;
      PCurrent := PTemp
    end;
    PCurrent^.info := datum
  end;
procedure RemoveElement;
  var
    PTemp: PointerToElementType; (* вспомогательный указатель *)
  function PPrevious: PointerToElementType; (* поиск предыдущего элемента *)
    begin
      PTemp := PFirst;
      while PTemp^.next <> PCurrent do
        PTemp := PTemp^.next;
      PPrevious := PTemp
    end;
  begin
    if PCurrent <> nil then
      if PCurrent = PFirst then begin
        PCurrent := PFirst^.next;
        dispose(PFirst);
        PFirst := PCurrent
      end else begin
        PTemp := PPrevious;
```

```
        PTemp^.next := PCurrent^.next;
        dispose(PCurrent);
        PCurrent := PTemp
      end
  end;
procedure ToNext;
  begin
    if (PCurrent <> nil) and (PCurrent^.next <> nil) then
      PCurrent := PCurrent^.next
  end;
procedure ToFirst;
  begin
    if PCurrent <> nil then
      PCurrent := PFirst
  end;
```

Doubly

В приведенном примере следует обратить внимание на то, что нет ни типа, ни переменной, обозначающих список. То, что здесь реализован именно список, программист может отметить только в комментариях.

Часто при реализации новый элемент добавляется в начало списка.

5) Список двунаправленный (Doubly-Linked List, рис. 4) — структура данных, состоящая из двусторонней последовательности элементов. При работе с таким списком обычно используют не менее трех указателей: на первый, на последний и на текущий элементы.

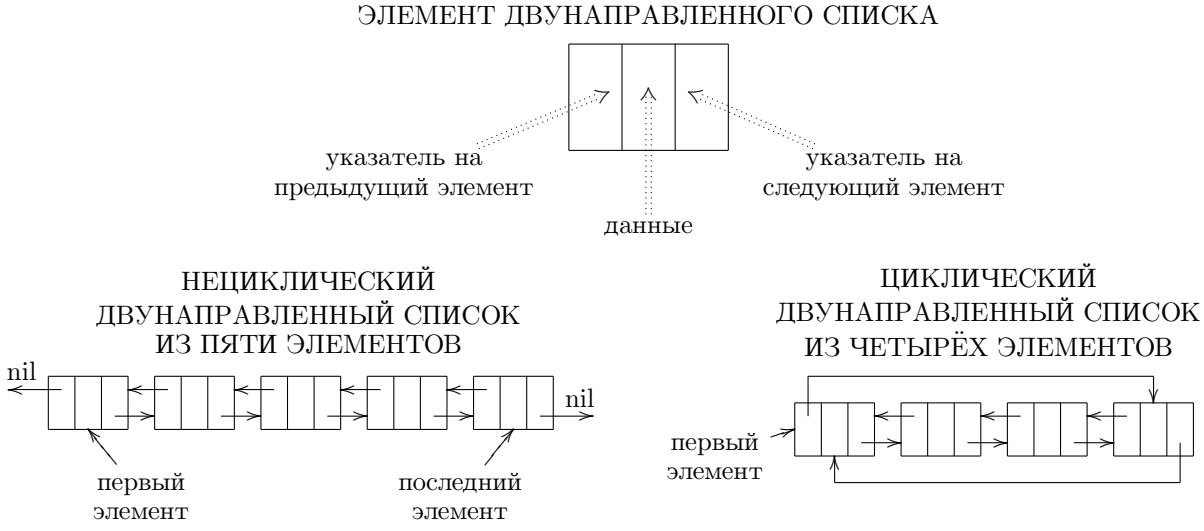


Рис. 4

К двунаправленным спискам можно применять 6 фундаментальных операций: добавить элемент в текущую позицию, удалить элемент с текущей позиции, получить доступ к следующему элементу, получить доступ к предыдущему элементу, получить доступ к первому элементу, получить доступ к последнему элементу.

Двунаправленные списки используются для хранения последовательности данных. Пример: книга. Они требуют больше памяти для реализации по сравнению с однонаправленными.

Двунаправленные списки, как и однонаправленные, реализуются обычно с использованием динамических переменных.

Пример реализации нециклического двунаправленного списка.

```
type
  PointerToElementType = ^ElementType;
  ElementType = record
    info: word; (* данное *)
    next, prev: PointerToElementType (* указатель на следующий и предыдущие
      элементы *)
  end;
const
  PCurrent: PointerToElementType = nil; (* указатель на текущий элемент -
    при старте программы список пуст *)
var
  PFirst, PLast: PointerToElementType; (* указатели на первый и последний
    элементы *)
```

```
procedure AddElement(datum:word);
  var
    PTemp: PointerToElementType; (* вспомогательный указатель *)
  begin
    if PCurrent = nil then begin
      new(PFirst);
      PCurrent := PFirst;
      PLast := PFirst
    end else begin
      new(PTemp);
      PTemp^.next := PCurrent^.next;
      PTemp^.prev := PCurrent;
      PCurrent^.next := PTemp;
      PCurrent := PTemp;
      if PCurrent^.prev = PLast then
        PLast := PCurrent
      else
        PCurrent^.next^.prev := PCurrent
    end;
    PCurrent^.info := datum
  end;
procedure RemoveElement;
  begin
    if PCurrent <> nil then
      if PCurrent = PFirst then
        if PCurrent = Plast then begin
          dispose(PFirst);
          PCurrent := nil;
        end else begin
          PCurrent := PFirst^.next;
          dispose(PFirst);
          PFirst := PCurrent
        end
      else if PCurrent = PLast then begin
        PCurrent := PLast^.prev;
        dispose(PLast);
        PLast := PCurrent
      end else begin
        PCurrent^.prev^.next := PCurrent^.next;
        PCurrent := PCurrent^.prev;
        Dispose(PCurrent^.next^.prev);
        PCurrent^.next^.prev := PCurrent
      end
  end;
procedure ToNext;
  begin
    if (PCurrent <> PLast) and (PCurrent <> nil) then
      PCurrent := PCurrent^.next
  end;
procedure ToPrevious;
  begin
    if (PCurrent <> PFirst) and (PCurrent <> nil) then
      PCurrent := PCurrent^.prev
  end;
procedure ToFirst;
  begin
    if PCurrent <> nil then
      PCurrent := PFirst
  end;
procedure ToLast;
  begin
    if PCurrent <> nil then
```

```
    PCurrent := PLast
end;
```

Главное преимущество двунаправленного списка по сравнению с однонаправленным в эффективности операции удаления элемента. Возможность обратного обхода может при реализации игнорироваться.

6) Дерево. Рассмотрим сначала бинарное дерево (binary tree, рис. 5) — структуру данных, в которой каждый узел дерева не может иметь более двух потомков. Фундаментальные операции: поиск элемента, добавление элемента, обход, удаление элемента. Указатель на предыдущий элемент может не использоваться — он полезен только при перестройке дерева или удалении элемента.

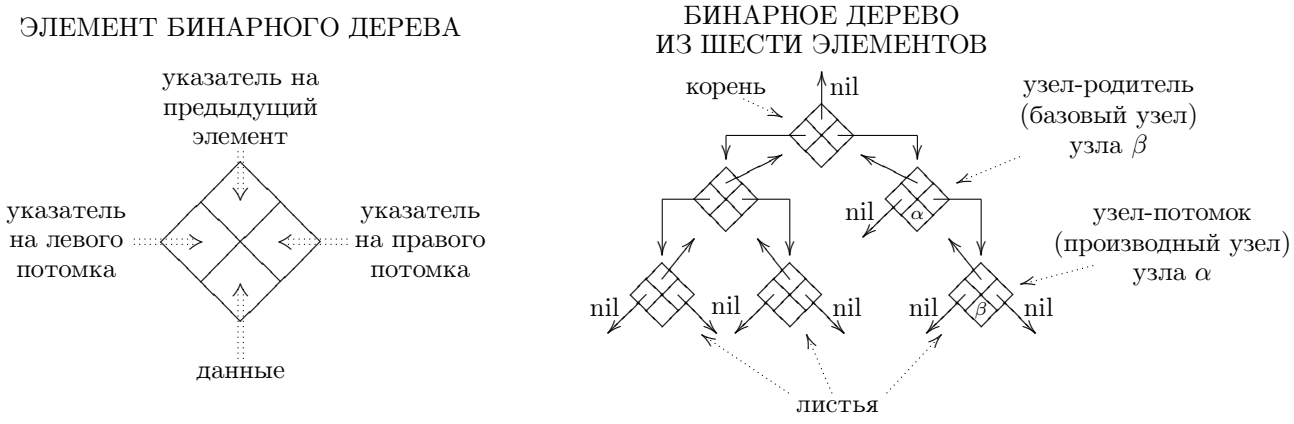


Рис. 5

Обеспечивают быстрый поиск нужных элементов. Скорость поиска в (бинарном) дереве пропорциональна логарифму количества элементов в нем.

Пример. Рассмотрим список чисел 6, 8, 4, 7, 9, 2, 5, 3. Подсчитаем сколько сравнений нужно выполнить в среднем, чтобы найти в нем некоторый элемент. Элемент 6 находится за одно сравнение, элемент 8 — за два, а элемент 5 — за 7. Итак, в среднем один элемент находится за $(1 + 2 + \cdots + 8)/8 = 36/8 = 4.5$ сравнения. Заданному списку соответствует бинарное дерево на рисунке 6 (в нем левый потомок всегда меньше родителя, а правый потомок всегда больше или равен родителю):

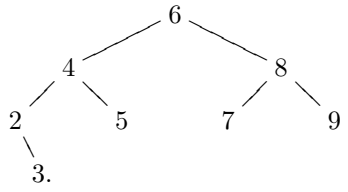


Рис. 6

Элемент 6 в нем находится за одно сравнение, элементы 4 и 8 за два, 3 — за 4, а прочие — за три. Следовательно, поиск некоторого элемента займет $(1 + 2 * 2 + 4 * 3 + 4)/8 = 2.625$ сравнений. С ростом числа элементов n отношение среднего времени поиска в бинарном дереве к времени поиска в однонаправленном списке есть функция пропорциональная log(n)/n, т.е. скорость поиска в бинарном дереве несравнимо быстрее.

Если заносить в дерево упорядоченные данные, то оно выродится в список. Сбалансированные деревья обеспечивают логарифмический поиск при любых исходных данных. Существуют несколько способов организации сбалансированных бинарных деревьев: АВЛ-деревья (AVL-tree), красно-чёрные деревья (RB-tree), деревья с подрезаемыми ветвями, ...

Рассмотрим, например, красно-чёрное дерево. В нём каждая вершина раскрашена в чёрный или красный цвет. Все листья и все потомки красных узлов — чёрные. Все пути от корня к листьям содержат одинаковое количество чёрных вершин.

Поддержание сбалансированности (перестройка дерева) требуется при каждом занесении нового элемента в дерево, но эта операция выполняется за время, пропорциональное логарифму числа узлов дерева.

2-3-деревья — это обобщение сбалансированных бинарных деревьев. В них у каждого внутреннего узла 2 или 3 потомка и все пути от корня до листьев имеют одинаковую длину. Б-дерево (B-tree) — это обобщение 2-3-дерева с любым числом потомков для узлов. Б-деревья — это основная структура баз данных.

Дерево позволяет обходить данные в прямом и обратном порядке. В этом его внешнее отличие от хэша (hash) — другой структуры данных, используемой для быстрого поиска. Проще всего обход реализуется рекурсией. Для бинарного дерева процедура обхода с параметром-узлом дерева, с начальным вызовом с параметром-корнем будет следующей.

