

Сравнительное обозрение языков программирования

Автор: Олег Цилюрик

Редакция **2.31**
21.10.2014г.

Оглавление

Обоснование.....	3
Направления развития.....	4
Демонстрационная задача.....	5
Сравнительные иллюстрации.....	7
Язык C.....	7
C++.....	9
Java.....	13
Python.....	16
Ruby.....	18
Perl.....	20
JavaScript.....	22
PHP.....	24
Lua.....	26
Командный интерпретатор bash.....	28
Язык Go.....	32
Scheme.....	38
Scala.....	41
Ocaml.....	44
Haskell.....	48
Другие языки.....	56
X10.....	56
Chapel.....	56
Haxe.....	57
Erlang.....	57
Обсуждение.....	58
Заключение.....	59
Указатель источников информации.....	60

Обоснование

Эти заметки относительно разных языков программирования, в разрозненном и отрывочном их виде, уже представлялись на различных ресурсах Интернет. В общей массе положительных отзывов и обсуждений, высказывались и критические сомнения, в том смысле, что:

- Сравнение различных языков программирования может иметь только академическую ценность или для любопытства, поскольку всякий язык программирования предназначен исключительно для своего класса задач, и уместен только для этого класса;

- Реальный средний программист практикует один, от силы пару, языков программирования, и для него не актуальны подобные расширенные сравнения.

Автор **принципиально** не согласен с такой позицией!

Все из упоминаемых ниже языков программирования являются **универсальными** языками. А сравнить исходные коды эквивалентных, в меру возможностей, приложений, реализованных на различающихся языках программирования — занятие и познавательное и поучительное:

- Начинающие программисты могут бегло взглянуть на имеющийся в их распоряжении арсенал средств и, в какой-то мере, утвердиться в том, в каком направлении им желательно развивать свои навыки.
- Имея в руках работающие приложения выполненные в разных технологиях, можно, экспериментируя, рассмотреть принятые техники доведения программного кода до работающего приложения (технология, методологию), которые в разных языках существенно различаются.
- Практик-профессионал, рассматривая код на языке, далёком от сферы его интересов, сможет выделить там отдельные специфические приёмы, а затем и смоделировать их в своей привычной инструментальной среде (подобным образом в обиход программирования много привнесли LISP, APL или FORTH).
- Специалист по обработке данных может позаимствовать отдельные идеи и структуры данных даже из совершенно экзотического для его целей языка.
- Студенты смогут оценить цельность базовых концепций программирования, которые едины и только лучше оттеняются контрастом разнообразия реализаций.
- Соискатель работы перестанет теряться во множестве загадочных наименований, фигурирующих в требованиях к вакансиям, когда работодатель сам достаточно часто плохо понимает что ему нужно.
- ... и вообще, профессиональный программист должен по внешнему виду программного кода распознавать на каком языке это написано.

Мы пройдём по реализациям однотипных приложений на разных языках программирования, и будем фиксировать примечательные особенности каждого из них¹. Важнейшим принципом здесь будет не сравнение по принципу «лучше-хуже», а намерение показать как подобные вещи могут выглядеть в разных языках.

Не следует искать какого-то особого значимого подтекста в том, в каком порядке представляются отдельные языки: порядок определяется, главным образом, той хронологией, в которой были готовы реализации задач для их показа. Тем не менее, сначала рассматриваются языки (10 штук), наиболее часто используемые в практических свободных и коммерческих проектах (C, C++, Python, Perl, ...). И только затем — языки, более известные в академической среде и образовании (Scala, Haskell, ...).

При переходе к очередному языку будет даваться краткая (не более одного абзаца) его характеристика: когда был впервые создан (это важно для понимания принципов), область и особенности применения. По менее известным (Lua, Go, ...), или особенно новым языкам (Go, Scala, ...), реализации которых отсутствуют в стандартных дистрибутивах Linux, приводятся все необходимые данные по установке соответствующего инструментария. Включение в обзор более редко, в силу разнообразных причин, используемых языков программирования («экзотических») интересно и важно из нескольких соображений:

- Это даёт возможность взглянуть на широкий пласт существующих языковых средств, не фигурирующих в лидерах коммерческих разработок. Это языки новые, экспериментальные, специальной ориентации и т.д. Знание их, хотя бы самое поверхностное, не будет во зло: то, что сегодня экзотика, завтра может стать популярным инструментом в использовании (так

1 Примеры кода написаны не как образец того, как **надо** писать, но представляют полностью работающие и проверенные реализации того, как **выглядит** код. Это совершенно естественно, поскольку к некоторым обсуждаемым языкам автор не прикасался в практической работе последние 20-30 лет (Lisp), а с некоторыми вообще познакомился во время подготовки этого обзора. Тем не менее, решается главная цель: примеры показывают как **может выглядеть** код на каждом из языков.

