Языки программирования, базовый курс

Введение

Меня дважды спрашивали: «Скажите, мистер Бэббидж, что случится, если вы введете в машину неверные цифры? Сможем ли мы получить правильный ответ?» Я не могу себе даже представить какая путаница в голове может привести к подобному вопросу. Чарльз Бэббидж

При изучении программирования возникает много проблем. Одна из них связана с тем, что мы изучаем язык, и, естественно, пытаемся провести параллели с уже известными нам языками - английским, немецким... Изучая английский, мы не спрашиваем, почему в нем нет падежей или почему эти глаголы неправильные, а не вон те, хотя лингвисты знают ответы и на эти вопросы. В отличие от естественных языков, языки программирования появились недавно, всего лишь около 60 лет назад, все статьи и книги, посвященные предмету, сохранились, и восстановить последовательность появления в языках программирования, например, объектов или генераторов, не составляет особенного труда.

Конечно, сказанное не относится только к программированию. Многие придуманные людьми вещи появляются в школьном курсе как данность. Предполагается их изучение и запоминание, но не выяснение вопроса «а почему, собственно, оно было сделано именно так, а не иначе».

Цель данного текста — заполнить образовавшуюся пустоту и, во-первых, рассказать о том, как появлялись и эволюционировали языки программирования, а во-вторых, показать общие принципы их построения. Для успешного освоения курса нужно знать хотя бы один язык программирования. Большинство примеров даны на C, Pascal или Python, причем, в большинстве случаев, без какого-либо предпочтения
тому или другому языку.

Этот текст написан под сильным влиянием предыдущей работы на эту тему - книги Дэвида Баррона «Введение в языки программирования», перевод которой вышел в издательстве МИР в 1980 году. Для своего времени книга была необычная, яркая и запоминающаяся, прежде всего легкостью изложения, а также взглядом на проблемы сверху, с высоты птичьего полета. К сожалению, в наши дни эта замечательная книжка безнадежно устарела. Многие языки, которые сравнивает автор, сейчас остались только в виртуальных музеях. Из книги Баррона я позаимствовал общую структуру изложения и манеру ставить эпиграфы. Некоторые явно устаревшие языки, вроде Fortran будут, тем не менее, подробно разобраны, поскольку разбор ошибок, допущенных при их конструировании, может быть полезным для последующих поколений изобретателей языков.

Немного истории

Первым полноценным языком программирования был Fortran, придуманный в 1954 году, и впервые реализованный в 1957-м. Годом позже появились Algol и Lisp. Удивительно, что большинство особенностей языка Fortran образца 1954 года в наше время считаются устаревшими, но, в отличие от него, Algol был вполне современным языком. Достаточно сказать, что популярные в наше время языки Pascal и C (а также Java, PHP и C#) сделаны на основе языка Algol путем добавления некоторых усовершенствований. А вот следующий по счету язык Cobol был опять снабжен невероятным количеством особенностей, которые не пригодились в дальнейшем.

Сам язык Algol в начале XXI века не используется вообще, а вот Lisp прожил полвека почти без изменений. Любопытно также, что развитие искусственных языков програмирования шло тем же путем, что и развитие естественных: от сложного к простому. Так же, как из русского языка исчезли несколько букв, пара падежей и двойственное число, из языков программирования исчезали слишком сложные особенности. Современные языки, как правило, проще старых (C++ можно считать исключением). Некоторые устаревшие особенности языков мы будем специально рассматривать.

Роль языка программирования

Внутри компьютера естественные языки выглядят неестественно
Алан Перлис

Посмотрим, как происходит процесс создания программы. Сначала необходима **постановка задачи**. То есть понимание того, что именно программа должна делать и зачем она нужна. С точки зрения нашей абстракции исполнения мы должны сказать, какие начальные состояния нас интересуют и какие конечные состояния мы хотим получить. Например, на входе — последовательность покупок в универсальном магазине, а на выходе — месячный отчет о продажах. На входе — текст, на выходе — список орфографических ошибок в этом тексте. И так далее.

Второй этап - придумывание алгоритма (или реализация уже известного). Часто этот этап не отделяют от третьего - кодирования. Разница между вторым и третьим этапами состоит в том, что алгоритм, может не иметь отношения к языку программирования. Алгоритм можно описать словами, картинкой или как угодно еще.

> 20 января 2000 года судья Южного Округа Нью-Йорка вынес решение, запрещающее распространять код программы, способной декодировать защиту DVD дисков. Программистское сообщество программистов отреагировало на этот запрет своеобразным видом протеста. Пусть запрещено публиковать текст программы, решили они, но есть статья Конституции, гарантирующая свободу слова. А значит, можно создать некое представление алгоритма, которое само не являлось бы программой, но по которому можно было бы легко восстановить текст запрещенной программы. Таким образом впервые в истории был проведен эксперимент по описанию алгоритма наибольшим возможным числом способов *без использования языка программирования*. Было создано несколько десятков различных способов представления алгоритма **decss**. В том числе: изложение в виде текста на английском языке, математических формул, чтения текста программы вслух, электронной схемы, рисунков, фильма и даже в виде хокку.

Язык определяет способ мышления

Согласно гипотезе лингвистической относительности, иначе называемой «гипотеза Сепира-Уорфа», люди, говорящие на разных языках, по-разному воспринимают мир и по-разному мыслят. Неизвестно,

верна ли эта гипотеза в отношении естественных языков, но для языков программирования в этом утверждении имеется большая доля истины.

Анализируя алгоритмы, которые используют программисты на Java, Pascal или C, легко заметить сходство приемов и методов. Дело в том, что, несмотря на изобилие способов представления алгоритмов, программисты обычно используют для придумывания алгоритма тот же язык программирования, на котором пишут программу, и сочиняют алгоритм, исходя из возможностей этого языка. Иначе говоря, они думают на языке программирования.

Такая двойственность алгоритма (с одной стороны независимость от языка, а с другой - тесная связь с языком) объясняется достаточно просто. Многие языки программирования схожи между собой. Соответственно, алгоритмы, придуманные для одного языка, легко (или не очень легко) переносятся на другой. Поэтому можно говорить о представлении алгоритмов на некотором «обобщенном» языке программирования, существующем в воображении программиста, но при этом особенности языка учитываются при поиске оптимальных решений. О сходстве языков мы будем много говорить в течение всего этого курса, а сейчас обратим внимание на различия. Анализ различий позволяет нам построить классификацию или как-то систематизировать предмет изучения.

Рассмотрим характеристики языков программирования.

Субъективные характеристики языков программирования

Для начала рассмотрим те характеристики языков программирования, которые не связаны с какими-то формальными особенностями этих языков. К числу субъективных характеристик языка относятся мощность, простота, красота (элегантность). Рассмотрим примеры.

Мощность языка — это возможность не слишком сложными конструкциями решать достаточно сложные задачи.

Простота языка обычно определяется так называемым принципом Оккама: не следует умножать число сущностей сверх необходимости.

Красота языка обычно понимается как соответствие формы содержанию и как элегантность решения тех или иных проблем проектирования языка. Любой искусственный язык это своего рода «игра ума», и некоторые языки с этой точки зрения действительно выглядят красиво. Существуют языки, у которых красота — одно из наиболее существенных достоинств. Таковы, например, Forth, Haskell и Prolog.

Язык Forth замечателен тем, что в нем любая конструкция может быть выражена через другие. Программист может создать свой собственный `IF`, свой собственный `WHILE` и так далее. Языки Haskell и Prolog построены на базе строгих математических теорий.

Внешняя форма

Говорят, что по одежке встречают. Внешняя форма языка - первое, с чем приходится сталкиваться при его изучении. К общим элементам внешней формы языков относятся следующие элементы:

Пробелы и разбивка. Почти все современные языки допускают свободное размещение текста программы - вставку произвольного числа пробелов в тех местах, где допустим хотя бы один пробел, а также перенос строки на тех местах, где допустимы пробелы. Однако так было не всегда. Ранние языки, такие как FORTRAN, требовали от программиста совершенно определенного размещения

текста программы на странице. А именно, в случае FORTRAN IV, первые 6 позиций строки должны были быть пробелами или содержать метку программы. Если в них появлялась буква C, то это означало комментарий.

```
C Это комментарий
C Это тоже комментарий
C На следующей строке находится оператор присваивания
      X = 10
C А следующая строка - ошибка, так писать нельзя
  X = 10
C А на следующей строке находится метка и оператор
 10   CONTINUE
 C Эта строка тоже ошибка, перед C не должно быть пробела
C А в конце программы обязательно должен стоять END
      END
```

Интересно, что в Fortran пробелы играют важнейшую роль в шести первых позициях каждой строки, но дальше они полностью игнорируются.

Из современных языков форматирование интенсивно используется в языке Python, где оно играет ту же роль,что и фигурные скобки в языке C или begin ... end в Паскале.

```
if x < 0:
   x = 0
   print ('Было слишком мало, теперь ноль')
elif x == 0:
   print ('Ноль')
elif x == 1:
   print ('Один')
else:
   print ('Много')
```

Зарезервированные слова. Большинство языков содержат так называемые ключевые слова, имеющие некое предопределенное значение в данном языке. Примером могут служить слова `for`, `while` и `if`, являющиеся ключевыми в очень многих языках. Существуют два основных подхода к ключевым словам. Согласно первому из них, ключевые слова не могут использоваться в программе ни в каком другом значении. Невозможно сделать переменную с именем `for`, или функцию с именем `if`. Второй подход состоит в дозволении подобных названий, и выяснении смысла происходящего из контекста. В некоторых языках используются оба подхода: часть ключевых слов «абсолютны», а часть программист может использовать по своему усмотрению.

Начиная с языка Algol-60, созданного в 1960 году, ключевые слова, такие как `for` и `if`, считаются чем-то неделимым, не поддающимся разбиению на отдельные символы. В некотором смысле до логического предела эту идею довели создатели компьютера Спектрум, в языке Basic которого ключевые слова *и были* отдельными символами. То есть при выводе на экран символа с кодом 235 отображалось слово **FOR**, а символ с кодом 250 соответствовал **IF**. Напротив, строки символов обычно рассматривают как изначально состоящие из отдельных символов. Так, операция поиска и замены

имеет смысл для строк, но не для ключевых слов языка. В некоторых языках программирования, например, в Prolog и Erlang имеется нечто среднее между символами языка и строками. Это нечто называется **атомы**, и это что-то вроде неделимых, то есть не состоящих из отдельных символов строк, которые определяет сам программист.

Комментарии. Уже в самых первых языках программирования появилась возможность вставлять в программу произвольный текст на естественном языке. Обычно используют два вида комментариев: блочные и строчные. Комментарии первого вида начинаются с определенного символа или сочетания символов, и продолжаются до другого предписанного сочетания символов. Основная проблема, которая возникает здесь, это так называемые вложенные комментарии.

```
int i;
/* это комментарий
/* а это вложенный комментарий */
   является ли этот текст комментарием? */
```

Если комментарий определяется как «текст, начинающийся с `/*` и заканчивающийся `*/`», то третья строка приведенного примера не считается комментарием. Если же комментарий это «текст, заключенный в скобки вида `/* */`», то третья строка - комментарий. Разные языки по-разному решают эту проблему. Второй вид комментариев проще - они начинаются с предопределенного «заклинания» и продолжаются до конца строки. В языке C++ и производных от него это сочетание `//`, встречаются также `#` и `--`. Набор символов. Чаще всего в языках программирования используют символы латинского алфавита и знаки препинания. Используемый в программировании набор символов, как правило, определяется стандартной клавиатурой, но с другой стороны, символы, находящиеся на стандартной клавиатуре, отчасти определяются часто используемыми языками программирования. Часто ли в обычных текстах встречаются фигурные скобки `{` и `}`? А в языке C они используются очень часто. До повсеместного распространения языка C на стандартной клавиатуре таких символов не было. В ранних стандартах этого языка даже были специальные сочетания общеупотребительных символов, которыми можно было заменять фигурные скобки.

Иногда изобретатели языков программирования выдумывают свои собственные символы. Таких много в языке ALGOL, а малоизвестный ныне язык APL в основном из них и состоит. Недостаток такого подхода очевиден - кроме самого языка, программисту приходится изучать еще и символы. (Это одна из проблем, возникающих при изучении некоторых естественных языков, например, японского или грузинского) Наверное, не все с ходу смогут это сделать. В заключение о наборе символов можно еще сказать, что в некоторых языках существует различие между прописными и строчными буквами, а в некоторых нет. В пользу и того и другого подходов есть аргументы, но в целом это дело вкуса и привычки.

Как правило, комментарии служат (вы, наверное, догадались) для комментирования программы. Однако, в некоторых случаях в язык вводят специальные виды комментариев. Например, в языке Java комментарии, начинающиеся с `/**` используются для автоматического создания документации к программе.

```
/**
 * Эта программа нажимает на кнопку.
```

```
 * <p>
 * Если у вас есть кнопка, ее можно нажать
 * вызвав эту процедуру
 * </p>
 * И еще какое-то длинное объяснение того, что тут
 * происходит.
 *
 * @param  btn это описание параметров
 * @return a это описание возвращаемого значения
 */
public int pushButton (Button btn) {
    ...
}
```

То же самое есть в языке Python

```
def pushButton(button):
    """ Это документация к функции pushButton
        которая нажмет на кнопку, если это
        зачем-то нам понадобится
    """
    ...
```

В отличие от Java, в программе на Python такие описания доступны даже во время выполнения программы. Формально в Python это не совсем комментарий, однако, эти строки никак не влияют на выполнение документируемой функции.

Еще один способ нетрадиционного использования комментариев состоит в том, чтобы давать указания компилятору. Например, в языке Python при помощи следующего причудливого комментария:

```
# -*- coding: utf-8 -*-
```

можно сообщить, что программа написана в кодировке utf-8

В языках C и Haskell тоже используются комментарии специального вида. Например, если программа на Haskell сгенерирована автоматически из какого-то исходника, и мы хотим видеть в отладчике не сгенерированную программу на Haskell, а исходник, нужно использовать такие комментарии:

```
{-# LINE 42 "source.program" #-}
```

Язык и пунктуация

Часто задают вопрос: почему основу большинства языков программирования составляют слова, заимствованные из английского языка? Нельзя ли сделать язык программирования на основе русского или суахили? Можно, но возникает целый набор неожиданных проблем. В английском языке нет родов

и падежей, простая структура фразы и не очень развитая грамматика. Все это позволяет легко конструировать искусственные фразы стандартного вида. Как на русский язык перевести оператор

```
for i=1 to N do
```

"Для i равного 1 до N делать"... "делай"? "повторять"? "повтори"? Фразы языков программирования выглядят не слишком красиво на естественном языке, но все уже привыкли. Мы также привыкли обозначать математические функции латинскими словами, мы пишем `sin x`, а не `син x`. Большая часть литературы по языкам программирования выходит на английском, так что создатели языков английский обычно знают.

И, кроме того, у этой проблемы есть простое и элегантное решение: сделать язык программирования вообще без заимствованных из естественного языка слов. Такие языки есть, например, APL, но они тоже не пользуются особой популярностью.

Команды и данные

> *Метапрограммы это такие программы, которые рассматривают самих себя и другие программы как данные.*
>
> Андерс Хейлсберг, создатель Delphi и C#

От внешнего вида программ перейдем к их функционированию. И тут же обнаружим одно важное разделение, а именно на программный код и данные. Обычно предполагается, что существует некое описание вычислительных или иных алгоритмических действий, отдельное от тех данных, над которыми эти действия выполняются. Надо сказать, что такое разделение существует не во всех языках. Иногда явного различия между данными и алгоритмом нет, и программист сам определяет, что считать данными, а что программой. Очевидно, что в таких языках возможна обработка программы как данных, и даже модификация выполняемого кода в процессе выполнения. Разумеется, почти в любом языке программирования можно открыть файл, в котором находится текст программы, и что-то с ним делать, но здесь речь идет именно об использовании данных самой программы в качестве исполняемого кода и наоборот.

Если бы язык Pascal давал нам такую возможность, мы бы могли писать что-то вроде:

```
var p,s: string; a:integer;
begin
  readln(s);
  p:='a:=2*' +s;
  execute(p);
  writeln(a);
end.
```

При вводе строки ‘2’ программа печатала бы нам число 4. Понятно, что такая возможность дает нам больше возможностей для ошибок, чем реальной пользы. Поэтому смешивать программы и данные обычно позволяется только в языках с очень простым синтаксисом, где вероятность ошибки существенно ниже. Однако, есть исключения, и одно из них - Javascript, который сегодня является одним из самых распространенных языков.

Правильность программ

Тестировщик заходит в бар и заказывает:
кружку пива, 2 кружки пива, 0 кружек пива, 999999999 кружек пива, –1 кружку пива, qwertyuip кружек пива.
Первый реальный клиент заходит в бар и спрашивает, где туалет. Бар взрывается.
Анекдот

Возможно ли определить, что программа содержит ошибки? Можно ли доказать правильность программы? И какова роль языков в этом процессе?

И самый интересный вопрос: раз уж возможно рассматривать текст программы как данные для какой-то другой программы, возможно ли написать такую программу, которая будет проверять правильность других программ? И сможет ли она заодно проверить правильность себя самой?

Тут мы прежде всего сталкиваемся со сложностями в постановке вопроса. Допустим, наша программа должна посчитать квадратный корень из некоторого числа, введенного с клавиатуры. Для отрицательных чисел правильного ответа не существует, и суть ошибки нам понятна. Если вместо числа ввести, например, фразу «Здесь был Вася», то никакой разумный ответ тоже получить не удастся. Слишком большие числа не удастся представить в машинном выражении, многие языки ограничивают «самое большое число». Назовем положительные числа, представимые в данном языке, «корректными исходными данными» для нашего примера. Таким образом, возможно несколько различных определений того, что такое правильно работающая программа.

1. Это программа, которая не зацикливается и выдает требуемый результат для определенного набора корректных входных данных. Например, чисел 4, 5 и 234.

2. Это программа, которая не зацикливается и выдает требуемый результат для любых корректных входных данных.

3. Это программа, которая не зацикливается и выдает требуемый результат для любых корректных входных данных и выдает сообщение об ошибке при некоторых некорректных входных данных.

4. Это программа, которая не зацикливается при вообще любых входных данных, выдает требуемый результат для любых корректных входных данных, и выдает сообщение об ошибке в случае любых некорректных входных данных.

Очевидно, что в случае 1 можно провести полное тестирование программы при всех оговоренных входных данных, а в случае 3 - при всех оговоренных некорректных данных и надеяться на то, что утверждения 2 и 4 соответственно тоже окажутся верными, то есть программа, правильно вычисляющая квадратный корень из 4, 5 и 234 окажется способной вычислить его и для 567, 789 и вообще для всех положительных чисел. Но тестирование программы при всех вообще возможных входных данных для достаточно серьезной программы обычно невозможно.

Что же делать? Можно попробовать доказать правильность программы примерно так же, как в математике доказывают теоремы. При том, что доказательство правильности отдельных небольших участков программы, как правило, тривиально, доказательство правильности больших алгоритмов может быть очень сложным. Хотелось бы заставить сам компьютер проделывать такие доказательства (вспомним о возможности рассматривать программу как данные). Но для этого нужно уметь

формально доказывать правильность любых алгоритмов. К сожалению, в общем лучае это невозможно. И это можно доказать.

Для того, чтобы доказать правильность алгоритма нужно, в соответствии с приведенными выше определениями правильности, доказать, что алгоритм вообще когда-либо завершится. Зацикливание программы вряд ли соответствует нашему представлению о правильности. Таким образом, нам в любом случае понадобится функция, назовем ее `isfinite()`, такая, что если ей в качестве аргумента дать имя файла с корректной программой на данном языке, то результат будет `true` если эта программа успешно завершается через какое-то, пусть даже очень большое, время, и `false` если данная программа зацикливается. Заметим, что почти на всех языках программирования можно так или иначе зациклить программу. Представим себе, что у нас есть эта функция, и напишем следующую несложную программу:

```
begin
  if isfinite('test.pas') then
      while 1=1 do;  (* зацикливаем программу *)
  else
      writeln('OK'); (* нормальное завершение *)
end.
```

Что будет, если в файле **test.pas** у нас окажется *сама эта программа*? Если функция **isfinite()** решит, что эта программа не зацикливается, то эта программа зациклится, и наоборот. Из полученного противоречия можно сделать вывод, что функция **isfinite()** невозможна. А значит, и невозможно доказать в общем случае правильность какой угодно программы. Здесь важно заметить, что во многих частных случаях такое доказательство, тем не менее, возможно. Таким образом, написать программу, проверяющую правильность других программ можно, но правильность *самой* этой доказывающей программы будет соответствовать лишь первому из наших определений правильности.

Этот вывод о невозможности выяснить, зацикливается ли произвольная программа, носит название «*проблема остановки*» и в теории алгоритмов доказывается несколько более формально. Но на этом проблемы не заканчиваются. Теорема Райса утверждает то же самое относительно *любого* нетривиального свойства алгоритмов. Нетривиальным тут называется такое свойство, которым некоторые алгоритмы обладают, а некоторые нет. Разумеется, это касается неких идеализированных алгоритмов, в нашем подходе к программам как к строкам текста никто не мешает выяснить, например, содержит ли данная программа оператор `if` или переменную с именем **X**. Но такие свойства не считаются свойствами алгоритмов.

Откуда есть пошли языки программирования

> *Низкоуровневый язык — это когда требуется внимание к вещам, которые никак не связаны с программами на этом языке.*
>
> Алан Перлис

Немного теории - команды, данные и адреса

Компьютер представляет собой устройство, способное выполнять команды, записанные в память. Команды выполняются над каким-то данными, которые тоже лежат в памяти. Память компьютера

можно представить себе как набор полок, на каждой из которых может лежать число, причем число в определенном диапазоне. У каждой такой полки есть номер, то есть еще одно число, по которому можно получить хранимое на полке число или положить туда другое, заменив им имеющееся. Обычно на одной такой полке можно хранить число в диапазоне от 0 до 255 (один байт). Число 326 оказывается слишком большим и не влезает на одну полку. Однако, его можно представить в виде 256+70 и уместить в два байта со значениями 1 и 70.

Команды процессора это элементарные операции, доступные программисту. Например, сложение, вычитание, помещение числа в память или извлечение его из памяти. Команды никак не отличаются от данных, это такие же числа, и прочитав содержимое какой-либо памяти, мы никаким способом не сможем отличить команды от данных. Например, последовательность байт 102, 131, 192, 2 может означать команду для процессора x86 «прибавить 2 к содержимому регистра ax", а может означать число 1719910402, а может означать и 4 числа 102, 131, 192, 2. Интерпретация зависит от того, как программа (а не процессор) использует эти данные. В некоторых процессорах память, в которой лежат команды отделена от памяти, в которой хранятся данные, но это не меняет ситуацию - программа и данные это просто числа, хранящиеся в памяти в соответствии с описанными правилами. Проводились эксперименты по созданию процессоров, различающих типы данных, но на сегодняшний день такие процессоры не получили распространения.

Программирование в символических адресах

Откуда вообще появились языки программирования? На самых первых компьютерах не было никаких языков. Не было также и клавиатур, дисплеев, и других удобств. Программы писали прямо в машинном коде и вводили в компьютер в двоичном коде с помощью переключателей. Одно положение переключателя означало 1, второе 0. Писать программы таким способом было крайне неудобно. Проблему переключателей решили довольно просто, приспособив к компьютеру уже имевшиеся устройства чтения перфокарт, а также электрическую пишущую машинку. Но осталась еще проблема адресов.

Предположим, у нас есть гипотетический компьютер, «понимающий» следующую систему команд: каждая команда состоит из трех чисел. Первое из них - код команды, два другие - адреса в памяти. Команды у нас будут такие: 25 - сложение, 35 - вычитание.

Тогда команда 25 10 100 означает «прибавить к содержимому ячейки памяти с адресом (номером) 10 содержимое ячейки памяти с адресом 100». Примерно так устроены команды у всех реально существующих процессоров, но коды команд и способы кодирования, естественно, различаются. При написании программы в машинных кодах программист должен был где-то для себя составить табличку, в которой записать, что по адресу 10 у него хранится, например, высота орбиты спутника, а по адресу 100 - вычисленная коррекция орбиты. Нечто подобное до сих пор делают, например, в программируемых калькуляторах или электронных таблицах, где у каждой ячейки есть номер строки и столбца, а смысл лежащего там числа известен только тому, кто работает с этой таблицей. Другая таблица, которой все время приходилось пользоваться, была таблица кодов команд, в которой было написано, что делает та или иная команда.

При необходимости изменить программу приходилось вручную выискивать все команды, ссылающиеся на измененные адреса и корректировать их. Работа кодировщика состояла в поиске в таблицах кодов команд и адресов, и в ручном вводе их в компьютер. Эту занудную работу, в ходе которой происходило множество ошибок, переложили на компьютер. Программисты стали обозначать

команды легко запоминающимися сокращениями, а адреса так и вообще произвольными идентификаторами, соответствие которых реальным адресам устанавливала уже специальная программа - ассемблер. Приведенная выше команда на языке ассемблера выглядела так:

```
    ADD Height,Correction
```

Несложная программа могла заменить ADD на 25, а для переменных Height и Correction выделить реальные алреса памяти, то есть составить таблицу, в которой символическим именам были бы сопоставлены числа. Таким образом был создан первый протоязык программирования - ассемблер.

Последующее развитие систем программирования и дальше шло по пути сокращения количества рутинной работы и перекладывания на компьютер тех задач, которые вызывали наибольшее число ошибок. Традиционно принято различать компиляторы и интерпретаторы языков. Определения их такие:

Компилятор преобразует текст программы в эквивалентный машинный код, который потом исполняется уже без участия программы-компилятора.

Интерпретатор, наоборот, считывает исходный текст программы и исполняет его безо всякого преобразования путем программного моделирования.

Однако, в современном мире программ все намного сложнее, и простое разделения на компиляторы и интерпретаторы не отражает все многообразие возможностей. По существу, в мире осталось очень мало программ, которые в полном смысле можно было бы назвать интерпретаторами. Обычно используются некоторые смешанные технологии, которые с точки зрения приведенного тут классического определения можно назвать и так и этак. К счастью, для анализа языков программирования это не очень важно. Вместо классификации программ, реализующих языки, мы займемся классификацией самих языков. Среди конструкций языка в этом смысле можно выделить три основных класса:

Машинно-зависимые элементы

При включении в язык таких элементов имеется в виду их реализация на какой-то конкретной машине. Под словом "машина" здесь и далее понимается определенная архитектура компьютера, набор команд и связанных со всем этим особенностей. Например, можно говорить о машинах IBM360, машинах I32, AMD64 итд. Название это используется по аналогии с термином "машина Тьюринга".

Машинно-зависимые элементы это элементы, непосредственно подсказанные особенностями той машины, с которой предполагается работать. Например, 32-битные целые числа и 8-битные байты стали фактическим стандартом, хотя не во всех компьютерах байт (единица адресации памяти) состоит из 8 бит. Были машины с 6-битными и 9-битными байтами. Размер в 8 бит был принят в качестве стандарта только в 1993 году. Для современных языков это не имеет особого значения, но, тем не менее, целые часто продолжают делать 32-битными. Или 64-битными.

Другим примером может служить Fortran, в котором есть некоторые крайне неудобные для програмиста, но «удобные» для машины конструкции. Например, машина IBM 704, для которой впервые был реализован Fortran, могла считывать только 72 символа с перфокарты, емкость которой

составляла 80 символов. Байт там был шестибитный, и в 36-битное машинное слово можно было упаковать 6 символов. Отсюда пошли 6-символьные идентификаторы в языке Fortran. Так что машинные особенности могут оказывать влияние не только на выполнение программ, но даже на внешний их вид.

Когда компьютеры были большими, а память и быстродействие их - маленькими, было естественно осложнить жизнь программисту и упростить - машине. Интересно, что компиляторы первых языков программирования были намного сложнее современных, а введение машинно-зависимых элементов в языки не дало в целом сколь-нибудь заметного положительного эффекта. В современных языках конструкций, подсказанных устройством железа, становится все меньше. Однако пережитки «железячной» эпохи еще остаются в некоторых языках программирования. В языке C неявно предполагается, что символ занимает 1 байт, и может быть преобразован к типу «целое число», у которого диапазон допустимых значений более широкий. Это плохое решение как с точки зрения современных требований к представлению естественных языков (256 символов достаточно для русского языка, но уже маловато для китайского), так и с точки зрения современных компьютеров, в которых уже идет переход от 32-битного к 64-битному представлению целых чисел. Однако в C символы остаются однобайтными, а при преобразовании к целому числу 7 байт оказываются просто лишними. Языки с большим количеством машинно-зависимых элементов часто называют «языками низкого уровня».

Отображаемые на «железо»

Некоторые языковые конструкции рассчитаны на определенные способы их реализации. Это не значит, что их нельзя реализовать как-то иначе, но авторы языков часто имеют в виду конкретные варианты перевода на машинный язык. Вызовы подпрограмм в современных языках программирования C и Pascal продуманы так, чтобы упростить их реализацию на современные машинах, в которых есть программно-доступный стек возвратов. В тех редких случаях, когда такого стека нет, например, для микроконтроллера AT90S1200, обычный компилятор C не реализован. С другой стороны, способ вызова подпрограмм, принятый в языке Pascal, был реализован в процессорах Intel. Таким образом, отображаемые на машину элементы языка иногда способствуют развитию аппаратного обеспечения. Разработчики процессоров вводят в свои процессоры новые команды, еще более упрощающие реализацию таких элементов.

Машинно-независимые

Это те элементы, которые не предусматривают какой-то очевидной и простой реализации в компьютере. Такие элементы требуют определенного объема программного моделирования или создания виртуальной машины, которая уже позволяет реализовать эти особенности. Примером могут служить строки в Паскале. Конструкция

```
var a,b,c: string;
b := 'Hello ';
c := 'World!';
a := b+c;
```

простого выражения в кодах процессора не имеет, и приблизительно переводится в следующий алгоритм:

- выяснить длину b и c
- сложить полученные числа
- выделить область памяти с длиной, равной полученному результату сложения
- скопировать туда сначала содержимое строки b, потом содержимое строки c
- сделать так, чтобы полученная строка была доступна под именем a.

Надо заметить, что процесс выделения памяти тоже никак не укладывается в одну-две процессорных команды.

Неисполняемые элементы

Это те части программы, которые непосредственно не вызывают порождение какого-то машинного кода, но нужны для решения задач, которые сам же язык и ставит перед программистом. Например, в языке C++ не допускается непосредственное присваивание переменной, объявленной как целое число (int) значения адреса памяти. Однако, написав так называемое "приведение типа", это можно сделать.

```
char *v;
int i = (int) v;
```

С точки зрения процессора заклинание (int) тут вообще ничего не делает. И int и адрес внутри многих (не всех!) компьютеров представляются совершенно одинаково. Но для програмиста на C такая запись это знак «Внимание! Тут происходит нечто не совсем обычное". В некторых языках имеются также чисто декоративные элементы, ни на что не влияющие вообще. Так, например,в языке Pascal есть конструкция program, которая ставится в начало программы и ничего там не делает. В большинстве реализаций Паскаля ее можно просто не писать.

Syntax sugar

Это сложнопереводимое высказывание означает возможость записать какое-то часто встречающееся сочетание операторов иным способом, проще и нагляднее. Например, в C++ можно вместо

```
int i;
for (i=1;…
```

написать

```
for (int i=1; …
```

Syntax sugar часто встречается в современных языках, поскольку облегчает жизнь и красиво смотрится, но не только. Реализация объектно-ориентированного программирования (ООП) в языке Lua представляет собой разновидность syntax sugar.

Такое разделение по отношению элементов языков к исполняющему компьютеру весьма условно, не существует четких границ между этими классами. Тем не менее, это деление является важным для понимания языков, поскольку обычно люди путают особенности языка с особенностями конкретной его реализации в виде компилятора или интерпретатора. Правильнее говорить не о «компилируемых» или «интерпретируемых» языках, а о реализациях в виде компилятора или интерпретатора. Язык, содержащий большое количество элементов, требующих программного моделирования, скорее всего, окажется интерпретатором, и наоборот. Существуют и исключения, например, в силу совершенно непонятных причин язык BASIC, в своем исходном варианте вообще не содержащий конструкций, требующих программного моделирования, чаще всего бывает интерпретатором и именно в таком виде был создан.

Совсем просто это деление на классы можно изобразить в виде лозунгов, под которыми могли бы быть созданы те или иные возможности языка.