```
procedure survey(node: pTreeNode);
```

```
  begin
     if node = nil then exit;
     survey(node^.left);
     writeln(node^.value);
     survey(node^.right)
  end;
```

Деревья можно использовать не только для быстрого поиска, но и еще, как минимум, для трёх целей: 1) для эффективного хранения данных; 2) бинарные — для представления вложенных списков; 3) для построения кодов.

Первый случай иллюстрируется следующей схемой словаря английских слов (знак $ — конец слова). Она представляет нагруженное дерево (trie, от слова re*trie*val — вы*бор*ка, по*луч*ение), иногда называемое луч или бор.

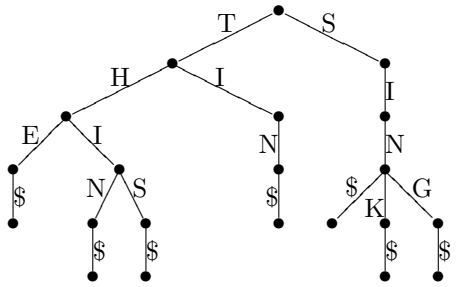


Рис. 7

Второй используется в языках, поддерживающих соответствующую структуру данных: лиспе, прологе, питоне, рубине, ...

Дерево обычно реализуют при помощи динамических переменных или массивов (файлов).

7) Хэш (hash) — структура данных, основанная на вспомогательном массиве. Каждый элемент массива состоит из данных и указателя на следующий элемент такого же типа. Фундаментальные операции: занесение, поиск и удаление элемента. Для работы с хэшем необходима определенная для него хэш-функция, которая использует данные для занесения, поиска или удаления как свой аргумент для вычисления индекса массива. После вычисления хэш-функции необходима проверка соответствия её аргумента данным в массиве по вычисленному индексу. Если они не совпадают, то производится переход к следующему элементу и опять делается проверка на совпадение. Если однонаправленный список элементов закончится, а совпадения не будет найдено, то искомых данных в хэше нет. Занесение элемента производится в начало списка, ассоциированного с найденным хэш-функцией элементом массива.

Скорость поиска данных в хэше в лучшем случае не зависит от размера хранимых в хэше данных, а в худшем вырождается в линейную зависимость от их размера. Скорость поиска данных в хэше зависит от хэш-функции и размера вспомогательного массива. Чем массив больше, тем хэш быстрее. Хорошая хэш-функция должна распределять данные равномерно по всем элементам вспомогательного массива, что невозможно для произвольных данных.

Пример реализации хэша для целых чисел.

```
type
  HashElementPtr = ^HashElement;
  HashElement = record
    data: word; (* данные *)
    next: HashElementPtr
  end;
const
  hashmax = 99;
var
  hashArray: array [0..hasmax]of HashElementPtr;
  prevHashElemPtr: HashElementPtr; (* для операции удаления элемента *)
procedure Init; (* должна вызываться при начале работы с хэшем *)
  var i: word;
  begin
      for i := 0 to hashmax do
        hashArray[i] := nil
   end;
function find(d: word): HashElementPtr;
  var
    ptr: HashElementPtr;
```

```
  begin
    prevHashElemPtr := nil;
    ptr := hashArray[d mod (hashmax + 1)];
                                      (* d mod (hashmax + 1) - хэш-функция *)
    if ptr <> nil then
      while ptr^.data <> d do begin
        prevHashElemPtr := ptr;
        ptr := ptr^.next;
        if ptr = nil then
          break;
      end;
    find := ptr
   end;
procedure AddElement(d: word);
  var
    ptr: HashElementPtr;
  begin
    new(ptr);
    ptr^.data := d;
    ptr^.next := hashArray[d mod (hashmax + 1)];
    hashArray[d mod (hashmax + 1)] := ptr
  end;
procedure RemoveElement(d: word);
  var
    ptr: HashElementPtr;
  begin
    ptr := find(d);
    if ptr <> nil then begin
      if prevHashElemPtr = nil then
        hashArray[d mod (hashmax + 1)] := ptr^.next
      else
        prevHashElemPtr^.next := ptr^.next;
      dispose(ptr)
    end
  end;
```

Вместо глобальной переменной можно для удаления элемента поддерживать двунаправленный список. Для вычисления хэш-функции может использоваться только часть данных для хранения в хэше.

Объектно-ориентированное программирование (ООП)

Реализация структур данных на ЯП наиболее естественна с использованием объектных типов, т.к. в этом случае каждой структуре данных можно сопоставить тип. При необъектном подходе структура данных — это набор разрозненных программных компонент, связь которых программист должен хранить только у себя в голове.

Использование объектных типов данных — это часть технологии ООП.

Три определения ООП:

1) расширение возможностей комбинированного типа добавлением в него подпрограмм и некоторых дополнительных особенностей;
2) инкапсуляция (защита абстракции), наследование и полиморфизм (реализует принцип “одна операция для разных типов данных”) — три краеугольных камня ООП;
3) ООП — это программирование в терминах обработки сообщений, в отличие от традиционного командного.

Первое определение чисто синтаксическое — оно скрывает подлинный смысл идеологии ООП.

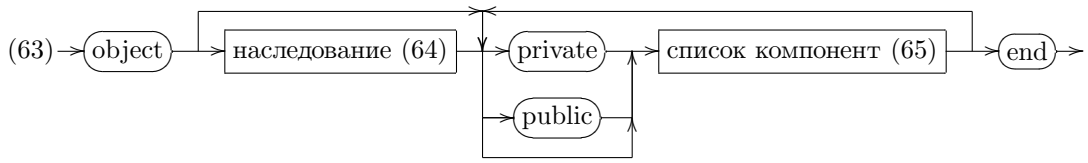
Третье определение отражает самое важное, т.е. смену идеологии программирования, но скрывает способ ее реализации.

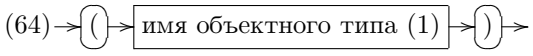
Второе определение формулирует основные проблемы, разрешаемые при использовании ООП.

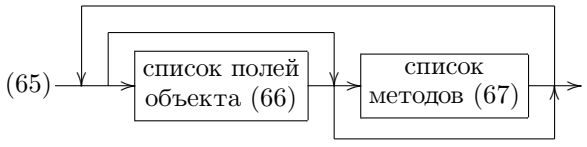
Использование объектов позволяет в программе создавать компактные структуры, адекватно отражающие моделируемые программой сущности.

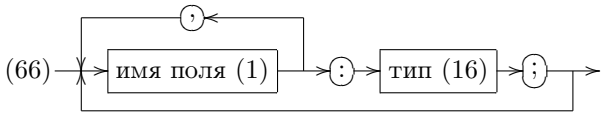
Объектный тип может описываться только в разделе описания типов. Подпрограммы объекта принадлежат не конкретному экземпляру объекта, но соответствующему объектному типу в целом.

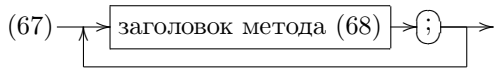
Синтаксическая диаграмма описания объектного типа:

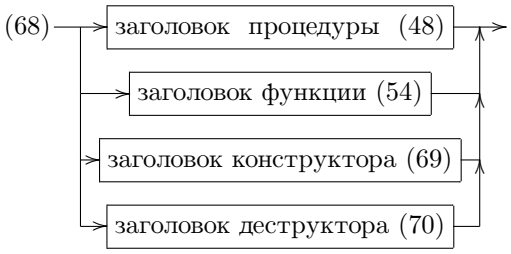




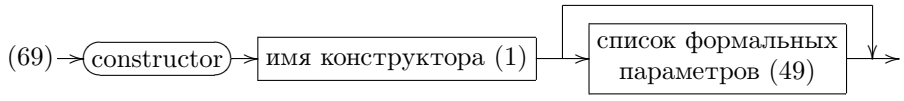








В заголовке процедуры, деструктора или функции объекта допускается директива virtual.



Итак, синтаксически объект без подпрограмм практически идентичен записи:

```
type                              type
  TPoint = object                   TPoint = record
    x, y: integer;                    x, y: integer;
    visible: boolean;                 visible: boolean
  end;                              end;
```

Синтаксически разница только в словах record и object, а также в необходимости знака “;” после каждого поля в объекте и в невозможности наличия у объекта вариантной части (см. синтаксические диаграммы). К объектам также можно применять оператор присоединения.

Параметр self

При вызове любого объектного метода в него неявно передается параметр self, который есть синоним имени экземпляра объекта, вызвавшего метод, например,

```
type
  o = object
    a: byte;
    procedure out;
  end;
procedure o.out;
  (* здесь можно подразумевать with self do *)
  begin
    writeln(a) (* вместо a можно написать self.a *)
  end;
const
  a: o = (a: 1);
  b: o = (a: 2);
begin a.out end.
```

Пример практического использования объектного типа

Рассмотрим построение объекта “точка”. Ясно, что точка имеет координаты, может быть как видима, так и невидима, точку можно переместить в новую позицию и т. п. Главное в этом примере понять идею, т. к. концепции ООП продолжает развиваться и их реализации меняются от транслятора к транслятору.

Пример представляет собой программу-клиента, которая печатает текстовые команды. Для показа графики необходимо ее вывод направить на вход соответствующей программы-сервера.

