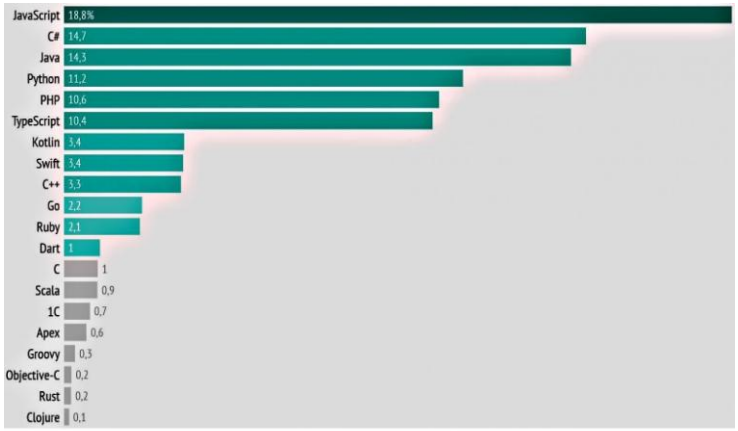
Глава 3. Алгоритмические языки

Пределы моего мира суть пределы моего языка.

Людвиг Витгенштейн

Теперь пора перейти к рассмотрению алгоритмических языков. Сейчас для практического программирования используется много различных алгоритмических языков. Периодически публикуются всевозможные рейтинги языков программирования, в основу этих рейтингов закладываются самые разные критерии. Например, можно составить рейтинг по популярности (предпочтению) языков среди программистов, рейтинг по вакансиям для программистов со знанием разный языков и т.д. Разумеется, в этих рейтингах такие языки, как Jawa, Python, C и C++ занимают достаточно высокие места.



Использование ЯП для коммерческих продуктов, 2022 год

Вот, однако, слева показана ещё одна диаграмма применение самых популярных из языков программирования на май 2022 года (в процентах к объёму реализованного коммерческого программного обеспечения).[1] Эта диаграмма заставляет задуматься… Какой же язык выбрать для изучения первым?

Практически все эти языки программирования разрабатывались как коммерческие, и ориентированными на определённую предметную область. Это представляет определённую трудность, так как вряд ли Вы сейчас точно знаете, в какой области будете в дальнейшем работать, и на каком языке программировать. В то же время, некоторые языки разрабатывались, чтобы служить именно первым (учебным) языком программирования. Именно на таком языке проще всего освоить основные (базовые) конструкции языка, стандартные типы данных и операции над ними. Кроме того, этот язык должен допускать надёжную реализацию, он обязан выявлять многое ошибки как на стадии компиляции, так и во время выполнения программы. Именно таким и является язык Паскаль.


Никлаус Вирт

Мы будем изучать два языка: стандарт Паскаля [1] и его конкретную реализацию язык Free Pascal [4,8,9]. Язык Паскаль был разработан Никлаусом Виртом в 1969 году, одной из главных целей считалось обучение учащихся структурному программированию. Последняя версия стандарта этого языка ISO/IEC 7185:1990 принята в 1990 году. На основе Паскаля разработаны и другие языки (Модула-2, Delphi, Free Pascal).

Стандарт Паскаля является *абстрактным* алгоритмическим языком, его, как и, например, машину Тьюринга, нельзя реализовать. Однако, если машину Тьюринга нельзя реализовать из-за её бесконечной ленты, то со стандартом Паскаля ситуация иная. Дело в том, что, в стандарте Паскаля многие его конструкции специально оставлены *не доопределенными*. Например, там сказано, что длина имени ограничена, но не сказано, *какова* его максимальная длина. Написано, что целочисленный тип имеет конечное число числовых констант, но не сказано, какое именно. «Открытым текстом» говорится, что результаты некоторых операций *не определены*. Например, если при сложении двух чисел результат выходит за допустимый диапазон, то этот результат не определён и т.д.

Всё это надо понимать следующим образом. Каждый, кто будет делать конкретную реализацию Паскаля, во всех этих случаях должен чётко указать, что у него будет происходить. Например, в язы-

---

[1] Как видим, языка Паскаль (и его производных) на этой диаграмме нет совсем, но и так любимый многими язык C тоже занимает только символический 1%, а язык C++ только 3,3%.

ке Free Pascal длина имени ограничена 127 символами, в целом типе `smallint` ровно 2^{16} чисел,[1] при выходе суммы за допустимый диапазон результат определяется текущим режимом работы и т.д.[2]

3.1. Вид программы на Паскале

Написать правильную программу сложно. Для этого необходимы знания и умения, которые молодые программисты ещё не успели приобрести. А чтобы приобрести их, требуется мыслить и анализировать, на что у многих программистов просто нет времени .Это требует такой самодисциплины и организованности, которые не снились большинству программистов. А для этого нужно испытывать страсть к профессии и желание стать профессионалом.

Роберт С. Мартин
«Чистая архитектура. Искусство разработки программного обеспечения»

Программа является записью алгоритма на языке программирования, это просто слово в некотором алфавите. Структура этого слова отличается для разных языков программирования. Например, для одного из первых языков Алгола-60, вся программ – это одна (длинная) строка. А вот для большинства других языков программа является текстом (или, как более привычно говорить для программистов, текстовым файлом).[3] По определению текстом является (непустая) последовательность строк символов ограниченной длины. К сожалению, русский язык здесь оказался не на высоте, одним словом «строка» обозначаются разные вещи. В английском языке всё понятно, та «неделимая» строка, которая является программой на Алголе, называется `string`, а вот каждая строка текста называется `line`.

Мы всё время сталкиваемся с текстами, но некоторые моменты всё же надо рассмотреть. Известно, что две последовательные строки текста можно объединить (склеить) в одну (если, конечно, не превышается максимальная длина строки). Например, при объединении двух строк

```
Маша ела
кашу.
```

в одну строку получается

```
Маша ела кашу.
```

Как видно, строки склеиваются через пробел. И, наоборот, разделить одну строку на две можно не в любом месте, а только по границам так называемых лексем (tokens) языка, для Паскаля мы с ними скоро познакомимся. Теперь становится понятным, почему длина имени в Алголе-60 не ограничена (вся программа одна строка), а вот в Паскале длина имени не превышает максимальную длину строки, так как иначе при склеивании строк (`lines`) внутри имени появляется пробел, которого там не должно быть.

Итак программа (например, на Паскале) состоит из символов алфавита языка. Здесь нас опять подстерегает трудность, связанная с особенностями русского языка. Например, в алфавите Паскаля есть такие символы, как `:=`, **`begin`**, **`end`** и другие, которые сами состоят из нескольких символов 🙁. Для англоязычного читателя здесь всё в порядке: символы алфавита Паскаля называются `symbol`, а символы в строке текста – `character`, так что `symbol` **`end`** состоит из трёх `character`: **`e`**, **`n`** и

---

[1] Отдельно стоит упомянуть стандартный тип `integer`. Free Pascal может работать в нескольких режимах и на разных аппаратных платформах. При включённом режиме `{$`**`mode`** `FPC}` в типе `integer` 2^{16} чисел, а в режиме `{$`**`mode`** `OBJFPC}` 2^{32} чисел.

[2] Иногда четко определить, что будет делаться в конкретной реализации тоже не удаётся, тогда будет так называемое *неопределённое поведение* программы (Undefined Behavior), о чём мы будем говорить далее.

[3] Во многих языках программирования (в том числе и в языке Free Pascal) программа может состоять из нескольких взаимосвязанных текстов (файлов), которые называются *модулями*.

d. Нам же придётся всегда чётко представлять, что мы понимаем под русскими терминами «строка» и «символ».

Итак, текст программы на Паскале есть непустая последовательность `lines`, состоящих из `character`. А уже из `character` собираются более крупные единицы языка, называемые лексемами (tokens). Это минимальные неделимые единицы языка, из которых строятся другие конструкции (как сами `symbols` языка, а также более крупные конструкции: выражения, операторы и т.д.).

В большинстве *естественных* языков более сложные языковые конструкции являются последовательностью лексем. Например, в русском языке слово может собираться из таких лексем, как приставка, корень, суффикс и окончание, в большинстве языков программирования выражения тоже строятся как цепочки лексем. В сложно устроенных естественных языках одна лексема может вставляться внутрь другой (раздвигая её символы). Например, в иврите (это государственный язык Израиля) или уже упоминавшемся языке Ифкýиль, суффикс вставляется внутрь корня (раздвигая его буквы), меняя при этом смысл слова.

3.2. Лексемы в алгоритмических языках

Лексема – языковая конструкция, по соглашению представляющая элементарную синтаксическую единицу.

ГОСТ 28397 89

Как Вы знаете, для русского языка из букв собираются такие лексемы, как корни, приставки, суффиксы, окончания и т.д. По счастью, почти во всех языках программирования лексемы практически одинаковые. Для каждой лексемы алгоритмического языка мы сначала опишем её синтаксис, потом неформальную (для человека) семантику, и, наконец, прагматику.

3.2.1. Имя

Что значит имя?
Роза пахнет розой,
Хоть розой назови её, хоть нет.
Уильям Шекспир.
«Ромео и Джульетта»

- Синтаксис лексемы имя нам уже хорошо знаком, это ограниченная (длиной строки-line) последовательность букв и цифр, начинающаяся с буквы. Стандарт Паскаля создавался в те времена, когда компьютеры были ещё маломощными, поэтому допускалось, чтобы имена *различались* друг от друга только по первым восьми символам. Например, имена `GreenGrass` и `GreenGrade` могли считаться компилятором одинаковыми. Сейчас, конечно, язык Free Pascal различает имена по всем 127 символам максимальной длины имени.
- В стандарте Паскаля всего пять объектов, которые могут иметь имена, это *константы*, *переменные*, *типы*, *процедуры и функции* и вся программа. В языке Free Pascal число объектов, которые могут иметь имена, естественно, больше.

По семантике Паскаля большие и маленькие буквы в имени не различаются (case insensitive), так что имена `Abc` и `abc` считаются одинаковыми (так же принято, например, в языках Алгол, Фортран, Lisp, именах сайтов Интернета и т.д.). В других языках (например, C, Python и т.д.) эти буквы, наоборот, различаются (case sensitive), так что программисту надо быть бдительным 🙄.

Особенно неудобно, когда (как, например, в языках C и Java) большие и маленькие буквы различаются не только в именах, придуманных программистом, но и в ключевых словах (служебных символах), так что там **`if`** служебное слово, а `If`, `IF` и `iF` – имена пользователя. Или, как в языке Java, есть имя стандартной функции `parseInt`, и написать его в виде `ParseInt` или `Parseint` нельзя ⚠. Также неудобно, когда в именах языка, например, можно было использовать только ПРОПИСНЫЕ буквы.

Все имена в Паскале делятся на три вида. Во-первых, это служебные имена, называемые также *ключевыми* именами (keywords), *зарезервированными* (reserved) или *служебными* словами. Они одновременно являются и символами (`symbol`) алфавита Паскаля. Чтобы отличать их от других имён, в текстах их обычно принято выделять **жирным** шрифтом. Всего в стандарте Паскаля 35 служебных (ключевых) имён:

and	**array**	**begin**	**case**	**const**
div	**do**	**downto**	**else**	**end**
file	**for**	**function**	**goto**	**if**
in	**label**	**mod**	**nil**	**not**
of	**or**	**packed**	**procedure**	**program**
record	**repeat**	**set**	**then**	**to**
type	**until**	**var**	**while**	**with**

Каждое такое имя имеет заранее известный смысл, изменять который программисту нельзя. В частности, нельзя этими именами называть свои переменные, константы и т.д. Постепенно мы познакомимся со смыслом всех служебных имён.

Отметим, что есть языки (например, PL/1) в котором несколько сотен ⚠ служебных слов, естественно, что «нормальные» программисты на этом языке все их не знают (полное описание этого языка содержит более 750 страниц текста). Некоторые языки (например, Фортран и упомянутый выше PL/1) допускают использования служебных имён в других смыслах. Скажем, в Фортране служебное имя **end** обозначает конец программного модуля. Программист, однако, может написать оператор присваивания `end=1`, и компилятору приходится по контексту «догадываться», что имелось ввиду. Известны случаи, когда это приводило к тяжёлым ошибкам. В языке Free Pascal программист может использовать для своих целей служебные имена, поставив впереди символ &, который имеет статус *буквы*, но только в первой позиции имени, например:

```
const &const=1; // У константы имя const
var &X,X: byte; {ОШИБКА, одинаковые имена X}
    X:=1; &X:=2; {Нет ошибки, одно имя X}
```

Далее идут стандартные (или предопределённые) имена, они тоже имеют заранее определённый смысл. Вот эти имена для стандарта Паскаля:

abs	arctan	boolean	chr	char
cos	dispose	eof	eoln	exp
input	integer	false	ln	maxint
new	odd	ord	output	pred
read	readln	real	round	sin
sqr	sqrt	succ	true	trunc
write	writeln			

Сейчас их не надо запоминать, многие из этих имён Вам уже известны, постепенно мы познакомимся с их смыслом и использованием. Стандартные имена отличаются от служебных тем, что, при необходимости, программист может *изменить смысл* этих имён. Например, имя `sin` определяет хорошо знакомую Вам тригонометрическую функцию. Поэтому, если программист станет искать корни квадратного уравнения

$$\sin^2-\sin+4=0$$

в виде $\sin_{1,2}=\ldots$, то будет зафиксирована синтаксическая ошибка, так как вещественные корни нельзя называть именем тригонометрической функции. Но, если программисту о-о-очень захочется использовать для корней именно имя `sin`, он может это имя *переопределить*, записав выше по тексту программы описание переменной

```
var sin: real;
```

Это и заставит Паскаль считать далее имя `sin` вещественной переменной. Отметим, что в большинстве языков программирования стандартных имён нет, есть только служебные имена и имена пользователя. Зачем же в Паскаль введены стандартные имена? Конечно, это сделано совсем не для того, чтобы экстравагантные программисты могли называть именем `sin` корни квадратного уравнения. Главное здесь именно в возможности *переопределения* стандартного имени. Например, если в программе надо вычислять синусы только в очень небольшом диапазоне углов, то может оказаться, что «стандартный» синус работает неэффективно, и тогда программист может написать свою функцию с этим именем, она и будет вызываться вместо стандартной. Или, например, можно переопреде-

лить стандартную процедуру вывода `Write` и заставить её выводить, скажем, комплексные значения, которые «обычная» процедура `Write` выводить не умеет.[1]

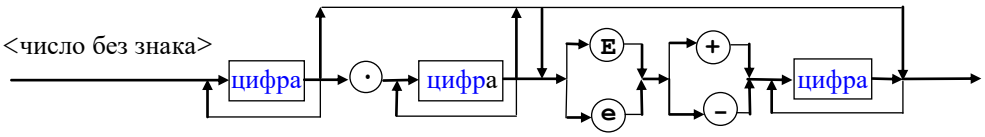
Все остальные имена в программе считаются именами пользователя. Для них действует жёсткое правило: перед использованием такого имени где-то выше по тексту программы про это имя должно быть чётко сказано, что это такое. Как говориться, имя пользователя обязательно должно быть перед использованием *описано* или *объявлено*, скоро мы узнаем, как это делается.[2]

3.2.2. Число

Все, что познаётся, имеет число, ибо невозможно ни понять ничего, ни познать без него.

Пифагор Самосский, VI век до н.э.

Следующей важной лексемой языков программирования является число. Ясно, что редко какая программа может обойтись без использования числовых значений. Мы, конечно, знаем, что числа бывают положительными и отрицательными. При формальном описании, однако, удобно сначала определить понятие «число без знака», а запись `-X` трактовать как *выражение*, т.е. значение `X`, к которому применена одноместная операция «минус». Итак, вот синтаксис понятия «число без знака» в стандарте Паскаля.



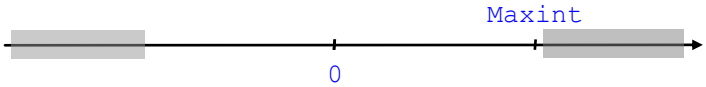
Давайте «поездим» по диаграмме и посмотрим, какие числа выводятся (т.е. считаются синтаксически правильными). Например, из этой диаграммы выводятся такие числа:

```
509
2.0777
33.102E+02
15e-10
```

Видно, что других по внешнему виду чисел не выводится. Те числа, в которые не входит ни точка, ни буквы `e` или `E`, называются целыми числами, а остальные – вещественными числами. Рассмотрим теперь *семантику* чисел. Буквы `E` и `e` по смыслу не различаются, обе они задают десятичный порядок *вещественного* числа, так что `33.102E+02`$=33.102*10^{2}$ и `15e-10`$=15.0*10^{-10}$.