было с Python).

- В этих экзотических языках любопытно наблюдать новые синтаксические и семантические конструкции. Наличие таких конструкций, особенно когда они необычные, во-первых, указывает, что эквивалентные им возможности реализованы с определёнными дефектами в традиционных языках, иначе не предпринимались бы усилия в поиске им замены. Во-вторых, такие конструкции подсказывают направления, в которых развиваются языковые возможности — завтра эквивалентные расширения могут появиться в спецификациях традиционных языков.
- Наконец, в-третьих, разбирательство с такими интересными конструкциями позволит их моделировать и средствами традиционных языков не ожидая их расширения.

Примечание: Материал для этого обзора нарабатывался достаточно продолжительное время, поэтому установка языковых пакетов, проводившаяся в разное время и на разном оборудовании, показывается в двух дистрибутивах, Debian и Fedora, с их различными пакетными системами. Но такое разнообразие даже полезнее для широты охвата...

Уже в ходе работы, намерения несколько изменились, и в обзор были включены сведения и по некоторым новым языкам программирования, которые динамично развиваются, и которые выглядят перспективными и интересными — даже если на них не реализована (пока) и не показана тестовая сравнительная задача. По таким языкам приводится краткая справочная информация: где взять, как установить, ... Возможно, в последующих редакциях этого обзора и для них будет показана сравнительная реализация.

Всё обсуждение ведётся и иллюстрируется примерами в операционной системе Linux (и, с тем же успехом, в любой POSIX системе). Это обусловлено только тем, что в Linux разнообразные языки программирования используются значительно шире, чем, скажем, в Windows. Но языковой инструментарий программирования на зависит от операционной системы, поэтому всё излагаемое может быть воспроизведено, с некоторыми техническими отличиями и в среде Windows.

Примечание: По каждой реализации будет показана команда запуска приложения и результат его выполнения, скопированные непосредственно с терминала. Хотя результаты работы приложений очень похожи, такая избыточность сделана специально для того, чтобы читатели, при желании, могли воспроизвести каждый запуск, прогнать его результат, и использовать его как отправную точку для последующих экспериментов. Рассматриваемые далее приложения почти идентичны, но это «почти» касается, главным образом, возможностей и техники обработки ошибок ввода пользователем. Доводить обработку ошибок до полной идентичности значило бы перегружать код частными ненужными деталями.

Направления развития

Следующим вопросом, который заслуживал бы рассмотрения, является оценка производительности (скорости выполнения) альтернативных реализаций на разных языках одних и тех же общеизвестных алгоритмов. Сравнить скорости — неблагоприятное занятие! Но совершенно реально прикинуть **порядки** временных затрат, требуемых разным реализациям: быстрее-медленнее в 10, 100, 1000 и т.д. раз. Основная часть такой работы уже проделана, но ожидает своего упорядочения. После этого она будет представлена либо как отдельная публикация, либо как составная часть расширенного настоящего текста.

Есть ещё один срез сравнения различных языковых реализаций, который отчётливо выявился только в последние 5-7 лет, и который абсолютно не отражён в литературе и обсуждениях. Речь идёт о возможности вообще **распараллелить** код задачи для выполнения его на симметричной **многопроцессорной** системе (SMP) и о последствиях **такой** реализации: код, написанный для N процессоров во сколько раз будет (или не будет) производительнее одно-процессорного варианта? Кроме того, разные языки могут предлагать совершенно различные модели выражения параллельности и синхронизации (семафоры Дейкстры, мониторы Хоара и др.).