```
procedure StartGraph; (* инициализация графической системы *)
  begin
    writeln('area 320 200 darkgreen')
  end;
procedure EndGraph;
  begin
    writeln
  end;
type
  TPoint = object
    private
      x, y: integer;
      Visible: boolean;
    public
      constructor Init(x0, y0: integer); (* здесь можно использовать procedure *)
      procedure Toggle;
      procedure Move(dx, dy: integer);
      function Status:boolean;
      function ui:qword;  (* число-идентификатор объекта для граф. сервера *)
  end;
function TPoint.ui:qword;
   begin
     ui := qword(@self)
   end;
constructor TPoint.Init(x0, y0: integer);
   (* во всяком объекте, как правило, есть процедура-инициализатор *)
   begin
     x := x0; y := y0;  Visible:=false
   end;
procedure TPoint.Toggle;
  begin
    if Visible then
      writeln('delete o', ui)
    else
      writeln('point o', ui, ' ', x, ' ', y, ' red');
    Visible := not Visible
  end;
procedure TPoint.Move(dx, dy: integer);
  begin
    x := x + dx;
    y := y + dy;
    if Visible then
      writeln('move o', ui, ' ', dx, ' ', dy)
  end;
function TPoint.Status: boolean;
  begin
    Status := Visible
  end;
var  (* далее идет пример работы с описанным объектом *)
  Point1: TPoint;
begin
  StartGraph;
  Point1.Init(2, 4);
  Point1.Toggle;
  writeln('pause');    (* ожидание нажатия 1-й кнопки мышки *)
  Point1.Move(70,70);
```

```
  writeln('pause');
  EndGraph
end.
```

Т.е. получена именованная (Point1) программная модель точки на экране. Сделать такое можно только с использованием объектов. Очевидно, что саму модель можно легко совершенствовать (добавить, например, проверку на допустимость смещения в процедуре Move). При традиционном (командном, BASIC) программировании для создания модели точки пришлось бы создавать несколько именованных программных объектов, поддержание связей между которыми становилось бы большой проблемой для программиста. При ОО подходе управлять нужно не компьютером в целом, а отдельными объектами.

Объектная видимость

Раньше рассматривали видимость глобальную и локальную. Правило объектной видимости состоит в том, что поля и методы из закрытой (приватной) части объекта видны только внутри методов объекта и подпрограмм дружественных объекту. Например, поля x, y и Visible любого объекта типа TPoint доступны внутри подпрограмм-методов Init, Toggle, Move, UI и Status и нигде больше. Правило объектной видимости составляет суть концепции инкапсуляции данных ООП.

Наследование и полиморфизм

Наследование позволяет при определении новых объектных типов использовать ранее определенные объектные типы.

Например, требуется промоделировать систему, состоящей из множество различных животных. Объектный подход позволяет описать сначала объектный тип “животное”, родовой для всех животных, затем на его основе создать типы “позвоночные” и “беспозвоночные” и далее “насекомые”, “рыбы”, “кошачьи” и, наконец, “тигр”, “корова”, “кролик” и т.д. Причем развитие и детализация каждого типа происходит независимо от других.

Аналогично на базе родительского типа TPoint можно определить дочерние типы круг и далее кольцо (в разделе описания типов):

```
  TCircle = object (TPoint)
    Radius: integer; (* радиус круга *)
  end;
  TRing = object (TCircle)
    Radius2: integer;
  end;
```

Тип TCircle наследует от типа TPoint все его поля и методы. Можно, например, описав переменную Circle1 типа TCircle, вызывать для нее подпрограммы Init, Status, Toggle и Move. Ясно, что подпрограммы Init (не инициализирует размер радиуса) и Toggle (рисует точку, а не круг) для круга не годятся и потребуется их переопределение. Если внутри дочернего объектного типа определить метод с таким же заголовком как и в родительском объектном типе, то он перекроет прямой доступ к родительскому, к последнему, однако, можно будет иметь доступ специальным образом. Функцию Status и процедуру Move можно будет наследовать без изменения.

Переменным базового объектного типа можно присваивать значения производных от него типов, но не наоборот. Аналогичным образом, переменным-указателям на базовый объектный тип можно присваивать указатели на производные от него типы, но не наоборот.

Если иметь указатель на базовый тип, указывающий на производный тип, в котором перекрывается метод базового типа (в модифицированной программе-примере таким методом будет Toggle в типе TCircle), то при вызове активизируется метод базового типа. Например, если установить указатель типа ^TPoint на объект типа TCircle и затем вызвать Toggle через этот указатель, то вызовется метод объекта TPoint. Если же необходимо вызывать метод из того объекта, на который указывает указатель, то нужно, чтобы этот метод был виртуальным. Также если использовать унаследованный метод, который использует метод, переопределяемый в объекте-потомке, то для его корректного использования объектом-потомком необходимо сделать этот переопределяемый метод виртуальным. Например, если бы Move использовал Toggle, то Toggle необходимо было бы сделать виртуальным, т. к. невиртуальный Toggle приводил бы к тому, что при применении Move к кругу вызывался бы Toggle для точки.

При вызове виртуальной подпрограммы происходит вызов именно той подпрограммы, которая соответствует вызывающему ее объекту. Виртуальные подпрограммы описываются при помощи директивы virtual. Объектный тип, содержащий виртуальные подпрограммы, обязан иметь конструктор (использование деструктора в этом случае рекомендуется, но не обязательно). Использование отдельных экземпляров такого типа должно предваряться вызовом конструктора. Подпрограммы, используемые как виртуальные, должны быть помечены словом virtual как в дочерних, так и в родительском типах.

Добавим теперь объект TCircle в программу.

Сначала модифицируем исходный тип TPoint, добавив слово virtual к заголовку процедуры Toggle:

```
TPoint = object
  private
    x, y: integer;
```

```
      Visible: boolean;
    public
      constructor Init(x0, y0: integer); (* здесь constructor обязателен *)
      procedure Toggle; virtual;
      procedure Move(dx, dy: integer);
      function Status: boolean;
      function ui:longint;
  end;
```

Опишем теперь полностью тип TCircle, добавив следующие строки:

```
type
  TCircle = object (TPoint)
    private
      Radius: word; (* радиус круга *)
    public
      constructor Init(x0, y0: integer; r0: word);
      procedure Toggle; virtual;
  end;
constructor TCircle.Init(x0, y0: integer; r0: word);
  begin
    inherited Init(x0, y0); (* служебное слово inherited позволяет
                            использовать перекрытый родительский метод *)
    Radius := r0
  end;
procedure TCircle.Toggle;
  begin
    if Visible then
      writeln('delete o', ui)
    else
      writeln('circle o', ui, ' ', x, ' ', y, ' ', Radius, ' cyan');
    Visible := not Visible
  end;
```

Напишем теперь программу, использующую новый объект. Удалим раздел операторов модифицируемой программы и добавим строки:

```
var
  Circle1: TCircle;
begin
  StartGraph;
  Point1.Init(21, 41);
  Circle1.Init(20, 40, 10);
  Point1.Toggle;
  Circle1.Toggle;
  writeln('pause');
  Point1.Move(68, 38);
  Circle1.Move(70, 40);
  writeln('pause');
  EndGraph
end.
```

Виртуальные подпрограммы вызываются несколько медленнее обычных (статических) и требуют больше памяти. Но эти замедление и больший расход памяти незначительны. Современные же тенденции в программировании таковы, что скоро возможен полный отказ от невиртуальных методов в языках программирования. Подобное случилось раньше с нерекурсивными функциями.

Виртуальные методы обеспечивают полиморфизм операции. В рассмотренном примере полиморфен метод Toggle: он работает по разному для объектов типа TPoint и TCircle.

ООП позволяет создавать более конкретные типы, чем те, которые существуют в рамках необъектного программирования. Можно, например, создать тип “собака”, “вертолет” и т. п. Для сравнения можно взять стандартный тип integer, не связанный с никаким конкретным понятием: необходимая связь данных целого типа с моделируемой сущностью поддерживается только программистом лично и не отражается в конструкциях языка.

Динамическое распределение памяти и ООП

Память для данных объектного типа, как правило, выделяется динамически, т. е. при помощи стандартной процедуры new. И очень редко при помощи описания объектной переменной в разделе декларации переменных.

В известных коммерческих, научных и прочих программах подавляющая часть всех использований данных объектного типа происходит динамически.

Использование процедуры new с объектными типами имеет свои особенности. Это связано с тем, что перед использованием каждого экземпляра объектного типа, содержащего виртуальные методы, необходимо вызывать соответствующий этому типу конструктор. Процедура new позволяет в этом случае в качестве второго параметра использовать вызов конструктора.

Продолжим рассмотрение примера из предыдущих лекций: модифицируем программу так, чтобы память под объекты выделялась динамически. Сначала удалим из программы все описания переменных (разделы, начинающиеся со служебного слова var) и раздел операторов. Добавим в конец полученного текста следующие строки:

```
var
  PCircle: ^TCircle;
  PPoint: ^TPoint;
begin
  StartGraph;
  new(PPoint, Init(21, 41));   (* выделение памяти *)
  new(PCircle,Init(20, 40, 10));
  PPoint^.Toggle;
  PCircle^.Toggle;
  writeln('pause');
  PPoint^.Move(68, 38);
  PCircle^.Move(70, 40);
  writeln('pause');
  dispose(PCircle);         (* освобождение памяти *)
  dispose(PPoint);
  EndGraph
end.
```

Деструкторы

Методы-деструкторы полезны лишь для объектов, память для которых выделяется динамически. Деструктор — это обычная процедура, вызываемая перед уничтожением экземпляра объектного типа. Семантически деструктор никак не связан с конструктором: могут быть объектные типы с конструктором, но без деструктора (тип с виртуальными методами, все экземпляры которого создаются статически), а могут быть и объектные типы с деструктором, но без конструктора (тип без виртуальных методов, некоторые экземпляры которого создаются динамически). Особый эффект от вызова деструктора проявляется только в случае его использования в процедуре dispose в качестве второго параметра.

Рассмотрим пример программы, в которой использование деструктора целесообразно. Опять удалим из полученной ранее программы все описания переменных и раздел операторов и добавим в конец полученного текста следующие строки:

```
var
  PCircle: ^TCircle;
  PPoint, PFigure: ^TPoint;
begin
  StartGraph;
  new(PPoint, Init(21, 41));
  new(PCircle,Init(20, 40, 10));
  PFigure := PPoint;
  PPoint := PCircle;
  PPoint^.Toggle;   (* рисует круг! *)
  PFigure^.Toggle;  (* рисует точку *)
  writeln('pause');
  PPoint^.Move(70, 40);  (* сдвигает круг *)
  PFigure^.Move(68, 38); (* сдвигает точку *)
  writeln('pause');
  dispose(PPoint);  (* плохо *)
  dispose(PFigure);
  EndGraph
end.
```

Указатель PPoint указывает на объект типа TCircle, который занимает в памяти больше места, чем объект родительского типа TPoint (он содержит, например, поле Radius). Поэтому вызов процедуры dispose(PPoint) может освободить только часть памяти, выделенной для объекта, на который указывает PPoint. Для того чтобы

в данной ситуации произошло корректное освобождение памяти требуется использование метода-деструктора объектного типа TCircle.

Итак добавим к описаниям типов TPoint и TCircle метод Done:

```
TPoint = object
   ...
   destructor Done; virtual;
end;
destructor TPoint.Done;
  begin
  end;
TCircle = object
   ...
   destructor Done; virtual;
end;
destructor TCircle.Done;
  begin
  end;
```

После этого можно заменить плохую строку в разделе операторов на dispose(PPoint, Done), которая работает корректно. Кроме того, полезно также заменить оператор dispose(PFigure) на dispose(PFigure, Done), хотя в данном конкретном случае они эквивалентны друг другу. Таким образом, dispose, как и new, может иметь второй параметр, вызов деструктора. Однако, в случае с new второй параметр опционален, например, new(PPoint, Init(1, 2)) означает то же самое, что и два последовательных оператора new(PPoint) и PPoint^.Init(1, 2). Использование же dispose без деструктора или деструктора вне dispose не дадут эффекта корректного освобождения динамической памяти.

Вызов деструктора необходим также в случае уничтожения объекта, который выделял себе память динамически, — этот вызов должен освободить эту память. При использовании наследования деструкторы, как правило, следует делать виртуальными.

Визуальное программирование

Это технология разработки программ, основанная на идеях ООП. Концепция визуального программирования расширяет понятие объекта. Кроме обычной части-описания объекта на выбранном языке программирования, при визуальном программировании объект может содержать средства для связи с сущностями, не описываемыми на этом языке, например, с рисунками или звуками. Эти сущности называются ресурсами. Связывание программных объектов с ресурсами происходит на этапе компиляции и компоновки. Результат такого связывания, как правило, отождествляется с соответствующими программными объектами.

Первая визуальная программная система была продемонстрирована на специализированных компьютерах фирмой Xerox в 1981 году. Эта система была написана на первом объектно-ориентированном языке Smalltalk. Все современые визуальные среды создавались с ориентацией на эту систему. В 1984 году появились первые массовые компьютеры с визуальным интерфейсом пользователя, Apple Macintosh и Atari ST. Тогда же в Массачусетском технологическом институте (MIT, США) была создана экспериментальная версия системы X Window для Unix. В 1985 появились визуальная среда для IBM PC совместимых компьютеров, Microsoft Windows, и первые мультимедийные компьютеры Commodore Amiga.

Одним из первых массовых средств для написания программ по технологии визуального программирования была библиотека Turbo Vision, для языков си++ и паскаль фирмы Borland в текстовых режимах среды MS-DOS. В среде Microsoft Windows все средства для разработки программ в той или иной степени визуальные. Примерами таких сред являются Microsoft Visual Basic, трансляторы языка си++, системы Borland Delphi (Linux Kylix) и Builder.

Основные понятия визуального программирования — это видимые элементы, сообщения и невидимые объекты.

Видимый элемент — это компонент программы, который можно показать на экране. Все видимые элементы являются объектами. Рамки окон, полосы прокрутки, меню, курсор мышки — все это видимые элементы. Видимые элементы могут объединяться для формирования более сложных компонент, таких, как различные виды окон.

Основные видимые элементы (см. рис. 8), как правило, стандартизируются на уровне графической оболочки (GUI — Graphic User Interface) операционных систем, что позволяет достичь унификации способов взаимодействия программ с пользователем: последнему достаточно хорошо изучить только одну прикладную программу, чтобы понять как работать со всеми прочими. Унифицированные способы взаимодействия программы с пользователем составляют интерфейс программирования приложений или API (Application Programing Interface). Графический интерфейс пользователя называют также WIMP — Windows, Icons, Menus and Pointers.

Средства для быстрого создания визуальных программ (RAD — Rapid Application Development), автоматически генерируют код-описание объекта, соответствующего созданным программистом ресурсам. Например, достаточно только нарисовать какое-нибудь окно-диалог, а описание объекта, ему соответствующего, на паскале, си++ или другом ЯП появится в программе автоматически.

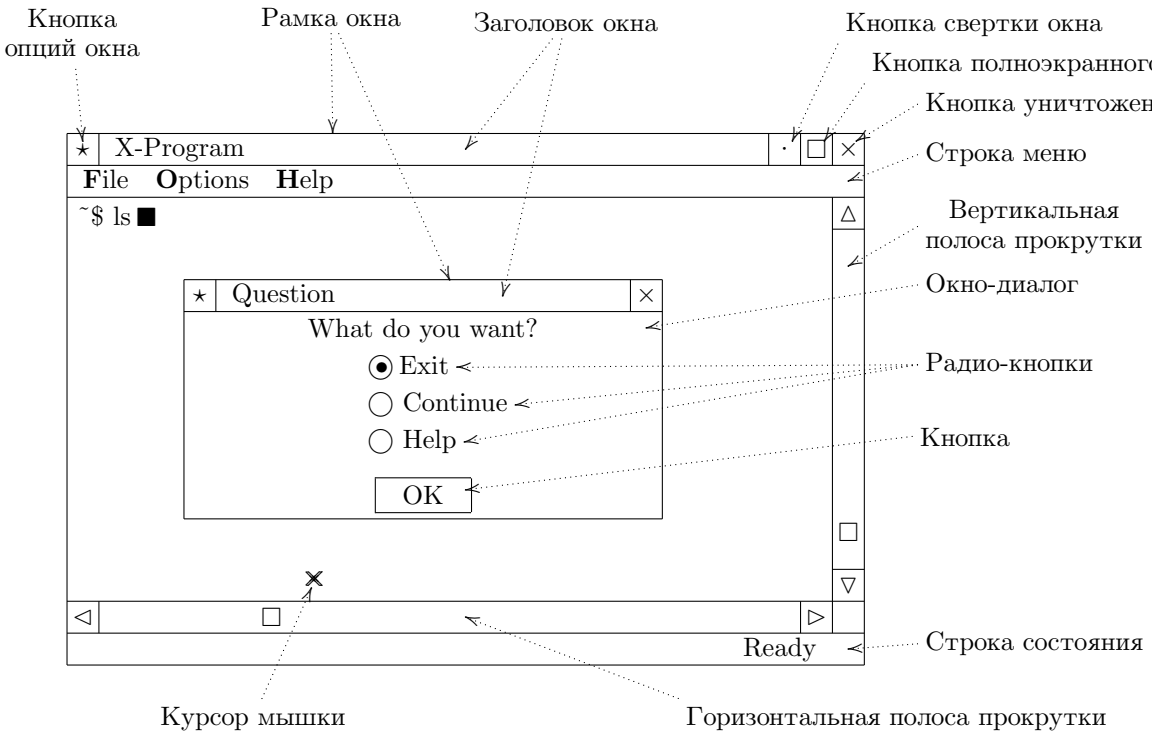


Рис. 8

Сообщение — это то, на что объект должен отреагировать. Сообщения могут приходить от клавиатуры, мышки и др. устройств или генерироваться внутри программ. Например, нажатие клавиши клавиатуры или мышки порождает соответствующее сообщение. Сгенерированные сообщения поступают в системную очередь и затем последовательно обрабатываются. Обработка сообщения состоит в его передаче всем соответствующим объектам системы. Например, нажатие клавиши мышки приведет к передаче сообщения об этом всем видимым в текущий момент на экране дисплея объектам. Каждый из этих объектов проводит анализ этого сообщения: если курсор мышки в момент нажатия клавиши не находился в зоне видимости элемента, то это сообщение игнорируется, в противном случае происходит обычно активизация и помещение этого элемента поверх других.

Невидимые объекты — это любые объекты программы, отличные от видимых элементов. Они “невидимы”, поскольку сами ничего не выводят на экран. Они производят вычисления, связь с периферией и т. п. Когда невидимому объекту необходимо вывести что-либо на экран, он должен связаться с видимым элементом. Типовые невидимые объекты как и видимые элементы могут создаваться в полуавтоматическом режиме.

Существуют технологии разработки программ с максимальным использованием средств визуализации даже при создании невизуальных компонент. Наиболее известная технология такого рода — это UML (Unified Modeling Language).

Модульное программирование

Все современные языки программирования высокого уровня при своем появлении делились на две группы: модульные и немодульные. Такие языки как фортран, си и ада были изначально модульными, а такие языки как бэйсик, алгол и паскаль изначально не поддерживали модульности. Практика показала, что модульные языки имеют целый ряд преимуществ перед немодульными. Например, программа на одном языке, поддерживающем использование модулей, может состоять из фрагментов, которые написаны на других модульных языках. Ныне как бэйсик, так и паскаль расширены средствами поддержки модульности.

Модуль — это программа или часть программы, которая проектируется, компилируется, отлаживается и используется отдельно. Модули в программах используются только в уже откомпилированном виде.

Некоторые модули, можно превратить в исполнимые программы, используя редактор связей (компоновщик, linker), но большинство модулей не предназначены для такой трансформации.

Присоединение модуля к программе означает добавление к ней набора готовых средств: подпрограмм, констант и т. п. Например, присоединение к программе модуля Dos, позволяет ей использовать средства для обработки файлов, процедуры assign, reset, rewrite, close и других.

Модуль подобно объекту в концепции ООП предоставляет пользователю набор средств, скрывая от него детали их реализации. Последнее позволяет повысить надежность программного обеспечения, т. к. модули, содержащие стандартные средства, например, для работы с графикой или базами данных, создаются программистами-профессионалами и закрываются для возможности их изменения пользователем. Использование модулей позволяет легко изменять характеристики среды программирования. Например, присоединение к программе на паскале модуля, содержащего средства для работы с матрицами, расширяет язык паскаль операциями линейной алгебры, которых нет в его стандарте.

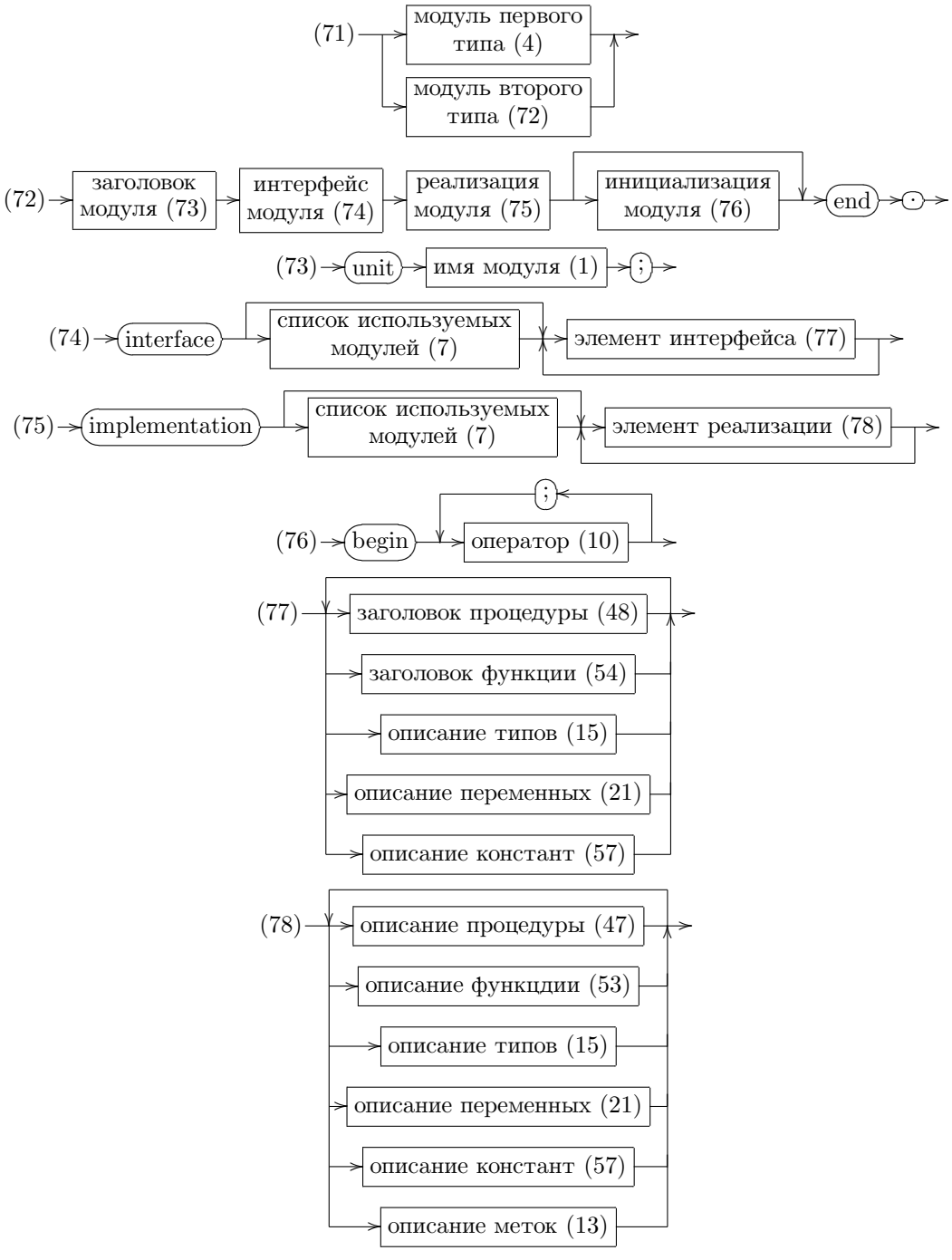
В отличии от объекта модуль реализует лишь инкапсуляцию данных. В модули часто объединяют объекты,

характеризуемые высокой степенью общности. Присоединяя такие модули к программе, получаем возможность из эти общих объектов-родителей создавать конкретные объекты-потомки.

Программные средства, предоставляемые разными модулями, можно сгруппировать в библиотеку, которая представляет собой отдельный файл. Специальные программы-библиотекари позволяют добавлять к библиотекам программные средства, удалять их из библиотек и создавать новые библиотеки. Библиотеку нельзя трансформировать в исполняемую программу. Особый класс библиотек составляют динамически связываемые библиотеки (shared library или DLL — Dynamic Link Library), широко используемые в среде современных ОС (например, Linux или Microsoft Windows): связывание с ними происходит не на этапе трансляции, а на этапе исполнения программы.

Текст модуля может быть двух типов. Первый тип — это текст модуля-программы, предназначенной для самостоятельного исполнения (до этой лекции рассматривались только такие). Второй тип — это текст программы-модуля, которую нельзя исполнить самостоятельно. Оба типа модулей могут присоединять к себе другие модули при помощи конструкции со служебным словом uses, но только 2-го типа.

Далее следуют синтаксические диаграммы описания модулей.



Пример программного модуля (файл должен иметь имя greeting):