В языке Free Pascal дополнительно считаются правильными вещественные числа без цифры перед порядком `E`, например, `-1.E2`. В языках Fortran и Python числа вида `.33` или `15.` тоже считаются правильными (вещественными) числами. В качестве упражнения измените диаграмму, чтобы такие числа тоже выводились. А вот, например, в языке Python между цифрами «для красоты» допускаются разделители подчеркивания, например `5_271_000_123`. Напишите синтаксическую диаграмму для целых чисел языка Python.

Ясно, что наша диаграмма позволяет выводить числа любой длины, поэтому прилагается семантический фильтр, который указывает, что правильными являются только целые и вещественные числа, попадающие в диапазоны *допустимых* (в конкретной реализации) целых и вещественных чисел. Таким образом, для целых чисел в стандарте Паскале будет такая числовая ось:



Как видно, на целочисленной оси только конечное число допустимых целых чисел, самое большое из них имеет в Паскаля стандартное имя `Maxint`.[3] К сожалению, самое маленькое целое число в стандарте Паскаля не имеет имени, более того, ниоткуда не следует, что это число равно `-Maxint`,

---

[1] Отметим, что в современных языках программирования для этого лучше использовать так называемые полиморфные имена, см. разд. 8.7.

[2] Про единственное исключение из этого правила будет рассказанов главе 13.

[3] В языке Free Pascal определена ещё стандартная константа `MaxLongint=`$2^{31}-1$.

наоборот, во всех реализациях Паскаля это неверно. Дело в том, что на ЭВМ под запись целого числа отводится некоторое количество бит, таким образом, всех целых чисел *чётное* количество. Так как в середине числовой оси стоит ноль, то оставшиеся целые числа поровну не делятся, поэтому обычно отрицательных чисел на единицу больше, чем положительных. Отсюда вытекает удивительное следствие: в языках программирования не у всякого отрицательного числа есть абсолютная величина 🙄.

Как видим привычные математические правила работы с числами в Паскале не всегда действуют, там царство особой, дискретной математики (есть специальный курс по этому предмету). Далее встаёт вопрос, а как в этой математике работают привычные нам операции над числами (сложение, вычитание, умножение)?

Например, в обычной математике для целых чисел `X` и `Y` всегда существует сумма `X+Y`. Ясно, что у нас это не так, ведь сумма может выйти за допустимый диапазон целых чисел. Стандарт Паскаля прямо говорит, что при выходе за допустимый диапазон значение суммы не определено. Как мы уже говорили, это означает, что в каждой реализации языка Паскаль должно быть чётко сказано, что в этом случае происходит. Рассмотрим, что на этот счёт сказано в языке Free Pascal.

Исполнитель алгоритма на языке Паскаль принято называть Паскаль-машиной. Это комплекс аппаратных средств (самого компьютера) и набора служебных программ. Так вот, Паскаль машина языка Free Pascal может работать в двух режимах: с контролем выхода результата за допустимый диапазон, и без такого контроля.

По умолчанию контроль выключен, для его включения предназначена специальная директива компилятора `{$R+}` (есть «длинный» синоним `{$RangeChecks ON}`), здесь главное слово Range – допустимый диапазон. Когда контроль включён `{$R+}`, то при выходе результата за допустимый диапазон возникает исключительная ситуация с выдачей соответствующей диагностики (Range check error).[1] А вот когда контроль выключен `{$R-}`, то при выходе результата за допустимый диапазон получается неправильный ответ, и Паскаль машина, как ни в чём не бывало, продолжает счёт программы 😕.

К сожалению, выдача неправильного результата операции может вызвать неопределённое поведение всей программы. Например, для условного оператора `if X+Y>0 then ... else ...` при выходе суммы `X+Y` за допустимый диапазон неизвестно, по какой ветви пойдёт выполнение программы. В таких случаях в языках программирования вводится явное понятие неопределённого поведения UB (Undefined Behavior). Это означает, что «как это будет работать толком неизвестно, но ничего хорошего не будет» 🐷. Например, на 2017 год в стандарте C и C++ было описано около 200 видов UB. В качестве часто встречающегося UB можно привести выход индекса за границы массива. Известны примеры программ, в которых из-за неопределённого поведения вызывалась функция, которая по логике алгоритма не должна была вызываться никогда.

Возникает вопрос, зачем продолжать счёт, если ответ всё равно будет неправильным? Чтобы с этим разобраться поймём, а как ЭВМ производит контроль выхода за допустимый диапазон? Ясно, что это делается с помощью каких-то машинных команд, а значит в режиме работы с контролем программа будет *больше* по размеру и работать *медленнее*. Поэтому в Паскале рекомендован такой режим: на этапе отладки контроль включается, а когда программист убедит себя, что ошибок в его программе больше нет 🙂, то контроль можно и выключить, чтобы программа была меньше по размеру и работала быстрее. Вероятно, в *надёжных* программах контроль отключать всё же не стоит. [i] [см. сноску в конце главы]

Современные компьютеры фирмы Intel могут выполнять арифметические операции над целыми числами в особом так называемым режиме с насыщением (with saturation). Эти операции выполняются на специальных векторных регистрах ЭВМ. При выходе значения такой операции за верхнюю или нижнюю допустимые границы своего типа, в качестве результата берётся эта верхняя или нижняя граница. Такая «хитрая» арифметика широко применяются при обработке мультимедийных данных (изображения и звука). Действительно, при сложении, например, двух «звуков» их громкость не мо-

---

[1] Когда нужно контролировать выход за допустимый диапазон только целых чисел (а не, скажем, ещё и вещественных), то лучше включить такой контроль директивой `{$Q+}` (синоним `{$OverflowChecks ON}`), этот контроль производится более эффективно машинными командами контроля компьютерных флагов переноса `CF` и переполнения `OF` (это тема курса по архитектурам ЭВМ).

жет выйти за некоторый предел (регулятор громкости имеет ограничение). В языке Free Pascal (только для процессоров Intel и для особых типов целых переменных `tmmxword`, необходимо подключить модуль `uses mmx;`) этот режим работы с целыми числами включается директивой `{$saturation+}`, а выключается директивой `{$saturation-}`.

Теперь посмотрим, как целочисленная ось реализована в языке Free Pascal. Когда программист работает с большим количеством чисел (например, большими массивами таких чисел), то у него возникает естественное желание отвести под них как можно меньше памяти ЭВМ. Например, если ему надо обработать массив из 100 миллионов целых чисел, все из которых в диапазоне от `-20000` до `+20000`, то под каждое такое число можно отвести всего два байта памяти ЭВМ и нет нужды, скажем, отводить по четыре или восемь байт. Исходя из этих соображений, на числовой оси выделено несколько расширяющихся подмножеств (диапазонов) целых чисел, у каждого подмножества (*типа*) своё стандартное имя.

В самом маленьком подмножестве с именем `shortint` («машинный» синоним `int8`) всего 256 чисел в диапазоне от `-128` до `+127`, под хранение каждого такого числа отводится один байт памяти ЭВМ. Следующий тип с именем `smallint` (синоним `int16`) содержит 2^{16} чисел в диапазоне от `-32768` до `+32767`, каждое число укладывается в два байта. Далее идёт целый тип с именем `longint` (синоним `int32`) с числами в диапазоне от -2^{31} до $+2^{31}-1$, помещающимися в четыре байта. Тип `integer` (единственный целый в стандарте Паскаля) в обычном режиме эквивалентен типу `smallint`, а при включённом режиме `{$mode OBJFPC}` эквивалентен типу `longint`. И, наконец, самый большой тип с именем `int64` содержит числа из диапазона -2^{63} до $+2^{63}-1$, длиной по 8 байт. Такие же диапазоны (возможно, с другими именами) есть во многих языках программирования. Как видно, программист сам выбирает поддиапазон числовой оси, который достаточен для работы с его данными.

В языке Free Pascal программист может записывать целые неотрицательные числа в других системах счисления, например, число 255:

`$FF` (16-ная система) `&377` (8-ная система) `%11111111` (2-ная система)

Для вывода чисел в этих системах счисления можно использовать стандартные функции, определённые в модуле `SysUtils`, они преобразуют целые числа из внутреннего представления в формат строки заданной длины, например:

```
uses SysUtils;
   . . .
x:=255;
write('$',HexStr(x,4)); { $00FF }
write('0x',HexStr(x,3),'h'); { 0x0FFh }
write('%',BinStr(x,9)); { %011111111 }
write('&',OctStr(x,4)); { &0377 }
```

Отдельно стоит упомянуть случай, когда все нужные программисту числа только не отрицательные или, как принято говорить, *беззнаковые*. Тогда программист может взять для работы с ними тип с именем `byte` (синоним `uint8`, т.е. `unsigned int8`) [1], содержащий числа в диапазоне `0..255`, тип `word` (синоним `uint16`) с диапазоном `0..`$2^{16}-1$, тип `longword` (синонимы `uint32`, `dword` и `cardinal`) с диапазоном `0..`$2^{32}-1$ или же тип `qword` (синоним `uint64`) с диапазоном `0..`$2^{64}-1$. [2] Числа этих типов занимают один, два, четыре и восемь байт памяти соответственно.

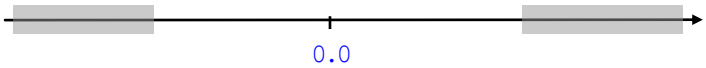
Любопытно, что есть языки программирования (например, JavaScript), в котором вообще нет целых чисел, только вещественные. Аналогично, в начальный период развития вычислительной техники существовали ЭВМ, в которых тоже были только вещественные числа. Некоторые языки (например, Python) могут работать с числами как фиксированной, так и произвольной длины, исполнитель сам выбирает, сколько памяти отвести под хранение конкретного числа. Говорят, что язык поддерживает *длинную арифметику*, правда, всё это работает ну о-о-чень медленно  .

---

[1] Так как большие и маленькие буквы не различаются, так что можно писать `Uint8`, `uInt8` и т.д.

[2] Такие синонимы описаны в разделе типов стандартного модуля `system`, например
`type dword=longint; Uint32=longword;` и т.д.

Рассмотрим теперь семантику *вещественных* чисел. Так как под его запись отводится конечное число бит, то в Паскале *конечное* число вещественных чисел. Вещественную числовую ось можно изобразить так:



Как и для целых чисел, здесь есть точка `0.0`, и диапазон чисел, ограниченный самым маленьким и самым большим вещественными числами (вне этого диапазона числовая ось изображена серым цветом) [ii] [см. сноску в конце главы]. В этом диапазоне *конечное* число представимых вещественных чисел. Эти числа распределены на числовой оси неравномерно: ближе к нулю они встречаются часто,[1] но, чем дальше от нуля, тем числа встречаются всё реже, и реже. Пусть числа `X` и `Y` представимы на нашей вещественной оси, как тогда выполняется, скажем, операция сложения `X+Y`? Когда результат выходит за допустимый диапазон, может возникнуть исключительная ситуация или получиться `±∞`. В противном случае, скорее всего, вещественное число `X+Y` *отсутствует* на нашей оси (попадает между двумя соседними точками), тогда результат *округляется* до ближайшей представимой точки. Таким образом получается, что, если результат операции над целыми числами представим, то он всегда получается точным, а вот для вещественных чисел результат будет, скорее всего, *приближённым*. [iii] [см. сноску в конце главы]

С математической точки зрения над машинными вещественными числами определено *отношение эквивалентности*: два числа эквивалентны, если они округляются в одну и ту же точку на вещественной оси ЭВМ. Таким образом, каждая точка на этой оси представляет (бесконечное) множество эквивалентных «настоящих» вещественных чисел (в этом случае множество точек на вещественной оси будет так называемым фактор множеством). В математике есть поле вещественных чисел, а в компьютерах вещественные числа не образуют ни поля, ни кольца, ни даже группы! Надо учить алгебру и матан … 🙂.

В языке Free Pascal, как и для целых чисел, предусмотрены расширяющиеся диапазоны вещественных чисел. Это тип `single` длиной 4 байта, тип `double` длиной 8 байт, тип `real` (длина зависит от реализации, обычно совпадает с типом `double`) и тип `extended` длиной 10 байт.[2]

В языке Free Pascal есть также забавный вещественный тип `currency`. Он относится к так называемым типам с фиксированной точкой. Так, в типе `currency` после десятичной точки всегда расположены четыре десятичные цифры (вычисления с такими числами округляются до 4-х цифр):

`XXXX…X , XXXX`

По существу, эти вещественные числа хранятся, умноженными на 10000, в целых числах типа `int64`, и операции над ними производятся как над целыми числами (это много быстрее, чем операции над вещественными числами). Далее, при выводе не печать, десятичная точка ставится перед 4-й десятичной цифрой. Этот тип имеет больше значащих десятичных цифр (19-20), чем в вещественных числах такого же размера типа `double` (15-16). Типом `currency` удобно пользоваться в бухгалтерских расчётах для хранения значений денежных сумм (поэтому и такое название). При подключённом модуле `uses SysUtils;` можно пользоваться функцией `CurrToStr`, преобразующей такие числа в «красивую» строку для вывода (там даже привычная запятая вместо десятичной точки: `123,4567` 🙂).

Дискретная математика, в которой работают компьютеры, сильно отличается от обычной математики, которую мы учили в школе. В частности, если коммутативный закон по сложению и умножению выполняется, т.е. всегда `a+b=b+a` и `a*b=b*a`, то ассоциативный и дистрибутивный законы не выполняются, т.е. в общем случае `(a+b)+c≠a+(b+c)` и `a*(b+c)≠a*b+a*c`.[3] Легко понять, что это происходит из-за того, что результат любой операции может выйти за допустимый диа-

---

[1] Например, в вещественном типе `single` между точками `1.0` и `2.0` располагаются `8388609` чисел.

[2] Этот тип реализован только на компьютерах фирмы Intel (и совместимых с ними).

[3] На современных компьютерах из-за особенностей машинных систем счисления ассоциативный и дистрибутивный законы выполняется для целых чисел, но не выполняется, если хотя бы одно числе в выражении вещественное. Это тема курса по архитектуре ЭВМ.

пазон. Кроме того, все действия с вещественными величинами производятся *приближённо*, так как результат операции округляется до ближайшего представимого вещественного значения.

Для того, чтобы более наглядно представить себе отличие дискретной математики от обычной, рассмотрим на вещественной прямой решение уравнения `X+A=A`. В обычной математике это уравнение имеет единственное решение `X=0.0`, а вот в дискретной могут существовать и отличные от нуля решения. Например, для типа `double` в языке Free Pascal есть корни `X=1.0` при `A=1.9E+19`, `X=1000.0` при `A=1.9E+22`, а для `A=1.9E+25` это будет уже корень $X=10^6$ 🙄. Так происходит потому, что после прибавления относительно маленького корня `X` к большой константе `A` результат округляется снова в `A`. Такие решения образно называются «грязными нулями». Легко понять, что и все числа, меньшие, чем `X`, тоже будут корнями этого уравнения.

Вещественные числа в языках программирования и сравнивать между собой надо очень осторожно, особенно числа разной длины. Например, рассмотрим на языке Free Pascal программу

```
var a: single=1/3; b: double=1/3;
    c: double;
begin c:=a;
  Writeln('a=',a:12:10);   {a=0.3333333433}
  Writeln('b=',b:17:15);   {b=0.333333333333333}
  Writeln('c=',a:17:15);   {c=0.333333343267441}
  Writeln('a=b -> ',a=b); {a=b -> FALSE}
  Writeln('b=c -> ',b=c); {b=c -> FALSE}
```

Как видим, даже «одинаковые» числа разных типов сравнивать между собой нельзя. Исходя из этого, некоторые языки программирования вводят два вида сравнения на равенство. В «обычном» сравнении (как в языке Free Pascal) две величины́ сначала приводятся к общему типу, а потом сравниваются `a=b ≡ double(a)=b`. При «строгом» сравнении величины сравниваются «как есть», без приведения к общему типу, в этом случае величины разных типов никогда не совпадут (т.к. имеют разную длину и кодировку). Например, в языке JavaScript присваивание задаётся операцией `=`, «обычное» сравнение операцией `==`, а «строгое» сравнение операцией `===` ⚠.

Некоторые языки программирования предоставляют возможность работать в 10-ой системе счисления, как привычный всем нам калькулятор (это, например, *вещественный* тип `decimal` языка Python, в примере ниже записан в виде комментария):

```
var a: single=1/3; {b: decimal=1/3;}
    Writeln(a+a+a=1.0); { FALSE }
{   Writeln(b+b+b=1.0);   TRUE  }
```

Конечно, скорость работы в таком «калькуляторе» падает на 2-3 порядка.