1. «Зачем что-то изобретать? У меня тут есть машина. То, что она умеет, я и включу в язык»
2. «Я придумал ловкий способ реализовать эту возможность. Люди с компьютерами других типов получат массу проблем, но это будут не мои проблемы»
3. «То, что я тут придумал, выглядит красиво и удобно, и какая разница, во что это обойдется. Компьютеры нынче дешевы»
4. «Извините, так надо в силу логики языка»
5. «Программисты будут мне благодарны за это маленькое упрощение их труда. Кроме того, это красиво»

Разнообразие языков программирования

Продолжим анализ различных характеристик языков программирования. Языки бывают:

Универсальные и специализированные

К универсальным языкам в самом общем смысле относятся такие, на которых теоретически можно написать что угодно. Разумеется, не учитывая особенности реализации, из-за которых, например, операционная система, написанная на PHP, будет работать чересчур медленно (и требовать еще одну операционную систему для работы). Примером универсального языка может служить C (или C++). К специализированным языки, созданные для какой-то конкретной цели, либо языки, созданные с определенной целью. Зачастую специализированные языки не позволяют реализовать какой-то специфический класс алгоритмов. В качестве специализированного языка можно указать язык запросов к базам данных SQL.

Интерактивные и неинтерактивные

Некоторые языки специально приспособлены для работы в режиме «калькулятора» - вводим строку, получаем результат. Одним из первых интерактивных языков был BASIC. Из более современных можно отметить Python, язык системы Matlab и другие. К неинтерактивным можно отнести C, C++, Pascal. В литературе режим непосредственного исполнения введенных команд обозначают сокращением **REPL** (Read-Evaluate-Print Loop), цикл чтения-вычисления-печати результата.

Императивные и функциональные

Императивными называются те языки, к которым мы больше всего привыкли. Программа в них состоит из набора команд, «приказывающих» некоторому исполнителю произвести ту или иную последовательность действий. Pascal, C, BASIC - это все императивные языки.

```
read (a);
read (b);
c := a*b;
print(c);
```

Перед вами программа на императивном языке, означающая: прочесть число, назвать его **a**, прочесть второе число, назвать его **b**, перемножить **a** и **b**, результат назвать **c**. Во всех императивных языках эксплуатируется одна важная идея - изменяемое состояние. Когда программа читает переменную **a**, ее состояние меняется. Когда программа присваивает переменной с значение суммы a и b, состояние переменной с меняется. В очень общем виде смысл алгоритма на императивном языке - привести нечто, что может иметь некий набор состояний из некоторого начального состояни в желаемое состояние. Это могут быть переменные в программе, файлы данных или внешние управляемые программой устройства.

В функциональных языках выполнение любой программы равносильно вычислению значения функции. Примерами функциональных языков могут служить языки LISP и Scheme.

```
(write (* (string->number (read-line))
      (string->number (read-line))
))
```

Это программа на языке Scheme, означающая: напечатать произведение двух чисел, каждое из которых получено преобразованием строки, прочитанной из стандартного ввода, в число. Это та же самая программа, хотя надобность в промежуточных переменных a,b и c отпала. Важно отметить наличие так называемого «побочного эффекта» функций, который в данном случае заключается в чтении и печати чисел. Если в императивном языке чтение числа происходит в момент исполнения команды, то в функциональном есть только описание способа вычисления функции.

Главной особенностью функциональных языков является то, что в них обычно стараются избегать главного свойства языков императивных: изменяемого состояния. В приведенном примере переменных нет вообще, но если они есть, их обычно делают неизменяемыми. Если в императивных языках переменные это «полки", на которые можно положить значения, то в функциональных языках переменные это в полном смысле слова имена для функций или значений. Эта парадигма ближе к естественному пониманию вещей, если вашу собаку зовут Рекс, вы обычно не склонны его менять и тем более не рассматриваете это имя как способ хранения разнообразных собак. Запись `X=X+1` во многих функциональных языках (там, где вообще возможна такая запись) не имеет смысла, X не может быть равен X+1. Тем не менее, в функциональные языки часто вводят императивные элементы, просто потому, что идея изменяемого состояния пришла к нам из аппаратных особенностей компьютеров, и представляется программистам вполне естественной.

Посмотрим еще раз на приведенную выше программу. Допустим, в качестве первого числа ввели 0. Наша система может быть настолько «умной», что сообразит, что читать второе число в общем-то и не нужно, результат все равно будет 0. Вопрос - может ли система не читать второе число? Ответ на этот вопрос неоднозначен. Эта неоднозначность присутствует во всех функциональных языках программирования, и везде она как-то решается.

Один из очевидных способов решения - выделить отдельный класс функций, пропускать вычисление которых нельзя. Тогда в случае, если первое число 0, второе число будет все равно прочитано, хотя умножение может и не производиться.

Объектно-ориентированные

Объектно-ориентированные языки могут основываться как на императивных, так и на функциональных, или на тех и других сразу. Они используют некую концепцию программирования, которую мы будем разбирать отдельно в посвященной этому главе. Пока что приведем примеры. Java, SmallTalk, ECMAScript - объектно-ориентированные языки.

Логические

К логическим языкам относятся те, которые используют в качестве основы ту или иную систему формальной логики. Одна из таких систем называется «дизъюнкты Хорна», в ней используются выражения вида

```
    u <= (p AND q AND t AND ... AND z)
```

что надо понимать как «для того, чтобы было истинно u, необходимо, чтобы были истинны p,q, ... z». Внешне эта запись похожа на присваивание в обычных языках. Вот пример программы на Прологе :

```
father('pete', 'paul').
father('john', 'jim').
father('paul', 'andrew').

grandfather(C, GF) :- father(C, F) and father(F, GF).
```

Первые 3 предложения определяют так называемые «факты», то есть утверждения, не зависящие от других. Четвертая строка определяет условное утверждение: GF является дедом C, если существует некто F, являющийся одновременно сыном GF и отцом C. С точки зрения Пролога это утверждение выглядит так: чтобы доказать, что grandfather(C, GF) истинно при некоторых значениях переменных (и одновременно найти, при каких именно C и GF), нужно сначала найти утверждение father(C, F), присвоить переменным C и F конкретные значения, в данном случае 'pete' и 'paul', а потом попробовать найти утверждение father(F, GF), используя уже найденное значение переменной F. Если не получится, то пробовать другие варианты. В этом примере условиям предиката grandfather удовлетворяют первый и третий факты. В результате выполнения программы C получается равным 'pete', а GF равным 'andrew'.

Несмотря на то, что такие языки предназначены для решения логических задач, они почти все относятся к универсальным языкам, и годятся для написания каких угодно программ. Смысл существования этих языков отчасти заключается еще и в том, что логические системы, на которых эти языки построены, являются хорошо известными и тщательно разработанными разделами математики. Кроме того, логические языки программирования требуют от программиста особенного стиля мышления, в ряде случаев весьма полезного. Первым логическим языком был Prolog, и большинство существующих сейчас логических языков (Alice, Goedel) основаны на нем.

Отдельную группу языков составляют основанные на понятии *rewriting*. Парадоксально, но для этого слова в Википедии не нашлось русского эквивалента, хотя один из языков этой группы, Рефал, был разработан в нашей стране. Смысл понятия rewriting очень простой. Это всего лишь обобщение идеи поиска и замены, используемых в любом текстовом редакторе.

Рассмотрим следующую простую задачу. Вы хотите задать функцию, которая определяет для любой заданной строки символов, является ли она палиндромом. Попробуем определить это формально:

1. Пустая строка является палиндромом.
2. Строка из одного символа является палиндромом.
3. Если строка начинается и оканчивается одним и тем же символом, то она является палиндромом тогда и только тогда, когда строка, полученная из нее путем удаления начального и конечного символов, является палиндромом.
4. Если не выполнено ни одно из вышеприведенных условий, строка палиндромом не является.

Обычно функциональные, логические и rewriting языки объединяют в одну группу под названием декларативных, намекая на то, что все они не задают последовательности вычислений, а декларируют некоторые факты или утверждения, ведущие к достижению цели.

Прочие

Реляционные языки программирования основаны на специально придуманной системе, которая называется реляционной алгеброй, или алгеброй отношений. Смысл, опять же, в том, чтобы «подложить» под язык стройную математическую теорию. К реляционным языкам относятся SQL и Prolog.

Языки запросов относятся к специализированным языкам и предназначены для извлечения информации из баз данных.

Эзотерические языки программирования

Некоторые языки программирования создавались с совсем иной целью, нежели все остальные. По крайней мере, никто из их создателей не предполагал, что на них всерьез будут писать программы. Создание эзотерических языков программирования - это игра ума, сродни сочинению стихов, каждое слово которых начинается с буквы Ч, или изобретению машин, разбивающих яйцо наиболее сложным способом. Иногда эзотерические языки программирования создают как пародию на существующие. Таков, например, язык **INTERCAL**, в котором вместо команды **GOTO** используется конструкция **COMEFROM**, которая пишется в том месте, *куда* происходит переход.

Многие эзотерические языки программирования были придуманы с целью создать язык, писать программы на котором будет максимально сложно. Причем написать какую угодно программу должно быть просто, а вот написать программу, делающую нечто конкретное - практически невозможно. На сегодняшний день эту цель можно считать достигнутой. На языке **Malbolge** написано всего несколько программ, и ни одну из них не написал человек. Все они созданы автоматически, методом перебора. Такие языки нужны во-первых, затем, чтобы эта область на становилась слишком уж скучно-академической, а во-вторых как полезное упражнение в той или иной области теории языков.

Здесь можно остановиться и задать вопрос: а зачем вообще нужны разные языки? Если они все одинаковы, то почему бы всем не выучить какой-то один? Если они все разные, то почему не выбрать лучший?

Ответом могут служить следующие цитаты:

«*программист, знающий один язык программирования, не знает ни одного*»

«*если становиться программистом, то программистом хорошим. Такого программиста отличает только постоянное желание стать еще лучшим программистом, а единственный путь для этого — стремиться овладеть несколькими языками, стать хорошим лингвистом в программировании. Несомненный вред наносят те довольно хорошие программисты, которые полагают, что язык, которым они пользуются, во всех смыслах является наилучшим*»

«*до тех пор, пока мы не избавимся от последнего пользователя только-C или только-Паскаля, для остальных нет шансов достигнуть профессионального уровня.*»

Язык программирования представляет ценность сам по себе, как произведение человеческого разума и изобретательности. Спрашивать, какой язык нужно изучить, чтобы уметь программировать – все равно, что спрашивать, какую книгу прочесть, чтобы знать литературу. Языки программирования во многом напоминают естественные, и полиглот всегда имеет преимущество в понимании языка. Попыток прекратить размножение языков программирования и создать «идеальный» язык предпринималось много, но все они оказались неудачными.

Языки, реализации и библиотеки

Очень часто эти вещи вызывают путаницу. Говоря, что такой-то язык «медленный", а другой «быстрый» или что один язык «позволяет делать то-то», а другой «не позволяет», люди смешивают в одну кучу собственно языки программирования, их конкретные реализации и библиотеки.

В большинстве языков программирования проводится различие между собственно элементами языка, такими как операторы, ключевые слова и т.п., и функциями, которые могут быть реализованы так или иначе, и вызываются неким единообразным способом. В некоторых языках, например, в Forth, такого различия нет. Кроме того, библиотеки функций делятся на те, которые описаны в стандарте языка или другом подобном документе, и те, которые написаны на этом языке другими людьми, не имеющими отношения к созданию и развитию данного языка. Популярность языка Python, а в не очень далеком прошлом и языка Delphi, не в последнюю очередь обусловлена наличием огромного числа сторонних библиотек, среди которых есть очень качественные, например, numpy или scipy.

Вопрос различий между реализациями сейчас стоит не так остро, как раньше, потому что у подавляющего большинства языков программирования есть реализации с открытым кодом, и их можно считать основными. Тем не менее, отличия производительности между разными реализациями

языка могут быть очень существенными, хотя это не всегда можно считать характеристиками самого языка. Иначе говоря, язык не виноват, что для него написали компилятор, выдающий медленный и неэффективный код, но для некоторых языков код всегда будет более медленным, чем для других.

Ответа на вопрос «какой язык программирования самый быстрый» не существует. В Интернете уже много лет проводится игра под названием «The Computer Language Benchmarks Game». В ней участники пытаются создать наиболее эффективные реализации различных алгоритмов на разных языках программирования. Под эффективностью понимаются как скорость выполнения программы, так и необходимый для ее работы объем памяти. Результаты многолетнего труда соревнующихся именно такие: «самый эффективный язык» невозможно определить однозначно.

Например, пусть компилятор языка A выдает код, который для некоторого алгоритма выполняется вдвое быстрее, чем код для такого же алгоритма, записанного на языке B. Но в то же время язык B позволяет легко распараллелить аэтот алгоритм и выполнить его на 4 процессорных ядрах одновременно, а язык A не позволяет сделать это просто и эффективно. Но это не значит, что на языке A нельзя написать более эффективную многоядерную реализацию алгоритма, просто это сделать сложнее. И тем более это не значит, что у данного алгоритма нет еще более эффективной реализации, про которую мы не знаем.

Немного о договоренностях и терминологии

В области компьютерной лингвистики на русском языке до сих пор нет окончательно устоявшейся терминологии, и многие важные термины переводят с английского, используя калькирование, что дополнительно все запутывает. Например, термин **Duck typing**, о котором мы будем говорить в разделе про ООП, вызывает в памяти рядового американца ассоциации с сенатором Маккарти, "охотой на ведьм" и вообще понятно, о чем идет речь. Русскому программисту надо специально растолковать, что вот был такой печально знаменитый Маккарти, и тогда станет ясно, почему оно так называется. А если термин еще и перевести, получится "Утиная типизация" - довольно уродливый термин и совершенно непонятный. Ну правда, не начинать же каждую лекцию про ООП с рассказа об истории послевоенных США.

В этом тексте используются английские термины там, где не находится подходящих русских эквивалентов.

Задания к части 1

1

Напишите программу, которая печатает свой собственный код. При этом программа не должна производить никакой ввод, не должна читать файлы, и не должна использовать какие-то особенности реализации или операционной системы. Такая программа называется куайн, в честь философа и лингвиста У. Куайна. Небольшая подсказка: мы можем написать программу, которая печатает `"Hello, world"`. Далее мы можем написать программу, которая печатает `print("Hello, world")`. Дальше надо всего лишь понять, как остановиться. Как это сделать, покажет еще одна подсказка. Решите логическую задачу:

Некий ученый, назовем его N, написал статью, которая потом была переведена и опубликована в японском журнале. N не знает японского. В конце статьи на японском были добвлены следующие примечания:

1. Я благодарен моему коллеге профессору Тамура за перевод данной статьи.
2. Я благодарен профессору Тамура за перевод вышеприведенного примечания.
3. Я благодарен профессору Тамура за перевод вышеприведенного примечания.

Почему благодарностей именно столько, сколько их есть, и как N удалось избежать бесконечного повторения примечаний?

2

Изучите какой-нибудь ассемблер и какой-нибудь эзотерический язык программирования.

3

[[TOC]]

Данные, имена и значения

Что для одного человека константа, для другого переменная
Алан Перлис

Следующая часть материала посвящена подробному исследованию элементов языков.

Прежде всего рассмотрим имена. Они есть почти в любом языке программирования, и можно сказать, что языки программирования были изобретены ради имен. Ассемблер иногда называют системой программирования в символических адресах, под термином "символические адреса" в этом определении понимаются имена, которые программист дает адресам и числам.

В этом, кстати, проявляется одно из фундаментальных отличий языков программирования от языков общения, как естественных, так и плановых. В последних имена рассматриваются как данность, относящаяся к объектам реального мира и внешняя по отношению к языку. Поэтому имена в литературных произведениях иногда бывает сложно переводить с языка на язык, тогда как их перевод в технических текстах обычно сводится к транслитерации. В программах на большинстве языков программирования невозможно обойтись без имен, придуманных самим программистом.

Попробуем перевести известный текст с естественного языка на компьютерный.

```
3 девицы D1,D2 и D3 под окном
Пряли поздно вечерком
...
```

В языках программирования высокого уровня роль имен значительно расширилась. Но чаще всего имя является названием для места памяти, где (возможно) что-то лежит. Рассмотрим фрагмент программы

```
X = X + 2;
```

Заметим, что два X здесь обозначают разные вещи. Первый X это место, куда должно быть записано значение. Второй X обозначает само значение. Фактически такая запись это syntax sugar для

```
X = value(X)+2;
```

где функция value(X) выдает значение переменной X. В таком выражении X справа и слева от знака присваивания означает одно и то же - место, где может содержаться значение. В некоторых языках, например в Forth, именно так и пишут. Но подавляющее большинство языков придерживаются договоренности о том, что переменная, появляясь по разные стороны от знака присваивания, изменяет свой смысл. Так удобнее.

Переменные

Различают так называемые “левый” и “правый” контексты для переменных. Еще раз внимательно разберемся с терминологией. Пусть переменная с именем X равна числу пи. Тогда у нас есть:

- Число, равное 3.141592
- Это число находится в памяти машины по некоторому адресу.
- Адрес это тоже число.
- Адрес называется X. Это название дал ему программист, но он не связывал конкретное число, являющееся адресом, с этим именем, такую связь за него сделал компилятор.

Обобщением понятия адреса является *ссылка*. Это такая сущность в языке, которая каким-то образом указывает, где находится какое-то значение. Для многих языков программирования, например, C или Pascal, ссылка это и есть адрес в памяти, однако, она может быть и чем-то большим. Ссылка может содержать информацию о типе, о состоянии объекта, итд. Проводя аналогии с объектами реального мира, если адрес это номер полки, на которой лежит значение, то ссылка это любое описание, по которому можно данную полку найти (например "справа от входа наверху"). В том числе и номер полки, конечно.

Простая переменная имеет следующий вид:

```
имя —> ссылка —> значение
```

Здесь стрелки означают, что зная имя, можно получить ссылку (адрес), а зная адрес, можно получить значение. Конкретный способ получения для нас сейчас не очень важен. Заметим только, что память, в которую можно записать значение, или адрес этой памяти, может ассоциироваться со ссылкой как в процессе компиляции программы, так и во время выполнения. Реальное значение адреса обычно скрыто от программиста, но во многих языках его можно получить, например, с помощью операции & в языке C, или @ в Pascal. В других языках связь между значением и ссылкой не такая простая, и получить адрес в непосредственно машинной форме невозможно. Но он всегда где-то есть, ведь значения переменных больше негде хранить, кроме как в памяти компьютера.

Константы

Константа имеет вид

```
имя —> значение
```

Внутренняя “механика” процесса получения значения по имени, опять же, может быть достаточно сложной, но смысл всегда остается тот же самый — по имени можно получить значение. Частным

случаем констант являются числа, тут можно считать, что имя отсутствует, а можно считать, что именем является сама запись числа. Это не существенно, важно лишь то, что константа не предполагает способа для изменения значения. Соответственно, слева от знака присваивания константа сама по себе появляться не может.

Принципиальная особенность констант состоит в том, что их значения нельзя изменять во время выполнения программы. К сожалению, некоторые языки в некоторых реализациях позволяют с помощью изощренных приемов (а то и ошибок) это сделать, но смысл константы состоит именно в том, что программист заявляет «эту величину я менять не буду». Языки программирования позволяют делать многие вещи, не предусмотренные их создателями. Это не значит, что так делать нужно, скорее наоборот, таких решений нужно старательно избегать.

Существует интересная разновидность констант, у которых есть имена, а вот фактические значения, им соответствующие, нас не интересуют. Так, например, кодируя пол человека, можно мужскому поставить в соответствие 1, а женскому 0, а можно мужскому 95, а женскому 34, и ровным счетом ничего от этого не изменится. Подобные константы в Паскале задаются посредством перечислимых типов, а в C с помощью перечислений (**enum**). В качестве примера рассмотрим описание колоды карт (например, для игры пасьянс в Windows)

Pascal:

```
type cardsuit = (clubs,diamonds,hearts,spades);
```

C:

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
```

В функциональном языке Erlang такие константы используются очень широко, и любое слово, не являющееся ключевым словом языка и начинающееся с прописной буквы, является с точки зрения языка константой с неизвестным программисту числовым значением. Некоторые такие константы по соглашению имеют предопределенный смысл, например, константа `true`.

Числовые типы и подтипы

Каждое значение в программе в момент использования имеет какой-то явно или неявно приписываемый ему тип. Не зная типа, невозможно правильно истолковать содержимое памяти компьютера, ведь все байты памяти одинаковы.

В большинстве случаев различают следующие числовые типы:

Натуральный, Целый, Вещественный, Комплексный. Не в любом языке программирования существуют все эти типы, но их обычно стараются вводить в язык, не в первой версии, так в десятой. Например, в языке C изначально не было комплексных чисел, но в стандарте C99 их ввели. Стандартные операции (`+ - * /`) для одних и тех же чисел разных типов выполняются по-разному. Для натурального и целого типов отличаются результаты операции вычитания. Значение выражения 10 - 20 для целых чисел будет равно -10, а для натуральных это скорее всего ошибка, хотя можно

получить достаточно неожиданный результат вроде 4294967286 или 65526. На разных машинах он может быть разным, но все равно неправильным. Разница между целыми и вещественными числами состоит в результате операции деления. Обычно деление целых чисел рассматривают как деление нацело, то есть 4/3 = 1.

Разные языки по-разному относятся к смешению разных числовых типов в одном выражении. Если мы складываем вещественное число и целое, то в действительности нужно сначала преобразовать целое в вещественное, а потом сложить эти два числа. В более сложных выражениях можно сначала преобразовать целые к вещественному типу, а потом выполнить все операции. А можно сначала поделить целые (нацело), а потом результат преобразовать к вещественному типу. В любом случае легко получить совсем не то, что хотел программист.

Сложно сказать, какой способ хуже. Во многих языках есть правила приведения типов, то есть автоматического преобразования типов в выражении. В языке MODULA-2, например, вообще запрещено смешивать целые числа с вещественными без явного их преобразования. Другое решение было принято в некоторых вариантах языка BASIC - там вообще нет целых чисел. В заключение темы попробуйте определить, что делает вот такая программа:

```
int i,j;
int a[10,10];
for ( i=0; i<10;i++){
    for ( j=0; j<10;j++){
      a[i,j] = (i/j) * (j/i);
    }
}
```

Числовые подтипы

В компьютере обычно имеется несколько вариантов представления чисел того или иного типа. Например, натуральное число может занимать 1 байт, 2 байта, 4 байта итд. Если 1 байт это 8 бит, то с помощью одного байта можно представить числа от 0 до 255, всего 256 значений. Логично было бы как-то выразить такую машинно-зависимую возможность в языке. Хотя бы с целью экономии памяти. С другой стороны, существует много видов данных, представляемых в виде ограниченных числовых диапазонов. Месяцев в году всего 12, дней в месяце не более 31, а химических элементов непонятно сколько, но вряд ли более 200. Для таких типов можно ввести в язык отображаемую на реальные типы данных возможность - указать диапазон изменения типа, а компилятор пусть подберет наиболее подходящее внутреннее представление. В некоторых языках идут еще дальше, проверяя, какие именно значения программист пытается присвоить переменным, и проверяя, какие действия можно, а какие нельзя совершать со значениями тех или иных типов.

Подтипы в языке Ada. Язык **Ada** имеет одну из наиболее развитых систем типов и подтипов, поэтому мы рассмотрим ее более внимательно. В этом языке есть две возможности. Во-первых, можно явно объявить один тип как подтип другого. При этом переменным базового типа можно присваивать значения производного типа.

```
type WeekDays is range 1 .. 7;  -- Дни недели это числа от 1 до 7
subtype WorkDays is WeekDays range 1 .. 5; -- Рабочие дни это числа от 1 до
```

```
5
A : WeekDays := 8; -- ОШИБКА!
B : WorkDays := 4;
C : WeekDays := B; -- OK
```

Другой пример:

```
-- Натуральные числа
subtype Natural  is Integer range 0 .. Integer'Last;

-- Положительные числа
subtype Positive is Integer range 1 .. Integer'Last;
```

Во-вторых, можно объявлять производные типы, сохраняющие все свойства базового,но несовместимые с ним.

```
type Apples is new Integer ;
type Oranges is new Integer ;
A : Apples := 8;
B : Oranges := A+B; -- ОШИБКА!
-- Яблоки нельзя складывать с апельсинами.
```

Атрибуты данных

Откуда компилятор знает, что можно делать с переменными, а что нельзя? Посмотрим еще раз на диаграмму чуть выше, изображающую переменную. Как мы уже говорили, ссылка это не просто адрес в памяти, но еще и информация о типе данных. Если программа компилируется, то эта информация зачастую просто “выбрасывается” после генерации кода. Все проверки соответствия типов производятся во время компиляции. Другой подход состоит в том, чтобы хранить где-то информацию о типе данных и проверять соответствие типов во время выполнения. Часто комбинируют оба подхода, или же один и тот же компилятор может сохранять какую-то информацию о типе, а может и не сохранять в зависимости от настроек. В компиляторах языка C++ эта информация называется RTTI (Run-Time Type Information).

В соответствии с тем, с какой частью переменной ассоциируется информация о типе, языки можно разделить на два класса.

Языки L-типа и R-типа. В языках L-типа информация о типе является частью ссылки, а в языках R-типа - частью значения. При этом неважно, когда именно используется информация о типе - во время компиляции или во время выполнения программы. Если в языке L-типа переменной каким-нибудь способом подсунуть данные не того типа, среда исполнения не будет «знать», что данные неправильные. Наиболее распространенным примером является выход за границы массива. В языках R-типа такой трюк невозможен. К языкам L-типа относятся, например, Pascal и C. Пример языка R-типа - PHP. В нем выйти за границы массива не удастся - они хранятся вместе со всем массивом.

Иногда используют другую классификацию, разделяя языки на **динамические** и **статические**. В динамических языках, или, точнее, в языках с динамическим типированием переменных, любой переменной можно присвоить значение любого типа, например, сначала строку, потом число. В языках со статическим типированием тип переменной известен компилятору и не может меняться.

Проблема состоит в том, что, хотя в языке L-типа всегда можно сделать проверки на безопасность, многие программисты полагают, что основное достоинство таких языков - это именно возможность делать что угодно, обходя проверки. Действительно, некоторые алгоритмы системного программирования работают быстрее и реализуются удобнее при наличии возможности обойти проверку типов. Поэтому возможности так или иначе «обмануть компилятор» рано или поздно добавляют в большинство языков L-типа. Например, в языке Pascal записать значение за границей статического массива невозможно, но уже в варианте Turbo Pascal такая возможность есть. Покажем, как в Turbo Pascal можно записать что-то в память за границами массива.

```
type
small_array = array[1..10] of integer;
big_array = array[1..100] of integer;
cheat = record
    case integer of
    1: (x: ^small_array);
    2: (y: ^big_array);
    end;
var test:cheat;
    v:small_array; (* это жертва нашего теста *)
begin
  (* v[11]:=5; *)  (* так сделать не получится *)
  test.x := @v;
  test.y^[11]:=3;  (* 11й элемент находится за границей массива *)
end.
```

Языки L-типа можно приблизить к языкам R-типа с помощью программного моделирования.

В дополнение к L и R языкам, формально можно выделить еще и языки **N-типа**, где атрибуты — часть имени. В практически вымершем уже диалекте языка BASIC все переменные, имена которых заканчиваются на $ являются строковыми.

```
Basic
REM ВЕЩЕСТВЕННОЕ
X = 5.5
REM СТРОКА
X$ = "SOME TEXT"
```

Некоторые пережитки подобной практики сохранились в языке PERL. Там приняты следующие обозначения:

```
$foo # переменная
@foo # массив
%foo # таблица
FOO  # файл
&foo # подпрограмма
 # и это все - разные имена
```

Логические значения

Практически в любом языке есть какой-то вариант операторов `if`, `while` и других, требующих для своей работы значения вида “правда”-“ложь”, иначе называемых логическими. Даже если в языке нет специальных значений этого типа, они все равно неявно присутствуют. Обычно существует набор операций, дающих в результате значения логического типа. Например <, >, >= и другие. Для представления логических значений используют следующие приемы:

1. Выделенный тип. В этом случае можно явно объявлять логические переменные, способные принимать всего 2 значения. Тип этот часто называют `Boolean`, `bool` или `logical`, а значения - `true` и `false`.

Pascal

```
var x : Boolean;
x := 4 < 5;
if x then
 …
```

2. Неявные значения. Переменную логического типа объявить нельзя, и выражения, дающие логический результат, могут появляться в строго определенных местах программы.

BASIC

```
IF X < 3 THEN GOTO 44
REM а так нельзя :
P = X < 3
```

3. Использование другого типа, например целого. Такой подход использован в языке C - любое целое число, не равное 0 означает true, а 0 означает false.

```
int x;
x = 4 < 5;
if (x) {
 …
```

Описания переменных

Некоторые языки, например Pascal и C, требуют описания переменных до их использования. В других языках первое появление имени переменной в тексте программы является описанием этой переменной. Иногда переменные можно описывать, но делать это не обязательно. Таким образом, можно выделить два основных признака языков по отношению к описанию переменных: *можно* ли описывать переменные в языке и *необходимо* ли это делать.

Другой важный вопрос относительно описаний состоит в том, какое значение содержит переменная сразу после описания? Присваивание начального значения переменной называется инициализацией. Здесь опять же возможны несколько подходов, и все они где-нибудь используются.

1. Переменная может содержать 0. Проблема с этим подходом состоит в том, что не все типы допускают 0 в качестве значения. Достаточно посмотреть на описанный выше тип Positive.

2. Переменная может содержать произвольное значение, то есть так называемый “мусор”. Вся ответственность за использование переменной до присваивания ей значения возлагается на програмиста. Этот подход частично устраняет вышеописанную проблему.

3. Использование неинициализированных переменных может быть запрещено или невозможно.

4. Для каждого типа существует свое собственное заведомо корректное начальное значение. В объектно-ориентированных языках этот способ расширяется довольно элегантным способом, который мы рассмотрим дальше. Обычно в язык вводят возможность объявлять переменные, сразу присваивая им значения. Например:

C

```c
int mass = 84; length = 50;
```

Часто описание переменной включает только название ее типа, например,

```c
int m;
```

Но часто из присваиваемого значения можно однозначно определить тип переменной. В этом случае в языке C++ можно вместо типа писать слово auto:

```c
auto m = 15; // m - целочисленная переменная
```

а в языке NIM просто не писать название типа:

```
var x, y: int    # объявляем две переменные типа int
var s = "abc"    # объявляем строковую переменную
```

Переменные-ссылки

Во многих языках есть так называемые адресные переменные или указатели. Это переменные, в качестве значения содержащие ссылку.

```
имя — > ссылка —> значение (само является ссылкой) —> значение
```

Иногда их так и называют - ссылочные переменные. Впервые такие переменные появились в языке Algol-68.

C

```
int *p; int q = 5;
p = &q;
*p = 3; // теперь q тоже равно 3
```

PASCAL

```
var p,q: ^integer;
new (q);
q^:=5;
p:=q;  // p и q теперь обозначают одно и то же место в памяти
p^:=3  // теперь q^ тоже равно 3
```

Обратите внимание на операторы * в C и ^ в Pascal. Они осуществляют так называемую *операцию разыменования*, или получение переменной по ссылке. Таким образом, если у нас переменная p является ссылкой на целое

```
**(p —> ссылка) —> ссылка на целое -> значение**
```

то *p это как если бы у нас была такая переменная с именем "*p", объединяющая в себе первые два элемента данной цепочки

```
***p — > ссылка на целое —> значение**
```

Важно заметить, что *p ведет себя как полноценная переменная - слева от знака присваивания она означает ссылку, справа — значение. Заметим также, что при начальной инициализации переменной это не так.