```
Unit Greeting;
  interface (* после этого слова описываются средства, доступные
               пользователю этого модуля *)
    procedure Hello;
```

```
  implementation (* после этого слова описывается способ реализации
                    этих средств - он невидим для пользователя *)
    procedure ShowMsg;
      begin
        case random(3) of
          0: writeln('How are you?');
          1: writeln('Hi, friend!');
          2: writeln('Hello, user.')
        end
      end;
    procedure Hello;
      begin
        Randomize;
        ShowMsg
      end;
end.
```

После компиляции его (файл с расширением ppu, tpu или o) можно использовать в следующей программе, приветствующей пользователя тремя разными способами.

```
uses Greeting;
begin
  hello (* вызвать ShowMsg здесь нельзя *)
end.
```

Язык программирования лисп

Большинство языков программирования (си, бэйсик, паскаль, да и многие другие) подобно первому из них, фортрану, включают в себя средства для обработки математических выражений, синтаксис которых сложился задолго до появления вычислительных машин. Стоит напомнить, что слово фортран — это сокращение от словосочетания “транслятор формул”. Средства для организации переходов, циклов и ветвлений первоначально имели очень простую структуру и предназначались в основном для организации вычислений по сложным математическим формулам. Впоследствии, когда стало ясно, что сфера использования ЭВМ гораздо шире нежели просто трансляция формул, часть синтаксиса языков программирования, не связанная прямо с вычислениями выражений, стала бурно развиваться, что в итоге и привело к тому, что синтаксис большинства современных языков программирования разделился на две плохосвязанные части: синтаксис выражений и синтаксис операторов и описаний. Это разделение усложняет синтаксис таких языков. В развитии языка Си, языке Си++, использование объектов и связанных с ними специальных средств позволяет рассматривать синтаксис выражений как частный случай синтаксиса операторов. Такие языки программирования, как форт и лисп, изначально не имели такого разделения.

Язык программирования лисп — это, в основном, язык программирования приложений для Искусственного Интеллекта (AI — Artificial Intelligence). Удивительно, но язык лисп был вторым после Фортрана языком программирования высокого уровня в мире. Он стал использоваться уже в конце 50-х годов. Лисп был разработан Джоном Маккарти в Массачусетском технологическом институте. Первоначально лисп был задуман как теоретическое средство для рекурсивных построений, но затем он быстро превратился в весьма мощное и универсальное средство для создания самых разных программных приложений. Язык лисп обладает математической ясностью и предельной четкостью своих конструкций, что выгодно отличает его от множества других языков программирования. Большинство программ, моделирующих те или иные аспекты интеллектуальной деятельности человека, написаны на лиспе. Конкурентом лиспа является язык пролог, популярный в некоторых европейских странах и в Японии. Однако, на сегодняшний день использование языка лисп несравнимо шире.

Лисп является бестиповым языком. Это не означает, что в лиспе нет данных различных типов. Любой переменной этого языка можно присвоить любое значение: число, символ, строку символов, список телефонных номеров и даже саму программу на языке лисп, которую можно выполнить. Набор типов данных в лиспе очень широк, но тип не фиксируется за каждой переменной в отличии от таких языков как си, паскаль и бэйсик.

Программы на лиспе, как правило, не компилируются перед исполнением, а исполняются программой-интерпретатором. Но есть и компиляторы с языка лисп. Лисп является одним из немногих языков программирования, для которого создаются специальные компьютеры, приспособленные на аппаратном уровне только для работы с программами на нем. Такие компьютеры называются лисп-машинами.

Базовый тип данных языка лисп — это атом. Атом — это либо число (константа), либо знак. Знак — это любая последовательность символов, не означающая число.

Из атомов строятся списки — основной тип данных языка лисп (слово lisp — сокращение от “обработка списков”). Список в лиспе — это упорядоченная последовательность, компонентами которой являются либо атомы, либо списки (подсписки). Списки заключаются в круглые скобки, компоненты списка разделяются пробелами, например, список (a b (c d) e) состоит из четырех компонентов, один из которых подсписок, состоящий из двух атомов. Список — это многоуровневая или иерархическая структура данных, в которой открывающие и

закрывающие круглые скобки находятся в строгом соответствии. Пустой список (), который является одновременно и атомом, обозначается NIL. Он играет в лиспе роль, подобную пустому множеству в теории множеств или нулю в арифметике.

Примеры правильных списков:

```
(+ 3 6)
(добрый день сказал бородатый мужчина)
(tail nil) ;это то же самое, что и (tail ()).
```

Комментарии начинаются от символа “точка с запятой” и заканчиваются концом строки.

Списки позволяют представлять знания, например, вот описание анкетных данных человека с помощью списка

```
(анкета 101
   (имя Эдуард)
   (возраст 22)
   (язык русский)
   (приметы
      (голова грушевидная)
      (волосы редкие длинные))
    (адрес Москва)).
```

Одним из основных отличий языка лисп от большинства прочих языков программирования является запись в виде списков не только данных, но и программ. Например, список (+ 2 3) можно интерпретировать в зависимости от окружения либо как действие, дающее в результате число 5, либо как трехэлементный список.

Лисп является функциональным языком, что означает, что все вычисления на этом языке представляют собой вычисление функций. В лиспе для записи вызова функции принята специальная форма, при которой имя функции и ее аргументы записываются внутри скобок, например,

```
(+ 3 (* 5 8))    ;означает 3+5*8
(* 3 (+ 2 7) 2)  ;означает 3*(2+7)*2.
```

Любой аргумент-список функции воспринимается лисп-транслятором как вызов функции. В тех случаях, когда необходимо, чтобы аргументом был именно список используется апостроф, например, ’(f x y) — это не вызов функции f с аргументами x и y (аналога математической записи — f(x,y)).

Некоторые основные функции языка лисп: car (кар) — возвращает в качестве значения “голову” списка, cdr (кудр) — возвращает в качестве значения “хвост” списка, cons — включает новый элемент в начало списка, setq — присваивание, eval — вычисляет аргумент. Например,

```
(car ’(+ 1 2 3))   ;результат-атом - +
(cdr ’(+ 1 2 3))   ;результат-список - (2 3)
(car ’((+ 1 2) 3)) ;результат-список - (+ 1 2)
(cons (+ 1 2) ’(2 1))  ;результат-список - (3 1 2)
(cons ’(+ 1 2) ’(2 1)) ;результат-список - ((+ 1 2) 1 2)
(cons (car ’(1 2 3)) (cdr ’(1 2 3))) ;результат-список - (1 2 3)
(setq x ’(* 4 7))   ;присвоить переменной x список (* 4 7)
(eval x)            ;выполнить x, результатом будет атом 28.
```

Лисп имеет средства для создания новых функций и большие библиотеки готовых программных средств.

Критики языка лисп утверждают, что лисп это только хороший язык для обработки списков. И добавляют, что аппарат для обработки списков есть и в языке пролог, в котором обработка списков — это лишь часть среди других разнообразных средств. Эта критика была бы верной, если бы в лиспе не было функции eval, наличие которой делает лисп мощнейшим языком программирования, недосягаемым для пролога. Лисп-программа способна порождать и исполнять новые программы, которые также могут порождать и исполнять новые программы и т.д.

Наиболее известны следующие приложения, использующие лисп: система автоматизации чертежных работ AutoCAD; редактор текстов Emacs (Unix); системы символьных вычислений Reduce и Maxima. На основе лиспа создан язык программирования Scheme.

Реляционные языки программирования

Сегодня для специалиста в области программирования знание только одного языка программирования или нескольких очень похожих, например, паскаля и си, лишит такого специалиста полноты восприятия современных возможностей программирования. Незнание хотя бы в общих чертах таких языков как лисп, пролог и SQL, порождает ограниченность кругозора и соответственно резко ограничивает набор средств для решения многих прикладных задач.

Пролог, как и лисп, считается главным образом языком Искусственного интеллекта, но это универсальный язык программирования, используя его, можно решать любые задачи.

История пролога начинается с начала 70-х годов, когда стали делаться первые попытки создать язык программирования, позволяющий непосредственно использовать на компьютерах язык математической логики (пролог — это сокращение от словосочетания “программирование логики”). Первая эффективная реализация языка

пролог была создана в середине 70-х усилиями ученых-математиков из Шотландии и Франции. Ныне пролог широко используется при создании экспертных систем и баз знаний.

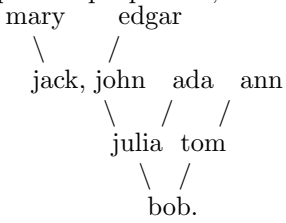
Пролог — это язык математической логики и его изучению должно предшествовать изучение большого числа математических понятий. Однако, как показала практика, знание математического фундамента пролога совсем необязательно для написания хороших программ на нем [фундаментально пролог проходят в курсе "Математической логика"].

Пролог-программа — это база данных (часто базы данных пролога называют базами знаний), которая состоит из набора фактов и правил.

Лучше всего понять то, на что похожа пролог-программа, можно рассмотрев какой-нибудь ее пример:

```
parent(mary,john).  %факт: "Мэри - родитель Джона"
parent(tom,bob).    %факт: "Том - родитель Боба"
parent(ann, tom).
parent(julia,bob).
parent(edgar,john).
parent(mary,jack).
parent(john,julia).
parent(ada,julia).
parent(edgar,jack).
```

Этот текст — база данных пролог-программы, описывающая следующие отношения:



К базе знаний можно задавать вопросы, например,

```
?- parent(mary,john).   %ответ будет "yes" (да)
?- parent(edgar,julia). %ответ будет "no" (нет)
?- parent(edgar,tom).   %ответ будет "no" (нет), т.к. в базе данных нет
                        %информации о родителях Тома.
```

Можно задавать и более содержательные вопросы, например: "Кто — родители Джона?"

```
?- parent(X,john).      %на этот запрос будут получены два
                        %последовательных ответа: X=mary и X=edgar
```

Можно задать и еще более общие вопросы, например, на вопрос: "Кто чей родитель?"

```
?- parent(X,Y). %ответом на этот вопрос будут всевозможные пары (всего 9)
                %из базы данных: X=mary, Y=jack; X=tom, Y=bob; ...
```

Можно задать вопрос о том, кто является прародителем:

```
?- parent(X,Y), parent(Y,bob). %ответами будут: X=john, Y=julia;
                               %X=ada, Y=julia; X=ann, Y=tom.
```

Запятая означает операцию логическое "И", а точка с запятой — логическое "ИЛИ". Каждое предложение в пролог-программе должно заканчиваться точкой. Комментарии следуют от знака процента до конца строки. Имена переменных должны начинаться с большой буквы, поэтому в рассматриваемой программе имена людей записаны со строчной буквы. Для записи констант-строк, начинающихся с заглавной буквы, с цифры, состоящих из нескольких разделенных пробелами слов и т.п., используют одинарные кавычки (апострофы). Например, вместо mary можно использовать 'mary', вместо Mary необходимо использовать 'Mary'. Имена предикатов должны начинаться со строчной буквы, но для них можно использовать практически любую символьную последовательность в апострофах.

Расширим исходную базу знаний следующими утверждениями:

```
male(john).   %факт: "Пол Джона - мужской"
male(edgar).
male(jack).
male(tom).
male(bob).
female(mary).
female(ann).
female(ada).
female(julia).
descendant(X,Y) :- parent(Y,X). %правило: "Для всех X и Y: X - потомок Y,
                        %если Y - родитель X" - так вводится новые
                        %отношения на основании фактов или других правил
```

```
mother(X,Y) :- parent(X,Y), female(X).
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X), X \== Y.
  %пример рекурсивного правила
ancestor(X,Y) :- parent(X,Y); parent(X,Z), ancestor(Z,Y).
```

Теперь к такой расширенной базе данных можно задать следующие вопросы:

```
?- brother(X,john). %ответ: X=jack
?- mother(X,bob).   %ответ: X=julia
?- ancestor(X,julia). %ответ: X=john, X=edgar, X=mary, X=ada.
```

Предложения пролог-программы могут содержать и командные действия, например, вывод текста на экран, которые с точки зрения логики рассматриваются как всегда истинные высказывания.

В языке пролог также есть мощные средства для обработки списков, вычисления значений выражений, поддержки ввода-вывода данных и др. Работа с пролог-программой — это постановка вопросов к базе знаний программы и расширение этой базы знаний. Такая работа похожа на разговор с экспертом в некоторой области. Если пользователь сам является экспертом, то он может научить программу-эксперта чему-то новому, если же пользователь — неэксперт, то он просто задает вопросы и получает на них ответы от программы.

Пролог называют реляционным языком или языком отношений. Отношение имеет имя и связываемые этим отношением сущности. Отношения могут быть либо фактами, либо правилами. Отношения-факты можно представлять в виде таблиц, например, отношение “родитель” из рассмотренного примера — это следующая таблица:

mary	john
tom	bob
ann	tom
julia	bob
edgar	john
...	

Таким образом, таблица — это множество строк или записей, разделенных на поля фиксированного формата.

Постановка вопросов пролог-программе очень похожа на запросы к базам данных, а сама пролог-программа — на реляционные БД.

Синтаксис, похожий на синтаксис языка пролог, используют языки меркурий (Mercury) и эрланг (Erlang).

Реляционные БД в отличие от баз знаний на прологе могут содержать только отношения, задаваемые таблицами, т.е. отношения-факты.

Язык SQL с начала 80-х годов — это стандарт в области языков управления БД.

SQL имеет средства для создания новых таблиц, заполнения таблиц записями и удаления записей из таблиц, а также организации запросов к таблицам. Средства языка SQL применяются исключительно к таблицам и результат их применения — всегда таблица. Запрос к базе данных — это, в общем случае, команда, создающая из некоторых полей записей опрашиваемых таблиц новую таблицу — результат запроса. SQL неявно тесно связан с языком математической логики. Например, для записи кванторов в SQL используют служебные слова EXISTS, ANY и ALL. SQL тесно и явно связан с теорией множеств, являясь языком манипулирования множествами. [Подробно язык SQL изучают в курсе “Базы данных”.]

Стековые языки

Стековая или постфиксная запись лаконична и легко интерпретируется для исполнения. Ее также называют польской обратной записью — RPN (reversed Polish notation). Как правило, при трансляции синтаксически традиционных языков программирования высокого уровня их сначала переводят в постфиксную форму. Поддержку стековой форме записи несложно и естественно реализовывать на уровне аппаратуры.

Например, запись выражения `sin(4+6*8)*ln(77-5*4)` в стековой форме выглядит как `4 6 8 * + sin 77 5 4 * - ln *`, т.е. скобки, запятые и большинство прочих разделителей совершенно не используются.

Польская обратная запись используется в консольном калькуляторе GNU dc, мощном калькуляторе orpie, в языках высокого уровня форт и постскрипт. Рассмотрим, например, как выглядит на постскрипте условный оператор паскаля.

```
if 5 > 4 then writeln(2*2) else writeln('Hello!'),
5 4 gt {2 2 mul} {(Hello!)} ifelse =
```

Команда `gt` (greater than) означает `>`, `mul` (multiplication) — `*`, `add` (addition) — `+`, `=` — `writeln`, `ifelse` — это условный оператор — он задействуют три верхних значения стека, т.е. это тернарная операция. Фигурные скобки группируют операторы и предотвращают их немедленное исполнение, а круглые — используются для задания строк-констант.

Определение функции для вычисления чисел Фибоначчи может выглядеть так.

```
/fib {
   dup 3 lt
     {pop 1}
     {1 sub dup 1 sub fib exch fib add} ifelse
} def
```

Для вычисления 31-го числа Фибоначчи ее можно использовать в форме `31 fib =`. Команда `dup` дублирует вершину стека, `sub` (subtraction) — это вычитание, `lt` (less than) — `<`, `pop` — отброс вершины стека, `exch` — смена мест вершины и предшествующего ей элемента в стеке.

Можно использовать не только рекурсию, но и итерацию. Например, распечатаем числа от 1 до 10 в цикле `for`, `0 1 10 {dup =} for` — в нем 1-я величина начальное значение, затем шаг и конечное значение и затем оператор, выполняемый в цикле. Вместо `for` можно использовать цикл `loop` — бесконечное повторение.
`0 {1 add dup = dup 10 ge {exit} if} loop`.

Типы данных в постскрипте — это числа, строки, массивы, словари (ассоциативные массивы), файлы, данные графики и другие. Во всех предыдущих примерах не использовались переменные — это общая особенность стековых языков. Можно связывать имена (имя может быть последовательностью из почти любых символов) со значениями, например, `/i 2 def (2x2=) = i i mul =` напечатает `2x2=4`.

Постскрипт большое внимание уделяется работе с графикой, в частности, графическим трансформациям. Он предназначен для работы со статической графикой для печати и поэтому область его применения ограничена типографским оборудованием.

Язык сценариев — рубин

Этот язык является типичным представителем названной группы языков. Отличается хорошей поддержек ООП. В нем все данные без исключения являются объектами. Например, выражение 2 + 3 означает `2.send "+", 3`, т.е. посылку двойкой сообщения “+” с аргументом 3.

Рубин использует традиционный, но чрезвычайно гибкий синтаксис, допускающий переопределение операций. Например, рассмотрим несколько способов распечатать числа от 1 до 10.

```
10.times {|i| p i + 1} #p - операция печати
(1..10).each {|i| p i}
for i in 1..10: p i end
1.upto(10) {|i| p i}
```

Рассмотрим также следующие эквивалентные условные операторы.

```
if a < 3 then b = 5 end
if a < 3: b = 5 end
b = if a < 3: 5 else b end
unless a >= 3: b = 5 end
b = 5 if a < 3
b = 5 unless a >= 3
```

А так можно распечатать “Hello, Hello, Hello, World!” — `p "Hello, "*3 + "World!"`.

Рубин как и большинство прочих языков сценариев:

- не фиксирует тип за переменными и не требует их предварительного описания;
- поддерживает ассоциативные массивы, называемые также хэшами или словарями, — в них индексом может быть величина любого типа;
- огромное внимание уделяет организации эффективной работы с текстовыми данными, в частности, поддерживает регулярные выражения;
- может эффективно работать со списками;
- имеет тысячи пакетов для поддержки решения задач в самых разных областях;
- использует автоматическую процедуру (уборщика мусора) для освобождения динамической памяти;
- для работы с памятью использует ссылки, а не указатели.

Ссылки, в отличие о указателей, не требуют специальной операции (`^` в паскале) для доступа к указываемому значению — этот доступ происходит сразу. Работа с переменными-ссылками отличается от работы с переменными-значениями, используемыми в паскале, си или бэйсике. Например.

```
a = "Test"; b = a; a[0] = "B"; p b #напечатает Best!
```

Присваивание в рубине соответствует присваиванию при передаче параметра по ссылке паскаля. Если бы было нужно, чтобы `b` была независима от `a`, т.е. нужно было бы присваивание в стиле паскаля, по значению, то его синтаксис — `b = a.dup` или `b = String.new(a)`.

Возможны последовательные (`a = b = 5`) и параллельные присваивания (`a, b = 5, 7`).

Имена типов и констант должны записываться с большой буквы, а переменных — с маленькой. Имена переменных-полей объекта должны начинаться с `@`, а имена переменных полей объектного типа с `@@`. Символы начинаются с двоеточия, имена глобальных переменные с $.

Объекты данных рубина — это числа (целые, дробные, произвольной точности и комплексные), строки (массивы символов), массивы-списки, хэши, процедуры (со значением), файлы, регулярные выражения, символы (атомарные строки) и другие.

Элементы массивов могут быть разных типов. Рассмотрим примеры работы со списками-массивами.

```
a = ["Test", 400, 5.2, [7, "Best"], proc {p "ok"}, 12] #заполнение массива, нумерация с 0
p a[0], a[1], a[3][1] #прямая индексация: Test, 400, Best
```

```
p a[-1], a[-4] #обратная индексация: 12, 5.2
p a[1,2], a[3,1] #получение подмассива: [400, 5.2], [[7,"Best"]]
a[1,4][3].call #вызов анонимной процедуры - печать ok
p a[0,2] + [3, /ok/] #объединение списков - печать ["Test", 400, 3, /ok/]
p a.size #печать 4
```

Далее пример работы с хэшем.

```
h = {1 => "start", "bad" => "cold", "good" => "summer"} #начальное заполнение хэша
h["wolf"] = "grey" #добавление/изменение элемента хэша
p h[1], h["good"] # печать start, summer
```

Рубин является функциональным языком — все его операторы производят значения. Например, можно написать `a = 5; p (if a > 1: a = 7 end)*4 #напечатает 28 и установит a=7`.

Определение функции для вычисления чисел Фибоначчи.

```
def fib(n)
   if n < 3
     1
   else
     fib(n - 1) + fib(n - 2)
   end
end
```

Эквивалентное определение.

```
def fib(n)
   n < 3 && 1 || fib(n - 1) + fib(n - 2)
end  # && означает И, a || - ИЛИ
```

Определение с итерацией и параллельным присваиванием.

```
def fib(n)
   c = p = 1
   while n > 2 do
     n, c, p = n - 1, p, c + p
   end
   p
end
```

Применение метода methods к любому объекту возвращает список всех методов этого объекта, например, `p 5.methods` напечатает все операции для целых чисел.

Компоненты-поля объекта недоступны напрямую вне объекта — к ним можно иметь доступ только через методы. Объектные типы являются открытыми — их можно дополнять, т.е., определив объектный тип, к нему можно потом добавить доопределение.

Пример работы с небольшой объектной иерархией.