К сожалению, во многих учебниках по программированию о таких «ненужных тонкостях» вообще не пишут (или же авторы этих учебников о них не знают 🙂), что приводит в реальных программах к семантическим ошибкам, которые трудно выявляются при отладке.

3.2.3. Строка в апостро́фах

Круглое невежество – не самое большое зло: накопление плохо усвоенных знаний ещё хуже.

Платон. «Федр»

Следующий класс лексем представляют из себя текстовые константы, они часто используются в программах. В стандарте Паскаля это так называемая строка в апостро́фах (apostrophe), это строка, заключённая в одиночные прямые кавычки `'`). У этой лексемы простой синтаксис:

```
<строка в апострофах> ::= '<элемент строки>{<элемент строки>}...'
<элемент строки> ::= {<символ-не апостроф>|<два апострофа подряд>}
<два апострофа подряд> ::= ''
```

Как видно, вся строка заключена в апострофы, внутри которых непустая цепочка элементов. Каждый элемент является либо любым символом (`character`!), не совпадающим с апострофом, либо это два апострофа подряд. Например:

```
'abc'      'ПОД''ЕЗД'      ''''''
```

Два апострофа подряд представляют в строке один символ апостроф, ясно, что это сделано для того, чтобы не путать, где апостроф-конец строки, а где апостроф, стоящий *внутри* строки.

Проблема, как вставить символ-ограничитель начала и конца строки внутрь самой этой строки существует во всех языках программирования и решается по-разному. Например, в языке Ассемблера MASM можно заключать строку как в апострофы, так и в двойные кавычки:

```
'abc'      "abc"      "ПОД'ЕЗД"      '""'
```

В языке Си введён специальный символ \ (обратный слеш), блокирующий следующий за ним символ от анализа на принадлежность к служебным символам

```
"abc"      "ПОД\"ЕЗД"      "\\\""
```

В языке Фортран для строки может и не быть ограничителей, просто указывается её длина перед символом H (cHaracter):

```
12HПривет, мир!    7HПОД'ЕЗД    9H2␣+␣3␣=␣5
```

У каждой строки есть главная характеристика – её длина, т.е. количество входящих в неё символов. В стандарте Паскаля пустые строки, не содержащие символов, запрещены, но разрешены в большинстве других языков (включая и наш Free Pascal). Во всех языках программирования строка (`string`) не может переходить с одной строки (`line`) текста программы на другую. В основном это связано с тем, что при таком переходе внутрь строки попадает «невидимый» пробел, и длина увеличивается.

В языке Free Pascal строки могут содержать произвольные символы, заданные их номерами в алфавите, например, запись

```
#8'ABC'#13#10'DEFG'
```

определяет строку длиной 10 символов, со служебными символами `#8` (в начале строки), `#13` и `#10` (в средине).

3.2.4. Лексемы-разделители

Не в совокупности ищи единства, но более – в единообразии разделения. 🙄

Козьма Прутков

В последний класс лексем входят так называемые лексемы-разделители. Внимательно рассматривая предыдущие три класса лексем, можно заметить, что все они в тексте программы не могут вплотную примыкать друг к другу, это либо будет синтаксической ошибкой, либо изменит смысл лексемы. Например:

`32ABC` (число+имя) – ОШИБКА;
`'123'abc` (строка+имя) – ОШИБКА;
`X777` (имя+число) – новое имя `X777` и т.д.

Именно поэтому между любыми лексемами первых трёх классов должна стоять ходя бы одна лексема-разделитель (ниже мы показали их в рамочках), например:

`32`[`*`]`ABC` `'123'`[`,`]`abc` `X`[`+`]`777` и т.д.

К сожалению, язык Free Pascal позволяет не ставить лексему-разделитель между «обычной» лексемой и служебным именем (символом алфавита языка Паскаль), например, допускаются конструкции

```
32do    '123'end    и т.д.
```

Кроме того, из-за старшинства операций (см. разд. 4.1) часто можно опускать лексемы-разделители «круглые скобки», так что арифметическое выражение `i++++j` будет трактоваться как `i+(+(+(+j)))=i+j`.

К лексемам-разделителям относятся почти все знаки препинания, пробел, круглые и квадратные скобки и другие символы, с которыми мы будем постепенно знакомиться. Многие лексемы-

разделители являются символами (`symbol`) алфавита Паскаля и состоят из двух символов (`characters`), например:

```
:=    <>    >=    <=    ..    (*
```

Лексема-разделитель *пробел* имеет особый смысл: между любыми двумя лексемами можно вставить пробел без изменения смысла алгоритма. В основном это сделано для повышения читаемости программ. Как следствие, разбиение одной строки текста программы (`line`) на две допускается только по границам лексем, так как при переходе на новую строку в текст вставляется пробел.

Комментарии.

Код никогда не лжёт, а вот с комментариями такое случается.

Рон Джеффрис
Основатель методологии
экстремального программирование 😳

Особой лексемой разделителем является комментарий, по семантике он эквивалентен пробелу:

<комментарий> ::= {<текст>} | (*<текст>*)

Это единственная лексема, которая может занимать несколько строк (`line`) текста программы. Конечно, при переходе с одной строки на другую в комментарий вставляется пробел, но, так как это не влияет на смысл алгоритма, то на это можно не обращать внимания.

Большинство реализаций Паскаля считают концом комментария парную закрывающую скобку, например, {комментарий { комментарий } комментарий}, но в стандарте Паскаля такое поведение не прописано.

Наличие двух лексем-ограничителей комментария `{ }` и `(*` `*)` связано с чисто историческими причинами. Просто не на всех клавиатурах первых ЭВМ были клавиши с фигурными скобками. Интересно, что на самых «древних» клавиатурах не было и квадратных скобок, вместо них писали лексемы `(.` и `.)`, кстати, так всё ещё можно делать в языке Free Pascal, т.е. вместо `X[i]` писать `X(.i.)`.

В языке Free Pascal (и во многих других языках) есть ещё один вид так называемых *концевых* комментариев, начинающихся с лексемы-разделителя `//`, например:

```
x:=1;  // Всё до конца строки есть комментарий
```

Как ясно из их названия, лексемы-разделители отделяют одну лексему от другой. Однако, как это часто бывает, всё может иметь и противоположный смысл. Например, кто-то может сказать: «Смотри, как в кирпичной стене цементный раствор отделяет кирпичи друг от друга». На что другой может ему возразить: «Ты ничего не понимаешь, раствор соединяет кирпичи и они образуют стену». Так и в языке программирования, разделители соединяют лексемы, образуя более крупные конструкции (выражения, операторы и т.д.).

3.3. Структура программы на Паскале

Раньше назначение программ заключалось в управлении нашими вычислительными машинами, теперь назначение вычислительных машин состоит в исполнении наших программ.

Эдгар Дейкстра

Итак, рассмотрим структуру программы на стандарте Паскаля:

<программа> ::= <заголовок><блок><конец программы>

Как видим, вся программа состоит из заголовка, блока и конца программы. Проще всего описать конец программы, это символ точки:

<конец программы> ::= . [1] [комментарий]

---

[1] В языке Free Pascal после **`end.`** можно писать комментарий:

```
end. Вот и закончилась, наконец, эта длинная программа!
```

Заголовок программы имеет следующий вид:

<заголовок> ::= **program** <имя> (<имя потока>{,<имя потока>}...) ;

Как видим, заголокок – это служебное слово **program**, за которым идёт имя программы, а затем в круглых скобках стоит не менее одного имени так называемого потока ввода/вывода. Через эти потоки программа вводит все свои входные данные, и выводит результат работы, вскоре мы будем это подробно изучать.

Основной частью программы является блок, это очень важное понятие в языках программирования. Достаточно сказать, что те языки, в которых есть блоки, так и называются *блочными* языками.

```
<блок> ::= [<раздел меток>]
           [<раздел констант>]
           [<раздел типов>]
           [<раздел переменных>]
           [<раздел процедур и функций>]
           <раздел операторов>
```

Как видим, блок в Паскале состоит из шести разделов (из которых только один последний является обязательным). Чтобы понять назначение разделов, вспомним, что алгоритм состоит из шагов, которые обрабатывают данные. В языках программирования шаг алгоритма обычно называют оператором (statement), а вот английское слово operator означает *знак операции* (сложение +, умножение * и т.д.). Операторы обрабатывают данные, которые могут быть константами и переменными. Константы, в отличие от переменных, не меняют свои значения в процессе вычислений.

Так вот, в разделе констант некоторым константам программист *даёт имена* (остальные остаются безымянными (их часто называют литеральными константами (literal constant), например, `11` или `3.14159`), в разделе переменных *некоторым* переменным даются имена, остальные остаются безымянными (anonymous variables). Аналогично, в разделе типов некоторым типам (что это такое мы вскоре будем знать точно) даются имена, а в разделе процедур и функций программист даёт имена всем своим процедурам и функциям и строго описывает, что они должны делать. В дальнейшем будем использовать термин *подпрограмма* (subroutine) для общего обозначения процедур и функций (когда нам будет неважно различие между ними). [1]

Таким образом, полное название этих разделов будет: «Раздел объявления имён констант», «Раздел описания имён переменных» и т.д. В дальнейшем будем использовать термин *подпрограмма* (subroutine) для общего обозначения процедур и функций (когда нам будет неважно различие между ними).

Во многих учебниках путают или не делают различия между терминами «объявление» (declare) и описание (define).[2] Между ними есть тонкое различие, которое заключается в том, что при описании объекта (например, переменной или процедуры) ему приписываются все необходимые атрибуты и объект размещается (allocation) в памяти программы, а при объявлении размещения в памяти не происходит (объект не порождается). Например, при *объявлении* константы в Паскале

```
const N=100;
```

программист требует, чтобы всюду ниже в программе имена N заменялись на *целочисленное* значение 100, но сама константа 100 в памяти *нигде не размещается*. А вот в Фортране константы описываются, т.е. не только объявляются, но и размещаются в памяти, например:

```
real, parameter :: Pi=3.14159
```

Такое различие связано со способам передачи параметров (см. раздел 8.1). Наряду с константами, в языках могут быть и так называемое задание *эквивалентности*, например:

```
{$define N:=100} {в языке Free Pascal}
#define N 100    {в языке C}
```

---

[1] Многие языки (C++, C#, Python и другие) допускают использование безымянных (анонимных) функций, они описываются прямо в месте их использования, но в стандарте Паскале их нет.

[2] В языках C и C++ термин «описание» (declaire) переведён на русский язык как «определение».

Здесь тоже требуется, чтобы ниже по тексту программы все имена N заменялись на значение 100 (это три символа '100' !), но это не константы, ниже в программе их можно переопределить:

```
{$define N:=200} {в языке Free Pascal}
#define N 200     {в языке C}
```

Близкая к константам директива есть в Ассемблере MASM:

```
N equ 100
```

При *объявлении* процедуры

```
procedure P(x: integer); forward;
```

программист не описывает, что делает эта процедура, а просто *объявляет*, что где-то в другом месте программы (скорее всего, ниже по тексту) будет «настоящее» *описание* процедуры P (после чего она и будет размещена в программе). Во многих языках допускается и объявление переменных, например:

```
var A,B: integer external; { в языке Free Pascal }
extern int A,B;                // в языке C
```

Здесь говорится, что переменные A и B объявлены и будут использованы в данном модуле, но порождены (им отведена память) в каком-то другом программном модуле. А вот при *описании* переменных A и B (variable declarations and definition):

```
var A,B: integer; { в языке Free Pascal }
int A,B;          // в языке C
```

они объявляются (declare) и порождаются (define), т.е. им отводится (allocate) место в памяти. Важное значение имеет точка программы, в которой порождается переменная, она определяет принадлежность переменной некоторому *блоку*, о чём будем подробно говорить при изучении подпрограмм. Как видим, здесь всё сложно…☹

Рассмотрим теперь синтаксис разделов более строго.

3.3.1. Раздел меток

*Оператор **goto** сам по себе слишком примитивен; он создаёт сильное побуждение запутать программу.*

Эдгар Дейкстра.
«О вреде оператора Go To»

В разделе меток даются «имена» операторам, в стандарте Паскаля в качестве таких «имён» выступают номера, т.е. неотрицательные целые числа. Впрочем, в языке Free Pascal (и в других языках) метки могут быть и обычными именами. Метки используются только в операторе перехода **goto**, все метки должны быть предварительно описаны в разделе меток.

Сейчас в программировании укрепилось мнение, что при использовании «хорошего» стиля, который называется структурным программированием, метки использовать нельзя. Это понимал и автор Паскаля Никлаус Вирт, но исключить **goto** из Паскаля он не рискнул, так как все существующие в то время языки программирования имели этот оператор. Современные языки (например, Python, Java, Ruby или Swift), этого оператора уже не имеют, не имел его и первый функциональный язык Lisp и *самый первый* алгоритмический язык Планкалкэль (Plankalkül – Исчисление планов), разработаный немецким инженером-конструктором ЭВМ К. Цузе ещё в 1946 году ⚠.

В современном программировании (на языках высокого уровня) весьма редко встречаются случаи, когда использование **goto** даёт более короткую, эффективную и понятную программу. Ещё Дональд Кнут говорил, что изредка бывают случаи, когда эффективность оператора **goto** перевешивает его вред для понимания смысла программы. Обычно это нужно в глубоко вложенных циклах. Впрочем, для этих случаев в языки включены «замаскиро́ванные» или ограниченные (restricted) формы **goto**, это такие «безметочные» операторы перехода, как `break`, `exit` и `continue`. Современная концепция обработки ошибок **try … finally** (см. главы 16 и 17) тоже содержит «замаскиро́ванные» **goto**. При написании примеров программ мы не будем использовать метки и операторы **goto**, `break`, `exit` и `continue`.

Отметим, что в языке Free Pascal по умолчанию установлен режим работы `{$GOTO-}` или `{$GOTO OFF}`, *запрещающий* использовать метки и операторы **`goto`**. Включить использование меток и **`goto`** можно директивой `{$GOTO+}` или `{$GOTO ON}`.

Метки являются *именами операторов* и в большинстве языков это *константы*, их нельзя менять. В редких случаях (например, в языке PL/I), есть и метки-переменные, принимающие значения меток, что-нибудь вроде:

```
Met:=L1; if x<5 then Met:=L2; ... goto Met;
```

3.3.2. Раздел констант

То, что для одного человека константа, для другого – переменная.

Алан Перлис,
первый лауреат премии Тьюринга

Обычно большинство констант в программе безымянные (литеральные), но, как уже говорилось, некоторым из них можно присвоить имена в разделе констант. Опишем синтаксис, семантику и прагматику именованных констант.

<раздел констант> ::= **`const`** <имя>=<константа>;{<имя>=<константа>;}…

Как видно, в разделе констант объявлено не менее одной константы, для каждой из них задано имя и значение. Мы пока знаем, как записываются целые, вещественные и строковые (в апострофах) константы, постепенно познакомимся и с остальными типами констант (символьными, логическими, ссылочной и константами-множествами). Например:

```
const N=100; K=-1; Pi=3.14159; T='Корень X=';
```

У именованной константы следующая простая семантика. Всюду ниже по тексту программы, где встречается заданное имя, надо заменить его на указанную константу. Иногда говорят, что задана *эквивалентность* имени и значения, т.е. можно имя константы заменять на её значение и наоборот, смысл программы от этого не меняется. Стандарт Паскаля делает замену имени константы на её значение достаточно «интеллектуально», чтобы не возникало синтаксических ошибок, например:

`X:=-K; {Будет не X:=--1 (ОШИБКА),`[1] а `X:=-(-1);}`

Теперь о прагматике. Когда программисту нужно давать константе имя? Можно указать два таких случая.

1. Константа длинная, присвоив ей короткое имя, программист сократит текст своей программы. Кроме того, повышается и надежность, так как при повторном наборе константы можно и ошибиться:
   ```
   x:=3.14159; y:=3.14169;
   ```
2. Константе лучше присвоить мнемоническое имя для хорошего понимания читателем алгоритма программы, например:
   ```
   const Number_of_Evaluetions;
   ```
3. Константа является *параметром* программы, т.е. таким значением, которое может меняться при последующих модификациях программы. Например, пусть программист решил, что 100 – это количество экспериментов, значения которых надо ввести, чтобы достичь заданной точности результата. И вот, через некоторое время он выясняет, что ста экспериментов недостаточно, надо 120. Далее ему придётся всюду в своей программе заменить число 100 на число 120. Самая плохая идея, которая при этом может прийти ему в голову, это приказать текстовому редактору заменить всюду число 100 на 120. Что будет, если где-то в его программе число 100 означает не количество экспериментов, а температуру кипения воды? Тут то ему и поможет именованная константа, надо описать `const N_exp=100;` и потом при изменении программы менять 100 на 120 только в одном месте.