```
int *p = 0;     // Это присваивание начального
                // значения ссылке (а не переменной)
*p = 0;         // Это копирование значения в то
                // место, на которое указывает ссылка
```

Для чего нужны ссылочные переменные? В основном они призваны решать три ответственные задачи.

1. Сделать так, чтобы две разные переменные обозначали одно и то же место в памяти.

2. Обеспечить возможность динамического распределения памяти, то есть выделения участков памяти под новые данные в процессе выполнения программы. Таким образом достигается экономия памяти.

3. Передавать параметры в процедуры, что является специальной разновидностью случая 1. Это мы будем подробно разбирать чуть позже, в разделе, посвященном процедурам и функциям.

При внимательном рассмотрении процесса разыменования возникает вопрос: куда указывает ссылка с самого начала? Дело в том, что ссылка вообще-то не обязана указывать на какую-то память, и для реализации динамического распределения памяти нам как раз и нужны ссылки, “ни на что не указывающие”, для которых мы впоследствии выделим память. Как мы помним, в языке L-типа определить, что именно содержится в данном участке памяти, во время выполнения программы невозможно.

Попытка читать и писать данные по произвольному адресу может привести к непредсказуемым последствиям. Таким образом, имеется существенное отличие между начальными значениями числовых переменных и начальными значениями ссылок. Для первых большинство (или все) возможные сочетания бит представляют собой разрешенные значения, за редкими исключениями, а для ссылок наоборот — подавляющее большинство адресов вызовет ошибку при попытке их разыменования (хотя являться значением ссылочной переменной они могут). Иначе говоря, далеко не все возможные адреса содержат осмысленные данные определенного типа, не все возможные адреса вообще есть в компьютере, не все возможные адреса являются адресами памяти, и не все возможные адреса памяти доступны данной программе.

Для решения этой проблемы в языках L-типа обычно выделяется специальная константа, называемая NULL или nil, которую можно присваивать любой переменной ссылочного типа. При этом гарантируется, что ни один “легальный” адрес такого значения иметь не будет. Чаще всего константа NULL численно равна 0, но это не обязательно так.

Присваивая значение NULL или nil, можно получить указатель, не ссылающийся ни на какую область памяти, и более того, используя это значение, можно проверить, что указатель ни на что не ссылается. Таким образом, “пожертвовав” одним значением из многих тысяч возможных, решили важную проблему с указателями.

C

```c
int *a = NULL;
…
if (a == NULL) // память не выделена
   a = malloc(100); // выделение памяти
```

Pascal

```pascal
type
   arr = array[1..20] of integer;
```

```
   parr = ^arr;
var p:parr;
begin
  p:=nil;
…
  if  p = nil then
      new(p);
end.
```

Область видимости переменных

Вводя в языки переменные, их создатели обеспечили програмистам возможность *разными* именами без всяких сложностей называть *разные* области памяти. Ссылки дают возможность *разными* именами обозначить *один и тот же* участок памяти. Логично было бы спросить, можно ли *одним и тем же* именем назвать *разные* вещи? Разумеется, можно. Представим себе, что было бы, если бы это было не так. Если бы каждое имя в программе было закреплено за строго определенным участком памяти, то, прежде всего, намного осложнилась бы совместная работа нескольких программистов над одной и той же программой. Им пришлось бы договариваться о том, какие имена кто из них использует. Например, один мог бы все идентификаторы начинать с alice_, второй с bob_ и так далее. Кроме того, многие переменные, не несущие смысловой нагрузки, например, вездесущий параметр цикла i, пришлось бы каждый раз называть по-новому. Да еще учитывать, какие имена уже использованы, а какие еще нет.

К счастью, в большинстве языков идентификаторы имеют строго определенную область видимости. Как правило, областью видимости идентификатора является блок, в котором он объявлен.

Блоком называется участок программы, заключенный между открывающей и закрывающей *операторными скобками*, например begin…end в Pascal или {} в C/C++ или выделенный каким-то другим способом.

Примеры

ALGOL 60

```
begin integer x,i,z;
  x := 3;
  begin real x
    x := 5.5;  comment: это уже другой x;
    i := 6;
  end;
end;
```

C++

```
{
  int x,i,z;
  x = 3;
  {
```

```
    double x;
    x = 5.5;  // это уже другой x
    i = 6;
  }
  x = 7;  // возвращаемся к прежнему пониманию x
}

// проблема :
for (int i = 0; i < 3; i++)
    cout << i << endl;
for (int i = 0; i < 5; i++)   // это та же самая i
    cout << i << endl;        // или другая или это ошибка?
```

В языке NIM (с синтаксисом, похожим на Python) для создания блока используется ключевое слово block

```
block myblock:
  var x = 25
echo x # не работает, x тут не определено
```

В связи с областями видимости идентификаторов возникает целый ряд своеобразных проблем. Первая из них может показаться надуманной: а что если программист, только что “закрывший” вложенным объявлением внешнее, вдруг решит все-таки обратиться к внешней переменной (она же никуда не девается во время выполнения блока, просто ее имя временно отдается другой переменной). Несмотря на явную абсурдность такого вопроса (зачем тогда было называть внутреннюю переменную тем же именем?), некоторые языки предоставляют такую возможность. Так, в C++ можно обратиться к переменной, находящейся на самом верхнем (глобальном) уровне, поставив перед именем двойное двоеточие **::x**

Другая проблема, пока что окончательно не решенная ни в одном императивном языке программирования, состоит в том, что некоторые значения могут получаться в разных ветках оператора **if**.

```
typedef enum { LESS, NOT_LESS } RELATION_A_TO_B;

RELATION_A_TO_B c;

if (a < b)
  c = LESS;
else
  c = NOT_LESS;
```

Вопрос: где должна быть объявлена переменная `c`? Если это сделать до оператора `if`, то в какой-то момент переменная не будет иметь никакого значения. В принципе это то, что нам нужно, но язык программирования об этом "не знает", и может запрещать использование переменной без инициализации. Если же присвоить `c` при объявлении какое-то значение, то у нас появляется какое-то

третье незапланированное значение для c. Можно, конечно, присвоить c одно из объявленных значений, но тогда создается ложное впечатление, что это значение используется по умолчанию. А теперь представьте себе, что значение `c` получается в результате *очень* сложного кода с множеством условных операторов и может иметь не два, а десять разных значений.

В принципе, компилятор может отследить, что значение c присваивается в каждой ветке условных операторов, но такого механизма пока что нет в императивных языках программирования. В функциональных же языках все намного проще:

Lisp:

```
(defconstant LESS 1)
(defconstant NOT_LESS 2)
(setq a 5)
(setq b 3)
(setq x
    (cond
         ((< a b) 'LESS)
         (t 'NOT_LESS)
    )
)
(write x)
```

Распределение памяти

От ссылок и указателей логично будет перейти к задаче распределения памяти при выполнении программы, или, что то же самое, отождествлению ссылок (переменных или констант) с конкретными физическими адресами памяти.

Статическое распределение памяти - наиболее простой способ. Каждой переменной ставится в соответствие строго определенный адрес памяти, по которому находится ее значение. Этот адрес не меняется на протяжении всего выполнения программы. К этому классу относятся переменные, описанные в языке C на верхнем уровне, вне функций, или с ключевым словом static.

C:

```
static int i = 10;
```

Автоматическое распределение памяти используется для тех переменных, которые в C, Pascal или PHP описываются внутри функций. При вызове функции или входе в блок переменная создается, ей выделяется какое-то место в памяти для хранения значения, при завершении функции или блока - место освобождается.

C

```
int function_auto()
{
```

```
    int p = 10; // переменная p - автоматическая
    p = p+1;
    return p;
}
// переменная p больше не существует
```

Динамическое распределение памяти это постановка в соответствие ссылочным переменным каких-то адресов памяти.

C

```
int main(){
    int *a;
    a = (int *) malloc(100);
    a[5] = 33;
    free(a);
}
```

Pascal

```
procedure testmem;

type intarray = array [1..100] of integer;

var a : ^intarray;

begin
  new(a);
  a^[5] := 33;
  dispose(a);
end;
```

Java

```
int [] a;
a = new int[100];
a[5] = 33;
```

Функция malloc в C, операторы new в Pascal, C++ и Java выделяют память для переменной. Заметим, что сама переменная-ссылка, которая используется для доступа к выделенной памяти, может относиться к классу автоматической памяти. В этом случае при завершении блока память останется распределенной, а единственное средство доступа к ней будет потеряно. В C и Pascal существуют средства (free и dispose соответственно) для того, чтобы освободить память перед тем, как средство доступа к ней будет потеряно. Почему память не освобождается автоматически? Вспомним п.1) из

списка “для чего нужны ссылки”. Две и более переменных могут указывать на один и тот же адрес памяти.

Может существовать другая ссылочная переменная, указывающая на тот же участок памяти. Освобождать память автоматически можно только тогда, когда не останется ни одной переменной, на нее ссылающейся. В C и Pascal следить за тем, чтобы вся полученная от системы память возвращалась обратно, должен программист. Однако в языке Java никакого аналога функции free мы не находим. Получается, что есть какой-то способ автоматически освобождать память? Есть, но он связан с большим объемом программного моделирования. Проверять, не найдется ли где-нибудь еще одна переменная, указывающая на тот же адрес, каждый раз при выходе из блока было бы слишком накладно. Поэтому такую проверку выполняют сразу для всех переменных и для всех выделенных блоков памяти. Те участки памяти, к которым нет доступа, освобождаются. Эта довольно сложная процедура носит название “сборка мусора”. Зато программисту не приходится думать о том, нужно ли освободить память при выходе из блока, где она выделялась.

Выражения и операторы

Программисты — не математики, как бы нам этого ни хотелось.
Ричард Гэбриел

Выражения

Рассмотрим два определения:

Выражение это набор инструкций по вычислению значения, записанных по определенным правилам.

Выражение это запись функции, состоящая из операндов (которые представляют собой значения) и операций.

Как мы уже говорили, существует два способа представлять себе процесс выполнеия программы. Вопрос состоит в том, какое из них удобнее использовать. В ранних языках программирования, таких как Автокод, существовало настолько много ограничений на способ записи выражений, что представление о выражении как о наборе инструкций более соответствовало тому, что приходилось писать программисту. В современных языках ограничения на запись выражений становятся все менее заметными.

Чаще всего для записи выражений используется привычная нам алгебраическая запись (слегка адаптированная), в которой знак операции ставится между значениями: 2 + 3. Такой способ записи называется *инфиксным*.

Но это не единственный способ. При записи функций мы ставим обозначение функции, такое как sin или tg, перед значением: sin x. Такая запись называется *префиксной*. В языках программирования тоже используют аналогичный подход. Но никто не мешает операции + - * / тоже трактовать как функции и писать, например, вместо 2 + 3 выражение + 2 3, или, в более традиционной форме, plus (2,3), или даже +(2,3). В тех языках программирования, которые используют префиксную запись, обозначение функции или операции обычно вносят внутрь скобок: (plus 2 3).

Записывая в виде функций все арифметические операции, мы немного потеряем в читаемости текста, зато приобретем целый ряд важных преимуществ. Во-первых, можно будет забыть о том, что умножение делается перед сложением. Порядок вычисления выражения всегда будет определяться

скобками. Сравните 2 + 3 * 4 и (plus 2 (mult 3 4)). При втором способе записи нет разночтений. Во-вторых, все операции и функции в выражениях будут записываться совершенно единообразно. Что существенно облегчает компиляцию программ и позволяет работать с выражениями как с данными. Основная проблема такого способа записи это обилие скобок.

Вот цитата из руководства по языку LISP: «PROG-выражение всегда заканчивается как минимум пятью скобками»

Но существует способ записи выражений, при котором можно обойтись вообще без скобок. Этот способ называется *постфиксным*, и состоит в том, что знак операции пишется после операндов.

Уже рассмотренное выражение запишется в нем как `3 4 * 2 +` или даже как `2 3 4 * +`. Чтобы понять последнюю запись, вспомним, как мы вычисляли значения выражений, когда изучали подстановки. Здесь можно делать то же самое - заменяем 3 4 * на 12, а 2 12 + на 14, и выражение вычислено. Но можно использовать и другой подход, очень важный с точки зрения теории. Рассмотрим структуру данных, называемую стек.

> СТЕК (англ. stack — стопка) — структура данных с методом доступа к элементам LIFO (Last In — First Out, последним пришел — первым вышел). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю. Добавление элемента, называемое также заталкиванием (push), возможно только в вершину стека (добавленный элемент становится первым сверху), выталкивание (pop) — также только из вершины стека, при этом второй сверху элемент становится верхним.

При использовании стека интерпретация постфиксных выражений определяется всего двумя правилами:

1. Если в записи выражения встретилось число, то занести его в стек.
2. Если встретился знак операции, то выбрать из стека столько чисел, сколько требуется для данной операции, выполнить над ними операцию, и результат занести обратно в стек.

По окончании процесса выражение должно закончиться, а в стеке должно остаться ровно одно число, оно и будет значением выражения. Недостаток постфиксной записи состоит в том, что каждая операция обязана иметь строго определенное число операндов, и вольности вроде той, что знак - может обозначать вычитание, а может и знак числа, не допускаются. Скобочная префиксная запись, наоборот, позволяет под одним знаком операции объединять сколько угодно операндов:

```
( PLUS 2 3 (MULT 3 5 8) 11 )
```

Заметим, что данное свойство зависит не от того, ставим мы знак операции до или после операндов, а от соглашения о том, сколько операндов может иметь та или иная операция. Однако, в большинстве языков программирования используются либо инфиксная запись, либо префиксная со скобками и произвольным числом аргументов, либо постфиксная с фиксированным числом аргументов у каждой операции. Возможно, это связано с тем, что выражение вида `\+ + 2 2 \* 3 5` читается хуже, чем `(+ (+ 2 2) (\* 3 5))`

Таким образом, любое выражение можно записать тремя различными способами: инфиксным, префиксным и постфиксным. Поскольку это все-таки одно и то же выражение, должен существовать

способ автоматического перевода из одного представления в другое. И он действительно существует. Несложно придумать, например, преобразователь постфиксной записи в инфиксную - для этого в вышеприведенных правилах нужно вместо того, чтобы вычислять выражение, просто заносить в стек соответствующую строку в скобках. Однако, обратное преобразование не такое простое. Чтобы понять, как это работает в общем виде, рассмотрим еще один способ записи выражений - в виде дерева.

```
          /
     +          *
(2       3) (4     8)
```

Это запись выражения `(2+3) / (4*8)`

Пишем в узлах дерева знаки операций, а на листьях - числа. Теперь преобразование дерева выражения в любую из форм записи достаточно просто. Для префиксной записи: если нижележащих узлов нет,то это число, пишем его. Если есть, то ставим скобку, пишем знак операции, стоящий в данном узле, а далее обходим нижележащие узлы, и для них выполняем те же действия. Для инфиксной записи алгоритм будет тот же самый, но сначала пишем первый аргумент, потом знак операции, потом второй аргумент. Что надо делать для получения постфиксной записи, догадайтесь сами.

На основе постфиксной записи построено несколько языков. Первым из них был FORTH, рассмотрим его более подробно. Любая программа на Форте является выражением, записанным в постфиксной форме. Имеется стек, в котором происходит вычисление этих выражений. Некоторые операции вызывают побочные эффекты, например . (точка) печатает число, находящееся на верху стека. Программа

```
3 3 * 5 5 * + .
```

вычисляет значение выражения `3 * 2 + 5 * 2` и печатает результат.

Для удобства работы со стеком существуют служебные операции, например, DUP дублирует верхушку стека. Так,

```
3 DUP +
```

это то же самое, что

```
3 3 +
```

Другая служебная операция, SWAP, меняет местами два верхних элемента стека. Например,

```
5 3 - .
```

печатает 2, и

```
3 5 SWAP - .
```

тоже печатает 2. Теперь посмотрим, как это можно использовать. Определение подпрограммы начинается с : (двоеточия), после которого идет имя подпрограммы, дальше текст, и в конце ставится ;

```
: SUM-OF-SQUARES  DUP * SWAP DUP * + ;
```

После того, как подпрограмма определена, ее можно использовать точно так же, как любую другую операцию:

```
3 5  SUM-OF-SQUARES .
```

Это то же самое, что просто написать

```
3 5 DUP * SWAP DUP * + .
```

Можно пойти дальше, и заметить, что возведение в квадрат в виде DUP * встречается у нас дважды.

```
:  SQUARED DUP * ;
:  SUM-OF-SQUARES SQUARED SWAP SQUARED + ;
```

Легкость выделения подпрограмм, а также то, что накладные расходы на вызов подпрограмм у Форта намного меньше, чем у других языков, приводят в результате к очень компактным программам.

Арность операций

Известные нам арифметические операции используют два операнда, и поэтому называются двуместными или бинарными. Если рассматривать знак числа как операцию (а это так и есть, например в выражении `-(2*Z - 3*X)`), то его, а также элементарные функции будут одноместными, или унарными операциями. Иногда встречаются и трехместные, или тернарные операции. Так, в языке C

```
y = x > 0 ? x : 0;
```

означает «если x > 0 то x, иначе 0». В ряде случаев число операндов может иметь значение само по себе, как характеристика операции. Обычно это происходит в тех языках, в которых программист может вводить свои собственные операции. Тогда это свойство операций, исходя из общей части названий “унарные”, “бинарные”, “тернарные” именуют арностью (arity).

Полиморфные операции

Как вы, наверное, уже заметили, одна и та же операция (-) может обозначать разные вещи, в зависимости от её арности. Выше говорилось, что результаты некоторых операций над разными типами могут различаться. Однако мы привыкли писать что-то вроде a+b, не задумываясь о том, целые a и b или вещественные.

Распространение этой практики на языки программирования привело к понятию *полиморфизма*, или возможности выполнять одинаково записанные операции над разными типами данных, получая различные результаты. Как правило, арифметические операции в языках программирования обладают встроенным полиморфизмом.

Что касается полиморфизма, вносимого программистом, то тут единства подхода нет. В ранних языках определять полиморфные операции и функции, как правило, было невозможно. Особенно удивительно это выглядит в языке Modula-2, в котором смешивать типы данных в выражениях запрещено.

Порядок действий

В школе нас учили, что умножение и деление выполняются прежде сложения и вычитания. Подавляющее большинство языков программирования заимствует и расширяет эту практику. Поскольку для программирования требуется несколько больше бинарных операций, чем для арифметики, на дополнительные операции, такие как and, &, &&, or, и другие, тоже распространяется идея приоритетов. 7-8-уровневая система приоритетов операций в языке программирования не редкость. Главная проблема тут состоит в том, что в разных языках одни и те же операции имеют разные приоритеты.

Например, в C

```
if (a<3 && a>0)
```

вполне легальное выражение, поскольку приоритет && ниже, чем у > и <.

А в Pascal

```
if a<3 and a>0 then
...
```

является ошибкой, потому что компилятор пытается интерпретировать его как

```
a < (3 and a) > 0
```

По какой-то причине создатель языка Pascal решил,что так будет лучше. В качестве радикального средства борьбы с разночтениями было предложено приоритеты отменить вовсе. Так было сделано, например, в языке APL. Однако, эта практика не прижилась, и приоритеты операций есть почти во всех языках, использующих инфиксную запись выражений. Немного проще тем, у кого выражения записываются в префиксной или постфиксной форме. Проблема расстановки приоритетов не вызывает на головы их создателей постоянных проклятий со стороны программистов. Вывод из рассуждения о приоритетах простой: сомневаетесь - ставьте скобки.

Получение значений

Привычной для нас является запись вида

```
x = x+1;
```

где x с разных сторон от знака присваивания означает разные вещи, и смысл данного символа выясняется из контекста. Но не везде это так, есть языки, в которых разыменование, или, что то же самое, получение значения переменной по ее имени, должно выполняться в явном виде. Один из таких языков - уже упоминавшийся Forth. В нем есть операция записи значения в переменную ! и операция взятия значения переменной и помещения его на стек @. Соответственно, **x = x+1** запишется на **Forth** (в постфиксной записи) как

```
x @ 1 + x !
```

что читается как “x взять, 1 прибавить, x записать”. Сама переменная x в этой записи просто обозначает ссылку на какое-то место в памяти.

Побочные эффекты

В большинстве языков предполагается, что вычисление выражений не имеет неявных побочных эффектов, то есть от того, будет выражение вычислено или нет, ничего не изменится. Иначе говоря, от замены

```
y = 2 * 2;
```

на

```
y = 4;
```

не изменится ровным счетом ничего. Однако, есть и исключения. В языках C и C++, а также в других, унаследовавших их синтаксис, операция ++ увеличивает значение переменной на 1. В связи с этим

новичков в C часто озадачивают вопросом: чему равно значение выражения

```
(x++) + (++x)
```

Попытки выяснить этот вопрос, откомпилировав и запустив программу, ответа на вопрос не дают. Потому что правильный ответ, который дает стандарт языка C, такой: значение этого выражения *не определено*. Стандарт не определяет такое значение именно вследствие общего принципа: вычисление выражений не должно менять среду, а если уж оно ее меняет, то пусть делает это предсказуемым образом.

Операции, определяемые программистом

По-видимому, всем создателям ранних языков программирования хотелось дать программистам возможность определять свои собственные операции. Однако, единого мнения по поводу того, как это следует делать, до сих пор не сложилось. В языке C++, например, можно переопределять операции, уже имеющиеся в языке, для своих типов. Например, операция << исходно означает сдвиг числа влево (умножение его на 2^N). Эта же операция, примененная к потокам вывода, означает вывод значения в поток с использованием формата по умолчанию.

```
cout << "Значение x " << x;
```

В языке NIM возможно определение вообще любых своих знаков операций, состоящих из символов `+ - * \ / < > = @ $ ~ & % ! ? ^ . |`

```
proc `+$+` (x: int): string =
...
# теперь у нас есть операция +$+

var s:string = "Hello " & +$+ 5
echo s
```

Операторы

В отличие от выражений, операторы как раз предназначены для изменения среды. И главным в этой роли выступает, конечно, оператор присваивания. Все операторы можно разделить на две группы операторов: присваивания и управления.

Операторы присваивания

Оператор присваивания меняет среду, и в этом состоит его основное и единственное назначение. Традиционно оператор присваивания записывается в виде, напоминающем математическое равенство:

```
e1 = e2
```

Это сходство с математической записью много лет являлось проблемой при освоении программирования, поскольку программистами чаще всего становились математики, привыкшие, что e1 = e2 означает не «вычислить значение e1 и записать его как e2», а нечто другое. Для математика e1 = e2 и e2 = e1 это примерно одно и то же. Чтобы математикам было проще воспринимать обозначения языков программирования, в языке Algol ввели новое обозначение e1 := e2, асимметричным видом оператора намекая на то, что e2 := e1 это не совсем то же самое. Изобретатель языка Pascal Никлаус Вирт и по сей день полагает, что такой вид оператора присваивания необычайно важен, хотя в наше время программистами чаще всего становятся люди, весьма далекие от математики. Гораздо хуже была другая ошибка, допущенная при проектировании сразу нескольких ранних языков. В PL/1 и Basic знаки присваивания и сравнения совпадают. Как, например, следует понимать оператор x = y = 2? Оказывается, это означает, что x присваивается результат сравнения y и 2. То есть, если y равен 2, то x будет присвоено значение true. Какими именно сочетаниями символов записываются операции присваивания и сравнения, не так важно. Главное, что они должны быть разными.

Совпадение обозначений этих двух операций не только мешает читать программы, но и делает невозможным введение так называемого кратного присваивания, основная идея которого как раз и состоит в том, чтобы x=y=2 понимать как “присвоить y значение 2, а x значение y”. У оператора присваивания известно два основных способа развития. Первый из них состоит в том, чтобы присваивать значения нескольким переменным одновременно. Тогда можно писать, например, x,y = 5,6 но это еще не самое интересное. Такой способ записи позволяет поменять местами значения двух переменных, не используя третью: x,y = y,x. А это уже нечто действительно полезное.

Тем не менее, программистам, учившимся языкам Basic или Pascal, бывает сложно привыкнуть к системе обозначений языка C, в котором операция сравнения обозначается как == И здесь надо сделать важное замечание относительно языка C и производных от него. В них присваивание является именно операцией, которая имеет результат. Вполне логично, что этим результатом является присваиваемое значение. Таким образом, y=x=2 в языке C означает “присвоить переменной y результат присваивания переменной x значения 2. Теперь зайдем немного вперед и посмотрим на такой оператор языка C:

```
if (x=y) printf (“yes”);
```

Результатом операции присваивания является присваиваемое значение. Поскольку в C нет логического типа, printf выполнится всегда, когда y исходно содержит любое ненулевое значение. Самое интересное, что после выполнения этого оператора x будет действительно равен y. Такие ошибки не так-то просто находить. Вторая важная особенность языка C состоит в том, что, поскольку = это операция, то `x = y+1` это выражение. Если в Pascal

```
x := y+1;
```

формально означает «вычислить выражение y+1 и результат присвоить переменной x, то в C семантика этого оператора формально состоит в том, чтобы просто вычислить выражение x = y+1. Значение попадет в переменную x в качестве побочного эффекта. Поэтому в C можно написать

```
y+1;
```

и даже просто

```
345;
```

Это означает «вычислить выражение, а результат забыть». В приведенных примерах это действие не имеет смысла, но в определенных ситуациях, которые мы рассмотрим в разделе «Функции и процедуры» оно оказывается полезным.

Оператор кратного присваивания мы встречаем в языках Python и Lua

```
x,y = 3,5 # присваивает x 3, а y - 5
```

А что будет, если написать

```
x,x = 3,5
```

Наверное, правильно было бы объявить, что результат такого присваивания не определен, как это и сделано в языке Lua.

Совместимость типов: что можно присваивать

В большинстве языков программирвания оператор присваивания делает нечто большее, чем просто копирование байтов из одной области памяти в другую. Например, при присваивании обычно производится приведение типа правой части к типу переменной левой части. Во многих объектно-ориентированных языках можно переопределить оператор присваивания так, что при его использовании с определенными типами или элементами структур будут происходить какие-то дополнительные действия. Например, в системе Delphi/Lazarus присваивание

```
...
Label1.Caption := 'Hello, World';
```

Приводит к цепочке довольно сложных действий, в результате которых на экране компьютера изменяется текст.

В Javascript тот же подход применяется в системе vuejs, где такое присваивание:

```
this.message = "Hello World";
```

приводит к изменению кода HTML, который отображается в данный момент.

В связи со всем этим возникает вопрос о том, что же можно присваивать, а что нельзя, и всегда ли мы можем написать `a = b;` если a и b - значения одного и того же типа? Ответ, как обычно, зависит от языка. Например, в C нельзя присваивать массивы.

Обычно авторы языков стараются сделать так, чтобы присваивания были по возможности универсальными.

Однократные присваивания

В некоторых языках программирования есть нечто промежуточное между константами и переменными, а именно переменные с однократным присваиванием.

```
let name = readLine(stdin)
# name = "Paul" # так нельзя
```

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;    // ошибка!
    println!("The value of x is: {}", x);
}
```

Переменным, значения которым присвоены оператором let в этих языках, менять значения запрещено. Это сделано для повышения качества кода, таким образом программист заявляет, что данное значение интересует его само по себе и не будет изменяться в дальнейшем. Кроме того, однократное присваивание гарантирует, что это имя не будет позже использовано для каких-то других целей, что уменьшает количество потенциальных ошибок.

Наконец, во многих функциональных языках присваиваний нет вообще, хотя часто встречаются внешне похожие на присваивание конструкции. Например, в языке OCaml

```
let x = 10;;
```

это не присваивание переменной x значения 10 (хотя очень похоже), а определение имени x и установление соответствия константы 10 этому имени. Для описания функции используется в точности такой же синтаксис:

```
let sum a b = a + b;;
```

Это похоже на однократные присваивания, но смысл (семантика) этих выражений другой. В языке Elixir знаком = так и вовсе обозначается сопоставление с образцом, так что можно написать:

```
{1,2,p} = {1,2,42}
```

при этом переменной p будет поставлено в соответствие число 42.

При изучении того или иного языка программирования, нужно найти ответы на следующие вопросы:

- есть ли в данном языке кратные присваивания?
- есть ли однократные присваивания?
- возможны ли вообще присваивания?

Операторы управления

С точки зрения терминологии более правильно некоторые операции C, такие как ++, --, = называть операторами. И все рассуждения о присваивании можно считать переходной частью от выражений к операторам.

Первый оператор, который мы рассмотрим, играет основополагающую роль в программировании и в подавляющем большинстве языков не обозначается вообще никак. Это оператор “следования”, он гарантирует нам, что вычисления выполняются в строго определенной последовательности. Как правило, операторы выполняются в той последовательности, в которой они написаны, однако бывают и исключения. Одним из таких исключений является язык Occam, в котором можно в явном виде написать:

```
SEQ
   x := x + 1
   y := x * x
```

и два присваивания будут выполнены последовательно. А можно написать:

```
PAR
   x := x + 1
   y := y + 1
```

тогда присваивания могут быть выполнены в любом порядке, и даже одновременно, если в компьютере есть несколько процессоров. Именно для работы с параллельными вычислениями и создавался язык Occam.

Почему про оператор следования обычно ничего не говорят? Видимо, вследствие кажущейся его очевидности. Тем не менее, в теоретических работах, таких как книга Э.Дейкстры “Дисциплина программирования” он рассматривается наравне со всеми остальными.

В распространенных языках программирования оператор следования, как правило, проявляется эпизодически, в виде гарантий того, что определенный участок программы будет выполнен обязательно до или после какого-то другого ее участка. Например, в C

```
if ( a == fun1(x) && c == fun2(y) )
```

Здесь разработчики языка гарантируют нам, что fun1(x) обязательно будет вычислено раньше fun2(y), хотя в выражении

```
x = fun1(x) + fun2(y)
```

такой гарантии нет.

С оператором следования тесно связана концепция составных операторов, про которые мы уже упоминали в связи с операторными скобками. Идея состоит в том, чтобы дать возможность программисту написать несколько операторов в любом месте, где можно написать один. В Algol и Pascal используются конструкции begin … end, а в C-подобных языках - фигурные скобки {}. В некоторых языках составных операторов нет вообще. Иногда это является недостатком языка, например в некоторых диалектах Basic после IF можно написать только один оператор. А в языках Modula-2 и Lua составные операторы не нужны, но это важное преимущество этого языка. Вместо использования каких-либо скобок там в конце каждого структурного оператора надо писать свой END.

```
IF x<y THEN
  writestr(“yes”);
  x:=y;
ELSE
  writestr(“no”);
END;

FOR i:= 1 TO 10 DO
 s:=s+1;
END;
```

```
x = 5
if x < 6 then
  print("yes");
  x=7;
else
  print("no");
end

s = 0
for i=1,10,1 do
 s = s+i;
```

```
end

print (s);
```

Такой способ записи добавляет к большинству программ несколько лишних END, но зато улучшает читаемость и вводит единообразие в структуру программы. Можно пойти по этому пути еще дальше, и заканчивать все if на end if; все for на end for, и так далее. В этом случае будет более понятно, какой именно оператор заканчивается. Так сделано в некоторых диалектах языка Basic. Еще один весьма странный вариант этого способа группировать операторы был применен в языке Algol-68. Там каждый оператор if, case, итп, предлагалось заканчивать тем же словом, но написанным задом наперед: fi, esac и так далее. Этот способ остался в языке интерпретатора команд системы UNIX, который называется bash.

Про необычный способ группировать операторы в языке Python мы уже говорили.

Вторым из обычно игнорируемых операторов является пустой оператор. Он вообще ничего не делает, но иногда бывает нужен, в основном для того,чтобы занять место, где по правилам синтаксиса должен быть оператор. В C и Pascal он обозначается одинаково, как действительно пустое место, после которого идет точка с запятой. Однако, в языке Fortran ничегонеделание обозначается длинным словом CONTINUE, а в языке Python коротким словом pass.

Два способа использования пустого оператора встречаются чаще всего. Первый - это ожидание какого-либо события.

```
while (! key_pressed());
```

Здесь, кроме проверки условия, ничего делать не надо.

Второй способ позволяет немного улучшить читаемость некоторых программ. Если в операторе if содержится очень сложное условие, и какие-то действия надо выполнить только в случае невыполнения этого условия, то вместо отрицания этого условия, что привело бы к появлению еще одной пары скобок, можно написать

```
if (a<b) or ( ( c<d) and (p=q)) and (a<2) then
   ; // ничего не делать
else
   do_something;
```

Интересно, что в языке Perl специально для этого случая придумана конструкция unless, по смыслу противоположная if (см. следующий раздел). Пустой оператор иногда путают с разделителями операторов. В качестве разделителей чаще всего используют точку с запятой, но, например, в языке Basic это двоеточие или перевод строки, а в FORTRAN каждый оператор должен начинаться с новой строки. Существенная разница тут имеется между C-подобными языками и Pascal. В языке Pascal точка с запятой разделяет операторы, а в C - заканчивает их. Поэтому запись

```
{
  print_number(4);
  print_number(5);
}
```

```
begin
  print_number(4);
  print_number(5);
end
```

в C означает просто два вызова подпрограммы, а в Pascal - два вызова подпрограммы, за которыми следует пустой оператор.

В других языках ситуация с разделителями еще более запутанная. В языке Javascript точку с запятой после оператора можно не ставить, но рекомендуется это делать, а в языке Python тоже можно, но рекомендуется как раз не делать этого.