```
class Point
    @@count = 0  #счетчик объектов, переменная типа
    def initialize(x, y)  #конструктор
       @x = x
       @y = y
       @@count += 1   # увеличение на 1
    end
end
point = Point.new(70, 40)  #создание объекта-точки
class Point  #доопределение
    def coord
       [@x,@y]
    end
end
p point.coord  #[70,40]
class Circle < Point  #наследование
    def initialize(x, y, r)
       super(x, y)  #вызов конструктора предка
       @radius = r
    end
    def info
       @@count
    end
end
```

```
circle = Circle.new(71, 38, 10)
p circle.coord  #[71,38]
p circle.info  #2
```

Используется, в частности, в системе для разработке интернет-приложений “Рубин на рельсах” (RoR — Ruby on Rails). Эта система является одним из лидеров в реализации концепции МПК — модель, представление, контроллер (MVC — Model, View, Controller).

Элементы общей теории языков программирования

При систематическом рассмотрении ЯП можно выделить пять основных позиций для такого рассмотрения: семиотическую, авторскую, технологическую, реализаторскую и математическую.

Технологическая позиция — это позиция человека, желающего использовать или пользующегося некоторым конкретным ЯП как технологическим инструментом на каком-либо из этапов создания и использования программных изделий.

Семиотическая позиция (семиотика — это наука о знаковых системах, примерами знаковых систем являются русский язык, система дорожных знаков и т. п.) естественна для человека, знакомого с некоторыми знаковыми системами и желающего узнать, чем выделяются такие знаковые системы, как ЯП.

Авторская позиция — это точка зрения автора (или авторов) на свое произведение. Автор создает ЯП, делает его известным программистской общественности, исправляет и модифицирует свой ЯП с учетом поступающих предложений и критических замечаний.

Математическая позиция — это точка зрения математика, рассматривающего ЯП как формальную систему — математическую модель некоторой сущности.

Реализаторская позиция возникает у тех, кто делает данный ЯП доступным для реального использования. Реализаторы не только создают трансляторы с данного ЯП, но и пишут к ним методические руководства, обучающие и контролирующие программы и т. п.

Рассмотрим более подробно ЯП с технологической позиции.

Как и всякое производство создание программ измеряется прежде всего своей эффективностью. Измерять эффективность того или иного производства разумно лишь по отношению к его цели или конечному результату. Цель производства программ — это не создание программ самих по себе, а предоставление программных услуг. Другими словами, программирование нацелено на обслуживание пользователей. А всякое обслуживание должно руководствоваться принципом: “Клиент всегда прав”. В применении к программированию этот принцип означает, что программы должны быть дружественными по отношению к пользователю. Последнее означает, что они должны, во-первых, — быть надежными, т. е. содержать минимум ошибок, во-вторых, сохранять работоспособность в неблагоприятных условиях эксплуатации, в-третьих, — содержать подсказки по способу использования своих возможностей и уметь объяснять ошибки пользователя, и, в-четвёртых, — иметь дружественный пользователю интерфейс.

Известно, что создание программ и предоставление других связанных с этим услуг является очень дорогим и относительно длительным делом, в котором трудно гарантировать высококачественный результат. Носит ли причина такого положения субъективный характер или связана с самой природой программирования? В настоящее время на этот вопрос можно ответить так: “Сложность — основная проблема программирования, она связана с самой его природой. Сложность программирования в некоторой области понижается по мере большего освоения классов задач, характерных для данной области”.

Первый источник сложности заключается в следующем. Компьютеры работают весьма быстро и набор их операций позволяет решать самый широкий круг задач. В таких условиях возникает принципиальная возможность настроить компьютер на предоставление услуг, очень далеких от элементарных возможностей, заложенных в аппаратуру. Для этого достаточно снабдить компьютер соответствующей программой. Но программа, реализующая сложные услуги, должна содержать огромное количество элементарных операций над огромным количеством элементарных объектов. Уже существуют программы, состоящие из миллионов команд. Но главная проблема — это огромное количество связей между программными объектами. Между тем возможности человека работать с взаимосвязанными объектами очень ограничены. В качестве ориентира при оценке этих возможностей указывают обычно на так называемое “число Ингве”, равное 7 ± 2. Другими словами, человек не в состоянии эффективно работать с объектом, в котором более семи компонент с произвольными связями. Итак, первый источник сложности в программировании — так называемый семантический разрыв — разрыв между уровнем и характером элементарных машинных объектов и операций, с одной стороны, и потенциально возможных услуг — с другой. Иными словами, эта проблема согласования масштаба — ювелирными инструментами предполагается строить города.

Для борьбы с такого рода сложностью для выбранного класса услуг выделяются характерные объекты и операции, в терминах которых и ведется написание программ. Сами же эти объекты и операции реализуются на уровне более приближенном к элементарному. Фактически такой подход означает создание адекватного данному классу услуг ЯП, который называется проблемно-ориентированным ЯП или ПОЯ. Использование подпрограмм, модулей, объектно-ориентированного подхода в обычных универсальных ЯП дает возможность создавать объекты и операции, свойственные решаемой задаче, из элементарных, свойственных выбранному ЯП, т. е. создавать ПОЯ.

Второй источник сложности состоит в отсутствии в компьютерах модели реального мира, согласованной с представлениями о мире у программистов и у пользователей. В общем случае компьютер не в состоянии контролировать указания программиста или действия пользователя с прагматической точки зрения, т. е. контролировать соответствие между действиями и теми целями, ради которых эти действия совершаются. Из-за этого самая мелкая с точки зрения создателя программы ошибка может привести к совершенно непредсказуемым последствиям. Хорошо известен классический случай краха одной из американских космических программ из-за того, что в программе на Фортране точка была перепутана с запятой.

С подобными сложностями борются развитием методов представления в компьютере знаний о реальном мире и эффективном учете этих знаний при создании и исполнении программ. Реальным примером таких методов является использование концепции типов данных.

Программное обеспечение (ПО) — это группа взаимодействующих друг с другом программ. Другое название ПО — МО — математическое обеспечение.

Типы ПО

Все ПО делится на три типа:

1. Прикладное — это программы, фактически выполняющие поставленную перед ними задачу, например, печать платежных ведомостей, инвентаризацию, резервирование билетов, прокладку маршрутов;
2. Системное — это программы, которые выполняются одновременно с прикладными программами. Системное обеспечение управляет ресурсами вычислительной машины, т. е. дисками, оперативной памятью, лентами, центральным процессором и т. п. Операционные системы и драйверы устройств попадают в эту категорию. Для некоторых архитектур вычислительных систем СУБД — это системное ПО;
3. Инструментальное — это программы, которые помогают программистам создавать ПО. Среди такого рода программ можно назвать трансляторы и отладчики.

Способы конструирования программ

Существует две основные традиционные технологии разработки программ, “снизу-вверх” и “сверху-вниз”.

При использовании технологии “сверху-вниз” требующая решения сложная задача разбивается на небольшое количество (как правило, не более 7–8) подзадач. Каждой такой подзадаче обычно соответствует подпрограмма. Если все или некоторые из полученных таким образом подзадач всё ещё слишком сложны для непосредственного программирования, то эти сложные подзадачи рассматривают как отдельные задачи, требующие решения, и опять разбивают на группу подзадач... Этот процесс продолжают до тех пор, пока каждая из полученных подзадач не станет достаточно простой для непосредственного программирования отдельной подпрограммой.

Технология “снизу-вверх” строится на принципах, обратных описанным: сначала создается большое количество типовых программ-модулей, из которых затем собирается программа, решающая поставленную задачу. Такой подход обладает одним неисправимым недостатком: он неуниверсален, т. к. невозможно заранее заготовить модули для решения любой задачи. Но при программировании решения задач в ограниченной области, технология “снизу-вверх” дает очень хорошие результаты.

Обычно программы создаются с использованием обеих технологий. По методу “сверху-вниз” реализуется основной алгоритм программы, в котором используется ряд уже готовых компонент, предоставляемых типовыми для решаемой задачи модулями. Например, программа, предназначенная для исследования алгоритмов сортировки, достаточно уникальна и её поэтому нужно разрабатывать самостоятельно, используя технологию “сверху-вниз”. Но сами алгоритмы сортировки носят типовой характер, поэтому их создавать заново не надо — можно взять готовые из стандартных библиотек подпрограмм, литературы и т. п.

Современные технологии разработки программ опираются в основном на методы объектно-ориентированного и ситуационного программирования, которые, в свою очередь, опираются на традиционные технологии.

При объектно-ориентированном подходе к созданию программ сначала выделяются общие понятия, соответствующие структурам данных, используемых в программе. Этим общим понятиям сопоставляются объектные типы-вершины иерархии объектных типов программы. Далее на основе этих типов создаются типы, соответствующие более конкретным понятиям. Для конкретным данных описываются типы-наследники обобщающих типов. При объектно-ориентированном подходе разделяются процессы создания и отладки каждого объектного типа от его использования.

При ситуационном (event-based) программировании главное внимание уделяется выделению и классификации всех событий, на которые должна реагировать программа, а также всех событий генерируемых программой. Собственно программирование заключается в написании обработчиков (handlers) событий. Обычно ситуационный подход сочетают с объектно-ориентированным.

Существуют и другие технологии, например, функциональное, описательное и доказательное программирование.

При функциональном подходе программа представляет собой описание функций и вызов функции, вычисляющей результат. Этот вызов, как правило, является сложной композицией функций. Писать функциональные программы можно, например, на лиспе. Писать функциональные программы можно на практически любом языке, поддерживающем рекурсию, но “самым функциональным” считается язык Haskell.

При описательном программировании программист лишь описывает все, что ему известно о задаче, требующей решения, в специальном формальном виде. После чего задает вопрос к полученной программе. Транслятор

на основании введенных данных самостоятельно находит ответ на поставленный вопрос. Так можно работать на прологе.

Доказательное программирование применяется там, где необходимо обеспечить проверяемость создаваемых алгоритмов. Суть этого подхода может быть проиллюстрирована на примере. Пусть имеются два алгоритма: один сложный, очень эффективный и быстрый, но правильность которого спорна, и другой очень простой, медленный и неэффективный, но правильность которого не вызывает сомнения. Тогда если удаться доказать, что на любых одинаковых входных данных оба алгоритма дадут идентичный результат, то это будет означать правильность первого алгоритма. Существуют специальные ЯП, предназначенные для доказательного программирования.