С другой стороны, не надо впадать в крайность и описывать константы вида

```
const Edinitsa=1;{🙄}
```

---

[1] В языке Free Pascal это не будет ошибкой.

В языке Free Pascal (как и во многих других языках) определены также так называемые константы времени компиляции. Это не «настоящие» константы, а значения выражений, которые можно вычислить на этапе компиляции программы, например

```
const N=100; SN=sin(2.5*N+1); C=succ(succ('B'));
```

3.3.3. Раздел переменных

Легко сделать что-то переменным. Хитрость в том, чтобы измерять продолжительность постоянства .

Алан Перлис,
первый лауреат премии Тьюринга

Нам будет удобнее сначала рассмотреть раздел описания переменных (variable declarations and definition), [1] а лишь затем раздел типов.

Из математики нам известно, что переменная – это нечто, принимающее различные значения. Теперь надо разобраться, как обстоят дела с переменными в языках программирования. Сначала надо сказать, что в некоторых, так называемых функциональных языках программирования, переменных может и вообще не быть.[2] Мы сначала рассмотрим это понятие в «традиционных» (так называемых *императивных* или процедурных) языках программирования, таких как Фортран, С или Паскаль, а потом сравним с другими, которые трактуют понятие переменной иначе.

Императивные (лат. imperativus – повелительный) языки программирования рассматривают обработку (входных) данных в виде задания последовательности шагов (команд, операторов, вызовов процедур и т.д.). Наши алгоритмические языки задают именно такой способ обработки данных. Альтернативный подход определяет обработку данных в рамках так называемого декларативного (лат. declarativus – описательный) программирования, которое часто делится на логическое и функциональное. При этом обработка данных определяется путём описания набора некоторых объектов, без явного задания последовательности действий с этими объектами.

Все императивные языки базируются на формальном описании компьютера в виде так называемой *машины фон Неймана*. Переменные при этом являются абстракциями *ячеек памяти* машины фон Неймана, это тема курса по архитектуре ЭВМ. В функциональных языках значения их «переменным» присваиваются только при их порождении, а в дальнейшем менять их нельзя. Обычно это достигается отсутствием оператора приваивания и передачи параметров только по значению, при этом из этих параметров можно только читать.

Под переменной в императивных языках программирования понимается место в памяти ЭВМ, в котором хранится текущее *значение* этой переменной. У каждой переменной есть атрибуты (свойства, характеристики), рассмотрим их. Обязательные свойства (которые есть у переменной всегда) будем заключать в рамку.

1. Имя (name) переменной (также называемое идентификатором). При *явном* описании имя приписывается переменной в разделе (описания имён) переменных, для *неявного* описания – при первом присваивании переменной значения (неявное описание мы вскоре рассмотрим в других языках, в стандарте Паскале его нет). Имя необязательный атрибут, переменные могут быть безымянными (anonymous variables). Например, если массив имеет имя `X`, но его элементы `X[i]` имён уже не имеют. Ясно, что, хотя `X[i]` и обозначает переменную, но, конечно, именем не является, это безымянная, как говорят, *переменная с индексом*. Безымянными являются и все так называемые динамические переменные, их мы будем изучать позже. Потом мы рассмотрим, как безымянные переменные могут *временно* получать имена.
2. Ссылка (reference) на месторасположение переменной в памяти ЭВМ (адрес). Это необязательный атрибут, если у переменной нет ссылки, то в данный момент она не существует.

---

[1] В большинстве англоязычных учебников этот раздел называется просто variable declarations. Тонкое различие между объявлением (declaration) и описанием (definition) некоторого имени мы рассмотрим несколько позже.

[2] В таких языках (Python, Haskel и др.) есть переменные с однократным присваиванием (по-существу, расположенные в памяти константы), после отведения им места в памяти и присваивания начального значения в дальнейшем они не меняются.

Итак, переменная может существовать не всё время, когда выполняется программа. Переменная порождается, когда ей отводится место в памяти ЭВМ, и уничтожается, когда это место у переменной отнимается. После уничтожения переменной отнятое у неё место в памяти ЭВМ может использоваться для порождения какой-нибудь (другой) переменной.

3. Значение (value) переменной, это величина, записанная в то место памяти, которое отведено переменной при её порождении. Когда переменная не существует, то и значения у неё, естественно, нет. В стандарте Паскаля при порождении переменной считается, что её значение не определено (что-то в этом месте памяти, конечно, содержится, но что – неизвестно). Как мы уже знаем, это означает, что всё зависит от конкретной реализации, о чём будет говориться далее.

Некоторые языки программирования содержат стандартную константу со значением «не определено», например, `undefined` в языке JavaScript, именно это значение и присваивается переменным сразу после их порождения. Есть и ЭВМ, позволяющие на аппаратном уровне приписать определённым переменным значение «неопределено» или «не число», и можно задать режим работы ЭВМ, при попытке использовать в котором эти переменные будет зафиксирована исключительная ситуация. Например, для компьютеров фирмы Intel это вещественные переменные в основной памяти компьютера с особым значением `SNaN` или переменные, хранящиеся на вещественных и векторных регистрах с тегом «не определено», это тема курса по архитектурам ЭВМ.

4. Тип (type) переменной. Тип в языках программирования можно определить весьма просто: это (конечное) множество различных значений, которые можно присвоить переменной этого типа (или которые могут храниться в памяти, отведённой для этой переменной).

Можно также сказать что тип – это множество значений, которые имеют одну *интерпретацию*, обычно это означает, что с каждым типом связан набор операций, которые можно выполнять над значениями этого типа. Так, все величины типа `byte` можно интерпретировать как множество целых значений в диапазоне `0..255`, с операциями целочисленного сложения, вычитания и т.д. Вообще говоря, теория типов в языках программирования опирается на *математическую* теорию типов. Это очень сложный раздел математической логики и дискретной математики.

Языки Паскаль, C, Java, Фортран и многие другие относятся к так называемым строго типизированным языкам (или языкам со статической типизацией). Это означает, что тип связывается с переменной до начала счёта (при написании программы) и во время выполнения программы меняться не может.

Статическая типизация позволяет до начала счёта программы (на этапе компиляции) проконтролировать правильность использования переменных во всех операциях, а во время счёта не тратить на такой контроль время. Например, можно считать ошибкой и не запускать программу на счёт, если программист пытается присвоить целой переменной вещественное значение. Таким образом, все такие ошибки выявляются до начала счёта.

А вот в языках с *динамической типизацией* (языки APL, Python и другие) приходится проводить этот контроль уже во время счёта и при каждой операции с переменной, на что тратится много машинного времени. Обидно, когда после часового счёта будет обнаружена ошибка, которую можно было бы найти на этапе компиляции. Особый вид динамической типизации под названием *позднее связывание* широко используется в объектно-ориентированном программировании.

Отметим, что в языке Free Pascal предусмотрен особый стандартный тип с именем `variant`, для которого используется динамическая типизация. С переменными этого типа программист может «отвести душу» и присваивать им практически любые значения, например:

```
var X: variant;
begin X:=1; X:=3.14; X:='abc'; X:=true; ...
```

Сама переменная хранит свой текущий тип в специальном служебном поле перед переменной. Для «дотошного» программиста предусмотрена функция `VarType(X)`, возвращающая текущий тип таких переменных в виде констант перечислимого типа (см. разд. 3.4.5): `VarEmpty`, `VarSingle`, `VarByte`, `VarUnknown` и т.д. Для обеспечения всех предоставляемых этим типом возможностей надо подключить модуль с именем `variants` (**`uses`** `variants;`). С помощью этого модуля можно даже присвоить переменной типа `variant` массив с элементами типа `variant` ⚠.

Эти переменные (как и родственные им переменные типа `Olevariant`) в программировании ненадёжны, их основное назначение – хранить различные значения, поступающие в программу из других вычислительных процессов. Эти значения оформляются по технологии межпрограммной коммуникации (обмена данными), которая раньше называлась в ОС Windows OLE (Object Linking and Embedding), а сейчас ActiveX.

Вас не должно удивлять, что переменная может не иметь одновременно всех своих атрибутов. Скажем, в русской литературе тоже был описан персонаж, у которого были только имя (его звали Киже) и тип (он был поручиком). А вот всего остального (тела, местожительства и т.д.) у него не было. Это, однако, не мешало ему успешно делать карьеру (дослужился до полковника), венчаться в церкви (была странная церемония) и даже быть арестованным и препровождённым в ссылку ⚠. При этом был издан приказ: «Арестант лицо секретное и фигуры не имеет», и конвоиры с примкну́тыми на винтовки штыками сопровождали пустое место по Владимирской дороге в Сибирь…

5. Класс памяти (storage class) переменной. Во многих языках у каждой переменной есть атрибут «класс памяти», который определяет *способ существования* переменной: как она порождается, сколько «живёт», как изменяется и уничтожается. Класс определяет и *область видимости* переменной, обо всём этом мы будем подробно говорить далее.[1] В языке Free Pascal есть три класса переменных.[2]

   а). Статические (static) переменные. Эти переменные порождаются до начала счёта программы и уничтожаются только после её полного завершения. Как следствие, они всё время «живут» в одном месте памяти (ссылки на них не меняется). Это самый простой для понимания класс переменных.

   б). Автоматические (automatic) переменные. Эти переменные существуют не всё время выполнения программы. Они порождаются исполнителем автоматически, когда программа достигает определённых точек (входов в так называемые блоки, вскоре мы изучим, что это такое). Далее, они автоматически уничтожаются при выходе из того блока, при входе в который они были порождены. При повторном входе в этот же блок такие переменные снова порождаются, и, вообще говоря, на другом месте памяти (с другими ссылками). Статические переменные можно образно сравнить с коренными жителями некоторого города, живущие на одном месте со дня его основания. В этом случае автоматические переменные похожи на командировочных, которые временно приезжают в город по своим делам, при этом могут каждый раз заселяться в гостиницы по разным адресам.

   > Забегая мемного вперёд скажем, что в большинстве языков программирования переменные, описанные в главной программе, будут иметь статический класс, а переменные, описанные внутри процедур и функций – автоматический класс памяти.

   в). Динамические (dynamic) переменные. Переменные этого класса порождаются и уничтожаются только по указанию программиста, когда он вызывает специальные процедуры для порождения (в разных языках они называются по-разному: `new`, `getmem`, `allocate` и т.д.) и уничтожения (`dispose`, `freemem`, `deallocate` и т.д.) переменных этого класса. Иногда их метко называют переменными с *ручным управлением* (по порождению и уничтожению). Надо, однако, отметить, что все такие переменные, ещё не уничтоженные программистом, конечно же автоматически уничтожаются по завершению программы.

В других языках классов переменных может быть больше (впрочем, иногда и меньше). Например, в языке C есть ещё классы со служебными именами **`register`** и **`volatile`**. Переменные класса **`register`** по возможности размещаются исполнителем на специальной быстродействующей, так называемой регистровой, памяти ЭВМ. Регистров обычно мало, и когда они понадобятся для других целей, то переменная «переселяется» с них в обычную память. Впрочем, оптимизирующий компиля-

---

[1] Отметим, что при описании стандарта Паскаля (в отличие, скажем, от языка C) термин «класс памяти» явно не используется, хотя это свойство у переменных, конечно, есть.

[2] В главе 11 мы познакомимся с ещё одним, внешним классом памяти, но переменные этого класса (например, файлы) программе не принадлежат (это «чужие» переменные). Отметим, что в некоторых языках программирования тоже существует внешние (external) переменные, но это не класс памяти, а просто переменные, описанные в других программных модулях, фактически они имеют статический класс памяти.

тор и сам стремится как можно больше часто используемых переменных хранить в быстрой регистровой памяти, так что «напрямую» программистами этот класс обычно не используется.

Класс памяти **volatile** очень «странный». Для переменных в языках программирования (да и в математике) мы предполагаем, что значение переменной меняется только после выполнения оператора программы, который сотрёт старое значение переменной и запишет туда новое значение. Так вот, переменная класса **volatile** может менять своё значение вне зависимости от выполнения каких-либо операторов в программе, и в произвольные моменты времени. Такую переменную компилятор не будет «трогать», т.е. оптимизировать, перемещать по памяти и т.д. Обычно эти переменные используются для обмена данными между параллельно работающими вычислительными процессами, это тема курса по операционным системам.

Отметим, что в переменные статического класса памяти в языке Free Pascal при порождении всегда помещается (конкретное) начальное значение, производится, как говорят, их инициализация (initialization). Обычно числовые переменные инициализируются нулевым значением, символьные – символом с нулевым номером в алфавите, логические – значением **false**, ссылочные – пустой ссылкой **nil** и т.д.

Для переменных автоматического и динамического классов памяти это чаще всего не делается, они будут иметь случайные значения (оставшиеся в этой памяти от предыдущей, уничтоженной переменной). Здесь дело в том, что статические переменные размещаются в памяти один раз и до начала счёта программы, так что сама программа не тратит своё время на присваивание им начальных значений. А вот переменные автоматического и динамического классов памяти порождаются и уничтожаются много раз и уже во время счёта программы. Таким образом, на присваивание таким переменным начальных значений было бы затрачено время программы, поэтому обычно это и не делается. [iv] [см. сноску в конце главы]

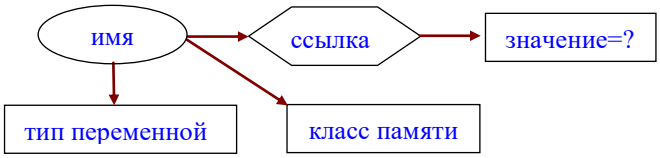
В языке Free Pascal программист при порождении переменной может явно задать её начальное значение, это особенно полезно для локальных переменных в подпрограммах, так как они имеют автоматический класс памяти и по умолчанию не инициализируются:

```
var x: char='A'; { а по умолчанию x=#0 }
procedure P;
  var x: real=3.14; { по умолчанию x не определён! }
```

Итак, переменная – это весьма сложное понятие [v] [см. сноску в конце главы]. В рассматриваемом нами разделе переменных в Паскале описываются только именованные статические и автоматические переменные. Сначала синтаксис:

```
<раздел переменных> ::= var <секция>;{<секция>;}...
<секция> ::= <имя>{,<имя>}...:<тип>;
```

Как видно, когда у нескольких переменных тип совпадает, то эти переменные можно собрать в одну секцию и тип указать только один раз. Семантика этого раздела простая: когда он просматривается исполнителем, то порождаются все описанные в разделе переменные, и с каждым именем связывается ссылка. Как мы говорили, класс памяти и тип связываются с переменной при написании программы. Представим это в виде такой схемы:



Итак, выполняя раздел переменных, исполнитель Паскаля *порождает* эти переменные, т.е. отводит им место в памяти. [vi] [см. сноску в конце главы]

3.3.4. Раздел типов

На данные свои взирая объективно,
Задумал типы я и идеал создал;
Козьма Прутков
«Безвыходное положение»

Итак, мы уже знаем, что тип – это конечное множество значений с одинаковой интерпретацией, а в разделе типов некоторым из этих множеств программист даёт имена.

Здесь надо сделать уточнение, что тип – это конечное множество констант во внутреннем машинном представлении. Таким образом, константы `1` и `1.0` принадлежат *разным* типам (целому и вещественному). Более того, например, константа `1`, в зависимости от размера занимаемой памяти (1, 2, 4 или 8 байт) тоже принадлежит к *разным* целым типам. Особенно пугаться этого не надо, так как чаще всего такие константы совместимы (compatible) по операциям, например, целые константы разной длины можно складывать и сравнивать друг с другом (но только в языках программирования высокого уровня, на машинном языке это не так!). Правда, при этом более короткая константа предварительно автоматически преобразуется к более длинной (точнее, величина *младшего* типа преобразуется в величину *старшего* типа). Когда по внешнему виду константы (например, 1) нельзя точно определить её тип, то Паскаль выбирает некоторый стандартный тип (обычно длиной 4 байта). Скоро мы будем говорить обо всём этом более подробно.

Дать типу имя очень важно, так как использование *безымянных* типов (а они в Паскале есть) в некоторых местах программы Паскаля запрещено. Синтаксис раздела типов:

<раздел типов> ::= **type** <имя>=<тип>; { <имя>=<тип>; }...

Здесь <тип> может быть как безымянным, так и задаваться именем уже существующего типа, что позволяет создавать идентичные типы, например (для любителей языка C 🙂):

```
type int=integer;
```

В большинстве языков программирования (но не во всех) к каждому типу «прилагается» набор операций, которые можно применять к величинам этого типа. Например, к величинам типа `real` Паскаля можно применять операции + и – (унарные и бинарные), умножения и деления, а также операции сравнения (равно, больше, меньше и т.д.).