Альтернатива

Оператор выбора, обычно называемый словом if дает возможность выполнить один из двух участков программы в зависимости от истинности некоторого условия.

```
if (a < b) print(a);
```

За исключением нескольких ранних флуктуаций, во всех процедурных языках он записывается примерно одинаково. Хотя в Perl придумали еще два дополнения. Во-первых, там можно писать `if` после оператора:

```
print(a) if a<b;
```

во-вторых, там есть инверсия if - оператор unless, который позволяет выполнить оператор при **не**выполнении некоторого условия.

```
print(a) unless a>=b;
```

Но в большинстве случаев разработчики языков используют 3 базовые формы оператора if:

Первая форма: проверить условие, в случае его истинности выполнить оператор p, в противном случае ничего не делать.

```
if <условие> then p;
```

Вторая форма: проверить условие, в случае его истинности выполнить оператор p1, в противном случае выполнить оператор p2.

```
if <условие> then p1 else p2;
```

(Здесь и далее мы будем использовать запись вида <что-то в угловых скобках> для обозначения различных категорий языковых конструкций. Что в данном случае имеется в виду под словом <условие>, интуитивно понятно. Ближе к концу курса мы введем по этому поводу более строгие определения. Сейчас достаточно понимать, что угловые скобки не являются частью оператора, а просто ограничивают описание на естественном языке)

Практика показала, что ситуаций, когда p2 в свою очередь, тоже является условным оператором, предостаточно. Поэтому во многих языках существует специальная

Третья форма (многовариантный if): проверить условие, в случае его истинности выполнить оператор p1, в противном случае проверить другое условие, в случае его истинности выполнить оператор p2, и так далее несколько раз; если все условия ложны, выполнить оператор pN.

```
if <условие1> then p1 elseif <условие2> then p2 elseif <условие3> then p3 else pN;
```

В одних языках используется написание elseif, в других elsif, else if или как-то иначе. В данном примере оператор p3 выполнится только в случае, когда <условие1> и <условие2> ложны, а <условие3> истинно.

С оператором if в тех языках, в которых есть операторные скобки, имеется одна проблема. Как следует понимать такую запись:

```
if  a>b then  if c<d then print ('1') else print ('2');
```

К какому if относится последний else? Вместо того, чтобы решать этот вопрос (а он, в отличие от проблемы с (x++) + (++x), имеет решение) надо просто всегда в таких случаях ставить операторные скобки. В языках, подобных Modula-2, Lua или Python, эта проблема вообще не возникает.

Оператор выбора

Если в третьей форме оператора if все условия являются взаимоисключающими и используют сравнение с одной и той же переменной, получается оператор case. Ruby:

```
case n
when 0 then         puts 'Ноль'
when 1, 3, 5, 7, 9 then puts 'Нечетное'
when 2, 4, 6, 8    then puts 'Четное'
else                puts 'Меньше ноля или больше девяти'
end
```

В нем значение некоторого выражения последовательно участвует в нескольких условиях. Если какое-то из них оказывается истинным, то выполняется соответствующий участок кода. Все это похоже на многовариантный if, хотя очень часто на условия накладываются некоторые ограничения. Но если операторы if в разных языках обычно похожи, то операторы case очень сильно отличаются. По какой-то причине создатели языков не могут прийти к единому мнению о том, как именно этот оператор должен выглядеть и что он должен делать.

Встречались даже совсем странные случаи:

Algol:

```
case i goto 11,3,6,23,1,44,11
```

Basic:

```
ON i GOTO 11,3,6,23,1,44,11
```

что значит: при i равном 1 перейти к метке 11, при i равном 2 перейти к метке 3 итд.

Противоположной экстремальной формой оператора case можно считать многовариантный `if`, в котором на условия не накладывается никаких ограничений.

Все, что по смыслу находится между этими двумя крайними формами, встречается в тех или иных языках в качестве оператора case. Изучая этот оператор в разных языках программирования, следует обратить внимание на следующие вопросы:

- какого типа может быть выражение для выбора варианта?
- могут ли в вариантах использоваться диапазоны значений?
- могут ли в вариантах использоваться произвольные условия (не только равенство)?
- существует ли вариант, который выполнится, если все условия ложны (else, default)?
- что происходит после выполнения выбранного варианта?

В языке Pascal выражение для выбора варианта должно быть перечислимого или целого типа. В условиях не могут использоваться диапазоны и произвольные условия. Вариант по умолчанию есть в некоторых диалектах и называется else. После выполнения условия происходит завершение выполнения оператора. Pascal:

```
case X of
  0: writeln('ноль');
  1, 5, 7, 9: writeln('нечетное');
  2, 4, 6, 8: writeln('четное');
  else writeln('другое');
end;
```

В языке C выражение для выбора варианта должно быть целым или приводимым к целому. В условиях не могут использоваться диапазоны и произвольные условия. Вариант по умолчанию есть и называется он default. После выполнения условия происходит выполнение кода следующего по порядку условия, независимо от его истинности. C:

```
switch(n) {
  case 0:
    printf("Ноль");
    break;
  case 1:
  case 2:
  case 3:
  case 4:
  case 5: printf("Меньше шести\n");
  case 6: printf("Меньше семи\n");
  case 7:
  case 8: printf("Меньше девяти\n"); break;
  default:
    printf("Не знаю, что с этим делать\n");
}
```

Оператор break завершает выполнение оператора case. Таким образом, при n равном 6 в этом примере будет напечатано :

```
Меньше семи
Меньше девяти
```

Таким образом, в C каждый case оператора switch аналогичен паре if - goto.

В языке Ruby, на котором был первый приведенный пример, выражение для выбора варианта может быть любого типа, допускающего сравнение. В условиях могут использоваться диапазоны и разнообразные условия. Вариант по умолчанию есть и называется он else. После выполнения условия происходит завершение выполнения оператора. Кроме того, case в языке Ruby может быть использован для получения значения:

```
kind = case year
    when 1850..1889 then "Blues"
    when 1890..1909 then "Ragtime"
    when 1910..1929 then "New Orleans Jazz"
    when 1930..1939 then "Swing"
    when 1940..1950 then "Bebop"
    else "Jazz"
end
```

Достаточно экстремальный вариант использования такого универсального оператора case приводит Ч. Уэзерелл:

```
select TRUE of
 (x=0): sign = 0;
 (x<0): sign = -1;
 (x>0): sign = 1;
end select;
```

Смысл такого фрагмента становится понятен не сразу. Разобравшись с ответами на эти вопросы, можно использовать оператор case (или switch, в языке C) и переводить программы с других языков. Например, для перевода оператора switch с C на Pascal в общем случае придется использовать набор команд if ... then ... goto. Однако, если каждый вариант в программе на C завершается оператором break, то программа на Pascal может быть написана с использованием case. Интересно, что в противостоянии «C-шного» и «паскального» операторов выбора окончательного решения так и не принято. В языке Java оператор switch полностью копирует вариант из C, тогда как в языке C#, созданном на основе синтаксиса C, switch пишется как в C, но имеет полностью «паскальную» семантику, то есть оператор break после каждого варианта писать строго обязательно.

Все эти особенности и странности оператора switch/case привели к тому, что в наиболее популярных сегодня скриптовых языках Lua и Python этого оператора вообще нет.

Оператор повторения

Третья фундаментальная конструкция обычно записывается как while и вызывает повторение одного и того же фрагмента программы.

```
x = 0;
while (x < 3)
{
   printf("x = %d\n",x);
   x = x+1;
}
```

Пока условие, написанное после while, остается истинным, участок программы повторяется снова и снова. Можно предположить, что условие продолжения цикла должно вообще-то изменяться в теле цикла. Но этого недостаточно, поскольку есть очевидные ситуации (увеличиваем x на четном шаге и уменьшаем на нечетном), когда цикл не завершается. Более того, существуют программы, в которых условие внутри цикла изменяется, но мы не знаем, завершится ли цикл. Примером может служить так называемая проблема Улама:

```
while n > 1 do
begin
   if odd(n) then
     n := 3*n + 1
   else
     n := n div 2;
end;
```

До сих пор науке неизвестно, завершится ли этот цикл при произвольном n.

Таким образом, оператор повторения является хорошим средством зацикливания программ и создания ошибок. Почти во всех языках существует специальная форма этого оператора под названием for. Она собирает в одном месте: первоначальное присваивание значения, проверку условия окончания, изменение переменной.

```
for (x = 0; x < 3; x = x+1)
{
   printf("x = %d\n",x);
}
```

В некоторых языках, например, в Pascal, оператор for организован так, чтобы можно было проверить, завершится ли он.

```
for x := 0 to 2 do
begin
   writeln('x = ',x);
end;
```

Поскольку начальное и конечное значения x известны, и на каждом шаге x изменяется на 1, то исход выполнения оператора можно предсказать. Однако, в языке C это полезное начинание полностью истреблено. Оператор for в языке C это просто механическое соединение трех выражений - для начала, окончания и изменения значения. В них могут использоваться произвольные переменные, и любое из них (и все три вместе) может отсутствовать. Другие языки, напротив, предлагают еще более экстремальную форму этого оператора

Ruby:

```
for i in 0..5
...
end
```

Python:

```
for n in range(1, 5):
...
```

Цель - подчеркнуть тот факт, что i будет последовательно принимать все целые значения из диапазона от 1 до 5 включительно, и никакие другие. Существует также две ситуации, когда `while` и `for` одинаково неудобны. Первая - это довольно странное с точки зрения чистой теории намеренное зацикливание программы. Это довольно часто встречается в жизни, поскольку существуют программы,

которые действительно никогда не должны заканчиваться. К таким программам относятся, например, программы управления различными объектами вроде электростанций, доменных печей или кофеварок. Остановка управляющей программы в этих случаях производится путем выключения компьютера. Вторая ситуация когда for и while неудобны возникает когда существует несколько независимых условий завершения цикла, причем проверка их осуществляется в разных местах этого цикла. Обычно в таких случаях пишут что-то вроде

`for(;;) ...` или

`while true do ...`

Но в некоторых языках есть специальная конструкция для этой цели: Modula-2

```
LOOP
...
  EXIT;
...
END;
```

Оператор EXIT используется в том случае, когда нужно все-таки прервать выполнение цикла. Следующим выполнится тот оператор, который стоит после END. Удивительно, но оператор бесконечного цикла менее популярен среди создателей языков программирования, чем цикл с проверкой в конце:

Pascal

```
x:=0;
repeat
  x:=x+1;
until x>=3
```

Обратите внимание, что условие окончания цикла не только переехало в конец, но и изменилось на противоположное. Теперь оно читается как «до тех пор, пока не станет x >= 3». В языке C есть похожий оператор.

```
x:=0;
do{
  x:=x+1;
}while(x<3);
```

Здесь смысл условия не изменяется. Зачем так сделали? Видимо, автор языка Pascal пытался приблизится к английскому языку и сказать что-то вроде ”repeat something until condition is satisfied”. Создатели же C смотрели на проблему с точки зрения программиста, которому удобнее, чтобы операторы были более единообразны. Вполне естественно, что оператор `break` (в других языках он

может называться `exit` или `leave`) можно использовать также внутри циклов `while`, `for` и `repeat...until`. Это удобно, например, при поиске в массиве.

Любой из операторов `for`, `loop` и `repeat...until` можно заменить оператором `while`. В случае `for` достаточно «разобрать» оператор на три части и вставить их в соответствующие места оператора `while`. Про бесконечный цикл мы уже говорили. Как превратить `repeat...until` в `while`, догадайтесь сами.

Гораздо интереснее тот факт, что любой оператор `if` тоже можно выразить через `while`. Делается это так:

```
if ( a == 2) printf (“yes”);
```

заменяется на

```
bool p = true;
while ( a == 2 && p )
{
  printf (“yes”);
  p = false;
}
```

Таким образом, все разнообразие операторов управления пока что свелось к одному единственному while. Зачем же их так много? Для удобства программиста, вспомните, что говорилось про «syntax sugar».

Оператор for, помимо его использования для организации предсказуемых повторений, имеет довольно тесную внутреннюю связь с массивами. Об этом мы поговорим позже.

Оператор GOTO, великий и ужасный

Кроме `while`, `if`, `repeat..until`, `case`, `unless` в арсенале операторов управления имеется еще один. Он называется `goto` и вызвал, наверное, самую продолжительную и ожесточенную дискуссию среди пользователей и авторов языков программирования. Проблеме goto были посвящены многие статьи, она обсуждается уже не один десяток лет, но решения пока что нет. Формулируется эта знаменитая проблема следующим образом: «если оператор goto настолько плох, может быть его совсем убрать?».

Этот ужасный оператор продолжает выполнение программы с нового места, отмеченного специальной меткой.

```
if(x > 10) goto l;
...
// 1000 строк кода
...
l: printf("x > 10")
```

То, что goto действительно плох, возражений в общем-то не вызывает. Действительно, лучшим способом безнадежно запутать любой код является неумеренное использование этого оператора. Почему же от goto так трудно отказаться? Причина проста. Рассмотрим часто встречающийся в жизни алгоритм поиска:

```
var
    a:array[1..10] of integer;
    i:integer;
    label 1;
begin
 // заполняем массив a
...
 // ищем в нем 0
  for i:=1 to 10 do
    if a[i] = 0 then goto 1; // ищем элемент, равный 0
  write('не ');
  1:  writeln('найдено');
end.
```

Аналогичная проблема возникает при необходимости преждевременно завершить сразу несколько вложенных циклов, например, при поиске в двумерном массиве. И вариант с goto в этом случае читается проще, чем альтернативы.

Еще одна ситуация, когда goto упрощает чтение программы это обработка аварийных ситуаций. Сравните два варианта одной и той же программы:

Pascal:

```
if x>=0 then
begin
  y := sqrt(x) + f(x);
  if y > 0 then
  begin
    z := 1/y;
    writeln (z);
  end
  else
    writeln ('error');
end
else
  writeln ('error');
```

и

```
if x<0 then goto 1;
y := sqrt(x) + f(x);
if y = 0 then goto 1;
```

```
z := 1/y;
writeln (z);
 ...
1: writeln ('error');
```

Программисты привыкли к goto, и авторы языков решают эту проблему сразу в двух направлениях. Обычно в языке оставляют оператор goto и в дополнение к этому для каждого случая, когда goto может быть полезным, вводят специальный оператор. Один из таких операторов мы уже видели - это break, завершающий цикл. Вместо «goto к началу цикла» в язык вводят оператор continue, а вместо «goto к концу процедуры» - оператор return. В языке Pascal исходно не было ни одного из этих «суррогатных» goto, а в языке C и в поздних вариантах Pascal они есть почти все.

Две задачи, которые не решаются при помощи суррогатных goto это выход из вложенных циклов и анализ причины завершения цикла. Решение первой проблемы предлагается в языке Java:

Java:

```
outerloop:  // метка цикла
  for (i=1; i <= 6; i++)
  {
    for (j=1; j <= 6; j++)
    {
        if (next_value(i,j) == 0) break outerloop;
    }
  }
```

Как видно из примера, здесь вводится метка с именем outerloop, но она обозначает не место в программе, куда надо перейти, а целиком весь цикл, из которого надо выйти. Частичное решение второй проблемы предлагается в языке Python:

Python:

```
for j in range(1, 10):
  if f(i,j) == 0:
    break
else:
  print('не найдено')
```

Здесь часть else относится не к if, а к for и выполняется в том случае когда цикл завершен нормальным путем, а не посредством break. Так что же, goto это объективное зло? Нет, это просто инструмент. Но инструмент, об который слишком легко порезаться. Вот несколько правил техники безопасности, соблюдая которые, можно писать вполне читаемый код с использованием goto:

1. Оператор goto применяется в двух случаях: для выхода из цикла (короткий goto вперед) и для создания бесконечного цикла (длинный goto назад).

2. В случае выхода из цикла оператор goto должен находиться не далее, чем в 20 строках (то есть на той же странице) от метки, на которую делается переход.
3. В случае зацикливания программы обязателен комментарий как к самому goto, так и к метке, на которую делается переход.

Надо отметить, что со стороны компьютера, который выполняет код, никаких проблем с goto нет. Все проблемы этого оператора связаны исключительно с программистом и с его способностью читать и понимать код. Наверное, наиболее серьезные проблемы оператор goto вызывает в тех языках, в которых без него обойтись невозможно, и прежде всего, в языке Basic. Если в Pascal или C наличие метки в коде само по себе говорит нам о том, что где-то в программе *возможно будет* относящийся к ней `goto`, то в языке Basic метка стоит у каждой строки кода.

Из только что сказанного следует еще одна причина для того, чтобы все-таки не убирать `goto` из языка: автоматическая генерация кода. Есть много программ, задачей которых является автоматическое создание фрагментов кода по каким-то спецификациям. Среди таких программ есть генераторы компиляторов, генераторы стандартных интерфейсов, генераторы конечных автоматов, и другие. Код, производимый ими, не предназначен для чтения человеком, так что оператор `goto` может использоваться в них без ограничений.

Ожесточенная дискуссия по поводу `goto` началась со статьи Э.Дейкстры, озаглавленной "Go To Statement Considered Harmful" и написанной в 1968 году. Эта дискуссия в какой-то момент привела к появлению статьи Френка Рубина ""Go To Statement Considered Harmful" considered harmful" (обратите внимание на вложенные кавычки).

Окончательной победой `goto` над академическими взглядами можно считать тот факт, что в реализации языка Modula-2, вышедшей в 1989 году, имелся этот оператор, хотя при разработке этого языка его там не предполагалось.

Функции и процедуры

Написав уже первые несколько программ, люди стали думать над тем, как этот процесс упростить. Исторически первым способом использования уже написанного кода стала организация подпрограмм или, иначе, процедур. В языке Fortran уже есть возможность описывать процедуры и функции. Быстро выяснилось, что кроме первоначального назначения их как способа не писать один и тот же код дважды, подпрограммы служат мощным средством структурирования программы и организации процесса программирования.

Действительно, можно легко разделить большую программу на части, если только описать, как именно будут взаимодействовать между собой процедуры. Можно даже поручить написание процедур разным людям. Поэтому самой сложной частью аппарата функций и процедур в языках программирования всегда была передача параметров и организация общих переменных. Те параметры, что написаны в тексте процедуры, называются формальными, а те, которые пишутся в вызывающей программе - фактическими. При вызове процедуры или функции формальным параметрам ставятся в соответствие фактические.

Вызов процедуры чем-то похож на оператор GOTO, только при вызове процедуры каким-то образом запоминается место, откуда она была вызвана, а по окончании выполнения происходит возврат (то есть еще один GOTO) на запомненное место вызова. Другая интерпретация семантики вызова

процедуры состоит в том, чтобы просто скопировать ее код в место вызова и подставить вместо формальных параметров фактические.

Некая терминологическая проблема состоит в том, что в разных языках процедуры и функции называются разными словами. В C все они называются функциями, в Algol, наоборот, процедурами, в Fortran используется слово «подпрограмма» (subroutine) итп. Не обращая особого внимания на принятую терминологию, мы рассмотрим два концептуальных класса объектов, которые будем называть «процедуры» и «функции», хотя последовательно разница между ними проводится в очень немногих языках, среди которых можно назвать Ada. Разница состоит не в написании, а в назначении: функции предназначены только для того, чтобы вычислять значение, а процедуры чтобы сделать что-то полезное. Все вместе процедуры и функции будем называть подпрограммами, имея в виду, что их основная цель - как-то обособить участок программы.

Макроподстановки

Программист должен быть ленив, это заставляет его больше думать и меньше делать и в итоге приводит к более качественному коду. Разумеется, никто не хочет писать один и тот же кусок программы несколько раз. Первый способ, который находит для себя начинающий программист, называется «китайский метод повторного использования кода copy/paste» - простое копирование кусков программы. Но даже этот способ ленивые программисты сумели автоматизировать. Макроподстановка или просто макро позволяет выделить какой-то текст, дать ему имя и далее копировать сколько угодно, причем средствами языка а не текстового редактора. При правильном использовании это дает более читаемый текст.

```
#include <stdio.h>
#define sayit printf(“I will always use Google before asking dumb
questions\n”)

main()
{
  sayit;
  sayit;
  sayit;
}
```

В макроопределения можно вводить параметры, и они будут подставлены в текст программы, обычно без какой-либо проверки. Макроподстановка - опасный инструмент, с его помощью можно легко запутать любую программу.

```
#define int float
#define float int
...
```

Исторически макроподстановки впервые появились в ассемблере, поскольку этот язык больше других располагает к созданию длинных текстов, состоящих их повторяющихся (почти) одинаковых фрагментов.

Но это еще не все. Некоторые языки программирования используют макроопределения и макроподстановки в качестве основного инструмента и главной идеи, на которой строится весь язык. Из этих языков наиболее известен **TRAC**.

Более последовательно идея макроподстановок проводится в более современных языках, например, в **nim**. Там для простых условных макроподстановок используется оператор `when`, почти идентичный оператору `if`:

```
when system.hostOS == "windows":
  echo "running on Windows!"
elif system.hostOS == "linux":
  echo "running on Linux!"
elif system.hostOS == "macosx":
  echo "running on Mac OS X!"
else:
  echo "unknown operating system"
```

Отличие только в том, что соответствующая строка будет не выполнена, а скомпилирована, и проверок во время выполнения произведено не будет. Если константа system.hostOS окажется равна "linux", то весь приведенный фрагмент будет в точности эквивалентен такому:

```
  echo "running on Linux!"
```

Передача параметров

Передать параметры в процедуру или функцию можно многими разными способами. Единственным в своем роде исключением тут можно считать некоторые ранние диалекты языка Basic, в которых передачи параметров в процедуры нет вообще. Хотите передать значение - кладите его в переменную. Во всех остальных распространенных языках используются следующие способы передачи параметров:

Передача значений

Процедуре передаются копии значений параметров. Соответственно, изменить какие-то переменные вызывающей программы процедура не может.

```
procedure test(x:integer);
begin
  writeln(x);
  x:=x+1;
  writeln(x);
end;

var  xx:integer;
begin
  xx:=1;
```

```
  test(xx);  (* xx не изменяется после выполнения процедуры *)
  writeln(xx);
end.
```

Заметим, что в языке C этот способ является единственным.

Передача ссылок

```
procedure test(var x:integer);
begin
  x:=x+1;
end;

var  xx:integer;
begin
  x := 3;
  test(xx);  (* xx после выполнения процедуры увеличится на 1 *)
end.
```

Отличие состоит в том, что в процедуру передается ссылка на переменную, и переменная x в процедуре и xx в вызывающей программе относятся к одной и той же области памяти. Этот способ передачи параметров является единственно возможным в языке Fortran. Возникает естественный вопрос: Что, если в качестве параметра будет передано выражение? Константа? Можно выражения в качестве параметров-ссылок вообще запретить. Так и сделано в языке Pascal. Но что делать, если других способов передачи параметров вообще нет в языке? Разработчики Fortran нашли весьма своеобразное решение этой проблемы. Если в качестве параметра указана переменная, то процедуре передается ссылка на эту переменную. А если там выражение, то создается специальная безымянная переменная, ссылка на которую передается процедуре. Все изменения этой переменной, произведенные процедурой, теряются. С константой все интереснее. Некоторые компиляторы Fortran не создавали копию константы, а передавали ссылку на место хранения самой константы, в надежде, что программист знает, что делает и не будет менять значение в процедуре. Таким образом Fortran давал возможность изменить значение константы. Что, разумеется, приводило к непредсказуемым последствиям.

Передача по имени.

Рассмотренные два способа передачи параметров являются основными. Но был еще один странный способ в языке Algol, который интересно рассмотреть. Он заключался в том, что переданное в процедуру выражение вычислялось каждый раз так, как если бы вместо формальных параметров были подставлены их значения, вычисленные в контексте вызывающей программы. По сути это нечто среднее между макроподстановкой и вызовом подпрограммы. Разобраться с этим способом нам поможет пример, называемый «прием Йенсена» по имени изобретателя.

```
begin

  procedure p (a, b);
    name a, b; integer a, b;
```

```
  begin
    for a:=1 step 1 until 10 do
      b := 0
  end p;

  integer i; integer array s [1:10];
  p (i, s[i])
end
```

При выполнении процедуры p на каждом шаге цикла вместо a подставляется i, а вместо b подставляется s[i], и эта процедура просто обнуляет массив s. Проблем с таким способом передачи параметров очень много, а из преимуществ можно назвать только прием Йенсена. Поэтому передача параметров по имени так и осталась курьезом из языка Algol.

Передача параметров по имени это нечто среднее между макроподстановкой и передачей функций как параметров. В современных языках программирования обычно используют передачу функций для достижения тех же целей. Вот реализация приема Йенсена на Javascript

```
  function p (f){
    var i;
    for (i=0; i<10; ++i)
      f(i,0)
   }

  s = Array(10);
  p( function(i,v){ s[i] = v;} );
```

Передача невычисленных выражений

(отложенное, или ленивое вычисление).

Рассмотрим пример на языке C

```
void f1(int a)
{
   if (a) printf("yes");
}

void f2(int a, int b)
{
   if (a || b) printf("yes");
}


main()
{
  int x = 1, y=2, z=0;
  f1((x==1) || (y/z == 1)); // работает
```

```
  f2((x==1),(y/z == 1));    // не работает, возникает деление на 0
}
```

Происходит это потому, что если в выражении `(x==1) || (y/z == 1)` первое условие выполнено, то дальше считать не имеет смысла, на результат это не повлияет. Но сделать то же самое внутри процедуры мы не можем, потому что значения параметров должны быть вычислены до вызова процедуры. Заметим, что в случае передачи параметров по имени оба способа будут работать.

В некоторых языках программирования существует еще один способ передачи параметров, который называется отложенными или «ленивыми» (lazy) вычислениями. Он похож на передачу параметра по имени, и отличается тем, что вычисление выражения происходит один раз, но только тогда, когда это значение действительно понадобится. В таких языках оба примера будут работать, если только параметры процедур объявлены как «ленивые».

Функции

Теперь поговорим о функциях. Цель функции — получить значение, в отличие от процедуры, цель которой — произвести какое-то действие. Если функция все-таки изменяет среду исполнения, например, выводит какое-то число на дисплей, это называется «побочным эффектом». К проблеме побочных эффектов в императивных языках есть два диаметрально противоположных подхода. В языке Ada побочные эффекты функций просто запрещены. Смысл в этом есть, ведь в выражении, подобном `(x==1) || f(y)` , функция `f(y)` может так и остаться никогда не вычисленной. Если она производит какое-то действие кроме вычисления значения, это действие так и не произойдет. Это может привести к довольно сложно обнаруживаемым ошибкам. С другой стороны, в языке С вообще нет разницы между функциями и процедурами. В С и процедуры и функции формально считаются функциями, причем процедуры возвращают специальное значение, принадлежащее к типу void, которое символизирует отсутствие результата. А тот факт, что в вышеприведенном выражении функция f(y) при x равном 1 не будет вычислена, считается дополнительным удобством для программиста. Наследники языка С идут в этом отношении еще дальше. Стандартный способ аварийного прекращения программы в языках PHP и PERL такой:

```
($x == 1) or die (“X не равен 1, аварийное завершение программы“)
```

Функция die выводит сообщение и завершает выполнение программы. Формально это выражение, состоящее из логического условия и вызова функции. Куда при этом девается результат? Как и в языке С - выбрасывается за ненадобностью.

Рекурсия

В математике часто используется определение функций через самих себя, рекуррентно. Например, факториал числа можно определить так:

N! это 1 если N=1 и N * (N-1)! в противном случае.

Запишем это определение на языке Python

```
```

```
def fact(N):
  if N == 1:
    return 1
  else:
    return N * fact(N-1)

print(fact(6))
```

В некоторых ранних языках программирования рекурсия была невозможна или затруднена. Обсуждалось даже использование специального ключевого слова recursive в определении функции для того, чтобы сообщитько компилятору, что функция будет вызывать саму себя. Д. Баррон в своей книге резонно заметил, что лучше было бы помечать некоторые функции как не-рекурсивные, поскольку в этом случае компилятор сможет сгенерировать более оптимальный код. В наши дни проблем с вызовом рекурсивных функций вроде бы нет, все функции могут вызывать сами себя.

Однако, некоторые проблемы остаются. Например, в функциональном языке Ocaml при определении рекурсивных функций следует использовать специальное ключевое слово rec. Зачем? Затем, что в OCaml определение функции это просто присваивание, и в момент обработки его правой части компилятор еще "не знает" имени функции, то есть имени той переменной, которой будет присвоена определяемая функция.

```
let rec fact n =
    if n == 1 then 1 else n * fact (n-1);;
```

Слово **rec** неким магическим способом решает эту проблему.

Очевидным недостатком прямого рекурсивного определения функций является необходимость хранить промежуточные результаты вычислений, причем иногда их бывает много. При вычислении `fact(6)` по приведенному выше алгоритму будут последовательно запомнены числа 6,5,4,3,2. И только после получения от вызова fact(1) значения 1 все они будут перемножены. Для решения этой проблемы в некоторых языках программирования, как правило, функциональных, применяется так называемое *устранение хвостовой рекурсии*. С формальной точки зрения рекурсивное определение функции состоит из двух частей, базы и шага. Базовая часть это обычно константа, а шаг представляет собой приближение к базовой части. В случае факториала база это fact(1), равный 1, а шаг это вычисление fact(n) через fact(n-1).

Если рекурсивный вызов в шаге рекурсии является в точности повторением вызова самой функции, то есть при этом не запоминается никаких промежуточных значений, то процесс рекурсивного вычисления функции становится намного проще. Покажем это на примере

```
def fact(N):

  def fact1(N,A):
    if N == 1:
      return A
    else:
```

```
        return fact1(N-1,A*N)

    return fact1(N,1)

print(fact(6))
```

Здесь использована внутренняя функция, вычисляющая факториал числа, и внешняя "обертка" для нее. Идея состоит в том, что мы говорим, что fact(6) это то же самое, что fact1(6,1), а это то же самое, что fact1(5,6) == fact1(4,30) == fact1(3,120) == fact1(2,360) == fact1(1,720) == 720. Определения базы и шага становятся менее ясными, но зато компилятор может сгенерировать намного более быстрый код. Такое описание рекурсивной функции из двух частей является стандартным приемом во многих функциональных языках, а если нам почему-то нужно написать программу, которая никогда не завершается (бесконечный цикл), то устранение хвостовой рекурсии является обязательным.

База рекурсии не всегда состоит из одного случая, их может быть два и больше. Рассмотрим, например, рекурсивное определение того, что такое палиндром (строка, которая читается одинаково от начала к концу и от конца к началу).

```
def palindrom(N):
    if len(N) == 0:     # пустая строка является палиндромом
        return True
    if len(N) == 1:     # строка длиной в 1 символ тоже является палиндромом
        return True
    if N[:1] != N[-1:]: # если первый и последний символы не совпадают
        return False    # то это не палиндром
    return palindrom(N[1:-1])

print(palindrom("adda"))
print(palindrom("ada"))
print(palindrom("axddaa"))
```

Здесь два базовых случая, для строк длиной 1 символ и 2 символа. Еще одна проблема с рекурсией состоит в формулировании базы. Посмотрим еще раз на функции, вычисляющие факториал числа. Если любую из них вызвать с отрицательным N, программа зациклится. Поэтому для отрицательных чисел нужно отдельно задавать базовый случай, который должен либо генерировать ошибку, либо возвращать какое-то специальноо говоренное число (например, 0).

Запись результата функции

В разных языках существуют различные способы сообщить, какое именно значение должна вернуть функция. Самый простой способ состоит в том, чтобы ввести специальный оператор возврата значения - return.

```
int sqr(int x)
{
    return x*x;
}
```

Другой довольно распространенный способ состоит в том, чтобы положить требуемое значение в специальную предопределенную переменную с именем, совпадающим с именем функции и имеющая тот тип, который функция должна вернуть.

```
function sqr(x:integer):integer;
begin
  sqr:=x * x;
end;
```

С виду все просто, но на практике такая запись оказывается очень неудобной. Если программист решит переименовать какую-то сложную функцию, то ему придется переименовывать и каждое вхождение переменной-результата. Более серьезная проблема состоит в том, может ли эта переменная появляться справа от знака присваивания? С одной стороны, это обычная переменная. С другой - в языке Pascal, где используется именно этот метод, вызов функции без параметров ничем не отличается от обращения к переменной.

```
var q:integer;

function interesting:integer;
begin
   interesting:=q;
   if interesting = 0 then
   begin
    q:=100;
    interesting := interesting+1;
   end;
   q := q - 1;
end;
```

Появляется неоднозначность: interesting в операторе if и справа от знака присваивания может означать либо рекурсивный вызов этой же функции, либо переменную-результат. Заметим, что этот странный пример будет работать в любом случае, хотя и будет возвращать разные числа.

В результате в языке Pascal появление переменной-результата в том контексте, где требуется значение, запрещено. Несмотря на неудобство, этот способ по историческим причинам оказался широко распространен. Со временем сложилась программистская практика, устраняющая основные недостатки переменных-результатов. Она состоит в том, чтобы самому объявить еще одну вспомогательную переменную, вычислить результат и в конце функции переложить его в переменную-результат.

```
function sqsum(a,b:integer):integer; // (a+b)^2
var res:integer;
begin
  res := a+b;
  res := res * res;
```

```
  hyp := res ;
end;
```

Метод оказался настолько удачным, что в языке Object Pascal (Delphi) ввели специальную автоматически объявляемую переменную с именем result.

Глобальные переменные

Использование подпрограмм вызвало еще одну проблему. У каждой функции или процедуры могут быть локальные переменные, которые, кроме нее самой, не доступны никому. Но очень часто нужно иметь доступ к каким-то переменным из нескольких процедур, или же сохранять значение какой-то переменной между последовательными вызовами функции. Самый распространенный пример - генератор последовательности чисел.

```
var seed:integer;

function nextval:integer;
begin
   seed:=seed+1;
   nextval := seed;
end ;

procedure setseed;
begin
  seed:=0;
end ;
```

Обратите внимание, что в данном примере присутствуют два варианта использования глобальных переменных. Функция **nextval** и процедура **setseed** обращаются к одной и той же глобальной переменной **seed**, а последовательные вызовы **nextval** используют тот факт, что переменная **seed** является внешней по отношению к этой функции и сохраняет присвоенное значение. Таким образом, один и тот же механизм глобальных переменных решает обе поставленные задачи.

Но так было не всегда. По какой-то непонятной причине разработчики языков **Fortran** и **Algol** считали, что процедура или функция не должна «видеть» ничего, находящегося за ее пределами. Поэтому упомянутые задачи решали по отдельности. В языке Algol было введено специальное ключевое слово own, означавшее, что значение объявленной таким способом переменной сохраняется между вызовами подпрограммы. На **Algol** функция nextval выглядела бы так:

```
real procedure nextval;
begin
   own integer seed;
   seed:=seed+1;
   nextval:=seed;
end;
```

Тут все вроде бы хорошо, но непонятно, как этому seed присвоить начальное значение. В языке Fortran, наоборот, основные усилия сосредоточили на решении второй задачи, сделав специальное ключевое слово COMMON.

```
      SUBROUTINE SETSEED
      INTEGER SEED
      COMMON /GLOBALSEED/ SEED
      SEED = 0
      END

      FUNCTION NEXTVAL()
      INTEGER NEXTVAL
      INTEGER SEED
      COMMON /GLOBALSEED/ SEED
      SEED = SEED+1
      NEXTVAL = SEED
      END
```

В отличие от Pascal и C, в Fortran каждая подпрограмма содержит свое собственное описание переменной SEED. Если эти описания будут разными, результат окажется непредсказуемым. Такое странное решение было принято с целью обеспечить возможность независимой компиляции подпрограмм.

Раздельная компиляция

Основная идея, лежащая в основе раздельной компиляции такова: сделать участки кода как можно более независимыми друг от друга, компилировать каждый из них только один раз и хранить где-то полученный машинный код. Это позволяло сэкономить время работы компилятора, которое было весьма дорогим на старых компьютерах. Если какая-то подпрограмма работает правильно, нет необходимости компилировать ее каждый раз заново, достаточно знать, как ее вызвать. Такие подпрограммы можно объединять в библиотеки, и вызывать по мере необходимости. На реализацию такой функциональности обращали особое внимание при создании языка Fortran, и тогда же сформировались три главные проблемы раздельной компиляции:

1. Программист должен знать, как вызывать функцию.
2. Хотелось бы, чтобы компилятор имел возможность проверить, правильно ли функция была вызвана.
3. Что делать, если нужно изменить библиотечную функцию, а ее уже кто-то использует?

Надо сказать, что ранние языки не предоставляли вообще никаких средств для решения этих проблем. Предполагалось, что для программиста будет написана некая инструкция, описывающая правильную работу с библиотекой. А программист все сделает правильно и в соответствии с инструкцией. Даже в языке C есть возможность вызвать любую подпрограмму сколь угодно неправильным способом. Например:

```
/* #include <stdio.h> */
main()
```

```
{
  int x = printf(0.5);
}
```

Формально компилятор должен выдать предупреждение о том, что тут происходит нечто не совсем правильное.

Основная идея решения проблемы проверки правильности вызовов состоит в том, чтобы создать какой-то отдельный файл с описаниями правил вызова библиотечных функций, но без их содержания. Компиляция этого файла тоже понадобится, но займет намного меньше времени. В языке C для этого служат так называемые include-файлы. В языке **Modula-2** они называются def-файлы. В **Turbo Pascal** файл с библиотечными подпрограммами делится на секции **interface** и **implementation**. Иметь один файл с описаниями намного лучше, чем дублировать описания в каждой подпрограмме. Одновременно получается документ, которым может пользоваться и программист. Разумеется, без комментариев такой файл имеет ограниченную полезность, но за много лет стало совершенно ясно, что задача заставить программиста комментировать код в процессе его написания находится где-то между «очень сложно» и «практически невозможно». Если в случае **Fortran** от программиста требовался довольно пространный текст, то в случае **C** или **Pascal** отделить описание функции, которое нужно компилятору для правильного ее вызова - дело простое, и комментарии требуются в меньшем объёме. В более поздних языках, например, в **C++**, вызов функции, не имеющей описания, вообще невозможен (что не мешает сделать неправильное описание).

Что касается третьей проблемы - несовместимости версий - то, несмотря на многочисленные попытки ее решить, она существует и сегодня.

Запись вызова функций и подпрограмм

Вопрос о том, как именно синтаксически оформить вызов подпрограмм, на первый взгляд кажется малосущественным. Однако, в языке Fortran использовалось специальное ключевое слово для вызова процедур.

```
CALL f(x)
```

В дальнейшем от использования ключевого слова отказались, и стали писать просто

```
f(x);
```

В записи же функций долгое время особого разнообразия не наблюдалось, поскольку запись была заимствована из языка математики. Некоторое разнообразие наблюдалось только в отношении функций и процедур, не имеющих аргументов. Вопрос стоял так: писать ли пустые скобки? В Pascal скобки писать не нужно, и это решение имеет два важных последствия, одно хорошее и одно плохое. Во-первых, можно писать программы, похожие на текст на английском языке:

```
begin
initialize_variables;
open_files;
perform_calculations;
write_results;
```

```
close_files;
end.
```

Про такие программы говорят, что они обладают свойством самодокументируемости. Про второе последствие мы уже говорили: функция без параметров синтаксически неотличима от простой переменной.

Функции с переменным числом аргументов

Тенденция к переносу средств ввода-вывода из языка в библиотеки, о которой мы будем говорить позже, привела к необходимости появления процедур с переменным числом аргументов (и функций тоже, но они нужны реже). Одна из первых попыток была предпринята в языке Pascal. Там есть процедуры с переменным числом аргументов, но это были «специальные» процедуры, по существу, операторы языка, лишь внешне напоминавшие обычные. Способа создать свою процедуру с переменным числом аргументов в классическом Pascal нет. Надо заметить, что на уровне машинных кодов традиционно рассматривают два способа передачи параметров в процедуру или функцию: "паскальный" и "сишный". Все современные компиляторы этих языков умеют использовать оба способа вызова, однако в каждом языке по умолчанию используется "свой" способ вызова. При использовании "паскального" способа вызова подпрограммы реализация функций с переменным числом параметров существенно затруднена.

В языке С процедуры с переменным числом аргументов уже можно создавать, и имеется стандартный механизм, позволяющий эти аргументы получать. Для этого нужно или иметь возможность определить число аргументов, анализируя первые несколько переданных значений, или принять соглашение о том, что какое-то определенное значение будет являться признаком окончания списка аргументов.

Таким образом, в С могут быть два вида вызовов процедуры с переменным числом аргументов:

`f(n, a1, a2 ... an);` и `f(a1, a2 ... an, NN);`

где NN - константа, которая не может встретиться среди значений аргументов, обычно это 0. Способа узнать, сколько параметров было в действительности передано, в С не существует, и программист может легко создавать трудноуловимые ошибки. Вот как выглядит описание функции с переменным числом аргументов:

```
int add_them_all( int arg_count, ... )
{
    va_list ap;
    int sum = 0 ;
    va_start( ap, arg_count );
    for(; arg_count > 0; --arg_count )
      sum += va_arg( ap, int );
    va_end( ap ) ;
    return sum ;
}
```

Еще один способ сделать то же самое использован в языках **Python** и **Javascript**. Фактически переданные параметры попадают в специальный массив, который в **Javascript** называется **arguments**, а в Python имя ему присваивает программист.

```
function sum()
{
   var s = 0;
   for(i=0; i<arguments.length; ++i)
      s = s+arguments[i];
   return s;
}

console.log(sum(1,2,3)); // напечатает 6
```

```
def sum(*arg):
  sum = 0
  for i in arg:
    sum = sum+i
  return sum

print(sum(1,2,3,4)) # напечатает 10
```

Существует и другой способ организовать подпрограмму в переменным числом аргументов. Для этого достаточно часть аргументов объявить необязательными. Самый, опять же, простой способ это сделать - присвоить им начальные значения. Если считать, что формальные аргументы сопоставляются фактическим слева направо, и необязательными объявлены один или несколько наиболее правых из них, то разночтений не возникает.

```
int func(int x, int y=0, int z=0);

func(1,2) // x=1, y=2, z=0
func(5) // x=5, y=0, z=0
```

Объекты первого класса

В описаниях языков программирования часто встречается термин `first class citizens` или `first class objects`. Это устоявшийся термин для обозначения того, как могут быть использованы те или иные сущности языка. Сущности "первого класса" имеют право появляться в следующих контекстах:

1. Они могут быть параметрами функций и процедур
2. Они могут быть результатами функций
3. Их можно присваивать переменным

В ранних языках программирования функции и процедуры не относились в полной мере к первому классу, но постепенно получали все больше "прав".

Ссылки на функции

Идея присвоить ссылку на функцию какой-нибудь переменной кажется вполне естественной. Функция, как и все, что есть в памяти компьютера, имеет адрес, этот адрес никто не мешает запомнить в какой-то переменной. Но в программах, написанных начинающими программистами, такие переменные встречаются на удивление редко. Видимо, это происходит вследствие того, что синтаксически функции выглядят совершенно иначе, чем данные. Поэтому вместо

```
void map(double *x, int length, double (*fun)(double))
{
  int i;
  for(i=0;i<length;i++)
    x[i] = fun(x[i]);
}
```

часто пишут что-то вроде

```
typedef enum { FN_SIN, FN_COS }  FUNCTION;
void mapnf(double *x, int length, FUNCTION f)
{
  int i;
  for(i=0;i<length;i++)
    switch(f)
    {
    case FN_SIN:
      x[i] = sin(x[i]); break;
    case FN_COS:
      x[i] = cos(x[i]); break;
    }
}
```

Недостатки второго способа очевидны: для добавления еще одной функции придется переписывать код. Тем не менее, его разновидности встречаются чаще, чем хотелось бы.

Типы результата

Традиционно в школьной математике считается, что функция возвращает число. Однако, в других разделах математики функция, в качестве результата имеющая матрицу или даже другую функцию, не являются чем-то необычным. Такую возможность довольно быстро включили и в языки программирования. Единственной сложностью, на первых порах замедлявшей введение подобных функций в языки, было опасение, что это приведет к низкой эффективности программ. Действительно, передача большого массива вызывает, как минимум, однократное копирование его содержимого. Тем не менее, идея оказалась настолько привлекательной, что возможность возвращать в качестве результата что-то помимо чисел имеется почти во всех современных языках программирования.

Типы функций и карринг

Допустим, у нас есть функция, возвращающая целое число, и имеющая один аргумент целого типа. Поскольку в языках со строгой типизацией «все имеет какой-то тип», логично спросить, какой тип имеет сама функция? В классических императивных языках вроде **C** или **Pascal** существуют типы для указателей на функции. Формально это обстоятельство предполагает существование типов самих функций, но создать переменную типа «функция» в языке **C** или **Pascal** не получится. В некоторых языках функция может сама по себе быть объектом «первого класса», в таких языках функции имеют типы. И на примере этих языков имеет смысл изучить типы, к которым принадлежат различные функции. Возьмем обозначения типов в языке **Haskell**:

Тип `Integer->Float` это вещественная функция целого аргумента. С точки зрения математики эта функция является отображением множества целых чисел Integer на множество вещественных чисел Float. `Float->Float` это функция, которая получает на входе вещественное число и возвращает вещественное число. Далее `Float->Float->Float` функция, которая получает на входе два вещественных числа и возвращает вещественное число. Но этот тип можно интерпретировать и по-другому: функция, которая получает одно число типа Float и возвращает функцию типа `Float->Float`. Действительно, если у функции с двумя аргументами зафиксировать один аргумент, то получится функция одного аргумента. Рассмотрим выражение

```
A + X
```

Если несколько необычным способом поставить скобки, то получится функция «A плюс» от X, прибавляющая A к аргументу:

```
(A +) X
```

Далее можно записать определение функции:

```
F = (A +)
```

где мы просто обозначаем нашу функцию, прибавляющую A, как F. Такой прием называется карринг, и широко используется в функциональных языках. Раз уж мы заговорили про функции как отдельные сущности, то попробуем пойти чуть дальше, и поговорить про

Лямбда

Идея, лежащая в основе одной из самых красивых концепций программирования, очень проста.

```
import math

def my_function(x):
    return math.sin(math.cos(x))

def sum_fun(f,x_min,x_max,step):
    s = 0;
    x = x_min
    while x < x_max:
        s = s+f(x)
        x = x+step
```

```
    return s

print(sum_fun(my_function,0,1,0.1))
```

В этом примере функция с именем **my_function** нужна только для того, чтобы передать ее в другую функцию, и имя ее нам совершенно не важно. Поэтому в языке **Python** есть возможность ее никак не называть:

```
import math

def sum_fun(f,x_min,x_max,step):
    s = 0;
    x = x_min
    while x < x_max:
        s = s+f(x)
        x = x+step
    return s

s = sum_fun( lambda x:math.sin(math.cos(x)), 0, 1, 0.1 )
print(s)
```

Заметим еще одну особенность такой записи: функция описывается там, где используется. Если в первом варианте, увидев запись `print(sum_fun(my_function,0,1,0.1))`, мы должны просмотреть код, чтобы узнать, что же такое my_function, то во втором она сразу видна. Недостаток языка Python, видимо, неустранимый, в том, что такая безымянная функция может состоять только из одного выражения.

В языке Javascript все проще:

```
function sum_fun(f,x_min,x_max,step)
{
  var s = 0;
  var x = x_min;
  while (x < x_max)
  {
    s = s+f(x);
    x = x+step;
  }
  return s;
}

var s = sum_fun(
    function(x){ return Math.sin(Math.cos(x)); },0, 1, 0.1
);
console.log(s);
```

Полученную таким способом безымянную функцию можно передать как параметр, присвоить переменной или вернуть в качестве значения другой функции. Фактически, конечно, при этом передается не сама функция (непонятно, что это такое), а ссылка на нее. Особую роль лямбда-исчисление играет в функциональном программировании.

Функции левой части

Исходя из соображений симметрии, часто хочется наряду с

```
x := min(a,b);
```

писать

```
min(a,b) := x;
```

Смысл этого вполне очевидный - переменной, которая имеет минимальное значение, присвоить значение x. Ссылки в языке C++ дают нам такую возможность.

```
int& min(int& x, int& y)
{
  if (x < y)
    return x;
  else
    return y;
}

...

min(a,b) = 5;
```

Перегруженные (overloaded) подпрограммы

В математике мы сталкиваемся с тем фактом, что, например, корень квадратный из натурального числа это не совсем то же самое, что корень квадратный из вещественного числа, хотя записываются они совершенно одинаково.

В программировании тоже логично было бы вместо преобразования типов использовать различные версии одних и тех же функций для различных типов аргументов. В ранних языках программирования эта задача решалась достаточно прямолинейно: придумывали разные имена для разных функций. В языке Fortran дело даже дошло до неформального соглашения: к названию функции приписывали букву, означающую используемый тип. Так, модуль вещественного числа назывался ABS, целого числа IABS, комплексного числа CABS, итд.

Возложить на компилятор обязанность различать функции по типам аргументов догадались не сразу. Первые попытки состояли в том, что компилятор «заранее знал» про некоторые функции и мог

подставить нужный код. Такие подпрограммы получили название полиморфных, или перегруженных. Программист не мог сам определить такую функцию. Разумеется, арифметические операции были полиморфными изначально, и в языке Algol-68 программисты получили возможность определять полиморфные операторы, но не функции. Причины, по которым введение полиморфных функций было отложено до 1976 года (язык ML), остаются не вполне понятными.

В мир императивных языков полиморфные функции были введены в 1983 году с языком C++. Изобретатель языка Бьерн Страуструп настолько боялся новой идеи, что постарался, как он сказал, «изолировать» полиморфные функции.

```
overload print (int x);
overload print (char *x);
overload print (int n, char *x);
```

Для этого использовалось ключевое слово overload, совершенно ненужное. Если программист определил функцию print с аргументом типа int, то ее так и следует вызывать, независимо от того, есть ли другие функции с тем же именем. И, наконец, непонятно, что будет, если функцию объявить как overload, но не написать никаких альтернатив. Что это - ошибка? А если две такие функции находятся в раздельно компилируемых модулях? В общем, во второй версии языка слово overload было отменено.

Никакой магии тут нет, если представить себе, что компилятор делает то же самое, что делали программисты на Fortran. Фактически при компиляции перегруженных функций создаются "внутренние" имена для них, и в эти имена включаются типы параметров. Например, перечисленные выше функции могут быть названы print@int, print@char и print@int@char. Если такие имена создаются для всех компилируемых функций, то слово overload оказывается не нужным. Здесь используется та особенность, что символ @ может появляться во внутренних именах функций, с которыми работает линкер, но запрещен в именах C++.

Компилятор может различать полиморфные функции по типу параметров, но не по типу результата.

Шаблоны (generics)

Альтернативой полиморфным функциям является концепция универсальных «шаблонных» функций. Рассмотрим ее на примерах из языков C++ и Ada.

```
template< typename T > void TSwap( T& a, T& b )
{
   T c;
   c = a;
   a = b;
   b = c;
}

int main()
{
   int a,b; float c,d;
...
```

```
    TSwap<int>(a,b);   // меняем местами целочисленные переменные
    TSwap<float>(c,d); // меняем местами вещественные переменные
}
```

Ada:

```
generic
  type Element_T is private;  --  Element_T это параметр
procedure Swap (X, Y : in out Element_T);

procedure Swap (X, Y : in out Element_T) is
  Temporary : constant Element_T := X;
begin
  X := Y;
  Y := Temporary;
end Swap;

procedure Swap_Integers is new Swap (Integer); -- создаем процедуру для
целых
procedure Swap_Chars is new Swap (Character); -- для символов

-- Используем функцию
a,b : integer; c,d: character;
Swap_Integers(a,b);
Swap_Chars(c,d);
```

В обоих случаях описывается шаблон - некая общая форма для работы с данными любого типа (или какого-то набора типов). У шаблона присутствует параметр - тип данных, с которым надо работать. При компиляции программы вместо формального типа подставляется фактический и создается новая функция. В языке C++ ее имя состоит из имени шаблона к которому приписано имя типа в угловых скобках; в языке Ada имя каждой функции нужно придумывать отдельно. Каждый такой шаблон эквивалентен целому «семейству» полиморфных функций. Разница между полиморфными функциями и шаблонами состоит в том, полиморфные функции для разных типов могут быть существенно различными, тогда как шаблоны внешне имеют один и тот же код для каждого типа, хотя полученный в результате компиляции машинный код может существенно различаться.

Замыкания

Нравится нам это или нет, но многие функции используют какой-то внешний контекст. Это могут быть глобальные переменные, константы, другие функции. В тех языках, где функции могут быть объектами первого класса, их можно возвращать из других функций в качестве результата и передавать как параметры. Вот как это происходит в языке JavaScript:

```
function differential (f, dx) {
  return function (x) {
    return (f(x + dx) - f(x)) / dx;
  };
}
```

Функция differential возвращает функцию для вычисления отношения заданных малых приращений другой функции к приращению аргумента. Чем это отличается от указателя на функцию? Очень просто: в случае указателя функция должна уже где-то существовать, в случае функции как значения она создается в процессе выполнения программы. Как именно это происходит - отдельный вопрос.

Заметим, что переменная dx, переданная в функцию differential, «замораживается» в возвращаемой безымянной функции. Следующий вызов differential породит другую безымянную функцию, с «вмороженным» в нее другим значением dx. Это еще один пример так называемого контекста, то есть всех сущностей, от которых зависит значение, возвращаемое функцией.

Функция вместе с контекстом в ряде языков представляет собой объект первого класса и носит название замыкания.

Возвращать как значение функцию, не имеющую контекста, технологически намного проще, чем замыкание, но в языке Ada, где такие функции выделены в отдельный класс, они не являются объектами первого класса. Замыкания впервые появились в функциональных языках, потом в языках, задуманных как интерпретируемые, и только в конце первого десятилетия 21 века начали появляться в традиционных языках, таких как C и Pascal.

Пример на Delphi

```
type
  TCounter = reference to function: integer;

function MakeCounter: TCounter;
var
  i: integer;
begin
  i := 0;
  result := function: integer
            begin
              inc(i);
              result := i;
            end;
end;
```

Генераторы

Одно из наиболее распространенных применений функций с внешним контекстом это создание генераторов, то есть функций, возвращающих каждый раз новое значение. Это может быть порядковый номер, псевдослучайное число или что-то еще. В любом случае функция должна как-то запомнить свое состояние и восстановить его при следующем вызове. Если при этом требуется какая-то инициализация, то необходима еще одна функция.

Рассмотрим пример

```
int number;

void init_number(int startvalue)
{
  number = startvalue;
}

int get_next_number()
{
  return number++;
}
```

Таким образом, для простой задачи нам нужны сразу три объекта, причем если генераторов нужно несколько, то задача дополнительно усложняется. Объектно-ориентированное программирование предлагает решение проблемы, состоящее в том, что все три компонента генератора «упаковываются» в целостный объект. Так гораздо удобнее, но проблема трех сущностей никуда не девается. Вот как решается эта задача в языке JavaScript, причем без явного объявления объектов:

```
function next_number(startvalue)
{
 var  number =  startvalue;
 while (true)
 {
    yield  number++;
 }
}

v=next_number(10);
a = v.next();  //  a=10
b = v.next();  //  b=11
```

Оператор yield возвращает специальный вид замыкания - генератор. К этому генератору можно обратиться посредством вызова next() чтобы получить очередное значение. Оператор yield можно рассматривать как разновидность оператора return. Он как бы останавливает вычисление функции и возвращает нечто, по сути своей похожее на замыкание. Вызов next() продолжает выполнение и возвращает очередное вычисленное значение. В языке Python генераторы выглядят более прозрачно:

```
# объявление функции
def countfrom(n):
    while True:
        yield n
        n += 1

# Использование

for i in countfrom(10):
     print(i)
```

На самом деле функция countfrom возвращает объект, но это совершенно не заметно.

То же самое в языке nim:

```
iterator countup(a, b: int): int =
  var res = a
  while res <= b:
    yield res
    inc(res)
```

Массивы и структуры

> *Программирование — это сложно. Основные правила, на которых все строится, очень просты, но по мере разработки программа сама начинает вводить свои правила и законы. Таким образом, программист строит лабиринт, в котором сам же может и потеряться.*
>
> Marijn Haverbeke

Поговорив столь подробно о простых данных, перейдем к составным. Составные типы или контейнеры представляют собой объединение данных. Обычно под составным типом понимают объединение данных, а когда говорят о контейнере, имеют в виду нечто, куда можно положить данные. Разница не вполне очевидна.

Общий принцип тут в том, что под одним общим именем в программе хранится некий набор данных.

Исследуя массивы, структуры, списки, контейнеры, словари и прочие составные типы в разных языках, следует обращать внимание на следующие особенности индексов и данных:

1. существует ли естественный порядок для индексов, то есть можно ли переходить от одного индекса к следующему?
2. если порядок задан, то является ли он полным, то есть следует ли из существования в массиве данных с индексами N и N+2 также существование и данных с индексом N+1
3. существует ли способ перехода от одного элемента *данных* к следующему?
4. возможно ли наличие в одном массиве данных разных типов?
5. является ли набор индексов раз и навсегда определенным при описании массива или структуры?

В соответствии с ответами на эти вопросы типы составных данных в разных языках называются по-разному. При этом системы в этих названиях, к сожалению, нет, и тип данных с одними и теми же свойствами в одном языке может называться "вектор", а в другом "список".

Массивы

Массивом мы будем называть фиксированный набор однотипных данных, доступ к которым производится по индексу, который является числом или чем-то подобным числу, то есть имеет естественные операции увеличения на 1 и уменьшения на 1. У массива в какой-то момент должно быть задано количество элементов, из этого количества следуют номера первого и последнего элементов, а все элементы между ними должны существовать.

Естественное деление массивов - на одномерные и многомерные. Как правило, в языке программирования есть и те и другие. Однако, начиная с языка Pascal, появилась некоторая путаница. Никлас Вирт, будучи математиком, предложил вполне логичное рекурсивное определение массивов:

```
var v: array[0..10] of array[0..10] of integer;

var w: array[0..10] of array[0..10] of array[0..10] of integer;

var z: array[0..10,0..10] of integer;
(* z и v это одинаковые объявления *)
```

То есть двумерный массив в Pascal это массив одномерных массивов. Все было хорошо до тех пор, пока array[0..10,0..10] было просто сокращенной записью для array[0..10] of array[0..10]. В языке C# это не так.

```
for(int i=0; i < n; i++)
  for(int j=0; j < n; j++)
     a1[i,j] = i*j;
и
for(int i=0; i < n; i++)
  for(int j=0; j < n; j++)
     a2[i][j] = i*j;
```

совершенно разные вещи. Здесь a1 - двумерный массив, a2 - массив массивов.

Описания массивов

В языках L-типа при описании массива должен задаваться его размер. Тогда компилятор сможет выделить память для массива. В языках R-типа обычно выделяют 2 действия : объявление массива и конкретизация его размера.

C#:

```
int [] arr;
arr = new int[10];
```

Индексы массивов

В процессе эволюции языков немало споров было вокруг индексов. Первым камнем преткновения был вопрос о том, с какого числа начинать нумерацию элементов массива. Было предложено три решения: с 0, с 1 и с любого числа по выбору программиста. Почему возникли эти вопросы? Логично считать, что первый элемент массива имеет индекс 1, а последний элемент массива из n элементов имеет индекс n. Однако, это приводит к достаточно неприятным проблемам в случае, когда над индексами нужно производить какие-то действия.

```
var x:array [1..100] of integer;
function element(ix,iy:integer) :integer;
(* возвращает элемент массива x, находящийся на строке ix, столбце iy,
представляя
    массив x как двумерную таблицу, записанную последовательно по строкам
*)
begin
   element := x[(ix-1)*10+iy];
end;
```

Обратите внимание на выражение `ix-1`. Если бы нумерация начиналась с 0, выражение было бы проще: `ix * 10+iy`. Надо сказать, что получив выражение `(ix-1)*10+iy`, компилятор вычтет из него 1 чтобы получить адрес первого элемента. Хорошо, если это выражение будет оптимизировано и приведено к виду `ix * 10+iy`. Но такая оптимизация возможна не всегда. Поэтому массивы, начинающиеся с 0 иногда приводят к более эффективному коду и не вынуждают программиста высчитывать, правильно ли отнята или прибавлена 1.

В языке Pascal было принято вроде бы правильное решение задавать первый индекс массива в явном виде. Пусть программист ставит 1 если хочет использовать натуральный счет, и 0 если ему нужна арифметика индексов. Но это решение привело к некоторым проблемам в ходе развития языка. Очень удобно передавать массивы в процедуры и функции, не задумываясь о совпадении размеров. В Pascal если параметр процедуры описан как `array [1..100]` of integer, то в качестве фактического параметра может быть использован только точно так же описанный массив. А нам хотелось бы иметь функцию, например, суммирования элементов массива, которая могла бы работать с массивом любого размера.

Есть два способа сделать это. Первый – передавать в явном или неявном виде обе границы – нижнюю и верхнюю. Второй – зафиксировать нижнюю границу равной 0 и передавать только одну границу – верхнюю. В языке Ada принят первый способ, а вот разработчики разных вариантов языка Pascal так и не смогли прийти к общему мнению. Оба способа имеют свои недостатки. В первом случае усложняются выражения для операций над индексами, так как вместо 0 придется писать что-то вроде `low(x)` чтобы получить значение нижней границы массива x.

Во втором случае программист вынужден мириться с тем, что его массивы внутри функций будут выглядеть иначе, чем в тех местах, где они были объявлены. Напомним, что функции и процедуры служат для двух различных целей - повторного использования кода и для структурирования программ. Во втором случае вполне возможно, что какой-то массив просто передается от функции к функции с целью его последовательной обработки.

На сегодняшний день индексация массивов начинаются с 0 в большинстве языков программирования, включая C++, Python, Java и Javascript. Индексация с 1 принята в языке Lua, а в Pascal можно начинать индекс с любого числа.

Доступ к элементам массива

Элемент массива может встречаться везде, где появляется обычная переменная, как в правой части, так и в левой. Начиная с языка Algol для индексации массивов используют квадратные скобки [], это

стало уже традицией. В языке C появилось оригинальное нововведение: запись обращения к элементу массива ничем не отличается от обращения к указателю.

```
int a[10];
x = *a;
x=a[0];
```

Это может быть иногда удобно, но приводит к тому, что многие программисты на C не понимают разницу между

```
char *string = “Hello,World”;
```

и

```
char string[] = “Hello,World”;
```

Первое это переменная типа char* с начальным присваиванием, второе - массив типа char. Разница будет обнаружена только когда программист попробует написать

```
string++;
```

В первом случае это сработает, во втором нет.

Массивы и for

Как мы уже говорили, оператор **for** тесно связан с идеей массива и во многих случаях является основным средством работы с массивами. Исторически первой идеей было создавать набор индексов посредством for, получать по этим индексам элементы массива и что-то с ними делать. Во многих случаях при этом нам не нужны сами значения индексов, а нужны только элементы массива. Рассмотрим пример:

```
sum = 0
for i in range(0,5):
  sum = sum + a[i]

print(sum)
```

Вполне естественно было бы исключить индекс из рассмотрения, раз уж он нам не нужен.

```

```

```
$a = [1,2,3];

$sum = 0;

foreach($a as $i)
   $sum = $sum + $i;

print($sum);
```

Специальная форма оператора **for** под названием **foreach** или **for_each** в некоторых языках, позволяет перебирать элементы массива, не обращая внимания на их индексы.

Операции над массивами

Понятие «операции над массивами» может иметь различный смысл. Во-первых, это может быть syntax sugar для циклов. Например, запись

```
A=B+C
```

может означать «к каждому элементу массива B прибавить элемент массива C с соответствующим номером, а результат записать в такой же по номеру элемент массива A.

Такие операции впервые появились как машинно-зависимые, поскольку в середине 80-х годов существовал целый класс суперкомпьютеров с так называемыми векторными процессорами. Они могли выполнять операции над массивами параллельно, за минимальное число тактов. В этом случае операция над массивом - это просто повторение операции над его элементами.

Чаще других реализуют операцию присваивания массивов

```
A=B
```

Во-вторых, можно определить самостоятельные операции, определяемые над типом «массив». Например, + может означать сцепление двух массивов в один. Такой смысл имеет широко распространенная операция сцепления строк. Ее часто ассоциируют со сложением.

```
string a = b+c;
```

Разнообразие этой категории операций над массивами ограничивается только фантазией автора языка, хотя часто тут используются возможности конкретной архитектуры компьютера.

Массивы в функциях и процедурах

Возможность передача массивов в подпрограммы всегда считалась важной для языка. В языке Fortran можно было описывать формальный параметр подпрограммы следующим образом:

```
      SUBROUTINE S(A,N)
      REAL A(N)
...
```

Это позволяло передать в подпрограмму массив и его размер в явном виде, и, таким образом, давало возможность писать подпрограммы, работающие с массивами разного размера.