Теперь нам пора разобраться, как в Паскале устроена система типов.

3.4. Типы данных в Паскале

Я полагаю, что естественной отправной точкой должна быть организация и классификация данных. Было бы по меньшей мере затруднительно создать алгоритм, не зная природы обрабатываемых им данных.

Алан Перлис,
первый лауреат премии Тьюринга

Во-первых, каждый язык программирования некоторые типы должен знать «по определению». В Паскале они называются стандартными типами (они, естественно, имеют стандартные имена). В других языках стандартные типы могут называться предопределёнными (predefined) или встроенными (built-in) и обозначаться служебными (ключевыми) словами.

В стандарте Паскаля есть четыре таких типа с именами `integer`, `real`, `char` и `boolean`. Все они относятся к скалярным типам, называемых также простыми или элементарными. Величины скалярных типов с точки зрения Паскаля не содержат внутри себя более мелких величин, в противовес, например, векторам, состоящим из элементов. Все скалярные типы упорядочены, т.е. для любых двух величин одного такого типа всегда известно, что одна величина больше или меньше другой, или же они равны между собой.

В «настоящих» языках это уже не так. Например, у нас некоторые величины типа `real` могут быть *несравнимы* (unorderd) между собой, то есть операция сравнения вырабатывает четыре ответа: меньше, равно, больше и «не сравнимы».

3.4.1. Целый тип

Бог создал целые числа, всё остальное – дело рук человека.

Леопольд Кронекер

Как уже упоминалось, к каждому типу обычно «прилагаются» операции, которые можно производить над значениями этого типа. Операции сложения, вычитания и умножения над целыми числами обозначаются привычными нам знаками `+`, `-` и `*`, семантику этих операций мы уже описыва-

ли. Различают унарные (одноместные) операции (unary или monadic operations), например, `-X`; и бинарные (двуместные) операции (binary или dyadic operations), например, `X+Y`. В некоторых языках (например, в PascalABC.NET, C и других) есть и тернарные (трёхместные) операции (ternary operations), но в стандарте Паскаля (и в языке Free Pascal) таких нет. Могут существовать и операции большей «арности». Вообще говоря, существуют и операцию «нулевой арности», это просто переменные и константы, а выполнение такой операции это просто чтение (и запись) значения.

Далее, например, одноместный и двуместный «минус» являются *разными* операциями, потом Вы узнаете, что в компьютере для них существуют две разные машинные команды с кодами **neg** и **sub**.

Заметим далее, что по сути, все бинарные операции являются функциями от двух аргументов, так что можно было бы обозначать операцию сложения как `+(X,Y)`. Более того, мы привыкли ставить знак двуместной операции *между* операндами (т.е. `X+Y`), такие операции называются инфиксными (infix). Можно, однако, договориться ставить знак операции *перед* операндами (т.е. `+X,Y`), такие операции называются префиксными (prefix), или же *после* операндов (т.е. `X,Y+`), такие операции называются постфиксными (postfix).

Постфиксные операции ввёл польский логик и математик Ян Лукасевич, поэтому они называются ещё обратной польской записью (reverse Polish notation). Любопытно, что в префиксной и постфиксной записи круглые скобки для изменения приоритета операций (operator precedence) не нужны, все выражения вычисляются слева направо. Именно поэтому при производстве вычислений на компьютере выгоднее использовать именно префиксную или постфиксную запись выражений. Вообще говоря, мы хорошо знакомы с бесскобочными выражениями, именно так мы выполняем вычисления на обычном калькуляторе ⚠.

Некоторые языки используют префиксные операции в явном виде (правда, со скобками). Например, в языке Lisp сложение переменных `X+Y` записывается как `(+ X Y)`, выражение `(5+X)*(Y-7)` как `(* (+ 5 X) (- Y 7))` и т.д. Каждая операция выполняется, как только за ней будут вычислены все её операнды (число этих операндов зависит от арности операции). Поймите, что круглые скобки теперь можно убрать, они просто синтаксическое оформление языка (для «наглядного» представления выражения в виде списка), без скобок будет `* + 5 X - Y 7`.

Вообще говоря, можно оставить только одну операцию *переменной* «арности», это круглые скобки `()`. Эту операцию, как в упомянутом выше языке Lisp, можно считать обобщением понятия функции. Тогда `sin(x)` будет бинарной операцией `(sin x)`, `abs(x,y)` – *тернарной* операцией `(abs x y)`, а `f(x+y)` – *бинарной* операцией `(f (+ x y))`. Более того, переменные и константы, как уже говорилось, можно рассматривать как операции *нулевой* арности. Любят теоретики запутать казалось бы простые вещи 🙄.

Сложнее обстоит дело с операцией деления, деления целых и вещественных чисел принципиально отличаются друг от друга. В отличие от вещественных, деление целых чисел даёт два результата: частное и остаток. Например, `13.0/4.0=3.25`, а `13:4=3` (частное) и `1` (остаток). В языках программирования редко реализуются операции с двумя результатами, например, вот такая операция в языке Python **`divmod`**`(9,4)→(2,1)`. В большинстве языков программирования для этого существуют две отдельные операции: деление нацело и вычисления остатка.

В разных языках по-разному выбраны обозначения операций вычисления частного и остатка. Например, в языке C для этого выбраны обозначения `/` и `%`, в языке Python это `//` и `%`, а в языке Фортран деление это `/`, а взятие остатка от деления решили вообще не реализовывать.[1] В Паскале выбраны «красивые» обозначения для этих операций в виде *служебных* слов **div** и **mod**.

Во-первых, надо понять, что нельзя вычислить частное без вычисления остатка, и наоборот (вспомним, как мы делим целые числа «уголком»). Во-вторых, возникают трудности, когда операнды могут быть отрицательными. Например, компьютер считает, что `-1:256=0` (частное) и `-1` (оста-

[1] В языках C и Фортран операция `/` рассматривается как деление нацело, если оба аргумента целые, иначе считается вещественным делением. А остаток, кому надо, можно вычислить как `X mod Y=X-(X/Y)*Y`. Здесь, однако, есть тонкие моменты, на которых мы не будем останавливаться.

ток), т.е. частное округляется в бóльшую сторону (как говорят, к *положительной бесконечности*). А вот в математике будет `-1:256=-1` (частное) и `255` (остаток), это округление частного в *меньшую* сторону (к *отрицательной бесконечности*), тогда остаток всегда *неотрицательный*.

В стандарте Паскаля сказано, что целочисленное деление должно выполняться «как в математике», однако все ЭВМ реализуют эту операцию первым способом. Исходя из этого, почти все реализации Паскаля отступают здесь от стандарта (и, в частности, наш язык Free Pascal, если не задать ему режим работы «как в стандарте» `{$mode ISO}`). Таким образом, в языке Free Pascal мы будем считать, что `-13 mod 4=-1` и `-13 mod -4=-1`, т.е. знак остатка совпадает со знаком делимого.

При конкретной реализации Паскаля допускается расширение языка, т.е. в него можно добавлять новые конструкции и возможности, однако стандарт должен входит в расширение как подмножество. К сожалению, это не всегда рационально, и в языке Free Pascal есть несколько таких отступлений от стандарта, мы их все отметим.

Далее, над целыми числами можно проводить привычные нам операции сравнения на равно, больше, меньше и т.д. «По научному» они называются операциями отношения (relational operators) между двумя величинами. Эти операции дают логический результат («верно» – «неверно», «истина» – «ложь», **`true`** – **`false`** и т.д.). В математике шесть операций отношения:

`=` `≠` `>` `<` `≥` `≤`

В разных языках программирования обозначение этих операций может различаться, например:

Язык Паскаль:	`=`	`<>`	`>`	`<`	`>=`	`<=`
Язык C:	`==`	`!=`	`>`	`<`	`>=`	`<=`
Язык Фортран:	**`.EQ.`**	**`.NE.`**	**`.GT.`**	**`.LT.`**	**`.GE.`**	**`.LE.`**
Ассемблер MASM:[1]	**`EQ`**	**`NE`**	**`GT`**	**`LT`**	**`GE`**	**`LE`**
Язык `bat` файлов:	`equ`	`neq`	`gtr`	`lss`	`geq`	`leq`

Вот и все наши операции над целыми числами. Отметим, что в стандарте Паскаля нет операции возведения в степень, только для возведения в квадрат есть стандартная функция `sqr(X)`.[2] Кроме того, есть и другие стандартные функции для работы с целыми числами: `abs(X)` для вычисления абсолютной величины, логическая функция `odd(X)`, принимающая значение **`true`** для *нечётного* `X`, и другие. Некоторые стандартные функции (`sin`, `cos`, `sqrt`) хоть и могут иметь *целый* аргумент, но всегда вырабатывают *вещественный* результат.

Рассмотрим теперь, как целый тип реализован в языке Free Pascal. Как уже говорилось, целый тип там разбивается на четыре расширяющихся знаковых поддиапазона `shortint` (`int8`), `smallint` (`int16`), `longint` (`int32`) и `int64`, и на четыре беззнаковых поддиапазона со стандартными именами `byte` (`uint8`), `word` (`uint16`), `longword` (`uint32`) и `qword` (`uint64`).

Самые длинные целые числа в языке Free Pascal это `int64` (знаковое) и `qword` (беззнаковое), они примерно по 19-20 десятичных цифр, на компьютере есть команды для операций над такими числами (сложение, умножение и т.д.). В старых 32-битных процессорах фирмы Intel были так называемые двоично-десятичной (binary-coded decimal) целые числа длиной до 31 цифры. Для выполнения операций над более длинными числами приходится использовать подпрограммы. Например, язык Python может делать операции над числами любой (разумной) длины, но, конечно, на несколько порядков медленнее, чем с помощью машинных команд.

Компьютер может выполнять арифметические команды (сложения, вычитания и умножения) только с операндами одинаковой длины, поэтому надо уметь преобразовывать числа из одного типа в другой. Для такого преобразования в языках программирования определены явные (explicit) и неявные (implicit) преобразования (coercion) типов.[3]

---

[1] Эти операции используются только в так называемых макросредствах Ассемблера, это тема курса по архитектуре ЭВМ.

[2] В языке Free Pascal при подключении математического модуля **`uses`** `Math;` можно использовать операцию возведения в степень, она обозначается как две звёздочки `**`, например `2**3=8, 2.0**3=8.0` и т.д.

[3] Точнее, это преобразование величин этих типов, но так не говорят.

Явные преобразования типов [1] в стандарте Паскаля отсутствуют, а в языке Free Pascal осуществляются с помощью стандартных функций, имена которых совпадают с именами целевого типа. Например, пусть есть описания переменных:

```
var x: shortint; y: integer; z: int64;
```

тогда явные преобразования в языке Free Pascal можно задать, например, такими операторами:

```
y:=integer(x); x:=shortint(z); z:=int64(x);
```

В языке Free Pascal для повышения надёжности *запрещены* некоторые «странные» явные преобразования типов, например:

```
real(true){ ОШИБКА, если «очень надо», может писать
            real(integer(true)) => 1.0 }
real('A'); boolean(3.14); byte(1.0) { ОШИБКИ!}
```

А вот когда в программе встречается выражение с числами *разного* типа, компилятор производит неявное (автоматическое) преобразование. Между числовыми типами в языке Free Pascal вводится отношение старшинства, обозначим его знаком ⋘.

```
shortint⋘byte⋘smallint⋘integer⋘word⋘
          longint⋘longword⋘int64⋘qword
```

Неявное преобразование (автоматическое) типов в двуместных операциях всегда преобразует все младшие типы к «естественному» типу (обычно `longint` или `longword`), или к самому старшему (`int64` или `qword`), а при присваивании результат преобразуется к типу переменной, например:

```
var x: shortint; y: integer; z: int64;
  y:=x+z; { Вычисляется как  y:=integer(int64(x)+z) }
  z:=x+y; { Вычисляется как  z:=int64(longint(x)+longint(y)) }
  if x=y then { Вычисляется как  if longint(x)=longint(y) then }
```

Преобразовании к старшему типу почти всегда проходит без потери информации, так как длина в байтах итоговой величины не меньше, чем преобразуемой, они ещё называются расширяющими преобразованиями (widening conversion). Исключением является преобразование между знаковыми и беззнаковыми или же целыми и вещественными числами, даже для одной длины оно может привести к искажению данных, например:

```
var x: shortint; y: byte;
    a: longint; {9 значащих цифр}
    b: single;  {6 значащих цифр}
begin x:=-1; y:=x; {y=255 🙄}
      b:=a;     {потеря значимсти}
```

Эта тема связана с машинным представлением целых чисел и изучается в курсе по архитектурам ЭВМ.

Преобразование к типу, занимающему меньшее количество байт, производится путём *усечения*, т.е. отбрасыванием необходимого числа старших байт, это ещё называется сужающим преобразованием (narrowing conversion). Это преобразование может привести к ошибке (переполнению). Как мы уже говорили, в режиме работы с контролем `{$R+}` Free Pascal при выходе за допустимый диапазон фиксирует ошибку, а в режиме без контроля `{$R-}` просто выдаёт неверный (усечённый) результат. Здесь важно учесть, что контроль производится не при *вычислении* выражений, а только при *присваивании* переменной нового значения.

Замечание на будущее. Неявные преобразования производятся и при передаче параметров в процедуру и функцию по значению, об этом будем говорить в главе 8.

---

[1] В языках, «основанных» на C это называют явным приведение (cast) типов.

3.4.2. Вещественный тип

В мире есть много трудных вещей, но нет ничего труднее, чем четыре действия арифметики.

Монах Бе́да Достопочтенный, VII век
(достала римская система счисления 🙁)

В стандарте Паскаля вещественный тип имеет имя `real`, над величинами этого типа определены все четыре привычные нам операции: сложение "+", вычитание "–", умножение "*" и деление "/". Чаще всего, как мы уже говорили, эти операции дают приближенный результат (округлённый до ближайшего представимого вещественного числа).

Один и тот же знак операции может иметь в языке программирования несколько разных смыслов. Например, для знака операции сложения "+" мы уже знаем четыре разных смысла:

1). унарный целый `+X`;
2). унарный вещественный `+X`;
3). бинарный целый `X+Y`;
4). бинарный вещественный `X+Y`.

По существу, это четыре разных операции, такое явление называется полиморфизмом (polymor–phism), а иногда говорят, что это перегружаемые операции (см. раздел 8.7). Синтаксис языка должен позволять однозначно выбрать конкретный смысл операции по месту её использования (по *контексту*). Скоро мы узнаем, что в стандарте Паскаля у операции "+" есть ещё и пятый смысл (объединение множеств), а в языке Free Pascal и шестой – сцепление (конкатенация) строк.

Полиморфными могут быть не только знаки операции, но и подпрограммы. Например, стандартная функция Паскаля `abs(x)` имеет разную реализацию для целого и для вещественного аргумента. В этом случае исполнитель должен вызвать конкретную функцию, в зависимости от типа параметра. Иногда этот вид полиморфизма так и называют *полиморфизм типов*, в отличие от рассмотренного ранее *полиморфизма операций*.

Многие языки программирования позволяют программисту определять свои собственные полиморфные операции и подпрограммы, обычно в этом случае говорят о перегрузке (overload) операторов и подпрограмм. Как это делать в языке Free Pascal рассказано в разделе 8.7.

Полиморфизм (греч. πολυμορφισμός – многофо́рмовый) [1] первоначально определён в биологии и обозначает способность организмов одного вида существовать на разных стадиях своего развития в непохожих друг на друга внешних формах (например, личинка – гусеница – бабочка). В программировании понятие полиморфизма ввёл в 1967 году Кристофер Стречи (Christopher Strachey), между прочем, автор машинного алгоритма генерации простых чисел и первой программы игры в шашки.

Когда синтаксис языка позволяет однозначно выбрать конкретный смысл операции или функции по месту использования (по *контексту*), то этот выбор делает компилятор (до начала счёта программы) и говорят о *статическом полиморфизме*. В этом случае говорят о раннем связывании операций и функций с конкретными типами данных. Иногда, однако, это сделать не удаётся и приходится производить такую связь уже во время счёта (позднее связывание), чаще всего оно применяется в объектно-ориентированном программировании, тогда говорят о динамическом полиморфизме.

Некоторые языки возлагают ответственность по выбору конкретной полиморфной функции на программиста. Например, в языке Фортран программист вызывает `abs(x)` для вещественного `x`, `iabs(x)` для целого, `cabs(x)` как модуль для комплексного числа (в Фортране есть и такой тип), и т.д. Теперь, при вызове, например, `cabs(-1)`, Фортран произведёт неявное преобразование типов и вызовет `cabs(-1.0+0.0*i)`. Вряд ли можно назвать это полноценным полиморфизмом.