Однако, в языке Pascal это полезное начинание было полностью отменено, и передача массивов в подпрограммы стала намного менее удобной. Массив как фактический параметр процедуры или функции должен был иметь тот же размер и тип элементов, что и фактический, заданный при описании подпрограммы. В результате написать, например, программу сортировки произвольного массива на Pascal было достаточно сложно. Зато тип передаваемого параметра всегда можно было проверить во время компиляции.

В следующем своем языке Никлас Вирт исправил эту ошибку. В Modula-2 можно было описывать формальные параметры как массив без указания размеров:

Modula-2:

```
PROCEDURE Display(a : ARRAY OF INTEGER);
VAR Index : CARDINAL;
BEGIN
  FOR Index := 0 TO HIGH(a) DO
...
  END;
END;
```

У переданных таким образом массивов нижняя граница всегда 0, а верхнюю можно было получить прои помощи встроенной функции HIGH. Можно выделить следующие способы передачи массивов в подпрограммы:

1. Передача ссылки на массив и передача размера массива в явном виде как отдельного параметра (C, Fortran)
2. Неявная передача границ массива вместе с массивом (Modula-2, Ada, Delphi)
3. Передача массива строго определенного размера (стандартный Pascal).

Структуры

В отличие от элементов массивов элементы структуры идентифицируются по именам. У каждого элемента структуры есть имя и тип, поэтому структуры иногда называют неоднородными массивами. В случае массива вычислением положения элемента занимается программист, а в случае структуры - компилятор.

```
```

```
struct rectangle
{
  int x,y;            // координаты
  int width,height;   // длина и ширина
  color c;            // цвет
  float rotation;     // угол поворота
};
```

В этом примере есть значения разных типов, но нас не интересует их положение относительно друг друга. Если мы поменяем местами x и rotation, программа на C, использующая данную структуру, не изменится. Составными частями структур могут быть переменные разных типов, массивы, а также другие структуры.

Во всех современных языках допускаются массивы структур и структуры, элементами которых являются массивы. Однако, описание структур не всегда было простым делом.

Описание структур

В языках PL/1 и COBOL структуры описывались следующим образом:

Cobol:

```
DECLARE 01 CUSTOMER
  02 NAME
    03 FIRSTNAME CHAR(100)
    03 MIDDLEINITIAL CHAR
    03 LASTNAME CHAR(100)
  02 SALARY FLOAT
```

то же самое на языке C:

```
struct customer
{
  struct name
  {
    char[100] firstname;
    char middleinitial;
    char[100] lastname;
  };
  double salary;
};
```

и на Pascal:

```
type customer = record
  name : record
```

```
    firstname : array[1..100] of char;
    middleinitial : char;
    lastname : array[1..100] of char;
  end;
  salary : real;
end;
```

Как правило, в современных языках программирования используются какие-то вариации на тему C и Pascal, однако в функциональных языках возможно некоторое разнообразие. В языке Elixir структуры описываются так:

```
defmodule User do
  defstruct first_name: "John", last_name: "Brown", age: 27
end

new_user = %User{first_name: "Georgia", last_name: "Brown"}
```

Ничего необычного здесь, вроде бы, нет, кроме того, что для first_name, last_name и age заданы значения по умолчанию. Однако, если мы попробуем выяснить тип переменной new_user, то это будет **map**. Что такое **map**, мы выясним чуть позже, а сейчас нужно понять, что в разных языках программирования разные с точки зрения реализации вещи могут называться одинаково и наоборот, одинаковые реализации могут носить разные названия. На всякий случай перечислим наиболее часто встречающиеся *названия* составных типов в языках, как на русском, так и на английском языке.

массив, структура, список, последовательность, вектор, ассоциативный массив, множество, словарь

vector, array, structure, list, sequence, map, dictionary, associative array, set

Обращение к элементам структур

Если где-то имеется переменная с типом структуры, то обращение к ее элементам в большинстве языков более или менее стандартно — имя элемента пишется через точку:

C:

```
customer Ivan;
Ivan.name.middleinitial = 'И';
Ivan.salary = 10000.0;
```

```
user = %User{first_name: "Ivan", last_name: "Smith", age: 33}
IO.Puts(user.first_name)
IO.Puts(user.last_name)
```

Кроме этого, в C и C++ есть операция `->` (стрелка), которая означает получение элемента структуры по ссылке на эту структуру. Интересно, что в C++ операцию `.` (точка) нельзя переопределить для

других типов, а стрелку `->` - можно.

В качестве исключения можно отметить язык PHP, в котором с самого начала не предполагалось никаких структур, и точка была использована как операция сцепления строк. Впоследствии составные типы все-таки добавили, и операцию доступа к элементу такого типа обозначили как `->`, заимствовав знак операции из C.

Присваивание структур

Вполне естественно рассматривать структуру как единое целое, раз уж набор разнородных данных обозначается одним общим именем, и присваивать структурные переменные так же, как обычные. Часто еще добавляют возможность описывать структурные константы.

C:

```
struct Point
{
  int x,y;
};

static struct Point pt = {10,20};
struct Point x = pt;
```

Хотя в языке C нет присваивания массивов, там есть присваивание структур. Парадоксально, но в языке C присваивание массивов возможно, если они являются элементами структур.

```
#include <stdio.h>

struct InArr
{
  int x,y;
  int a[2];
};

int main()
{
    struct InArr pt;
    pt.a[0]=33;
    pt.a[1]=34;

    struct InArr x;
    x.a[0]=0;
    x.a[1]=0;

    x = pt;

    printf("%d %d\n", x.a[0], pt.a[1]); // напечатает 33 34

    int a[2];
```

```
    int b[2];

    // a=b;  // ошибка!

}
```

Если у нас есть переменные-структуры, то у них есть адреса в памяти, следовательно, можно сделать указатели на структуры.

Использование структур

Польза от структур в основном выражается в двух совершенно не связанных областях. Во-первых, структуры позволяют уменьшить количество имен, с которыми приходится иметь дело «на верхнем уровне». Предположим, у нас есть какое-то количество точек на плоскости (как в предыдущем примере). У каждой точки есть две координаты, x и y. Если бы структур не было, пришлось бы заводить по два массива на каждый набор точек, по две переменных на каждую точку, и со всем этим как-то разбираться. Структуры позволяют «спрятать» одинаковые элементы внутрь, и работать с единым элементом «точка». Это уменьшает пространство имен и одновременно улучшает понимание, ведь всегда удобнее мыслить категориями «время», «точка на плоскости», чем «x-координата точки и y-координата этой же точки в другой переменной».

Интересно, что такая организация пространства имен иногда приходит в противоречие с современными аппаратными средствами. Дело в том, что у процессора есть так называемый кэш, сверхскоростная память, куда копируется содержимое медленной основной памяти. Предположим, что у нас есть массив структур типа "точка на плоскости", состоящих из координат x и y. Пусть нам нужно переместить объект на экране и для этого прибавить какое-то число ко всем x - координатам точек массива. При этом по сравнению с отдельными массивами для координат x и y в кэш поместится вдвое меньше точек. Если в структуре есть еще что-то, например, цвет точки, то снижение скорости будет еще более существенным.

Вторая польза от структур состоит в возможности организации динамических данных. Динамические данные это такие данные, общий размер которых выясняется только в процессе исполнения. Например, произвольную последовательность чисел можно описать как структуру, состоящую из числа и указателя на следующую структуру.

C:

```
struct list
{
  int value;
  struct list* next;
};
```

Такое описание рекурсивно: список это или специальный объект, называемый «пустой список», или элемент списка, за которым следует список. Моделирование таких объектов при помощи простых переменных и массивов возможно, но не очень просто. Работа с такими данными производится

обычно при помощи рекурсивных процедур и функций. Например, посчитаем сумму чисел, составляющих список:

C:

```
int sum(struct list* i)
{
    if(i == NULL) return 0;
    return i->value + sum(i->next);
}
```

Сначала мы проверяем, не является ли переданный нам объект пустым списком. Для нас пустой список это просто нулевой указатель. Если да, то сумма равна 0. В противном случае это число, соответствующее данному элементу списка плюс сумма всех остальных элементов списка.

Похожим образом можно реализовать и другие интересные виды данных. Например, стек это такой же список, для которого есть подпрограммы для добавления элемента в начало и удаления первого элемента:

C:

```
// добавление элемента
struct list* push(struct list* i, int value)
{
    struct list* t = (struct list*) malloc(sizeof( struct list ));
    t->next = i;
    t->value = value;
    return t;
}

// удаление элемента
struct list* pop(struct list* i)
{
    if(i == NULL) return NULL;
    struct list* t = i->next;
    free(i);
    return t;
}

// печать списка
void print(struct list* l)
{
    while(l)
    {
        printf("%d ",l->value);
        l = l->next;
    }

    printf("\n");
```

```
}

// использование всего этого
int main()
{
  struct list *l = 0;

  l = push(l,33);
  l = push(l,34);
  l = push(l,35);

  print(l);  // напечатает 35 34 33

  l = pop(l);

  print(l); // напечатает 34 33

  printf("%d\n",sum(l)); // напечатает 67

}
```

При добавлении элемента мы сначала выделяем память, а потом вставляем новый элемент в начало списка. При удалении мы сначала проверяем, не попросили ли нас удалить элемент пустого списка. Если нет, то мы сначала получаем указатель на следующий элемент списка (возможно, это NULL), а потом освобождаем память для первого элемента. Функция печати списка показывает, как можно использовать нерекурсивные методы для работы со списком. При помощи стека можно сделать программу, вычисляющую выражения в постфиксной записи.

Другой полезной формой организации данных является очередь. Она похожа на стек, только элементы добавляются в начало, а удаляются с конца.

Более подробно об организации данных можно прочитать в посвященных этому книжках, например, в книге Никласа Вирта «Алгоритмы и структуры данных». Для нас сейчас важно, что структурные типы в языке позволяют работать с динамическими данными.

Списки

В некоторых языках списки встроены в синтаксис. Термин "список" мы будем понимать как нечто, аналогичное массиву, но без индекса. Для списка должны быть определены три операции: получения первого элемента списка, получения остатка списка без первого элемента и получения нового списка из первого элемента и списка путем присоединения этого элемента в начало списка.

Нельзя сразу получить N-й элемент списка, но можно отсчитать N элементов с начала списка. И эту операцию часто встраивают в язык, так что список может быть неотличим от массива.

```
list = [1, 2, 3]
```

```
(write (list 1 2 3 ))
```

Стандартные названия для трех списочных операций пришли из языка Lisp. Его название это сокращение от **LIS**t **P**rocessor, так что списки составляют основу этого языка. Функция, возвращающая первый элемент списка, называется CAR, функция для получения списка без первого элемента называется CDR, а функция, присоединяющая первый элемент к списку, носит название CONS. Однако, в языках семейства ML по каким-то причинам первые две функции называются hd и tl (от слов head и tail).

Списки удобны тем, что составляют основу для многих алгоритмов, независимо от того, какие способы хранения используются. Например, в стандартной библиотеке C++ есть набор базовых операций для прохода по элементам списка, а конкретная реализация этого списка может оказаться стеком, очередью или даже простым массивом.

Для списков также существуют стандартные операции map, reduce и filter.

Аргументы **map** — список и функция с одним аргументом. В результате выполнения операции map функция вызывается для каждого элемента и из результатов формируется новый список.

```
for(int i=0; i < n; ++i)
    result[i] = f(a[i])
```

Результат операции **map** — список того же размера, что и исходный. В языках L-типа элементы списка должны быть одинакового типа, что автоматически достигается при объявлении типа элементов.

PHP

```
function cube($n)
{
   return($n * $n * $n);
}

$a = array(1, 2, 3, 4, 5);
$b = array_map("cube", $a);
```

Аргументы функции **reduce** - список, функция с двумя аргументами, и, возможно, начальное значение того же типа, что и элемент списка. Функция последовательно применяется к промежуточному результату и каждому элементу списка. Если используется вариант без начального значения, им может быть первый элемент списка, тогда применение **reduce** к пустому списку может быть не определено.

```
for(int i=0;i<n;i++)
    result = f(result,a[i])
```

Результат операции — скаляр или список меньшей размерности.

PHP

```php
function rsum($v, $w)
{
    return $v + $w;
}

$a = array(1, 2, 3, 4, 5);
$b = array_reduce($a, "rsum");
```

filter - двуместная операция. Ее аргументы - список и функция с одним аргументом, которая возвращает логическое значение. Она последовательно вызывается для каждого элемента списка, и из тех элементов, для которых она вернула true, формируется список-результат.

```php
function rfilt($v)
{
    if ($v == 2) return true;
    if ($v == 5) return true;
    return false;
}

$a = array(1, 2, 3, 4, 5);
$b = array_filter($a, "rfilt");
```

Последовательности

Последовательность похожа на массив, и у нее также есть индекс, но нет заранее заданного размера. В последовательность можно добавлять элементы, можно удалять их и еще есть способ получить текущую длину последовательности. В современных динамических языках последовательность является основным видом контейнера.

Например, в языке Python добавление элемента в последовательность производится вызовом **append**, удаление называется **del**, а длину можно узнать при помощи функции **len**.

```python
a = []
a.append(22)
a.append(33)
a.append(45)

print(a)        # [22, 33, 45]
print(len(a))   # 3

del a[1]

print(a)        # [22, 45]
print(len(a))   # 2
```

В языке Lua длину последовательности можно узнать при помощи операции #, добавление элемента в последовательность производится присваиванием элементу, следующему за последним, а удалить элемент можно, используя **table.remove**

```
a = {}

a[#a+1] = 33
a[#a+1] = 44
a[#a+1] = 55

print(unpack(a)) -- 33  44  55

print(#a)        -- 3

table.remove(a,2)

print(unpack(a)) -- 33 55

print(#a)        -- 2
```

Вызов функции unpack тут нужен только для красивой печати значений.

В Javascript добавление элемента называется **push**, удаление **delete**, а длину можно получить, используя **length**

```
a = [];


a.push(33);
a.push(44);
a.push(55);

console.log(a);         // [ 33, 44, 55 ]

console.log(a.length);  // 3

delete(a[1]);

console.log(a);         // [ 33, , 55 ]

console.log(a.length);  // 3
```

Похоже, что в Javascript произошло что-то не то. В последовательности образовалась дырка. Тут мы столкнулись с одной из фундаментальных проблем, связанных с последовательностями. Рассмотрим следующий код:

```
a = [1,1,2,2,3,3,4,4,5,5];

for(i = 0; i < a.length;++i){
  if(a[i] == 3)
    delete(a[i]);
}
```

Это часто встречающаяся в императивных языках задача - при последовательном просмотре удалить элементы последовательности в зависимости от какого-то условия. Проблема состоит в следующем: допустим, цикл дошел до **i** равного 4. Программа удалила элемент с индексом 4 (он равен 3), и последовательность стала такой: `[1,1,2,2,3,4,4,5,5]`. Следующее значение **i** будет 5, и элемент с этим индексом теперь имеет значение 4. Вторая тройка так и не будет удалена из последовательности. В зависимости от тех операций, которые предполагается проделать, возможен как пропуск элементов, так и обращение к несуществующим элементам последовательности (представьте себе, например, что мы вычислили длину последовательности заранее и присвоили ее переменной).

В языке Javascript была сделана попытка избавиться от этой проблемы, и удаление элементов последовательности не меняет индексы других элементов. Но при этом в последовательности индексов остаются "дырки".

Правильное решение этой проблемы - никогда не удалять и не добавлять элементы в последовательность в цикле, а формировать новую.

```
a = [1,1,2,2,3,3,4,4,5,5];

var t = [];
for(i = 0; i < a.length;++i){
  if(a[i] != 3)
    t.push(a[
      i]);
}
a = t;
```

Этот способ всегда используется в функциональных языках, однако начинающие программисты, работающие с императивными языками, часто пытаются его избежать. Эффективность того и другого подходов зависит от особенностей реализации последовательностей в данном языке, и формирование новой последовательности может быть не медленнее, чем модификация существующей.

Другой подход к изменению последовательностей состоит в получении так называемых вырезок.

Вырезки из последовательности

Удалим элемент из последовательности.

Python

```
```

```
a = [ 33, 44, 55 ]

print(a[0:1]) # [33]
print(a[2:3]) # [55]

a = a[0:1]+a[2:3]

print(a)  # [33, 55]
```

Вместо того, чтобы удалять элемент последовательности, мы вырезали два куска и склеили их вместе. Способ этот хорош, но эффективность его сильно зависит от реализации. Попробуем сделать то же самое в Javascript

```
a = [ 33, 44, 55 ];
a.splice(1,1);   // удаляем один элемент, начиная с первого

console.log(a);  // [ 33, 55 ]

console.log(a.length);  // 2
```

В Lua ничего подобного нет, хотя можно написать процедуру, которая будет удалять элементы последовательности. Как видим, способы работы с последовательностями существенно различаются от языка к языку, но в целом имеется два подхода к этой задаче. Можно удалять и добавлять элементы по одному, а можно рассматривать последовательность как единое целое.

Кортежи

Кортеж (tuple) объединяет в себе свойства массива и списка. Это составной объект, доступ к элементам которого производится по индексу, но добавление элементов в него невозможно. Кроме того, сам кортеж обычно делают неизменяемым, то есть присвоить что-то его элементу невозможно. Кортеж можно рассматривать как структуру, у которой элементы не имеют имен, а имеют числовые индексы. Основное достоинство кортежей состоит в том, что их не надо объявлять заранее.

```
a = (1,"word",2.5)
```

Такую функциональность можно включать даже в языки L-типа, поскольку типы кортежей можно выяснить при компиляции. Однако этого обычно не делают, и в языке Nim кортежами называются структуры, то есть массивы с постоянной длиной и именованными элементами.

Словари

В языке Python, PHP и некоторых других есть еще один тип данных, похожий на массив - словари или ассоциативные массивы.

```
```

```
$a = array("one"=>"один", "two"=>"два", "three" => "три");
```

Это такие списки, у которых индексами служат произвольные значения, как числового так и строкового типа. При этом не предполагается, что числовые индексы идут подряд.

```
a = {1:"one",20:"twenty",5:"five"}
print(a[1])
print(a[2]) # ошибка
```

Кроме функций добавления, удаления и изменения элементов, в языки обычно включают еще функции для получения всех значений и всех индексов данного словаря.

```
a = {1:"one",20:"twenty",5:"five"}
k = a.keys()
for key in k:
  print(key)
```

Составные данные в языках Javascript и Lua

В языках программирования Javascript и Lua сделан еще один шаг в сторону унификации всех составных типов. В этих языках записи

```
a["x"]
```

и

```
a.x
```

означают одно и то же. Если индекс является числом, то вторая запись, конечно, невозможна. Таким образом, нет разницы между структурами и ассоциативными массивами.

Сводная таблица составных типов

Как видно, терминологию в области составных типов нельзя назвать устоявшейся. В этом тексте были использованы следующие термины:

1. **Массив** - набор из фиксированного числа однородных элементов, доступ к которым происходит по числовому индексу.
2. **Структура** - набор из фиксированного числа разнотипных элементов, доступ к которым происходит по имени.

3. **Последовательность** - набор из переменного числа однотипных или разнотипных элементов, доступ к которым происходит по числовому индексу. Возможно добавление и удаление элементов.
4. **Кортеж** - набор из фиксированного числа однотипных или разнотипных элементов, доступ к которым происходит по числовому индексу. Удаление, добавление и изменение элементов невозможны.
5. **Словарь** - набор из переменного числа однотипных или разнотипных элементов, доступ к которым происходит по индексу произвольного типа. Возможно добавление и удаление элементов.

ООП

Магия перестаёт существовать после того, как вы понимаете, как она работает.
Тим Бернерс-Ли

ООП как Syntax Sugar

Знакомство с объектно-ориентированным программированием лучше всего начать с разбора типичной задачи. Предположим, что нам надо сделать библиотеку интерфейса пользователя, в которой основным "действующим лицом" будет окно, которое выводится на экран. Окна можно рисовать, показывать, убирать с экрана, перемещать, красить в разные цвета и т.д. Рано или поздно выясняется, что существенная часть функций такой бибилиотеки выглядит примерно так:

```
window_move(window* w, int x, int y);
window_size(window* w, int width, int height);
window_color(window* w, int color);
```

С точки зрения естественного языка это выглядит как тавтолгия: Окно_Переместить(Вот это окно, ...) Однако, если назвать функции просто move, size и color, легче не станет, потому что эти понятия слишком неспецифичные, мало ли что можно переместить или покрасить. Давайте введем такую договоренность: запись

```
window.move(x,y);
```

это то же самое, что и

```
move(window,x,y);
```

Правда же, это просто? И из вот этой простой идеи в результате выросла одна из самых сложных и мощных концепций программирования. Давайте посмотрим, что еще можно сделать с этой идеей.

Полиморфизм и методы

При наличии такой записи нет надобности называть функции различными именами, вроде `window_move`, компилятор сам разберется, что имелось в виду. window.move и car.move могут существенно отличаться, при этом у программиста тоже не возникает проблем с пониманием того, какая функция имелась в виду.

Возникает хорошая идея описывать функции прямо внутри описания структур, как это делается с другими их элементами. Так и делается в языке **Java**.

```
class val{
    int data;
    public void print(){
      System.out.println(data);
    }
};
...
val a;
...
a.print();
```

Есть небольшая сложность с тем, что если код какой-то функции очень длинный, то программист, желающий всего лишь прочитать описание структуры, будет вынужден прокручивать страницы ненужного ему кода. Поэтому считается хорошим тоном размещать данные, которые находятся внутри структуры, выше описаний кода. Заметим также ключевое слово `class`, которое говорит нам о том, что дальше идет описание "структуры с приклеенной к ней функциональностью". Для функций, описанных внутри класса, тоже придумали новое название - методы. Итак, у нас есть классы, внутри которых есть методы.

Экземпляры класса и this

В приведенном примере метод print обращается к переменной data без всякого объявления, как если бы все элементы структуры были описаны внутри метода как переменные. Не во всех языках это так, в PHP и Javascript придется использовать специальное ключевое слово.

```
class Val {

    public $data = 42;

    function print() {
        echo "My value is " . $this->data;
    }
}

$x = new Val;
$x->print();
```

```
```

```
a = {
  data : 42,
  print:function(){
     console.log("My value is " + this.data);
  }
}

a.print();
```

Вспомним, что мы имеем в виду под вызовом `x.print()`. Это просто другая запись для print(x). И вот этот самый **x**, который в скрытом виде передается функции print, внутри нее обозначается словом `this`.

Описание класса, как и описание структуры, дает нам шаблон, по которому строятся реальные данные. Можно создать много экземпляров класса, и у каждого из них будут свои данные, а вот методы будут, как правило, у всех одинаковые. По крайней мере, такова была исходная идея объектно-ориентированного программирования.

В языке Python описание методов более прямолинейно. Вместо создания специальной переменной `this`, Python обязывает программиста в явном виде давать имя дополнительному параметру.

```
class Val:
  def __init__(self):
    self.data = 42

  def print (this):  # не делайте так!
    print("My value is ",this.data)

x = Val()
x.print()
```

Принято (и не надо нарушать это правило, как мы это сейчас сделали) называть этот параметр `self`, хотя язык позволяет давать ему любое имя. Еще одно нововведение, которое мы тут видим, это метод со странным именем `__init__`, который автоматически вызывается при создании экземпляра объекта. Такой метод называется конструктор, и он очень удобен для присваивания начальных значений данным, поэтому он есть почти во всех объектно-ориентированных языках. Название его везде разное, но в целом есть три основных способа сказать, что данный метод является конструктором. В языке **Python** используется специальное имя, в **Delphi** есть ключевое слово `constructor`, а в C++, Java и других производных от C языках конструктором является тот метод, имя которого совпадает с именем класса.

```
class Val {
public:
   int data;
```

```
    Val(){        // конструктор
      data = 42;
    }
    void print(){
      cout << data << endl;
    }
};
```

Для создания экземпляра класса традиционно используется ключевое слово (или функция) `new`. `x = new C()` означает буквально следующее: выделить память под структуру и вызвать конструктор, который заполнит там поля. В языке **Python** вместо этого нужно использовать имя класса как функцию: `x = C()`.

Инкапсуляция

Итак, у нас есть структуры (теперь они называются объекты), у которых есть методы, и эти методы в основном используют данные из самих этих структур. Это значит, что у нас становится меньше глобальных сущностей, и это хорошо. Кроме этого, обычно в реализации ООП добавляют возможность в явном виде разделить все методы объекта на те, которые предполагается использовать "извне" для работы с объектом и те, которые нужны только для каких-то внутренних целей. Первые обозначаются ключевым словом `public`, вторые - `private`. Описание конструктора и методов, объявленных как`public`, составляет интерфейс объекта, то, что с ним могут делать другие программы. Все остальное - внутреннее дело самого объекта. Обычно предполагается, что все, описанное как `private`, может быть изменено в любой новой версии кода. В случае правильно спроектированной программы на работе функций, использующих объект это не должно сказаться.

Наследование

Предположим, у нас есть класс, описывающий окна программы. Он называется Window, и в нем есть какие-то методы для управления оконной системой. Например, переместить окно, открыть или закрыть окно, и так далее. Пусть теперь у нас появилось новое свойство окон, про которое мы раньше не подумали. Например, цвет. Если просто добавить поле с именем color в описание класса окна, все уже написанные функции никак не изменятся. Они просто не заметят наше добавление, ведь к элементам структуры они обращаются по именам. Если мы теперь напишем методы, работающие с цветом окон, мы получим новую реализацию оконной библиотеки, у которой какие-то методы будут совпадать со старыми, какие-то будут немного изменены, а какие-то будут написаны заново.

Концепция наследования позволяет не переписывать код и не менять интерфейс объекта, а добавлять новую функциональность.

```
class Window{
    int coord_x,coord_y;
    ... // описание каких-то еще данных
    public:
    Window(int x,int y,int height,int width);
    void move(int dx,int dy);
```

```
};

class ColorWindow: public Window{
    Color window_color; // цвет
    public:
    ColorWindow(int x,int y,int height,int width,Color c);
    void setColor(Color c);
};

ColorWindow cw = new ColorWindow(10,10,640,480,BLUE);
cw.move(20,20);
cw.setColor(GREEN);
```

Здесь метод move в классе ColorWindow достался в наследство от класса Window, метод setColor был добавлен, поскольку для класса Window он не имел бы смысла, а конструктор был изменен.

Поскольку классам-наследникам иногда нужен доступ к данным базового класса, появляется еще один вид доступа к данным и методам. Кроме уже описанных `public` и `private` добавляют еще `protected`, то есть разрешение доступа к таким элементам изнутри класса и для всех наследников данного класса.

Иерархию классов обычно строят всю целиком, а не добавляют туда свойства по запросу. И такая иерархия сама по себе является способом посмотреть на задачу в целом. Для примера напишем на Python систему классов для работы с геометрическими фигурами на плоскости.

```
class Figure:
    # Здесь мы описываем
    # наиболее общие свойства геометрических фигур.
    # У них есть координаты
    def setCoord(self,X,Y):
        self.x = X
        self.y = Y

    def getCoord(self):
        return (self.x,self.y)

    def __init__(self):
        self.x, self.y = 0,0

class Triangle (Figure):   # Треугольник это такая фигура
    # У треугольника есть все, что есть у фигуры, и кроме того
    def setVertices(self,P1,P2,P3):
        self.vertices = [P1,P2,P3]

    def getVertices():
        return self.vertices

    def __init__(self,P1,P2,P3):
        Figure.__init__(self)
```

```
        self.setVertices(P1,P2,P3)

    def Area(self):
        pass


class Circle (Figure):   # Круг это тоже такая фигура
    # У треугольника есть все, что есть у фигуры, и кроме того
    def setRadius(self,R):
        self.radius = R

    def getRadius():
        return self.radius

    def __init__(self,R):
        Figure.__init__(self)
        self.setRadius(R)

    def Area(self):
        pass
```

Экземпляры созданных таким образом классов можно, например, положить в массив:

```
a = [Circle(10),Triangle((10,10),(32,10),(50,10)),Circle(20)]

for f in a:
    print(f.Area())
```

В результате работы цикла будут напечатаны значения площадей разных фигур, причем для их вычисления будут вызваны методы разных классов. Заметим, что для этого нам нигде не пришлось проверять тип объекта, как это надо было бы сделать без ООП.

Задание: напишите код методов для вычисления площадей треугольников и кругов. Добавьте описание класса, реализующего квадраты на плоскости.

В языках L-типа ситуация немного сложнее. Так как тип данных должен быть полностью известен во время ко��пиляции, декларируется, что "*ссылки на типы объектов совместимы по присваиванию сверху вниз*". Это означает следующее: мы не можем просто так взять и скопировать объект с большим количеством данных в объект с меньшим количеством данных. Однако, мы всегда можем разместить эти данные в памяти так, чтобы метод объекта базового класса, получив ссылку, "думал", что он работает со "своим" объектом. То есть переменной, которой можно присвоить ссылку на объект базового класса всегда также можно присвоить и ссылку на объект производного класса (но не наоборот!). Это позволяет нам работать с иерархиями объектов. Но при этом возникает одна проблема. Посмотрите еще раз на описание Figure, Circle и Triangle. Вычисление площади неизвестно какой фигуры не имеет смысла. А компилятор, рассматривая базовый класс Figure, не должен ничего

знать про то, что мы где-то когда-то напишем метод, вычисляющий площадь конкретной фигуры. Figure - вещь слишком абстрактная, чтобы иметь хоть какую-то площадь.

Для таких случаев придуман механизм абстрактных классов. Если в языке такой механизм есть, можно объявить внутри Figure метод Area (и компилятор будет знать, что такой у нас есть), но объявить его абстрактным (`abstract`), чтобы компилятор знал, что реализация метода будет где-то в наследниках данного класса. Заодно наличие хоть одного абстрактного метода говорит о том, что создание экземпляров объектов такого класса не имеет смысла.

Перепишем наш пример на Java:

```
abstract class Figure{
    // Здесь мы описываем
    //наиболее общие свойства геометрических фигур.
    // У них есть координаты
    int x,y;

    void setCoord(int X,int Y){
        x = X;
        y = Y;
    }

    Point getCoord(){
        return new Point (x,y);
    }

    Figure(){
        x = 0; y = 0;
    }

    abstract int Area(); // абстрактный метод

};

class Triangle extends Figure{   // Треугольник это такая фигура

    Point vertices[];

    void setVertices(Point P1,Point P2,Point P3){
        vertices[0] = P1;
        vertices[1] = P2;
        vertices[2] = P3;
    }

    Point[] getVertices(){
        return vertices;
    }

    Triangle(Point P1,Point P2,Point P3){
        super();
        setVertices(P1,P2,P3);
    }
```

```
    @Override
    int Area(){
        return 0;
    }

}

class Circle extends Figure{   // Круг это тоже такая фигура

    int radius;

    final void setRadius(int R){
        radius = R;
    }

    int getRadius(){
        return radius;
    }

    Circle(int R){
        super();
        setRadius(R);
    }

    @Override
    int Area(){
        return 0;
    }
}
```

Здесь мы видим некоторые нововведения по сравнению с вариантом на Python. Во-первых, класс Figure объявлен как абстрактный. И в нем есть абстрактный же метод Area. Во-вторых, в производных классах добавлено (необязательное в Java) описание @Override. Это способ сказать компилятору, что у нас тут имеется конкретизация ранее объявленного абстрактного метода.

Кроме того появляется новое описание `final`. Оно значит, что данный метод *не* будет меняться в наследниках класса.

Объектно-ориентированное проектирование

Одним из наиболее важных преимуществ ООП является возможность "приблизить" понимание проблемы заказчиком и понимание решения этой проблемы программистом. Дело в том, что мы в обычной жизни привыкли рассуждать в категориях предметов, а не в категориях алгоритмов. И если до изобретения ООП заказчик и программист были вынуждены долго выяснять, что хочет заказчик, исходя из внешнего вида отчетов, то сегодня они могут сравнительно просто договориться об объектах, с которыми должна работать программа. Например, постановщик задачи может спросить: с какими объектами реального мира должна работать программа? Пусть это будут покупатели. Что мы хотим знать про них? Список покупок, возраст, адрес, номер телефона и что-то еще, и весь этот список

можно довольно просто представить в виде полей и методов структуры, представляющей в системе покупателя.

Можно представить себе объект как устройство с кнопками, нажатие на которые приводит к выполнению каких-то действий. Что находится внутри устройства и как именно оно работает, нам не слишком интересно. На первый план выходит удобство нажатия на кнопки и хороший внешний вид самого устройства.