В программах часто бывает необходимо выполнять смешанные операции, например, складывать целые и вещественные числа. Таких машинных команд нет, и Паскаль производит автоматическое (неявное) преобразование целого типа в вещественный (т.е. вещественный тип считается старше, чем целый). Для человека здесь всё просто, например, `12 ⇒ 12.0`. Для компьютера эти кодировки силь-

---

[1] Согласно греческой мифологии, бог сна Морфей мог являться людям во сне в самых разнообразных формах – это истинный полиморфизм 🙂.

но различаются (и часто имеют разную длину в байтах), однако есть машинные команды, которые весьма эффективно делают такое преобразование.

Обратное преобразование (вещественное в целое) необходимо, когда вещественное значение надо присвоить целой переменной. Многие языки (например, Фортран, C и другие) делают такое преобразование автоматически (неявно), однако в Паскале и некоторых других языках (например, в языке Java) это запрещено. Дело в том, что такое преобразование неоднозначно. Действительно, вещественное число 12.5 можно округлить и в целое 12, и в 13, в Паскале принято решения, что выбор одного из этих способов может сделать только сам программист.

Первое из них называется *усечением*, дробная часть при этом просто отбрасывается, это делается вызовом стандартной функции `trunc(x)`, например, `trunc(7.8)=7`, `trunc(-4.5)=-4` и т.д. Второе преобразование называется *округлением*, оно делается стандартной функцией `round(x)`, по определению эта функция в стандарте Паскаля работает так:

$$\mathtt{round(x)}=\begin{cases}\mathtt{trunc(x+0.5)}, & \text{при } x\ge 0\\ \mathtt{trunc(x-0.5)}, & \text{при } x<0\end{cases}$$

Поймите, что функция `trunc` производит округление к нулю, а функция `round` в стандарте Паскаля округляет последнюю цифру `5` к ±∞ (это так называемое математическое округление, его по умолчанию реализуют в пакетах символьной математики, например, в пакете MATLAB).

Внимание! В языке Free Pascal здесь существенное отступление от стандарта. Вместо округления к ±∞ функция `round` производит так называемое «банковское» округление к ближайшему чётному целому числу. Ясно, что различие только в округлении цифры `5` (в десятичной системе счисления), ну, или цифры `1` (в двоичной системе). Таким образом:

Стандарт Паскаля		Язык Free Pascal
`round(1.4)=1`		`round(1.4)=1`
`round(1.5)=2`		`round(1.5)=2`
`round(-1.5)=-2`		`round(-1.5)=-2`
`round(2.5)=3`		`round(2.5)=2`
`round(-2.5)=-3`		`round(-2.5)=-2`

В «оправдание» языка Free Pascal можно сказать, что процессоры современных ЭВМ по умолчанию делают именно такое округление чисел, при этом минимизируются ошибки округления. В хороших школах тоже учат делать округление именно так. Отметим, что при работе с вещественными числами в процессоре можно включить один из четырёх режимов округления, программист может сделать это, подключив «математический» модуль языка Free Pascal с именем `Math` и вызывая функцию `SetRoundMode` из этого модуля.

Начиная с версии Free Pascal 3.0.2 желающие могут вернуться к округлению, принятому в стандарте Паскаля ISO 7585:1990, задав директиву `{$mode ISO}`. Необходимо однако учитывать, что в этом случае по стандарту станет работать и операция взятия остатка **mod**, давая всегда неотрицательный результат.

Дополнительно в модуле `Math` языка Free Pascal реализованы стандартные функция, которые для вещественного аргумента возвращают такие результаты:

`int(x)` – целую часть `x`, например, `int(-2.26)=-2.0`
`frac(x)` – дробную часть `x`, например, `frac(-2.26)=-0.26`
`ceil(x)` – ближайшее целое, большее `x`, `abs(x)<=Maxint`, например, `ceil(-1.7)=-1`
`floor(x)` – ближайшее целое, меньшее `x`: `floor(-1.7)=-2`

Как уже говорилось, в языке Free Pascal реализованы четыре вещественных типа со стандартными именами `single` (длиной 4 байта), `real` и `double` (длиной 8 байт) и `extended` (длиной 10 байт). У их типов естественное отношение старшинства:

`single` ⋘ `real` ⋘ `double` ⋘ `extended`

В языке Free Pascal определён ещё тип с именем `ValReal`, в каждой реализации он равен «самому большому» вещественному типу (обычно это `extended`). А в модуле `Math` определён ещё тип

`float=extended` (привет C и другим языкам 🙂). Некоторые ЭВМ умеют работать и с типом `real16` (длиной 16 байт).

Над вещественными числами можно делать и операции сравнения (больше, меньше и т.д.), которые дают логический результат (истина или ложь). Сравнение вещественных чисел, однако, надо делать с осторожностью. Во-первых, как уже говорилось, некоторые вещественные числа могут быть *несравнимы* друг с другом. Во-вторых, вполне может оказаться, что `1.0000001=1.0000002`, так как эти числа будут попадать в одну точку на дискретной вещественной оси. Аналогично, если взять `a<b`, `n>1`, и `h=(b-a)/n`, то, скорее всего будет `a+n*h≠b`.

Почти все современные ЭВМ для вещественных чисел и операций над ними придерживаются международного стандарта ANSI/IEEE 754-1985 (Standard for Binary Floating-Point Arithmetic).[1] Стандарт разработан под руководством американской ассоциации Института инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronics Engineers). Любопытно, что этот стандарт практически целиком разработал всего один человек, профессор математики Вильям Каган (William Kahan), который в то время работал в Калифорнийском университете в Беркли. Отметим, что в 2008 году вышел обновлённый стандарт IEEE754-2008, в него, в частности, добавлены сверхдлинные 16-байтные вещественные числа `real16`.

Как уже упоминалось выше, некоторые из представимых вещественных чисел используются для служебных целей. Во-первых, это уже упоминавшееся ранее специальное значение «не число» (`NaN` – Not a Number). При попытке производить арифметические операции над такими «числами» в арифметико-логическом устройстве может возникать исключительная ситуация. Во-вторых, это два специальных значения `±∞`. Эти значения выдаются в качестве результата арифметических операций с вещественными числами, если этот результат такой большой по абсолютной величине, что не представим среди множества машинных вещественных чисел. Компьютер «разумно» (по крайней мере, с точки зрения математика) производит арифметические операции над такими «числами». Например, пусть `A` любое представимое вещественное число, не равное нулю, тогда

```
±A/0 = ±∞;  ±A/±∞ = ±0;  A * ±∞ = ±∞;
∞ + ∞  = ∞;  (-1.0)*(-∞) = +∞;
0*(±∞) = -∞+∞ =  ±∞/±∞ = ±0/±0 = sqrt(-1.0) = NaN;
A ± NaN = NaN ± NaN = NaN ± ∞ = NaN * ∞ = NaN и т.д.;
1.0^NaN = 1.0;  NaN^0.0 = 1.0 ⚠ (хотя таких машинных операций и нет)
```

Таким образом, наш компьютер может делить на ноль, получается `±∞`, при выводе Free Pascal обозначает эти «бесконечности» как `±Inf`. В модуле `Math` языка Free Pascal есть стандартная константа `Infinity` (т.е. `+∞`) и константа с именем `NaN`.

Необходимо также учитывать, что существуют два вещественных нуля `±0.0`, поэтому

```
+∞/(+0.0)+∞ = +∞;  +∞/(-0.0)+∞ = NaN;
```

К сожалению, как уже говорилось, вещественные числа не являются полностью упорядоченными, некоторые из них нельзя сравнивать между собой, причём не только на больше и меньше, но и на равно и не равно! Например, операции сравнения не работают, если хотя бы один из операндов `NaN`. В языке Free Pascal некоторые выражения будет давать «странные» результаты, например, `NaN=1.0` будет **`true`**, а `NaN<>1.0` будет **`false`** 🙄. Для проверки таких «числовых» значений в модуле `Math` предусмотрены специальные логические функции `IsInfinite(x)` и `IsNaN(x)`. Похожие функции реализованы и во многих других языках программирования.

---

[1] Термин Floating-Point (плавающая точка) «намекает» на то, что при выполнении операций и нормализации результата точка, отделяющая целую часть числа от дробной, «плавает» между значащими цифрами:

$20.34*2^2=2.034*2^3$

3.4.3. Символьный тип

Три самых дорогостоящих ошибки всех времён были вызваны изменением одного символа в ранее корректных программах.
Стивен Макконнелл
«Совершенный код»

В Паскале символьный тип обозначается стандартным именем `char`. Константами этого типа являются символы алфавита (те, которые `character`). Кроме обычных (графических) символов, которые можно изобразить на бумаге или экране, в алфавит входят и *служебные* (невидимые) символы (`Esc`, `BS`, перевод строки, возврат каретки, звонок и другие). Считается, что все символы (в том числе и служебные) в алфавите перенумерованы, начиная с нуля, таким образом символы упорядочены, их можно сравнивать между собой.

В стандарте Паскаля не сказано, сколько символов в алфавите и какие номера у конкретных букв, требуется только, чтобы символы цифр шли в алфавите подряд (все алфавиты удовлетворяют этому требованию). В языке Free Pascal определены два символьных типа (два алфавита). В одном из них с именем `char` ровно 256 символов,[1] в другом с именем `widechar` (это один из так называемых алфавитов Unicode) содержится 2^{16} символов. В наших примерах будет использоваться только символьный тип `char`.

На самом деле, как и в типе `char` для языка Free Pascal, в `widechar` некоторые подряд идущие символы обозначают дополнительные, так называемые *суррогатные* символы. Вообще же говоря, алфавит версии Unicode 15.0.0 2022 года содержит около 149 тыс. символов. Там есть всё, включая древнеегипетские иероглифы, шумерскую клинопись, пляшущие человечки Конан-Дойла, символы искусственных языков и смайлики (рожицы) Эмóдзи (а по оценкам в мире используется около 200 тыс. различных знаков). Для представления их всех приходится использовать кодировку UCS-4 (UTF-32), где под символ отводится 4 байта, или кодировку UTF-8, где под символ отводится от 1 до 4-х байт.

На Паскале *графические* символы, как и строки, принято записывать в апострофах, например, `'A'`, `'+'` и т.д. Ясно, что сам символ апострофа и служебные символы так задать нельзя. Как быть?

Для определения номера символа в алфавите предназначена функция Паскаля `ord(x)`. Например, во многих алфавитах `ord('1')=49`. Для работы со служебными символами в Паскале предусмотрена стандартная функция `chr(x)`, которая для целого неотрицательного параметра `x` возвращает символ с этим номером в алфавите, например, `chr(49)='1'`. Ясно, что функции `ord` и `chr` взаимно обратные, например, `chr(ord('A'))='A'` и `ord(chr(49))=49`.

В языке Free Pascal у типа `char` есть синоним `Ansichar`. Далее, *константы* типа `char` можно записывать в виде `#<номер символа>`, например, `#49=chr(49)`, так можно задавать любые символы, даже не имеющие графического представления. Правда, запись `#49` является символьной *константой*, а `chr(49)` символьным выражением (функцией), поэтому можно писать строку `'AB'#49'CD'`, но нельзя `'AB'chr(49)`, приходится (если это допустимо) использовать *выражение* `'AB'+chr(49)`.

Кроме того, служебные символы с номерами в алфавите от 1 до 26 можно записывать в виде латинской буквы (независимо от регистра) с предшествующим символом `^`, например, `^A=^a=#1`, `^B=^b=#2` и т.д. Вообще говоря, после символа `^` можно писать любой символ, при этом часто получаются символы со «странными» номерами в алфавите. Мы этим пользоваться не будем.

Единственными операциями над величинами символьного типа являются операции отношения (сравнения), как уже говорилось, символы упорядочены по их номерам в алфавите. Все остальные операции запрещены, например, для `var x: char;` приведённые ниже выражения ошибочны:

---

[1] Точнее, в типе `char` языка Free Pascal 256 имволов в основном алфавите и 255 символов в дополнительном, символы которого кодируются двумя байтами, первый из которых нулевой, более подробно об этом будет говориться далее.

```
x+1     4*x     sin(x)     { ОШИБКИ }
```

Почему так сделано? Ведь во многих других языках (Фортран, С и т.д.) символьный тип объявлен младшим, по сравнению с целым типом (т.е. `char≪integer`) и в выражениях производится неявное преобразование символьного типа в целый. В этих языках `x+1` выполняется как `ord(x)+1`. Можно, однако, заметить, что трудно вложить какой-либо разумный смысл, скажет, в выражение `4*'A'`. Многие языки трактуют его как `4*ord('A')`, а вот язык Python как `'AAAA'`, но чаще всего это просто ошибка программиста. И, в конце концов, если он хочет трактовать `4*'A'` как `4*ord('A')`, то пусть так и пишет `4*ord('A')`. Заметим, что в языке Free Pascal тоже есть функция `DupeString('123',4)`, которая выдаёт «учетверённую» строку `'123123123123'`.

3.4.4. Логический тип

Но да будет слово ваше: «да, да»; «нет, нет»; а что сверх этого, то от лукавого.
Евангелие от Матфея (5:37)

Логический тип обозначается стандартным именем `boolean`, в нём всего две константы. В стандарте Паскаля они обозначаются *стандартными* именами `false` и `true`. Логические константы пронумерованы числами 0 и 1, т.е. `ord(false)=0` и `ord(true)=1`, соответственно считается, что `false<true`.

Тип назван в честь математика и логика Джорджа Буля (George Boole). В 1854 году он опубликовал книгу «Исследование законов мышления, на которых основываются математические теории логики и вероятностей», в этой книге развита так называемая алгебра высказываний, получившей затем название *булевой алгебры*. Эта алгебра является инструментом разработки сложных электронных схем, служит для оптимизации числа логических элементов, из которых строятся ЭВМ. Она же является основой общепринятой двоичной системы счисления, лежащей в основе работы компьютеров. Впервые логический тип появился в языке Алгол-60.

Так как в стандарте Паскаля логические константы обозначены *стандартными* именами, т.е. их можно переопределить, например: `type true=char;`. Не надо думать, что теперь в программе логическое значение «истина» исчезает, эта константа просто становится *безымянной*. Такую программу будет тяжело понимать, отлаживать и модифицировать. Исходя из этого, многие реализации Паскаля отступают здесь от стандарта, считая эти имена не стандартными, а служебными. Мы тоже будем так делать, и писать эти имена жирным текстом **`false`** и **`true`**. К сожалению, поздние версии языка Free Pascal снова сделали эти имена из служебных стандартными.

Над логическими величинами определены все операции сравнения (больше, меньше и т.д.), они естественно, дают логический результат. Кроме того, из математики в языки программирования пришли и специальные *логические* операции. Логические операции, как и арифметические, бывают унарные (одноместные) и бинарные (двуместные). Унарная операция всего одна, и обозначается в Паскале служебным словом **`not`**, она совсем простая: **`not false=true`** и **`not true=false`**. Отметим, что в математике операция **`not`** a обозначается как ¬a или $\bar{a}$, в языке С как `!a`, в языке Фортран как **`.NOT.`** a и т.д.

Как уже отмечалось, двуместные операции над логическими величинами, дающие логический результат, по существу являются функциями от двух аргументов. Так как каждый из аргументов может принимать только два значения, то существуют всего 16 разных логических операций. Каждую операцию можно однозначно определить в виде так называемой *таблицы истинности* (true table), в которой просто перечисляются все четыре значения над возможными комбинациями значений аргументов. В стандарте Паскаля определены 8 таких операций (см. рис. 3.1). Для компактности, значение **`false`** обозначено цифрой 0, а значение **`true`** цифрой 1.

Операция **`or`** называется *дизъюнкцией* или логическим сложением (в математике обозначается как ∨, в языке С как `||`, в Фортране как **`.OR.`**), а операция **`and`** называется *конъюнкцией* или логическим умножением (в математике обозначается как ∧, в языке С как `&&`, в Фортране как **`.AND.`**). Проще всего запомнить, как они работают, если понять, что **`or`** определяет *максимум*, а **`and`** – *минимум* от значений своих аргументов.

x	y	**or**	**and**	=	>	<	>=	<=	<>
0	0	0	0	1	0	0	1	1	0
0	1	1	0	0	0	1	0	1	1
1	0	1	0	0	1	0	1	0	1
1	1	1	1	1	0	0	1	1	0

Рис. 3.1. Таблицы истинности операций стандарта Паскаля.

Существуют и так называемые многозначные логики, где более двух логических констант. Например, в трёхзначной логике эти константы можно образно назвать «неверно», «может быть» и «верно», с порядковыми номерами 0, 1 и 2 соответственно. Однако и там, что хорошо, **or** определяется как максимум, а **and** – как минимум своих аргументов.

Во многих языках есть ещё логическая операция **xor** (исключающее или, не эквивалентность, сложение по модулю 2, в математике обозначается как $\oplus$), однако её таблица истинности совпадает с операцией `<>`, так что она, вообще говоря, избыточна. Заметим также, что одни операции легко выражаются через другие, например:

```
x<>y=x xor y=(x or y) and not(x and y)
```

Можно, например, выбрать операции **not**, **or** и **and** как базовые и выразить через них все остальные операции.

Конструкторы электронных схем предпочитают в качестве базовых вместо операции **or** использовать операцию `x nor y=not (x or y)` (она называется стрелка Пирса, обозначается как `↓`), а вместо **and** операцию `x nand y=not(x and y)` (она называется штрих Шеффера, обозначается как `|`). Дело в том, что их реализация в электронных схемах требует на один транзистор меньше. Более того, и стрелку Пирса, и штрих Шеффера можно взять как единственную базовую операцию, выразив через неё все остальные операции, а также и логические константы, например, для **nand**:

```
not X ≡ X|X;  true ≡ X|(X|X);
X or Y ≡ (X|X)|(Y|Y);
X and Y ≡ (X|Y)|(X|Y);  и т.д.
```

Как и для символьного типа, для логических значений определена стандартная функция `ord`, дающая порядковый номер этой величины в своём типе: `ord(false)=0` и `ord(true)=1`. Для логических величин в Паскале запрещено их неявное преобразование в другие типы (в частности, в целый тип), исходя из этого приведённые ниже выражения ошибочны:

```
1+true    true-'A'    sin(false) и т.д. { ОШИБКИ }
```

Если программист считает, что ему это нужно (зачем 🙄), то пусть преобразовывает типы явно:

```
1+ord(true)   ord(true)-ord('A')   sin(ord(false))
```

В конкретной реализации для хранения логического значения достаточно одного бита компьютерной памяти. В языке машины, однако, (почти) нет команд, операндом которых был бы бит, минимальным объёмом обрабатываемых данных является байт (8 бит). Наилучшим машинным представлением логических величин в виде байта являются `false`=00000000_2 и `true`=11111111_2, так как тогда машинная команда `not x` преобразует их друг в друга.

В то же время, в широко распространённом языке C нет явного логического типа данных и логических констант. Вместо этого, любое «ненулевое» значение трактуется как **true**, а «нулевое» как **false**, таким образом длина логических констант в языке C получается от 1 до 8 байт. Это неудобно, поэтому в языке C++ логический тип уже есть. В языке Free Pascal, чтобы была возможность «программировать как в C 🙂», предусмотрены дополнительные беззнаковые целые типы с именами `ByteBool`, `WordBool`, `LongwordBool` и `QWordBool`, длины 1, 2, 4 и 8 байт соответственно. Они совместимы (compatible) по присваиванию и сравнению с логическим типом.

В языке Free Pascal целые значения могут *трактоваться* логическими операциями как вектора логических величин. Например, целые числа типа `word`:

```
var x,y: word;
```

имеют длину 2 байта=16 бит и могут трактоваться логическими операциями **not**, **and**, **or** и **xor** как логические вектора

```
var x,y: bitpacked array[0..15] of boolean;
```
[1]

Как видим, при описании массива мы использовали служебное слово **bitpacked**, в этом случае получается так называемый упакованный массив. В таком массиве каждое логическое значение занимает в памяти всего один бит, а не целый байт, как для обычных логических величин, при этом бит с нулевым значением трактуется как **false**, а со значением единица как **true**. В этом случае логическая операция, например, `x:=x or y` будет выполняться как

```
for i:=0 to 15 do x[i]:=x[i] or y[i]
```

Например:

```
x: 1000 0001 1010 1101
y: 1001 0101 0000 0010
x: 1001 0101 1010 1111
```

Операция `x:=x xor y` выполняется как побитовая операция <> (не равно), например

```
x: 1000 0001 1010 1101
y: 1001 0101 0000 0010
x: 0001 0100 1010 1111
```

Формально такие операции дают целочисленные результаты. Определена и одноместная побитовая логическая операция **not**, она меняет значение каждого бита на противоположное:

```
    x: 1000 0001 1010 1101
not x: 0110 1110 0101 0010
```

С точки зрения арифметики (если рассматривать аргументы как целые числа) эта операция может давать «странные» результаты, например, `not 1 = -2`. Почему так происходит Вы узнаете из курса по архитектурам ЭВМ.

Дополнительно над целочисленными данными определены операции сдвига **shl** (SHift Left – поразрядный сдвиг вектора налево) и **shr** (SHift Right – поразрядный сдвиг вектора направо). Например, операция `x shl n` будет выполняться сдвигом всех бит переменной x на n бит влево, на освободившиеся места записываются нули. Формально это можно записать как

```
for i:=1 to n mod 8*sizeof(x) do begin
    for j:=14 downto 0 do x[j+1]:=x[j];
    x[0]:=0
end
```

Например:

```
      x: 1000 0001 1010 1101
x shl 6: 0110 1011 0100 0000
```

Битовый вектор n раз сдвигается влево на единицу, а нулевая позиция обнуляется (получает значение **false**). Ясно, что, так как тип word имеет длину 16 бит, то задавать `n>15` не имеет смысла, получится полностью нулевой вектор. Здесь надо понять, что циклы просто описывают семантику таких операций, а в компьютере эти операции выполняются очень быстро (одной машинной командой).

В языке Free Pascal у операций сдвига есть синонимы, можно использовать обозначение `<<` вместо **shl** и `>>` вместо **shr** (привет, язык C 🙂). Главное назначение операций сдвига заключается в быстром умножении и делении целых чисел на степень двойки. Сдвиг влево на единицу (пока результат помещается в переменной) эквивалентен умножению числа на два:

```
x:=2*x ≡ x:=x+x ≡ x shl 1 ≡ x << 1
```

---

[1] Вообще говоря, на языке машины биты нумеруются справа-налево, так что лучше было бы писать

```
var x,y: bitpacked array[15..0] of boolean;
```

Но так на Паскале написать нельзя (`15..0` не есть тип, см. ниже ограниченный тип данных), и приходится мириться с этой неточностью.

Это верно как для беззнаковых, так и для знаковых чисел. А вот сдвиг вправо **shr** на единицу эквивалентен делению числа на два, но только для беззнаковых целых чисел (т.е. типов byte, word, longword и qword). Для знаковых типов shortint, smallint, longint, integer и int64 операция **shr** для отрицательный значений дают неверный результат в смысле операций деления на степень двойки, например:

```
var x: smallint=-3;
x:=x shr 1; write(x) {даёт не x div 2=-1, а 32766}
```

Для сдвигов величин знаковых типов предназначены стандартные, так называемые встраиваемые (inline) функции (см. разд. 8.1) с именами SarShortint, SarSmallint, SarLongint и Sarint64 (для величин типов shortint, smallint, longint и int64 соответственно).

Однако важно. При использовании этих функций правильный результат в смысле деления на степень двойки получается не всегда, а только для неотрицательных чисел и отрицательных чётных чисел. Для отрицательных нечётных чисел результат получается на единицу меньше требуемого. Впрочем, это можно хитро скорректировать, например:

```
var x: longint;
    x:=x div 4; { будет всегда то же самое, что и }
    x:=SarLongint(x,2)+ord((x<0) and odd(x));
```

Причину этого Вы узнаете из курса по архитектурам ЭВМ 🙁. Для всех операций сдвига существуют соответствующие машинные команды, и наоборот, для всех команд сдвига есть стандартные функции языка Free Pascal, например, для машинной команды **ror** есть функции RORByte, RORWord, RORLongword и RORQword.

3.4.5. Дискретные типы данных

По сути, я утверждаю, что разница между плохим программистом и хорошим заключается в том, считает ли он более важным свой код или свои структуры данных. Плохие программисты беспокоятся о коде. Хорошие программисты беспокоятся о структурах данных и их взаимоотношениях.

Луис Торвальдс. Создатель Линукса

Итак, изначально Паскаль (как и большинство других языков программирования) «знает» только стандартные скалярные типы. При необходимости работать с другими наборами величин, программист должен описать (сконструировать) свой собственный тип. Сейчас мы начнём изучать способы описания новых типов (некоторые из этих типов будут безымянными, другим мы дадим имена в разделе типов).

Сначала введём определение *дискретного* (или порядкового типа). Дискретным или порядковым (ordinal) называется такой скалярный упорядоченный тип, для любой величины которого (кроме первой и последней в этом типе) однозначно определено *следующее* и *предыдущее* значения. Заметим, что целый, символьный и логический типы являются дискретными, а вот вещественный уже нет. Действительно, какое число следующее, скажем, за 4.1? Может 4.1000001 или 4.1000002? Ясно, что это зависит от конкретной реализации Паскаля, более того, в языке Free Pascal ответ будет разным например, для вещественных типов single и double. Кроме того, как уже отмечалось, вещественный тип не является и полностью упорядоченным. [vii] [см. сноску в конце главы]

Для вычисления следующего и предыдущего значения дискретного типа в Паскале предусмотрены стандартные функции `succ(x)` и `pred(x)`, [1] например:

```
succ(5)=6       succ('5')='6'    succ(false)=true
pred(-7)=-8     pred('9')='8'    pred(true)=false
```

Конечно, для целого типа это эквивалентно прибавлению и вычитанию единицы, но для других дискретных типов следующее и предыдущее значения так не получить.

---

[1] Successor – последователь, predecessor – предшественник.

В языке Free Pascal для присваивания следующего и предыдущего значений дискретным переменным предусмотрены полиморфные процедуры `inc(x)` и `dec(x)`:

```
var x: integer; y: char; z: boolean;
begin x:=5; y:='5'; z:=false;
  inc(x); {x=6} inc(y); {y='6'} inc(z); {z=true}
  dec(x); {x=5} dec(y); {y='5'} dec(z); {z=false}
```

Существуют также варианты этих процедур с двумя аргументами `inc(x,n)` и `dec(x,n)`:

```
begin x:=5; y:='5'; z:=false;
  inc(x,3); {x=8} inc(y,3); {y='8'}
  inc(z,3); (* z=true в режиме {$R-} и АВОСТ при {$R+} *)
  dec(x,2); {x=6} dec(y,2); {y='6'} dec(z,2); {z=true}
```

3.4.6. Ограниченные типы

Определить – значит ограничить.
Оскар Уайльд
писатель, поэт, философ и эстет ⚠

Для всех дискретных типов в Паскале определён способ описания (конструктор) нового, так называемого ограниченного типа (subrange type).[1] По сути, это просто один непрерывный диапазон (сегмент) значений конкретного дискретного типа. Синтаксис описания ограниченного типа:

<ограниченный тип> ::= <константа>..<константа>

К этому определению прилагается семантический фильтр: обе константы должны быть одного дискретного типа, и первая из них не больше второй (так как пустые типы в Паскале запрещены). Например:

```
1..100       { ограниченный целый тип }
'0'..'9'     { ограниченный символьный тип }
true..true   { ограниченный логический тип }
'-'..'+'     { ОШИБКА, если в алфавите '-'>'+' }
```

Как видим, каждый ограниченный тип имеет свой базовый тип (base type), из которого он получен. В показанных выше примерах базовые типы подчёркнуты.

Семантика ограниченных типов очень проста: любая величина ограниченного типа полностью синтаксически эквивалентна любой величине своего базового типа. Другими словами, если в правильной программе заменить величину ограниченного типа на величину её базового типа (и наоборот), то программа останется синтаксически правильной. Некоторых учащихся это смущает, например:

```
var x: integer; y: 1..100; z:-10..100
```

Тогда в синтаксически правильной программе оператор `x:=y` можно заменить на `y:=x` или на `y:=z` и программа по-прежнему будет компилироваться без ошибок. Спрашивается, а что будет, если `x=200`? В стандарте Паскаля сказано, что значение переменной `y` при этом будет *не определено*, т.е. зависит от конкретной реализации. Для языка Free Pascal ответ Вы уже знаете: при включенном контроле выхода величин за допустимый диапазон (была директива `{$R+}`) произойдёт аварийный останов, в противном случае переменной `y` (скорее всего) будет присвоено *неправильное* значение.

Здесь любознательных учащихся часто интересует, а что же будет присвоено в режиме `{$R-}` переменной `y` в языке Free Pascal? Так как переменная `x` типа `integer` (в обычном режиме) занимает в памяти 2 байта, а переменная `y` типа `1..100` всего один байт, то при присваивании `y:=x` значение `x` просто усекается до одного (младшего байта). Таким образом:

```
x:=200; {200=128+64+8=0000 0000 1100 1000₂}
```

---

[1] Иногда их называют ещё интервальными типами, скоро мы поймём, почему.

```
y:=x;    {y=1100 1000₂=200 – правильно !}
x:=400;  {400=256+128+16=0000 0001 1001 0000₂}
y:=x;    {y=1001 0000₂=144=400 mod 256 – не правильно}
z:=x;    {z=1001 0000₂=-112 ⚠ – это архитектура ЭВМ}
```

Рассмотрим теперь прагматику ограниченных типов, другими словами, зачем они нужны. Допустим, программист знает, что по условию его задачи величина y, хоть и является целочисленной, но всегда должна лежать в диапазоне `1..100`. Тогда он описывает y как переменную ограниченного целого типа, и во время отладки программы включает контроль `{$R+}`. Удивительно, как много ошибок при этом находится в на первый взгляд правильной программе. Так что рекомендуем 🙂.

Ограниченные типы являются *безымянными*, при необходимости им можно дать имя в разделе типов, например:

```
type Small=1..100; var y: Small;
```

Отметим, что в языке Free Pascal многие *стандартные* типы данных по существу являются ограниченными целыми, например:

```
type shortint=-128..127; byte=0..255; { и т.д. }
```

3.4.7. Перечислитый тип данных

Знаки и символы правят миром, а не слово и закон.

Конфуций, V век до н.э.

Часто в программах приходится работать с наборами величин, которые не являются ни числами, ни символами, ни логическими величинами. Это, например, список городов, набор цветов, перечень деталей машины, дни недели и т.д. Почти все языки программирования в этом случае предлагают просто перенумеровать такие объекты целыми числами, например, `"понедельник"=0`, `"вторник"=1` и т.д. При этом, однако, появляется возможность производить над такими объектами обычные арифметические операции над целыми числами (сложение, умножение и т.д.). В то же время никакого разумного смысла выражениям `"понедельник"+"суббота"` или `4*"среда"` придать нельзя, скорее всего это просто семантические ошибки в программе.

Именно для таких случаев Паскаль позволяет создавать принципиально новые, так называемые перечислимые (enumerated) типы данных. Так как тип – это просто набор констант (с одинаковыми операциями над ними), то предлагается создать новый тип, просто перечислив в круглых скобках все константы этого типа, предварительно присвоив им имена. Например:

```
var x: (ПН,ВТ,СР,ЧТ,ПТ,СБ,ВС);
```

Для повышения читаемости и выразительности мы здесь пошли на «вольность», присвоив для константам дней недели русские имена, конечно, в «настоящей» программе на Паскале надо использовать `(Mo,Tu,We,Th,Fr,Sa,Su)`. Итак, мы описали новый (безымянный) тип, в котором всего 7 величин, упорядоченных в порядке их перечисления. Вам важно понять, что это не числа и не строки. Здесь учащиеся часто спрашивают, а чему равен ВТ? Ну, чему равен вторник, конечно, только самому вторнику 🙂. Вообще говоря, таковы все константы языка, например, чему равна (какое значение имеет) целая константа 5 ? Здесь же можно спросить, а чему равен символ языка Паскаль **begin** ? Отметим, что во всех языках программирования тоже есть «символы, значениями которых являются только они сами и ничего больше» 🙄.