В области ООП имеется множество пособий, программ и технологий, среди которых наиболее известна система Rational Rose фирмы IBM. В этой системе можно нарисовать диаграммы с объектами, диаграммы взаимодействия между объектами, диаграммы взаимодействия пользователей с системой и многое другое. Результатом моделирования, рисования диаграмм и их обсуждения с заказчиком может являться почти готовый код, в который останется только добавить реализацию уже документированных действий, таких, как загрузка данных, отображение их и обработка. Для моделирования систем разработан специальный графический язык UML (не являющийся языком программирования).

Модели ООП

Как мы только что видели, в языках L-типа и R-типа вызовы методов объекта немного отличаются. В Java компилятор даст вызвать только те методы, которые заранее описаны для данного объекта. В Python можно вызвать любой метод объекта, и если он у объекта есть, он выполнится, а если его нет, возникнет ошибка. Для такого случая придуман специальный термин "*посылка сообщений объекту*". В языке Smalltalk это единственный метод для работы с объектами.

```
Transcript open.
Transcript show: 'Hello world'.
```

В первой строке мы посылаем объекту `Transcript` сообщение `open`, во второй - сообщение `show` с параметром 'Hello world'.

Полезность такого подхода особенно проявляется при работе с графическими интерфейсами (GUI). При этом в системе существует набор сущностей (окна, кнопки, линейки прокрутки итп), которыми управляет программа, и набор возможных внешних воздействий, таких как нажатие кнопки мыши, перемещение мыши, нажатие клавиши на клавиатуре, срабатывание таймера. Возникает вопрос: как информацию обо всех этих событиях события передать программе? Очевидно, хотелось бы, чтобы каждое такое событие приводило к вызову какого-то метода. Если наша программа написана на языке L-типа, то каждый такой метод должен быть где-то описан. В системе Windows количество возможных сообщений, приходящих от операционной системы к пользовательской программе, превышает 1000. Значит, где-то должен быть описан объект с 1000 методами, каждый из которых получает сообщение и ничего не делает. Тогда в производном классе можно будет переопределить методы и в них делать то, что нужно. Это не очень хороший подход. Обычно используют другие, например, в явном виде связывают метод класса с сообщением.

```
void my_callback( Fl_Widget* o, void*  ){
```

```
    // получает управление при нажатии на кнопку
}


 Fl_Button btn( 10, 150, 70, 30, "Click me" );
 ...
 btn.callback( my_callback );
```

Однако, и этот способ нельзя назвать правильным, потому что управление получает функция, внешняя по отношению к тому объекту, в котором была создана кнопка. Создаются новые глобальные сущности, логика управления оказывается разбросанной по разным функциям. Нам хотелось бы, чтобы создание интерфейса и реакция на события были сосредоточены в одном объекте. Это достигнуто в системе Qt, но там для этого сделан специальный предкомпилятор, который просматривает исходный код на C++ с дополнениями Qt и транслирует его в код на C++, который уже может быть понят стандартным компилятором.

В Python и других языках R-типа все намного лучше. В некоторых библиотеках GUI нужно в явном виде связывать тип события с обработчиком, например, в wxPython:

```
...
# добавляем пункт в меню
exitItem = fileMenu.Append(wx.ID_EXIT)
...
# связываем его с обработчиком
self.Bind(wx.EVT_MENU, self.OnExit,  exitItem)
...
# а вот обработчик
def OnExit(self, event):
        """Close the frame, terminating the application."""
        self.Close(True)
```

В других библиотеках достаточно описать метод со стандартным именем, и он будет вызван.

```
# библиотека libui
class MyWindow(Window):
    ...
    def onClose(self, data):
        super().onClose(data)
        app.stop()
```

И в том и в другом случае наличие информации о типах во время выполнения сильно облегчает задачу. Вызывающая программа может использовать примерно такой алгоритм:

ЕСЛИ в связанном с событием объекте есть метод с данным именем, ТО вызвать этот метод ИНАЧЕ игнорировать сообщение

Интерфейсы и Duck typing

Как мы только что видели, для того, чтобы какая-то программа могла пользоваться объектами, достаточно, чтобы класс этого объекта реализовывал методы, которые данная программа собирается вызвать. Условно говоря, если у нас есть неизвестное устройство с кнопками, и назначение некоторых из этих кнопок нам известно (например, они подписаны), то мы можем пользоваться некоторыми функциями этого устройства, ничего не зная об остальных. В языках R-типа проблем не возникает, мы можем вызвать любой метод любого класса, не задумываясь о его существовании. Но в тех языках, где информация о типе в основном недоступна во время выполнения, компилятор должен решить, какой метод вызывать. Для этого была придумана концепция интерфейса. Интерфейс в ООП это набор методов, которые должны быть реализованы в объекте. Например:

```
interface Moveable {
    void moveAbs(int x,int y);
    void moveRel(int dx,int dy);
}

class Triangle implements Moveable{
    void moveAbs(int x,int y){
        ...
    }
    void moveRel(int dx,int dy){
        ...
    }
}
```

Здесь мы описали интерфейс, не связанный по сути своей с геометрическими фигурами на плоскости, и описали фигуру, которая *реализует* этот интерфейс. Это значит, что методы moveAbs и moveRel обязаны там присутствовать (иначе компилятор выдаст ошибку). Теперь мы можем написать программу, которая будет использовать наш интерфейс, не зная ничего о реальном типе переменной.

```
...
    void moveByPath(Moveable something, Point [] path){
        for(Point pt:path){
            something.moveAbs(pt.x,pt.y);
        }
    }
...
```

Имея информацию об интерфейсе, компилятор может решить, какие методы надо вызывать в данном случае. В языках R-типа такие описания не нужны, и вместо этого применяется концепция, которая называется duck typing: объект относится к некоторому классу, если он реализует все методы, которые реализует этот класс. "Если оно крякает как утка и ходит как утка и плавает как утка, то это утка и есть". Это приводит к несколько размытому представлению о том, что может делать объект, и немного мешает взаимодействию между программистами. В языке Javascript, например, ничто не мешает сделать объект, у которого будет один набор методов по понедельникам, другой по вторникам и так

далее, и сочинитель такого объекта даже сможет его использовать, но попробуйте объяснить необходимость такого странного решения другим людям. Вряд ли они согласятся.

Наследование классов и наследование прототипов

Соответственно двум описанным только что моделям ООП, различают два вида наследования. Первый вид используется в основном в языках L-типа и называется наследованием классов. При нем имеется формально прописанный набор методов и свойств класса, причем сам класс тоже может быть экземпляром класса более высокого порядка (метакласса), а наследование просто добавляет функциональность в производные классы. Здесь класс является как бы контрактом, в соответствии с которым объект должен иметь в своем составе те или иные методы.

Другой вид наследования называется наследованием прототипов и согласно ему, производный объект должен быть "совсем как вон тот объект, но еще должен уметь то-то и то-то". В языке Javascript никаких описаний классов нет, вместо этого конструктор просто создает объект, добавляя в него нужные свойства и методы.

```
class Circle {
  constructor(center, radius) {
    this.center = center;  // свойства center и radius нигде заранее не
были описаны
    this.radius = radius;  // в отличие от C++ и Java
    this.area = function(){
        return 3.14*radius*radius;
    }
  }
  ...
}
```

В результате получающиеся объекты будут почти одинаковыми, но могут быть и разными, если это зачем-то нужно.

Методы класса и методы экземпляра

Если в языке есть понятие класса, на сам класс тоже можно смотреть как на экземпляр какого-то класса, представляющего собой классы программы. И у этого класса тоже могут быть свойства и методы.

```
class Rectangle:
    def __init__(self,X,Y,H,W):
        self.coord = (X,Y)
        self.height = H
        self.width = W

    def print(self):
        print("Rectangle "+str(self.width)+"x"+str(self.height))

    @classmethod
```

```
    def square(cls,X,Y,S):
        return cls(X,Y,S,S)

r1 = Rectangle(10,10,50,80)      #создает прямоугольник 50x80
r2 = Rectangle.square(10,10,50)  #создает квадрат 50x80

r1.print()
r2.print()
```

Методы класса, как правило, используются или для альтернативного конструкторам создания экземпляров класса, или для работы со свойствами класса. Здесь есть прямая аналогия с глобальными и локальными переменными в функциях и процедурах, но с инкапсуляцией этих переменных внутрь класса. На глобальном уровне остается только имя класса, все объекты имеют доступ к переменным класса как к "частным глобальным" сущностям.

Виртуальные и невиртуальные методы

Другая сторона особенностей реализации ООП в языках L-типа состоит в том, что когда компилятору заранее известен класс объекта, можно не выяснять, экземпляр какого класса в действительности лежит по ссылке, и вызвать тот метод, который описан в декларации класса. Так как разработчики **C++** стремились прежде всего к эффективности и скорости, этот способ в языке **C++** используется по умолчанию. Например:

```
class Base {
public:
    int x;
    void print(){
        cout << "I am base class\n" << endl;
    }
};

class Derived: public Base {
    void print(){
        cout << "I am derived class\n" << endl;
    }
};


...
   Base *b = new Derived();
   b->print();
...
```

В данном случае вызов `b.print()` приведет к печати строки **"I am base class"**. Для того, чтобы был вызван метод того класса, объект которого в действительности лежит по ссылке, надо в описании метода добавить слово `virtual`. Эта особенность **C++** доставляет много проблем программистам, которые из обычных руководств по этому языку выносят представление о том, что методы, описанные

как `virtual`, это какие-то особенные методы, тогда как в действительности все обстоит наоборот, виртуальные методы **C++** это с точки зрения большинства объектно-ориентированных языков, совершенно обычные, стандартные методы, тогда как не-виртуальные методы **C++** это особенность языка, нужная для повышения производительности.

Множественное наследование

Другая особенность C++, вызывающая много проблем, это так называемое множественное наследование. Класс в C++ можно произвести сразу от нескольких базовых классов. Производный класс в этом случае наследует всю функциональность тех классов, от которых его произвели. Проблемы появляются тогда, когда имена каких-то методов в базовых классах совпадают. Тогда непонятно, от какого класса брать метод. Тем более непонятно становится, когда метод базового класса перекрывается в наследнике. Еще интереснее проблема становится, когда базовые классы, от которых произведен данный, сами являются наследниками одного класса.

```
class A{
public:
    virtual void print(){
         cout << "I am A\n" << endl;
    }
};

class B: public A{
public:
    virtual void print(){
         cout << "I am B\n" << endl;
    }
};

class C: public A{
public:
    virtual void print(){
         cout << "I am C\n" << endl;
    }
};

class D: public B,C{

};

...
    A* d = new D();
    d->print();
...
```

Попытка скомпилировать эту программу приводит к ошибке, компилятор не может решить, какой метод использовать. Если добавить в D добавить метод `print`, все будет интереснее. Строка

```
A* d = new D();
```

не будет скомпилирована, а строка

```
D* d = new D();
```

будет работать. Как же так? Мы уже привыкли, что ссылки на объекты базового класса совместимы со ссылками на объекты производных классов.

Теоретические дискуссии на эту тему привели к тому, что создатели языков программирования стали опасаться множественного наследования, а отсутствие такового считается преимуществом языка.

Множественного наследования нет в Java, C#, Ruby и некоторых других языках. Создатели Java утверждают, что любая программа, написанная с использованием множественного наследования, может быть переписана с использованием интерфейсов.

Мультиметоды

Иногда объекты устроены так, что непонятно, к какому классу должен относиться метод. Например, во многих библиотеках для работы со строками есть функция join, которая из массива строк делает строку, соединяя их с помощью разделителей. В языке Javascript это метод класса Array:

```
var letters = ["Alpha", "Beta", "Gamma", "Delta"];
var alphabet = letters.join(","); // "Alpha,Beta,Gamma,Delta"
```

А в языке Python это метод класса String

```
letters = ["Alpha", "Beta", "Gamma", "Delta"]
alphabet = ",".join(letters)  # "Alpha,Beta,Gamma,Delta"
```

Решением этого вопроса могут быть мультиметоды, то есть такие описания методов, которые позволяют им как бы принадлежать двум классам одновременно. Однако, при всей привлекательности такой идеи, результат более всего напоминает вызов обычных перегруженных функций с параметрами, соответствующих типов.

Примеси (миксины)

При использовании интерфейсов каждый класс, включающий в свой состав интерфейс, обязан реализовать функциональность, требуемую этим интерфейсом. Но иногда интерфейс класса устроен так, что реализация его достаточно очевидна, и, более того, одинакова для разных классов. Предположим, что для каких-то целей мы решили пронумеровать некоторые объекты в программе так, чтобы каждый объект "знал" свой номер. Причем эти объекты могут относиться к разным классам. Реализация этого интерфейса простая, нужно добавить свойство вида "целое число" в каждый объект и написать метод getNumber. Но тут возникает проблема: у каждого объекта будет свой код для getNumber, повторенный много раз. Другое решение - реализовать getNumber достаточно рано в иерархии классов, тогда код будет один на всех, но номера получат все производные объекты, возможно, даже те, которые мы и не собирались нумеровать.

Решение этой проблемы называется mixin ("примесь"), и состоит в возможности написать интерфейс вместе с реализацией и включать это все в какие-то классы. Это похоже на множественное наследование, но, возможно, выглядит не так страшно.

Шаблоны проектирования

Самая главная особенность ООП это наличие очень развитой теоретической базы. На тему того, как лучше организовать взаимодействие между объектами и как реализовывать сами объекты, написано много книг. Одна из самых известных - книга "**Приёмы объектно-ориентированного проектирования. Паттерны проектирования**", которую написали Эрих Гамма, Ричард Хелм, Ральф Джонсон, и Джон Влиссидес, они же «Банда четырёх».

В этой книге разбираются 23 так называемых «шаблона проектирования», то есть типовые решения для часто встречающихся задач. Рассмотрим два таких шаблона. Первый называется "Singleton" или Одиночка. Это глобальный объект, который может существовать в единственном экземпляре. Пример - класс, осуществляющий запись журнала событий или класс, реализующий глобальное создание уникальных номеров. Обычно для реализации такого класса используется либо специальная глобальная функция, при первом вызове создающая экземпляр класса, а при всех последующих возвращающая ссылки на этот единственный экземпляр, либо методы класса с подобной же функциональностью. Основная задача тут - запретить создание экземпляров класса всем, кроме этой единственной функции.

```
public class Singleton {
  private static Singleton instance;    // свойство instance и конструктор
объявлены private
  private Singleton () {}               // ими могут пользоваться только
методы этого класса

  public static Singleton getInstance() {  // все остальные могут
использовать метод getInstance
    if (instance == null) {                // который при первом вызове
сконструирует объект
      instance = new Singleton();
    }
    return instance;                       // и возвратит его в качестве
результата
  }
}
```

Принципы проектирования

Наряду с шаблонами проектирования, которые относятся к внутренним особенностям разработки программ, существуют наборы более общих принципов проектирования. Один из таких наборов называется **SOLID**, по первым буквам его составляющих.

Single Responsibility Principle - Принцип единственной ответственности. Это означает, что каждый класс должен выполнять только одну какую-то задачу. Это, наверное, самый простой принцип. Не надо делать класс, который будет делать вычисления и отображать результаты на экране. Лучше сделать

два класса. Заметим, что многие системы быстрой разработки пользовательского интерфейса, такие как Delphi, сильно провоцируют нарушение принципа единственной ответственности. Когда система автоматически создает для программиста класс, представляющий собой окно программы, и в нем создает обработчик нажатия на кнопку "прочитать файл", естественно будет "научить" этот же класс читать файлы с диска. В итоге это приводит к коду, который сложно поддерживать и сложно развивать.

Open Closed Principle - Принцип открытости/закрытости. Это буквально значит следующее: «программные сущности должны быть открыты для расширения, но закрыты для модификации.» Если у вас есть уже работающий набор классов, и вам надо добавить в него дополнительную функциональность, то лучше делать это путем создания производных классов, а не созданием новой версии базового набора.

Liskov Substitution Principle - «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» Назван в честь Барбары Лисков. Идея состоит в том, что производный класс должен дополнять, а не изменять базовый. В определенном смысле этот вопрос касается терминологии. Если в базовом классе есть, например, метод `print`, служащий для печати сообщений, то в производном классе метод с именем `print` не должен, например, готовить к работе принтер.

Interface Segregation Principle - Принцип разделения интерфейса. «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.» Другая формулировка это же принципа такова: при изменении метода класса не должны меняться программные сущности, которые этот метод не используют.

Dependency Inversion Principle - Принцип инверсии зависимостей. Идея состоит в том, что абстракции, специально придуманные для разработки программ, не должны зависеть от особенностей реализации. Также сущности верхнего уровня должны зависеть не от реализации сущностей нижнего уровня, а от абстракций.

Строки

Строковые и текстовые данные

В первых языках программирования этому предмету не уделялось достаточно большое внимание. Считалось, что компьютеры нужны прежде всего для вычислений, а строкам отводилась скромная роль оформления результата при печати. В языке Fortran первой версии строки выглядели совсем непривычно:

Fortran:

```
100  FORMAT (1HHELLO WORLD)
```

Странная конструкция `11HHELLO WORLD` означает следующее: строковая (первое H) константа длиной 11 символов с текстом HELLO WORLD.

Таким образом, программисту приходилось каждый раз вручную считать количество символов в строке. Если оно не совпадало с имеющимся, компилятор, пересчитав фактически имеющиеся

символы, выдавал ошибку. Естественно было переложить обязанность подсчета символов на компилятор. Первая идея состояла в том, чтобы заключать строковую константу в кавычки:

Pascal

```
s := 'Hello world';
```

C++

```
s = "Hello world";
```

При этом естественным образом возникала проблема: как быть, если в строке должна появиться кавычка? Первое решение этого вопроса было не совсем изящным, но работающим

C++

```
s = "I will say \"Hello, world\"";
```

Обратная косая черта, поставленная перед кавычкой, как бы экранирует ее, превращая в обычный символ. Читается это плохо, приводит к ошибкам, поэтому в некоторых языках ввели два разных символа для ограничения строк.

PHP

```
$s1 = "I will say 'Hello, world'";
$s2 = 'I will not say "Hello, world"';
```

Так уже лучше, но что, если в строке должны встретиться и одинарные и двойные кавычки? Решение, предложенное в языке Python — использование **тройных** кавычек:

Python

```
s = """I'll say "Hello, world" """
```

Все, вроде бы, хорошо, но обратите внимание на пробел в конце строки. Если его не будет, компилятор выдаст ошибку, потому что увидит **четыре** кавычки подряд. И, все-таки, что, если в строке должны появиться три кавычки подряд? Решение называется heredoc, оно появилось впервые в языке командной оболочки Unix, после было перенесено в язык Perl и другие языки. Выглядит оно так:

Perl

```
$s = <<END;
I will say anything ' ''  """, ", ""
Yes, anything at all.
Except for
END
```

Идея состоит в том, что в первой строке между << и ; записывается произвольная строка (в данном случае это END), которая *не должна* встречаться в том тексте, который нам нужно поместить в строковую константу, конец которой определяется этим же сочетанием символов. айти строку, которой нет внутри нашей константы всегда возможно. В языке Lua это делается несколько изящнее:

Lua

```
  s = [===[Any text]===]
```

Ограничивающие строку конструкции формируются из двойных квадратных скобок и знаков = между ними в произвольном количестве (в том числе и нулевом). Такой код читается легче, чем обычный heredoc.

Операции со строками

Раз уж мы научились формировать строковые константы, давайте посмотрим, что можно со строками делать. Первая операция называется конкатенация, или сцепление строк, и встречается почти во всех языках, где есть строки. В Javascript, например, она обозначается, как и сложение, знаком +

Javascript:

```
a = 'Hello';
b = a + ' world';
```

В некоторых языках для сцепления строк используется специально выделенный знак операции. В PHP это точка, в Lua две точки подряд (a..b). У каждого из способов есть определенные преимущества и недостатки. Если это знак +, то, с одной стороны, программисту не нужно запоминать еще один знак операции (о котором, в отличие от математических символов, договориться не удалось), а с другой, если в выражении смешиваются числовые и символьные данные, результат может быть не тот, который ожидается.

Javascript:

```
a = 'Number';
b = 3;
c = a + b; // Number3
d = b + a; // 3Number
e = a + 3 + b; // Number33
f = 3 + b + a; // 6Number
```

Поведение программы в случае использования специального оператора сцепления строк более предсказуемо, даже если аргументами являются числа, этот оператор предполагает, что они будут представлены в виде строк или будет выдана ошибка.

Это, кстати, еще две операции со строками — представление числа в виде строки и превращение строки в число. Разница между этими операциями в том, что если первая возможна всегда, то при выполнении второй могут возникнуть ошибки, которые придется как-то обрабатывать. Самый простой и одновременно неудобный способ используется в языке C. Функция `atoi` возвращает число, соответствующее переданной строке, но если интерпретировать строку как число не удалось, поведение ее не определено. Обычно функция возвращает 0. Таким образом, отличить ошибку от строки "0" невозможно.

Сравнение строк

Сортировка по алфавиту является одной из типовых задач, так что операции «равно», «меньше» и «больше» для строк были бы полезны. Другое дело, что две последние операции в случае текстов на естественных языках представляют собой сложную и иногда неразрешимую задачу, поскольку в разных языках одни и те же буквы (их числовые представления) могут находиться на разных местах в алфавите.

Точное сравнение в смысле совпадения битовых представлений строк намного проще, если, конечно, для представления символов не используется Unicode.

Регулярные выражения

Кроме полного сравнения строк есть еще так называемые *регулярные выражения*, или сравнения строки с образцом, тоже заданным в виде строки специального вида. И в некоторых языках регулярные выражения встроены в синтаксис.

```
$x = 'a';

if ($x =~ m/[abc]/){
    print "yes"
}
```

Регулярное выражение /[abc]/ сопоставляется со строкой, состоящей из одного символа, который может быть 'a', 'b' или 'c'. Для того, чтобы задать это выражение в программе, используются специальные ограничители / / вместо кавычек.

Особенности реализации строк

В целом к строкам в языке есть три различных подхода. Первый применяется, например, в языке C, где строки это просто массивы символов. Второй состоит в создании специального типа данных, как это сделано в языках Pascal и Java. Третий способ используется в языках с развитыми объектно-ориентированными возможностями. В языке C++ строка `string` это с точки зрения языка обычный

объект со всей необходимой функциональностью. При этом ничто не мешает создать свой собственный класс объектов, которые будут обеспечивать тот же (или другой) набор операций.

На нижнем уровне используется два основных способа представления строк. Это "строка со счетчиком" и "строка с нулем на конце". В первом случае выделяется память под символы строки и дополнительно под целое число, в котором хранится количество символов. Это число находится в памяти непосредственно перед символами строки, так что имея адрес этой структуры, можно сразу же получить длину строки.

Во втором случае символ со значением 0 используется для обозначения конца строки. Оба способа имеют свои недостатки. В случае строки со счетчиком необходимо при разработке языка или компилятора принять решение о том, какой максимальной длины может быть строка. В случае строки с финальным нулем проблемы возникают уже при выполнении программы. Дело в том, что определить длину строки можно только просмотрев ее всю и найдя тот самый конечный 0. При этом алгоритм операции сцепления строк состоит в выяснении длины строк-операндов, выделении памяти под результат и копирования строк в эту память.

Из-за того, что выделение памяти это обычно довольно долгая процедура, в некоторых стандартных библиотеках языков программирования делают специальную конструкцию для быстрого сцепления множества строк.

```
StringBuilder b = new StringBuilder();
for (int i = 0; i < 10000; ++i){
    b.Append("something");              // добавляем много строк
}
string s = b.ToString();
```

Объект StringBuilder не сцепляет строки и не выделяет память при добавлении новых строк. Он просто запоминает указатели на эти строки и суммирует их длины. Только при вызове ToString происходит фактическое создание строки-результата.

Языки программирования, ориентированные на работу со строками

Строки - наиболее универсальный тип данных. Поскольку в большинстве языков программирования программа это строка символов, то все, с чем умеет работать язык программирования, представимо в виде строки символов. Неудивительно, что во многих языках программирования строки - основной, а иногда и единственный тип данных. Пионером тут выступил в 1962 году язык Snobol (StriNg Oriented and symBOlic Language), в котором исходно был всего один тип данных. Впоследствии, в версии Snobol4 добавили и другие. Сам по себе язык Snobol особой популярностью не пользовался, в основном из-за очень непривычного синтаксиса, но оказал влияние как на регулярные выражения, так и на логические языки. Последнее может показаться странным, но алгоритм сравнения строки с образцом в языке Snobol в основном совпадает с алгоритмом логического вывода в языке Prolog.

Универсальность строк как способа представления данных также используется при организации взаимодействия между программами и в различных системах ввода-вывода и хранения данных. Если заранее неизвестно, на какой машине будут записываться или считываться данные, и по каким-то причинам возможна несовместимость между представлением данных на разных машинах, можно

попробовать представить данные в виде строк. Здесь тоже возможны проблемы, например, из-за того, что в зависимости от системных настроек число "сто тысяч и две десятых" может быть представлено как `100000.2`, как `100,000.2` или как `100000,2`.

Ввод-вывод

> *Обычно ввод-вывод — наименее удовлетворительная часть языка. возможно, потому, что здесь разработчик не может полностью игнорировать реальный мир.*
>
> Д.Баррон

Идеология ввода-вывода, как и идеология языка программирования, может определяться железом или моделью системы ввода-вывода. С некоторых пор большинство операционных систем используют примерно одинаковые модели воода-вывода, а именно производные от той модели, которая когда-то сложилась в системе UNIX. В этой модели есть два основных вида "объектов внешнего мира" - файлы и потоковые устройства. Однако, в более древних операционных системах ввод-вывод был устроен иначе, например, система могла выделять пользователю фиксированный участок жесткого диска с адресами, заданными в виде чисел. На этом участке программист должен был организовываать свои файлы.

В ранних языках программирования ввод-вывод пытались оформить в виде операторов языка, однако, скоро стало ясно, что возможности операционных систем меняются, и существующие возможности ввода-вывода перестают им соответствовать.

Современный подход к системе ввода-вывода языка состоит в том, чтобы не делать ее вообще, а возложить эти функции на какую-то библиотеку. Некоторая ирония, возможно, есть в том, что возможности ввода-вывода у современных ОС примерно одинаковые.

Тем не менее, функции ввода-вывода обычно входят в описание стандарта языка.

Форматный вывод

Форматный или строковый вывод составляет большую часть всего ввода-вывода. Идея состоит в том, чтобы преобразовать любое выводимое значение в строку, которая потом выводится на экран или записывается в файл.

Наибольшее разнообразие подходов тут обнаруживается в деле преобразования чисел в строки. Для этого обычно применяется строковое же заклинание под названием "спецификация формата". Она может быть выражена количеством знаков, отводимых под целую и дробную части вещественного числа, например, `F10.2` (FORTRAN) или `%10.2f` (C). Это также может быть шаблон, `#####.##` как в языке Basic. C-шный способ форматирования чисел получил широкое распространение и используется, например, в языке Python

```
print("%10.3f"% (123.456))
```

напечатает " 123.456" - строку длиной 10 символов, из которых 3 будут отведены на дробную часть числа. Все эти форматы были придуманы много лет назад в языке Fortran и в основном рассчитаны на то, чтобы красиво вывести числа в колонки *одинаковой ширины* на бумагу или экран. При этом использовались принтеры или терминалы, у которых все буквы были тоже *одинаковой ширины*. В наше время такие устройства используются очень редко. Как правило, нужно форматировать текст в соответствии с используемым шрифтом, а эта функциональность зависит от операционной системы,

экрана и много чего еще. Ее невозможно уместить в одну функцию и одну спецификацию формата для каждого числа.

Двоичный вывод

Двоичный вывод это запись в файл данных в том виде, в котором они хранятся в памяти. Результат такого вывода зависит от представления чисел в машине и данные, записанные таким образом, могут быть не переносимы на другую машину. Дело в том, что в разных процессорах целые числа, занимающие больше одного байта, могут храниться в памяти по-разному. Различают два основных способа записи таких чисел: big-endian и little-endian.

Предположим, у нас есть число 22222, и оно занимет два байта памяти. 22222 = 86*256+206, так что в одном байте будет храниться число 86, а в другом число 206. В памяти эти числа будут располагаться по соседним адресам: в случае big-endian сначала 206, потом 86, а в случае little-endian сначала 86, потом 206. Если мы запишем число 22222 на машине с одним порядком байт, а прочитаем на машине с другим порядком, то получим 52822.

Двоичный вывод обычно используется для хранения временных или локальных данных, а также в случае необходимости обеспечить совместимость с каким-то оборудованием, например, принтером.

Сериализация

Кроме простой возможности записывать байты в файл, многие системные библиотеки языков программирования предоставляют возможность так называемой *сериализации*, или систематического, связанного с языком и реализацией, способа записать на диск данные с возможностью их последующего восстановления.

Эта возможность восстановить объекты языка такими, какими они были перед записью - как раз то, что не гарантирует простая двоичная запись. Объект может содержать в своем составе адреса памяти, служебные данные и что-то еще, что может изменяться при каждом запуске программы. Фактически, из двоичной записи можно непосредственно востанавливать только очень простые данные: числа, массивы чисел и структуры, в которых есть те же числа, массивы и структуры. Сериализация гарантирует, что более сложные сущности, такие как списки, последовательности или словари, тоже будут корректно восстановлены из записи.

Сериализация - процесс, непрозрачный для программиста, последовательность байт, записанная на диск, может изменяться при изменении версий языка или в зависимости от особенностей машины, на которой она производится. Единственная базовая возможность сериализации это сохранение данных в файле и восстановление данных той же программой. В конкретных реализациях в библиотеках разных языков программирования возможны какие-то дополнительные удобства вроде гарантии восстановления данных независимо от версии или особенностей машины.

XML

Язык XML, не являющийся языком программирования, представляет собой противоположность сериализации, хотя сериализация в XML тоже бывает. Основная идея XML - создать обобщенный формат хранения любых данных, причем запись этих данных должна быть понятна для человека. Пример:

```
```

```
<letter>
    <to>Alex</to>
    <from>Justas</from>
    <heading>Bug in your code</heading>
    <body>Please fix it.</body>
</letter>
```

Смысл этой записи более или менее понятен, даже если не знать, какая программа должна работать с такими данными. Объект `letter` содержит в своем составе поля `to`, `from`, `header` и `body`. На практике чаще используют другую возможность XML - атрибуты.

```
<letter to="Alex" from="Justas" heading = "Bug in your code">
    Please fix it.
</letter>
```

Сам по себе стандарт XML накладывает очень мало ограничений на данные и совсем никак не регулирует способы представления объектов языков программирования. "Правильный" XML должен начинаться со специальной строки вида

```
<?xml version="1.0" encoding="UTF-8"?>
```

и остальная часть файла должна содержать ровно один раздел верхнего уровня:

Нельзя

```
<?xml version="1.0" encoding="UTF-8"?>
<letter> ... </letter>
<letter> ... </letter>
<letter> ... </letter>
<letter> ... </letter>
```

Можно

```
<?xml version="1.0" encoding="UTF-8"?>
<letters>
    <letter> ... </letter>
    <letter> ... </letter>
    <letter> ... </letter>
    <letter> ... </letter>
</letters>
```

Остается один вопрос: а что вообще может быть внутри `letter`? Какие поля там обязательны, какие можно пропустить, а каких там не должно быть вообще. Для решения этой задачи применяются две основные технологии, которые называются DTD и XML Schema. Это тоже такие языки (XML Schema построена на основе XML), с помощью которых можно описать структуру файла XML.

Главные достоинства XML состоят в том, что он относительно легко может быть понят человеком, а также что для него написано великое множество разнообразных программ и библиотек. Кроме того, это стандарт, поддерживаемый организацией W3C, и любую программу можно проверить на соответствие этому стандарту.

JSON

В отличие от XML, формат под названием JSON скорее является стандартом *de facto*, то есть большинство реализаций этого формата вам (почти) ничего не гарантируют. Возникновение этого формата связано с особенностью языка Javascript, в системной библиотеке которого есть функция `eval`, вычисляющая значение своего аргумента, заданного в виде программы на Javascript. JSON в базовом варианте это просто запись данных на языке Javascript:

```
[
    {
        from:"Alex",
        to:"Justas",
        heading:"Bug report"
    },
    {
        from:"Justas",
        to:"Alex",
        heading:"Re:Bug report"
    }
]
```

В JSON есть числа, строки, массивы и объекты, и в Javascript для чтения JSON требуется ровно одна строка кода:

```
eval('v='+json);
```

Недостаток такого, слишком прямолинейного, подхода в том, что вместо данных в строке может быть вообще что угодно, в том числе и вредоносный код. Поэтому в реализации Javascript были добавлены функции для работы с JSON, которые хоть что-то проверяют. Это также привело к введению некоторых ограничений на формат, например, приведенный выше JSON такими функциями прочитан не будет, им надо, чтобы имена полей также были заключены в кавычки:

```
[
    {
        "from":"Alex",
        "to":"Justas",
```

```
        "heading":"Bug report"
    },
    {
        "from":"Justas",
        "to":"Alex",
        "heading":"Re:Bug report"
    }
]
```

JSON - более компактный формат, чем XML, в нем есть типы данных, и нет той неоднозначности, которая изначально была заложена в XML. Библиотеки для работы с JSON написаны почти для всех современных языков программирования.

Базы данных

Базы данных дают возможность хранить большие объемы данных при сохранении независимости от системы. Для этого все операции по записи и чтению данных выносятся в отдельный процесс, а между ним и программой вводится промежуточный язык, называемый языком запросов, на котором происходит общение между программой и базой данных. Программа формирует запрос на таком языке и отправляет его программе - серверу базы данных, сервер обрабатывает запрос и возвращает результат.

В этом подходе имеется очень много достоинств. Сервер базы данных можно вынести на отдельный компьютер и тем повысить производительность. Сервер обычно умеет обслуживать сразу много программ - клиентов. Типичный сценарий работы в такой архитектуре состоит в том, что программы-клиенты часто запрашивают примерно одни и те же данные, и сервер может держать эти данные в оперативной памяти, ускоряя обслуживание.

Но есть и недостатки. Главный из них - наличие специального языка, на котором программа общается с сервером. Проблема тут не в том, что программист должен знать уже минимум два языка, и даже не в том, что ему приходится писать на двух языках одновременно, а в том, что предложения на языке запросов должны автоматически генерироваться программой. Например, мы хотим узнать число сотрудников с зарплатой более 100000 денежных единиц. Для этого мы пишем следующий запрос:

```
select count(*) from employee where salary > 100000
```

Теперь попробуем сформулировать более общий вопрос: найти число сотрудников с зарплатой более N денежных единиц. Для этого нам придется сконструировать запрос в нашей программе:

```
n = 100100
s = "select count(*) from employee where salary > " + str(n)
```

О наличии ошибки в сгенерированной строке запрса мы узнаем, только отправив запрос на сервер, или, возможно, не узнаем никогда. Если, например, вместо числа **100100** в переменную n каким-то образом попадет строка

`"0; drop table employee;"`

то при неправильной организации базы данных все данные о сотрудниках будут уничтожены. Этот подход называется **SQL injection**, и является одним из основных инструментов хакеров. Сложности с созданием запросов и опасности для инфраструктуры привели к тому, что в большинстве библиотек для работы с базами данных используются специальные средства для "правильного" создания запросов. При этом проверяется все, что возможно, но **SQL injection** искоренить не удалось.

Другой способ организации баз данных называется `No SQL`, и он предполагает написание запросов на том же языке, на котором пишется основная программа.

```
db.employee.find(
    {"salary": {"$gt": 100000}}
)
```

Здесь, как можно видеть, тоже есть некий своеобразный язык запросов. Он полностью интегрирован в Python, это хорошо с точки зрения надежности кода, но плохо потому, что на другом языке программирования это же запрос может выглядеть не совсем узнаваемо. Вот тот же запрос на Java:

```
findIterable = collection.find(gt("salary", 100000));
```

У баз данных, построенных по принципу `No SQL` есть еще много разных достоинств и недостатков, но они не имеют отношения к языкам программирования.

Графический интерфейс пользователя

Аварийные ситуации

При вычислении выражений основной аварийными ситуациями могут быть деление на 0, извлечение корня из отрицательного числа и подобные проблемы. Их не так много, и их всегда можно предусмотреть, проверяя аргументы функций. Но в случае ввода-вывода аварийные ситуации бывают намного более сложными. Предположим, программа пытается прочитать файл на диске. При этом нужно рассматривать следующие возможности: файл отсутствует, диск поврежден и файл не читается, у программы нет прав доступа к файлу, файл занят другим процессом, достигнут конец файла, итд. Реакция программы на такие ошибки должна быть различной.

Заранее проверить все возможности было бы слишком сложно. Кроме того, при взаимодействии с окружающим миром ситуация может меняться, и проверка сделанная сейчас может быть недействительна уже через секунду. Ошибки ввода вывода (как и любые ошибки программы) надо обрабатывать. Для этого придумали несколько основных методов.

Первый метод : проверка значения

```
`err = fread(b,size,n,F);`
```

в случае ошибки или конца файла функция чтения возвращает 0. Получив такой ответ, можно вызвать функцию (ferror) чтобы узнать точнее о том, какая ошибка произошла.

В некоторых языках (стандартных библиотеках) есть специальная переменная или функция, значение которой нужно проверять после каждой операции.

Недостатки такого способа очевидны: программа превращается в набор `if`-ов, проверяющих всевозможные варианты ошибок.

```
if файл открылся:
    if данные прочитаны:
       ...
       закрыть файл
    else:
       # что делать, если файл открыт, а данные не читаются
       закрыть файл
else:
    # файл не открылся
```

В той части программы, где все идет хорошо, логика обычно бывает простая, но исследование вопроса о том, что надо делать в случае ошибки, во-первых, сложно, во-вторых зависит от того состояния, в которое программа пришла, сделав безуспешную попытку открыть файл.

Второй метод : GOTO

Уже в языке Fortran его создатели подумали о необходимости обрабатывать ошибки ввода-вывода. Для этого они, наряду с проверкой значения, использовали вариацию на тему GOTO.

Оператор

```
READ(5,100,END=30,ERR=40)
```

пробует читать данные, и производит безусловный переход (GOTO) на метку 30 в случае, если произошла попытка читать данные за концом файла и на метку 40, если произошла какая-то еще ошибка.

Подобный подход был использован в языке Basic, но не во всех его диалектах можно было указывать тип ошибки. Обычно предлагалось писать что-то вроде

```
ON ERROR GOTO 230
```

После выполнения такой строки возникновение **любой** исключительной ситуации приводило к передаче управления на строку 230, а там уже надо было разбираться, что именно произошло. Отдельный вопрос, касающийся такого способа обработки ошибок, состоит в том, что должна делать программа, если после перехода вследствие ошибки произойдет еще одна программная ошибка.

Третий метод : вызов процедуры или функции в случае ошибки

Этот способ похож на предыдущий, только вместо `GOTO` происходит вызов подпрограммы. Это решение лучше в том смысле, что происходит локализация обработки ошибок, одна и та же подпрограмма может обрабатывать ошибки, возникающие в разных местах основной программы, и возможна более детальная раскладка ошибок по обработчикам. Например, в библиотеке jQuery, написанной на Javascript, можно задать обработчики для успешного и для неуспешного выполнения запроса:

```
$.ajax({
  url: "test.html",
  success: function1,
  error: function2
});
```

Функция function2 будет вызвана при возникновении любой ошибки. Однако, можно сделать индивидуальные обработчики для различных ошибочных ситуаций:

```
$.ajax({
  url: "test.html",
  success: function1,
  statusCode: {
    404: function3,
    500: function4
  }
});
```

Четвертый метод : предварительный вызов функции

Эта функция должна определить, можно ли выполнить операцию безопасно. В некоторых API есть возможность проверить, не возникнет ли ошибка при выполнении какой-либо операции.

```
if (can_write(f,s))
    write(f,s);
else
   //  обрабатываем ошибку
```

Преимущество такого способа обработки ошибок в том, что, во-первых, код обработки ошибки упрощается, и во-вторых, при отрицательном ответе от проверяющей функции вся система находится (по крайней мере должна находиться) в предсказуемом состоянии, а именно в том, в каком она находилась до проверки. Это следует из общей идеи об отсутствии побочных эффектов у функций. Однако, иногда создатели операционных систем экономят на количестве вызовов и делают одну и ту же процедуру для проверки и для самой операции.

Все эти методы обладают одним общим основным недостатком: они сигнализируют о том, что что-то не так, но не предполагают никаких методик написания программы, которая может работать в условиях возникновения ошибок. Если в результате ошибки ввода-вывода произошел переход к какому-то месту

программы, то что должен делать программист? Допустим, все это происходит внутри какой-то функции. Тогда можно, например, вернуть код ошибки. То есть превратить первый способ во второй.

Хорошо бы обработку ошибок сделать более предсказуемым действием. Для этого придуман

Пятый метод : перехват ошибок при помощи try — catch

```
try
{
    read(...);
}
catch(Exception e)
{
    // что делаем в случае ошибки
}
```

или

```
file = open(...);
try
{
    read(file,...);
}
finally
{
    close(file);
}
```

В случае возникновения ошибки при выполнении read в первом фрагменте управление будет передано ветке `catch`, во втором фрагменте - ветке `finally`. Часть `finally` будет выполнена как в случае ошибки, так и в случае нормального завершения операций.

У этого способа есть множество преимуществ. Во-первых, логика обработки ошибок всегда одна и та же. Во-вторых, начиная обрабатывать ошибку, программист уже имеет о ней дополнительную информацию, и в некоторых языках можно для каждой из возможных ошибок писать свой catch. И в третьих, легко передать информацию об ошибке на верхний уровень.

Представим себе типичную ситуацию: программа читает файл с какими-то сложными структурированными данными. При этом есть какая-то процедура нижнего уровня, которая просто читает байты, что-то с ними делает и передает прочитанное вызвавшей ее процедуре. У процедуры нижнего уровня нет никакой информации о том, что именно программа делает в данный момент и она не может принять решение о том, можно ли проигнорировать неправильно прочитанные данные, или можно ли их исправить или надо сообщить об ошибке пользователю.

Технология обработки исключений состоит в том, что если исключительная ситуация не обработана при помощи try-catch или try-finally на каком-то уровне вложенности вызовов процедур, то выполнение прекращается и исключение передается на более высокий уровень.

В языке Java разработчики пошли еще дальше и обязывают программиста описать в явном виде, собирается он обрабатывать ошибку или передавать ее дальше.

```
public int myDivide(int a,int b) throws ArithmeticException
{
   return a/b;  // возможно возникновение ошибки
}

public int callMyDivide1(){     // ошибка компиляции
    system.out(myDivide(2,0));
}

public int callMyDivide2 throws ArithmeticException() {     // так можно,
передаем исключение выше
    system.out(myDivide(2,0));
}

public int callMyDivide1(){     // так тоже можно, обрабатываем ошибку сами
    try{
        System.out.println(myDivide(2,0));
    }
    catch (ArithmeticException e) {
        System.out.println("ошибка вышла");
    }
}
```

Недостаток такого подхода в том, что обработка исключительных ситуаций делегируется вызывающей программе, которая в общем случае может и не знать о проблемах, локальных для вызываемой процедуры. Например, в рассмотренном случае процедуры высокого уровня, читающей структурированные данные и процедуры нижнего уровня, читающей байты, последняя может столкнуться с проблемами самого разного вида, например, со сбоем на диске, и если информация об исключительной ситуации передается наверх, процедура высокого уровня должна уметь обрабатывать все множество низкоуровневых ошибок. Для этого случая предусматривается оператор throw, который позволяет повторить обработку исключения на верхнем уровне. Например:

```
try
{
    read_bytes(...);
}
catch(Exception e)
{
    if(...) // мы можем обработать исключение здесь
      process_exception(); // обработать его
    else
      throw(e);
}
```

Оператор `throw` можно также использовать для создания собственных типов исключительных ситуаций. Обычно в языке либо предусматривают объектный тип `Exception`, который можно расширять путем наследования, либо, как это сделано в C++, допускают `throw` с любым типом аргумента.

Шестой метод - использование специальных типов

Если язык программирования, на котором пишется такая обработка ошибок, относится к R-типу, можно просто возвращать из функции, осуществляющей ввод-вывод, какое-то нужное значение, в случае успеха и значение другого типа в случае ошибки. Например:

```
function myDivide(a, b)
{
    if(b == 0)
        return "Ошибка: деление на 0";
    else
        return a/b;
}

$a = myDivide(3,0);
if(is_string($a))
    print($a);

// или просто

print( myDivide(4,0));
```

Для удобства и единообразия можно сделать объектную "обертку" для возвращаемого значения и даже организовать автоматическое преобразование этой обертки к нужному типу. Здесь тоже работает принцип "реши проблему сам, если можешь или передай ее на верхний уровень"

Одновременное выполнение программ

> *Все, начинающие изучать параллельное программирование, думают, что они его понимают. Потом они сталкиваются с ситуациями, которые, как они полагали, в принципе не могут возникнуть, и в итоге приходят к пониманию того факта, что они никогда не понимали, с чем имеют дело*
>
> *Herb Sutter*

Как мы уже говорили, некоторые участки программ могут выполняться параллельно. Возможность одновременного выполнения нескольких программ появилась довольно давно, и появилась она в связи с тем, что значительную часть времени обычной вычислительной программы занимает ввод-вывод. Пока результат распечатывается или записывается на диск, процессор может считать что-то еще, а процессорное время стоило дорого. Поэтому были придуманы операционные системы, способные выполнять несколько программ одновременно. После появления компьютеров с несколькими процессорами появилась и другая задача: разделить программу на участки, которые могли бы выполняться одновременно на нескольких процессорах. В ходе решения этих задач было придумано несколько концепций параллельной обработки данных.

Сопрограммы

Это амый простой, но одновременно неочевидный по синтаксису способ организации параллельных вычислений. Подпрограммы просто передают друг другу управление в определенных точках.

```
co = coroutine.create(function ()
      for i=1,10 do
         print("co", i)
         coroutine.yield()
      end
     end)
coroutine.resume(co)
```

Это похоже на генераторы, которые мы уже рассматривали раньше, но с одним отличием: сопрограммы умеют передавать друг другу данные. При этом параллельного выполнения кода нет.

Сопрограммы используются в основном в сценариях последовательной обработки данных - "конвейерах". Представим себе две процедуры, первая из которых читает файл с данными и отбирает некоторые из них, а вторая производит какую-то обработку данных. В зависимости от алгоритмов отбора и обработки взаимодействие между этими процедурами может быть довольно сложным, и не всегда понятно, какая из процедур должна вызывать другую. Сопрограммы решают эту задачу довольно простым способом: "главной" процедуры нет, и процедура чтения передает (yield) полученные данные второй подпрограмме.

Важным свойством сопрограмм является то, что у них не существует проблемы одновременного доступа к каким-то переменным или другим ресурсам; в каждый момент времени выполняется одна какая-то процедура, которая не может быть прервана в произвольный момент времени.

Параллельные процессы

Теперь поговорим о действительно параллельных программах. Первый случай, в котором требуется выполнять какой-то код если не параллельно, то по крайней мере асинхронно, это ввод-вывод. Выполнение программы может продолжаться во время ожидания завершения какой-то операции

Квазипараллельный ввод-вывод

PL/1

```
GET LIST EVENT(Z) A,B,C
...
WAIT(Z)
```

Команда чтения переменных GET LIST создает некую сущность под названием event, причем выполнение программы продолжается без ожидания завершения ввода-вывода. Дальше програма может делать что-то еще, не связанное с переменными A,B и C, которые должна прочитать команда GET LIST, а потом подождать (WAIT) завершения чтения. Заметим, что единственной связью между GET LIST и WAIT является переменная Z. Здесь уже возникает проблема, связанная с тем, что между

GET LIST и WAIT присваивать что-то переменным A,B и C или использовать их значения было бы некорректно. Конечно, хотелось бы, чтобы компилятор мог это отследить и хотя бы предупредить программиста. Однако, наличие в языке условных операторов и GOTO делают эту задачу принципиально невыполнимой.

В современных языках такая фукциональность реализуется при помощи процедур.

```
jQuery.get( "test.html", function( data ) {
  // делаем что-то с данными
});
```

Процедуре jQuery.get передается адрес в сети, по которому надо обратиться, и функция, которую нужно вызвать, когда чтение завершится.

В этом случае функция обработки может выполняться как параллельно с какой-то другой процедурой, так и вызываться подобно сопрограмме. Во втором случае должна быть какая-то очередь выполнения, в которую ставится функции, соответствующие завершившимся действиям ввода-вывода.

Участки одной программы выполняются параллельно

В некоторых языках параллельные участки программ можно объявлять в явном виде. Среди современных языков такая возможность есть в Chapel

```
writeln("2: ### The cobegin statement ###");

cobegin {
  writeln("2: output from spawned task 1");
  writeln("2: output from spawned task 2");
}
```

Все операторы блока `cobegin` (в данном случае это вызовы функций) будут выполнены параллельно. В языках C++ и Fortran возможность параллельного выполнения была введена независимо от спецификации языка, при помощи указаний компилятору.

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Здесь `#pragma omp parallel for` означает указание компилятору распараллелить выполнение дальнейшего кода. Это указание относится не к языку C++, а к надстройке под названием OpenMP, которая использует тот факт, что в большинстве императивных языков применяются одни и те же конструкции (for, while, if), и параллельное выполнение кода можно организовать одним и тем же набором директив.

Запуск параллельного процесса в явном виде

Самый простой и слабо зависящий от языка способ распараллелить процессы это запускать их в явном виде с помощью процедуры стандартной библиотеки. Процедура fork в языке C создает копию существующего процесса.

```
pid_t pID = fork();  // создает копию данного процесса
       // выполнение копии и оригинала продолжаются
       // с одного и того же места, но результат
       // функции fork у них разный.
if (pID == 0)  // кто я?
{
     // Я - дочерний процесс.
     // Делаем что-то полезное
     // и завершаем программу
     exit(0);
}
else if (pID > 0)
{
     // Я - родительский процесс.
     // pID – идентификатор дочернего процесса
     waitpid(pID);  // ждать завершения процесса.
 }
 else
 {
     // ошибка
     exit(1);
 }
```

Процедура fork не имеет отношения к языку C, а является частью интерфейса операционной системы UNIX. Использовать этот интерфейс может программа на любом языке программирования.

Запуск параллельной процедуры (threading)

В современных операционных системах есть как минимум 2 класса процессов. Первый класс это то, что обычно называют программами, то есть процессы операционной системы. Второй - процессы внутри процесса или потоками выполнения. Отличие между этими классами в том, что процессы ОС изолированы друг от друга (мы на это надеемся), и у каждого такого процесса есть своя область памяти, в которой он работает. В отличие от них, все потоки одного процесса имеют общую область памяти и вследствие этого нужны некоторые особые меры по предотвращению конфликтов.

Запуск потоков в Java производится путем создания класса, производного от специального класса Thread:

```
public class MyThread extends Thread
{
   private int tn;  // номер потока

   public MyThread(int n) // конструктор
   {
       System.out.println("Создаем поток "+n);
       this.tn = n;
   }

   @Override
   public void run()  // то, что должно выполняться параллельно
   {
        System.out.println("Я поток " +  this.tn);
   }
}
```

Использовать этот класс можно так:

```
    MyThread t1 = new MyThread(1);
    MyThread t2 = new MyThread(2);
    MyThread t3 = new MyThread(3);

    t1.start();
    t2.start();
    t3.start();
```

Метод `start` класса `Thread` создает параллельный поток выполнения и вызывает в этом потоке метод `run`, переопределенный в нашем классе `MyThread`.

Если несколько потоков одновременно пишут или читают одну и ту же область памяти, то возможна ситуация, когда какой-то поток запишет в эту память одно число, а прочитает другое, потому что другой поток уже успел записать туда что-то свое.

Для решения этой проблемы существует много разных способов, мы рассмотрим один, который называется mutex (от mutual exclusion). Этот способ хорош прежде всего тем, что не связан ни с ООП ни с особенностями языка. Mutex это такой совершенно непрозрачный объект, он не предоставляет никаких данных. Особенности его реализации скрыты от программиста. С этим объектом можно сделать две вещи - захватить и отпустить. При этом система, реализующая многопоточность, гарантирует, что невозможно захватить mutex дважды. Код, использующий mutex, выглядит примерно так:

```
mutex m;
```

```
m.lock();
// делаем что нам надо с общей памятью, другие такие же
// функции будут ждать, пока mutex не освободится.
m.unlock();
```

Недостаток mutex в том, что он не гарантирует недоступность общей памяти для всех возможных потоков выполнения, он вообще никак не взаимодействует с общей памятью и "не знает" о ней. Поэтому если какой-то поток не содержит вызова m.lock(), то он может вызвать ту самую проблему, для решения которой используется mutex.

Другая проблема состоит в том, что если mutex занят, то вызов lock() приведет к блокированию потока выполнения до тех пор, пока mutex не освободится. Например, такая программа может выполниться, а может и зависнуть:

```
mutex m;
cout << "ready" << endl;
m.lock();
cout << "set" << endl;
m.lock();
cout << "go" << endl;
m.unlock();
m.unlock();
```

Для таких ситуаций предусматривается вызов try_lock, который пытается захватить mutex и возвращает true в случае если это удалось сделать.

В языке Python используются те же самые mutex-ы, но с другими именами методов:

```
mutex = Lock()

mutex.acquire();
# ....
mutex.release();
```

В функциональных языках программирования общая память потоков может отсутствовать, поэтому параллельное выполнение потоков выглядит намного проще. Например, в языке Alice можно написать:

```
val n = spawn factorial 35
```

и функция `factorial` выполнится параллельно с основным потоком выполнения. Завершения ее вычисления можно в явном виде подождать, написав

```
await (spawn factorial 30)
```

(В этом языке программирования скобки для параметров функций писать не нужно, они используются лишь для задания порядка вычислений. Без скобок запись `await spawn factorial 30` означала бы вызов await с параметрами `spawn`, `factorial` и `30`)

Грамматики языков программирования

Компиляция это процесс, который обычно делят на две фазы: лексический анализ и синтаксический анализ. Если в нашем языке программирования есть ключевые слова, то не имеет смысла исследовать, из каких символов они состоят. Но можно пойти еще немного дальше. Посмотрим на оператор

```
a = b + 2;
```

Когда мы проверяем правильность синтаксиса, нам не очень важно, как именно поименованы переменные и какое конкретно число прибавляется к b (лишь бы оно было правильно написано). Поэтому вполне разумно провести предварительный этап, на котором удалить комментарии и выделить те элементы, смысл которых понятен. Это обычно числа, ключевые слова, операторы. После лексического анализа наш опреатор примет примерно такой вид:

```
<идентификатор> = <идентификатор> + <число>
```

В угловых скобках здесь записаны некие категории языка, а одиночные символы оставлены "как есть". Следующая стадия анализа программы - синтаксический анализ - состоит в том, что компилятор делает более глобальные обобщения на основе каких-то правил. Например, рассматриваемый нами оператор компилятор сначала сведет к виду

```
<идентификатор> = <выражение>
```

А потом отнесет к категории операторов присваивания. Одновременно строится так называемое дерево разбора, или AST (abstract syntax tree), которое отражает структуру программы. Например, фрагмент программы

```
a = 0;
while (a < 5){
    if(a == 3) print("3");
    a = a + 1;
}
```

будет представлен примерно так:

```
программа
    оператор присваивания
        переменная(a)
        выражение
            число(0)
    оператор цикла
        условие
            выражение
                знак операции(<)
                    выражение
                        переменная(a)
                    выражение
                        число(5)
```

```
        тело_цикла
            оператор if
                условие
                    выражение
                        знак операции(==)
                            выражение
                                переменная(a)
                            выражение
                                число(3)
                часть then
                    вызов функции (print)
                        аргументы
                            строка("3")
                оператор присваивания
                    переменная(a)
                    выражение
                        знак операции("+")
                            выражение
                                переменная("a")
                            выражение
                                число(1)
```

Читать это следует примерно так: программа состоит из оператора присваивания и оператора цикла. Оператор присваивания состоит из... и так далее.

По такому описанию уже не так сложно сгенерировать код. Для этого используется процесс, обычно состоящий из нескольких стадий, в том числе это могут быть оптимизация, генерация промежуточного кода, оптимизация промежуточного кода, генерация кода для конкретного процессора, итп.

Формальное описание языка

Форма Бэкуса - Наура придумана для описания языка Algol в 1958 году. Она состоит из правил, которые в свою очередь состоят из следующих элементов:

- классы или нетерминальные символы. Это любые слова или наборы слов, заключенные в угловые скобки.
- символы языка или терминальные символы. Это слова языка, заключенные в кавычки (иногда кавычки опускают)
- операция ::= (читается "определяется как"), слева стоит нетерминальный символ, справа выражение
- операция выбора | - описывает возможность выбора одного из вариантов
- квадратные скобки [] - заключенное в них выражение может отсутствовать.
- в конце правила ставится точка.

Например, возможно такое определение:

<арифметическая операция> ::= "+" | "-" | "*" | "/".

вот описание оператора `if` языка Pascal:

<условный оператор if> ::= "IF" <булево выражение> "THEN" <оператор> ["ELSE" <оператор>]. <булево выражение> ::= "NOT" <булево выражение> | "(" <булево выражение> ")" | <булево выражение> <логическая операция> <булево выражение> | <выражение> <операция сравнения> <выражение>. <логическая операция> ::= "OR" | "AND". <выражение> ::= "(" <выражение> ")" | <переменная> | <строка> | <число>. <операция сравнения> ::= "=" | "<" | ">".

Надо заметить, что БНФ языка могут соответствовать неправильные программы, например, фрагмент

```
IF 3 < 'klm' and 2 > 'abc' THEN
```

соответствует приведенному тут описанию оператора IF. Также мы ничего не сказали про то, что такое <строка> и <число>. Можно привести формальные определения в рамках БНФ, например

<число> ::= <цифра> | <цифра> <число>. <цифра> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0". <строка> ::= """ [<последовательность символов>] """. <последовательность символов> ::= <символ> | <символ> <последовательность символов>

Как видно, определения могут быть рекурсивными. Однако, для формального определения нетерминала <символ> нам понадобится перечислить все возможные символы, которые могут появиться в строке.

При описании синтаксиса языков программирования часто используются следующие конструкции: одно или более повторений элемента и ноль или более повторений элемента. Например, аргументы функции это ноль или более описаний аргументов:

<описание функции> ::= <идентификатор> '(' <список аргументов> ')'. <список аргументов> ::= [<идентификатор>] | <идентификатор> <список аргументов>.

Эти конструкции часто включают в форму БНФ как расширения.

Классификация грамматик

Первым языком, синтаксис которого был описан при помощи БНФ, был Algol. Возникает естественный вопрос: любой ли язык можно описать таким образом? Для ответа на этот вопрос была разработана теория формальных грамматик и классификация грамматик.

Формальная грамматика состоит из множества терминальных символов (слов языка), множества нетерминальных символов ("понятий"), и набора правил вида

```
левая часть -> правая часть
```

где левая и правая части состоят из каких-то терминальных и нетерминальных символов. Терминальные символы мы будем обозначать строчными буквами, а нетерминальные - заглавными. Также один из нетерминальных символов объявляется начальным, обозначим его S.

Синтез текста происходит путем замены левой части на правую, а анализ текста состоит в подборе подходящих правых частей и поиске пути, по которому мог бы происходить синтез данного текста.

Синтез заканчивается, когда в тексте не остается нетерминальных символов, а анализ - либо когда по цепочке правил удалось дойти до начального символа, либо когда удалось доказать, что данный текст при помощи данного набора правил сгенерировать невозможно.

Обычно это демонстрируют на специальных примерах. Рассмотрим грамматику:

S -> L L -> E L -> EcL E -> a E -> b

Возможно ли при помощи этого набора правил произвести строку `asdf`? Конечно, нет, в грамматике нет терминальных символов `s`,`d` и `f`. А строку `acbca`? Попробуем:

```
S -> L -> EcL -> acL -> acEcL -> acbcL -> acbcE -> acbca
```

Получилось. А можно ли вывести строку `acbcac`? Нет, нельзя, но доказательство этого будет чуть более сложным, чем для строки `asdf`. Заметим, что получившаяся у нас грамматика похожа на описание каких-то элементов, разделенных запятыми, в качестве которых здесь выступает терминальный символ `c`.

Для более сложных грамматик анализ текста производится автоматически, специальными программами, которые, в свою очередь, тоже создаются автоматически из описаний грамматик.

Формальные грамматики можно разделить на четыре типа. Для отнесения грамматики к тому и ли иному типу необходимо соответствие всех её правил некоторым схемам.

Тип 0 — неограниченные

Непустая последовательность символов любого вида преобразуется в другую последовательность символов без каки-либо ограничений. Практического применения в силу своей сложности такие грамматики не имеют.

Тип 1 — контекстно-зависимые

Это грамматики, у которых в левой части всех правил есть хотя бы один нетерминальный символ. Приведем пример контекстно-зависимой грамматики, описывающей язык, все тексты которого состоят из нескольких символов `a`, за которыми следуют столько же `b` и потом столько же символов `c`. Примеры текстов: `aabbcc`, `abc`, `aaabbbccc`, и тд.

S -> A A -> aABC A -> aBC CB -> BC aB -> ab bB -> bb bC -> bc cC -> cc

Рассмотрим вывод первой из строк, приведенных в качестве примера:

```
S -> A -> aABC -> aaBCBC -> aaBBCC ->  aabBcc -> aabbCC -> aabbcC -> aabbcc
```

Тип 2 — контекстно-свободные

Это грамматики, у которых левые части всех правил состоят из какого-нибудь нетерминального символа. Такие грамматики находят самое широкое применение в языках программирования ввиду

того, что они простые, легко поддаются анализу и для них созданы различные средства автоматической генерации синтаксических анализаторов.

Однако, не все языки можно описать при помощи грамматик типа 2. Например, язык, который мы описали выше при помощи грамматики типа 1, описать грамматикой типа 2 не получится. Это не самая большая проблема при создании компиляторов, потому что, если бы нам, например, действительно зачем-то понадобился язык со строками из одинакового количества символов `a` `b` и `c`, мы могли бы описать более простой язык, в текстах которого могут быть произвольные ненулевые количества `a` `b` и `c` (например, aabbbbc была бы текстом такого языка), а потом просто *посчитать* количества символов `a` `b` и `c`, и в случае их несовпадения выдать сообщение об ошибке.

Среди грамматик типа 2 есть несколько подклассов, среди которых наиболее важными для построения компиляторов являются LL(k) и LR(k).

Грамматика LL(k) это такая грамматика, в которой решение о том, какое именно правило верхнего уровня должно быть применено для дальнейшего вывода, всегда может быть принято после просмотра не более чем k следующих терминальных символов. Например, в языке Pascal, последовательно читая программу, мы видим один из нескольких терминальных символов: `var` означает, что дальше будет описание переменной, `procedure` означает, что дальше будет процедура, `type` означает, что дальше будет описание типа, и т.д. Число, ключевое слово `for` и многое другое не могут появиться там, где мы ожидаем увидеть слова `procedure` или `var`. Pascal - типичный язык, который можно описать грамматикой LL(1). Разбор программы происходит по принципу сверху вниз, поскольку на основании просмотренного символа можно из всей грамматики выделить набор правил, применимых в данной ситуации.

Алгоритм разбора LR(k) на основании просмотра не более чем k следующих терминальных символов может принять решение о том, какое именно правило нижнего уровня здесь подходит. Например, в языке **C** при разборе строки

```
static int x()
```

мы не знаем, о чем вообще идет речь, до тех пор, пока не увидим открывающую скобку. До тех пор это может быть описание переменной, а может быть и описание функции. Но после того, как появилась скобка, мы знаем, что это функция, и можем однозначно применить соответствующее правило.

Подкласс грамматик, которые могут быть разобраны при помощи алгоритма LR(k) включает в себя все грамматики LL(1), но сам этот алгоритм работает немного медленнее, чем алгоритм разбора LL(k).

Большинство программ для автоматической генерации синтаксических аналозаторов (`yacc`, `antlr` и др.) используют один из этих алгоритмов.

Тип 3 — регулярные

Эти грамматики делятся на два эквивалентных класса, у которых все правила имеют вид:

1. A -> Bc
2. A -> cB

где c - терминальный символ, а A и B - нетерминальные. Языки с грамматиками типа 3 могут быть интерпретированы с помощью регулярных выражений. Отсюда следует ответ на часто незадаваемый вопрос о том, можно ли проанализировать HTML (а также C или Python) при помощи регулярных выражений. Поначалу многим кажется, что это вполне возможно, учитывая мощь аппарата регулярных выражений, но теория говорит об обратном.

Зная теорию формальных грамматик, вы, например, никогда не сделаете такой ошибки: часто в программах требуются конфигурационные файлы. Часто для этих файлов придумывают свой собственный формат. И, глядя на сложные конфиги таких программ как `Apache` или `Nginx`, люди придумывают слишком сложный формат, который в общем случае не поддается интерпретации с помощью регулярных выражений.