Таким образом, над величинами перечислимого типа в Паскале определены только операции сравнения. Очевидно, что `ПН<СР`, а `ПТ>ВТ`. Перечислимый тип является дискретным, для него определены стандартные функции `ord`, `succ` и `pred`, например:

```
ord(ПН)=0, succ(ВТ)=СР, pred(ВС)=СБ
```

Конечно же, в стандарте Паскаля сказано, что `succ(ВС)` или `pred(ПН)` не определены. Для работы с перечислимыми типами этим типам рекомендуется давать имена, например:

```
type Week=(ПН,ВТ,СР,ЧТ,ПТ,СБ,ВС);
var  x: Week;
```

Далее, как и для всякого дискретного типа, его можно использовать как базовый и ограничить:

```
var y: ПН..ПТ; { Будние дни } z: СБ..ВС; { Выходные дни }
```

Разумеется, ограниченному типу тоже можно задать имя, например:

```
type Week=(ПН,ВТ,СР,ЧТ,ПТ,СБ,ВС);
     WorkDays=ПН..ПТ;
```

Здесь важно понять, что описывая перечислимый тип, мы одновременно описываем и его константы, так что, например, константу с именем Green можно описать двумя способами (точнее, одним из двух):

```
const Green=2;                  { 1-й способ }
var   x:(Red,Yellow,Green);     { 2-й способ }
```

В языке Free Pascal константам перечислимого типа можно задавать свои произвольные порядковые номера, например:

```
type
  Memory=(b=1,Kb=1024,Mb=1024*1024,Gb=1024*1024*1024);
  Colors=(Black,Red,Yellow=5,Green);
{Тогда}
  ord(Black)=0; ord(Red)=1; ord(Yellow)=5; ord(Green)=6
```

В языке C есть внешне похожий перечислимый тип, переменная этого типа X будет описана как

```
enum Colors {Black,Red,Yellow=5,Green} X;
```

При этом, однако, для переменной X определено *неявное* преобразование в целочисленных тип, т.е. можно писать `3*X+1` (что это за цвет 😳). Такая «несуразность» исправлена в современных языках C++, C# и Java 5.0, где величины типа **enum** хотя и имеют целочисленные номера, но использовать эти величина в челочисленных операциях, как и в Паскале, нельзя, т.е. `X=1` будет ошибкой.

Отметим, что близкий к Паскалю перечислимый тип существует в уже упоминавшемся ранее языке Ада. Похожий тип можно сконструировать и в языке Haskel:

```
data Week = Su | Mo | We | Th | Fr | Sa
```

Правда, в отличие от других языков, этот тип не будет скалярным, это просто множество величин, при необходимости функции succ и pred программист должен написать сам.

В языке Free Pascal для скалярных типов, переменных и констант этих типов определены стандартные функции с именами low и high, которые выдают соответственно минимальное и максимальное значение типа, например:

```
type Week=(Mo,Tu,We,Th,Fr,Sa,Su);
     int=-13..245;
var w:Week; i:int;
{ low(w)=low(Week)=Mo; high(w)=high(Week)=Su }
{ low(i)=low(int)=-13; high(i)=high(int)=245 }
{ low(byte)=0; high(byte)=255 }
{ ⬇⬇ по умолчанию считает константу 1 типа shortint ⚠}
{ low(1)=-128; high(1)=127;
{ low(char)=#0; high(char)=#255 }
```

Язык Free Pascal позволяет выводить значения перечислимых типов в виде имён их констант и вводить значения перечислимых типов в виде целых величин, например:

```
type Week=(Mo,Tu,We,Th,Fr,Sa,Su);
var  x: Week;
begin x:=Tu; Write('x=',x)  { ВЫВОД x=Tu }
  Read('ord(x)=',ord(x))    { ПУСТЬ ВВОД 5 }
  Write('x=',x,',ord(x)='ord(x)) { ВЫВОД x=Sa,ord(x)=5 }
```

Прагматика перечислимых типов очевидна: когда программист хочет уберечь «особые» переменные от случайного изменения обычными числовыми операциями, он описывает новый перечислимый тип. Это, несомненно, увеличивает надёжность программ.

Вопросы и упражнения

Образование есть то, что остаётся после того, когда забывается всё, чему нас учили 🙂.

Альберт Эйнштейн

1. Что такое программа на стандарте Паскаля?
2. Чем понятие `character` отличается от `symbol` ?
3. Чем понятие `line` отличается от `string` ?
4. Что такое лексемы?
5. Чем в Паскале отличается стандартное имя от служебного?
6. Чем лексемы целых чисел отличаются от лексем вещественных чисел?
7. Какие законы арифметики для целых чисел не выполняются в дискретной математике?
8. Почему в языках программирования целых и вещественных чисел конечное количество?
9. Почему целочисленная ось в языках программирования несимметрична?
10. Как выполняются операции с целыми и вещественными числами?
11. Зачем в языке Free Pascal несколько целых и вещественных стандартных типов?
12. Как вычисляется длина строки в апострофах?
13. Для чего нужны лексемы-разделители?
14. Когда нужно давать имена константам?
15. Как понимать, что переменная не имеет ссылки?
16. Как порождаются переменные а программе?
17. Что такое тип?
18. Чем отличаются простые типы от сложных?
19. Что такое дискретные типы данных?
20. Как в языках программирования вычисляется остаток от деления целых чисел?
21. Чем отличается неявное преобразование типов от явного?
22. Как определяется, какой из двух типов старший?
23. Объясните, что в языке Free Pascal делает оператор присваивания
    ```
    x:=Week((ord(x)+1) mod 7);
    ```
24. Что такое таблица истинности?
25. Почему вещественный тип не является дискретным?
26. Пусть `var x: integer;` Определить, какую математическую функцию `f(x)` задаёт выражение
    ```
    f(x)=ord(x>0)-ord(x<0); { ? }
    ```
27. Почему нельзя сделать компилятор со стандарта Паскаля?

---

[i] Для продвинутых читателей. Большинство ЭВМ при работе с целыми числами используют так называемую систему счисления с *дополнительным кодом*. В этой системе (ограниченная) целочисленная ось как бы замкнута в кольцо: при прибавлении единицы к самому большому числу `Maxint` получается самое маленькое (отрицательное) число, и наоборот. Операции над целыми числами при этом выполняются как вращения в этом замкнутом в кольцо *подмножестве* всех целых чисел. Математики говорят, что такие *целые* машинные числа образуют (по операциям сложения и умножения) *кольцо вычетов* по модулю 2^N, где N – разрядность числа (обычно 8, 16, 32 или 64 бита).

Как ни странно, для наших ЭВМ при работе с выключенным контролем и выходе результата за допустимые пределы, программа в итоге может дать и *правильный* результат. Дело в том, что при выполнении машинной команды сложения целых чисел, если сумма выходит за допустимый диапазон, то аварийного завершения программы не происходит. Вместо этого выдаётся *неправильный* ответ и поднимается так называемый флаг переполнения, который, при желании, программист на Ассемблере может проанализировать после команды сложения. Так вот, если не обращать на переполнение внимания, и результат выйдет за допустимый диапазон, но при последующих операциях снова попадёт в этот допустимый диапазон, то ответ будет правильным ⚠.

Можно считать это одним из достоинств системы счисления с дополнительным кодом, хотя у этой системы есть и свои недостатки. При программировании на языке Free Pascal это можно использовать, если не задавать директивы `{$R+}` и `{$Q+}` для контроля выхода величин за диапазон их допустимых значений. Стоит, однако, заметить, что для целых беззнаковых (их нет в стандарте Паскаля) и вещественных чисел такой трюк уже не работает.

[ii] При реализации на ЭВМ крайние точки на вещественной оси резервируются за особыми значениями ±∞. Эти значения обычно выдаются в качестве результата арифметических операций с вещественными числами, если этот результат такой большой по абсолютной величине, что не представим среди множества машинных вещественных чисел. Процессор достаточно «разумно» (по крайней мере, с точки зрения математика) производит арифметические операции над величинами ±∞. Например, пусть A любое «обычное» вещественное число, не равное нулю, тогда:

±A/0 = ±∞; ±A/±∞ = ±0; A * ±∞ = ±∞; ∞ + ∞ = ∞; (-1.0)*(-∞) = +∞;

Разумеется, результаты некоторых операций не существуют, тогда они обозначаются специальным вещественным значением «не число» NaN, например:

±A ± NaN = NaN; ±0/±0 = NaN; ∞ - ∞ = NaN; sqrt(-1.0) = NaN;

Это тема курса по архитектурам ЭВМ.

[iii] Для продвинутых читателей. Приближенный результат операций над вещественными числами является следствием *округления* точного результата до ближайшей представимой точки на вещественной оси. При выполнении бинарной операции ошибки округления результата тем больше, чем больше *порядки* двух операндов отличаются друг от друга. В компьютере ошибка округления каждой машинной операции над вещественными числами очень мала, но при выполнении длинной цепочки таких операций имеет тенденцию накапливаться и далеко «уводить» полученное решение задачи от точного решения.

Большинство используемых систем счисления имеют чётные основания, к ним принадлежит и наша десятичная, и все машинные двоичные системы счисления. В таких системах счисления математическое ожидание величины накопленной ошибки округления не равно нулю, что может приводить к нарастанию ошибок округления. Существуют разные способы округления (см. далее описание стандартной функции round), но даже в лучшем (так называемом банковском), способе округления встречаются случаи, когда существуют два кандидата на лучшее округление, и между ними приходится делать выбор.

В то же время среди систем с *нечётным* основанием существуют так называемые *сокращённые* системы счисления, в которых простейшее округление путём отбрасывания лишних разрядов имеет *нулевое* математическое ожидание ошибок округления. Такая простейшая троичная система счисления описана в приложении 1 (Системы счисления). К сожалению, по чисто техническим причинам в компьютерах такие системы счисления реализуются крайне редко.

Существуют, однако, алгоритмические приёмы, как в вычислениях с вещественными числами снизить ошибку округления в длинных цепочках операций. В качестве примера рассмотрим алгоритм так называемого компенсационного суммирования, который предложил Уильям Кэхэн (тот самый, который и разработал стандарт вещественных чисел IEEE-754). Пусть надо просуммировать вектор вещественных чисел

```
var x: array[1..N] of double;
```

Будем хранить частичную сумму вектора в виде двух чисел S_i и ds_i, где ds_i – это величина, связанная с накопленной при суммировании ошибкой округления. Предлагается такой «хитрый» алгоритм суммирования массива:

```
var S: double=0.0; ds: double=0.0; y,z: double;
     . . .
for i:=1 to N do begin
    y:=x[i]-ds; z:=S+y; ds:=z-S;
    ds:=ds-y; S:=z
end;
```

На каждом шаге величина полученной суммы S_i компенсируется величиной y, которая зависит от очередного слагаемого x_i и предыдущей накопленной ошибки округления ds_i. Ошибка ds_i, в свою очередь, перевычисляется на каждом шаге суммирования. Таким образом, в идеале должно быть $S_i+x_{i+1}=S_{i+1}+ds_{i+1}$. При обычном суммировании погрешность имеет среднее квадратичное отклонение $O(\sqrt{N})$, а при компенсационном суммировании эта погрешность не зависит от N. Конечно, при этом сложность операции суммирования возрастает в несколько раз. Погрешность возрастает, когда порядки слагаемых сильно различаются, поэтому данный метод лучше всего работает, если сначала отсортировать массив.

[iv] Для продвинутых читателей. Существование неинициализированных переменных (со случайным значением) является проблемой для программиста, так как его алгоритм становится не детерминированным. Например, переменная порождается, но затем в неё не вводятся входные данные и ей не присваивается никакого конкретного значения. При этом программа может при разных запусках вести себя по другому, что сильно осложняет отладку.

Для решения этой проблемы в стандарте представления *вещественных* чисел предусмотрено специальное значение «не число» `SNaN` (Signaling Not a Number). Программист может инициализировать свою переменную этим значением, тогда можно включить режим работы, когда при попытке совершения с ней операций (например, сложения) будет возникать исключительная ситуация.

При порождении сложной переменной (массива, структуры и т.д.) может понадобится инициализировать её весьма «хитрым» образом (например, присвоить каждому элементу `X[i]` значение `i`). В современных языках программирования (и в языке Free Pascal) для этой цели программист может создать специальные функции-конструкторы, которые автоматически вызываются при порождении такой переменной.

---

[v] Для продвинутых читателей. В большинстве языков с динамической типизацией (например, в языках Python, Ruby или JavaScript), где тип переменной может *меняться* во время счёта, существует только переменные *динамического* класса памяти. При этом понятие переменной трактуется по другому. В императивных языках это место в памяти, с которым (возможно) связывается имя и значение, то сейчас это, наоборот, имя, с которым связано место в памяти и значение переменной. Место в памяти этим переменным отводится при каждом присваивании имени переменной нового значения, в этот момент определяется и тип переменной. Таким образом, имя является как бы ярлыком, (label) который вешается на место памяти, и при выполнении каждого оператора присваивания этот ярлык перевешивается на новое место памяти.

Например, для языка Python:

```
X=3.14
```

Здесь переменная с именем `X` связывается (binding) с новой областью памяти вещественного типа и значением `3.14`.

```
X="abc"
```

А теперь переменная `X` связывается с новой областью памяти строкового типа и значением `"abc"`, при этом старая область памяти (со значением `3.14`) уничтожается, т.е. освобождается и в дальнейшем может быть отдана для другого значения. Такие объекты называются иммутабельными (immutable – неизменяемыми), после размещения в памяти их нельзя менять, это своеобразные константы!

Конечно, в языке Python есть и мутабельные типы (mutable type) данных, объекты этих типов, это «настоящие» переменные, их можно менять в памяти ЭВМ. В частности, это почти все объекты сложных типов (массива, множества, структуры и т.д.) Понятно, что так и должно быть, если, скажем, сделать массив иммутабельным, то при изменении любого его элемента придётся уничтожать старый массив, и порождать новый, что неприемлемо по времени.

В сложных языках (скажем, в так называемых объектно-ориентированных), тип *может меняться*, например, в разных частях программы переменная (объект) `X` данного типа (класса) имеет разный размер. Как видим, всё достаточно сложно 🙁.

[vi] Для продвинутых читателей. Первые простые языки программирования могли кодировать тип переменной прямо внутри её имени. Например, в первых версиях языка Basic тип определялся окончанием имени, например:

```
A   – вещественная переменная,
A!  – вещественная переменная одинарной точности,
A#  – вещественная переменная двойной точности,
A%  – целая переменная,
A$  – строковая переменная и т.д.
```

Для ускорения доступа к переменным, исполнитель может отводить под них больше места в памяти, чем им необходимо (почему так происходит изучается в курсе по архитектуре ЭВМ). Например, в языке Free Pascal под скалярные переменные отводится по 16 байт памяти (это называется *естественный размер* переменных). Говорят, что происходит *выравнивание* переменных в памяти на границу 16 байт. Таким образом, однобайтная переменная (например, типа `char`) всё равно будет занимать в памяти 16 байт. Такое стандартное выравнивание не выполняется для элементов массива и можно отключить внутри записи (см. главу 9), для полей которой можно задать своё выравнивание директивой `{$`**`align`** `N}`, где `N=1,2,4,8,16` или `32` – границы выравнивания полей в записи.

В разделе переменных Паскаля производится явное описание переменных (explicit declaration). Некоторые языки (Python и др.) допускают неявное описание переменных (implicit declaration), при этом переменные порождаются при присваивании им значения. Есть языка, которые используют смешанную стратегию, например, если в языке Фортран переменная с некоторым именем не описана явно, но ей присваивается значение, то порождается новая переменная, тип которой определяется по первой букве имени: `I`, `J`, `K`, `L`, `M`, `N` – целый тип, остальные буквы – вещественный тип.

Использование неявного описания переменных наносит серьёзный ущерб надёжности программ, в частности могут порождаться переменные при явных «очепатках» программиста. Так, запущенный к Венере космический корабль Маринер-1 был взорван вскоре после старта, так как в программе управления кораблём (написанной на Фортране) была ошибка. По одной из версий в заголовке оператора цикла

```
DO 17 I = 1, 10
```

вместо запятой была поставлена точка, в результате этого оператор цикла превратился в оператор присваивания `DO 17 I = 1. 10`, здесь переменной с именем `DO17I` присвоено вещественное число `1.10` (пробелы в именах и числах игнорируются 🐷).

[vii] Для продвинутых читателей. Из матанализа Вам должно быть известно, что в математике у каждого вещественного числа нет ни непосредственно следующего за ним, ни непосредственно предшествующего ему вещественного числа. В большинстве компьютеров, однако, вещественные числа представлены в стандарте ANSI/IEEE 754-1985, (представимых) вещественных чисел там *конечное* число, поэтому на них можно искусственно определить дискретность, впрочем, она будет разная для разных типов (`single`, `double` или `extended`).

Для случая неотрицательных вещественных чисел можно воспользоваться таким свойством этих чисел. Пусть есть два вещественных числа $X_{вещ}$ и $Y_{вещ}$ и причём $0 \le X_{вещ} < Y_{вещ}$. Можно взять битовые представления этих чисел и рассматривать их как целые числа X_{int} и Y_{int}, тогда будет $X_{int} < Y_{int}$.

Например, возьмём вещественные числа типа `single` (4 байта) и целые числа типа `longint` (4 байта), тогда `succ(single)≡succ(longint)`. На языке Free Pascal это можно реализовать, используя просто наложение переменных в памяти ЭВМ (об этом будем говорить позже):

```
var x: Single; y: Longint absolute x;
   { x:=succ(x) ≡ y:=succ(y) }
```