Демонстрационная задача

Перед сравнительной иллюстрацией разнообразия языковых реализации встаёт одна трудность: нужна адекватная задача для реализации. Тривиальная задача типа «Hello World» не будет показательной для сравнения — она слишком проста для того, чтобы отобразить какие-то значащие особенности языка. А задача свыше 100-150 строк будет уже перегружена ненужными деталями, громоздкой и неинтересной. Поэтому нам предстоит выбрать более-менее адекватную задачу для наших иллюстрационных целей...

Для сравнительной реализации была выбрана задача расчёта параметров 2D (на плоскости) **треугольника**, заданного координатами своих вершин, а именно: расчёт **периметра** и **площади**. По ходу изложения эта начальная формулировка будет несколько расширена.

Координаты вершин будут выражаться как комплексное значение — это естественно для физического мира, так как комплексные величины это и есть отображение точек 2D-плоскости. Но самое главное, что такой подход с самого начала потребует работы со структурными объектами (2-х компонентные комплексные значения). А геометрическая фигура (треугольник) естественным образом подталкивает к использованию понятий класса и объекта. Есть где разгуляться!

Но прежде, чем приступить к реализациям, нужно сделать минимальный экскурс в теорию комплексных вычислений. Каждое комплексное число представляется суммой вещественной и мнимой компонент:

$$z = \text{real} + i * \text{image}$$

Здесь `real` — это вещественная часть числа, а `image` — мнимая его часть (`real` и `image` здесь конкретный числовые, вещественные значения для данного конкретного комплексного числа).

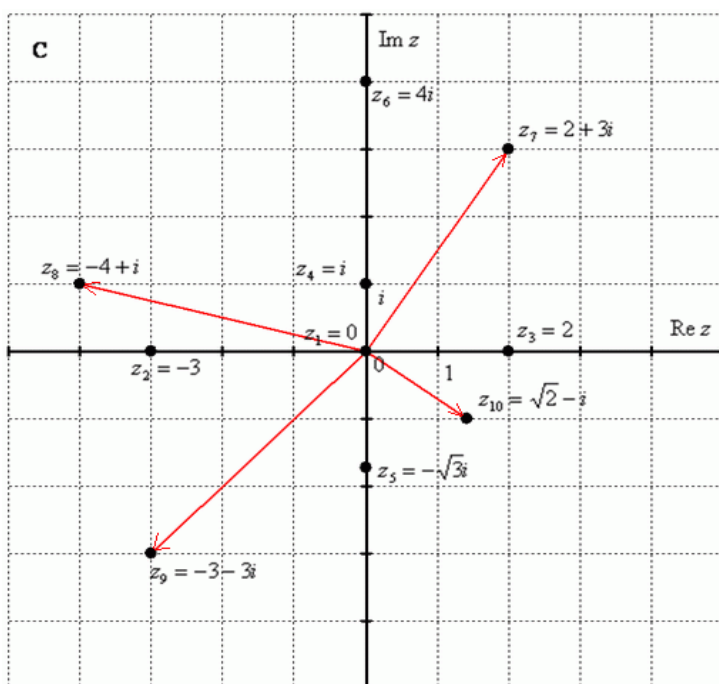
Примечание: Здесь я мог бы рассказать ещё, что i — это константная величина, равная $\sqrt{-1}$, и что это означает ... но это ровно ничего не добавит к целям нашего рассмотрения).

На вещественной плоскости (2D) комплексное число z отображается точкой плоскости, для которой: `real` — это координата точки по горизонтали (ось X), а `image` — это координата точки по вертикали (ось Y). Также каждое комплексное число имеют другую, альтернативную форму представления, так называемую экспоненциальную, вида:

$$z = \text{abs} * \exp(i * \text{arg})$$

Здесь значение `abs` — это длина вектора z (от точки 0,0), а `arg` — фазовый угол наклона вектора относительно оси X (выраженный в радианах), исчисляемый против часовой стрелки.

Эти две формы представления описывают одну и ту же точку плоскости, и между ними существуют взаимно однозначные соответствия (это эквивалентные представления). Они связаны соотношениями (все показанные математические функции присутствуют в библиотеке **любого** языка программирования):



$$\begin{aligned} \text{real} &= \text{abs} * \cos(\text{arg}) \\ \text{image} &= \text{abs} * \sin(\text{arg}) \\ \text{abs} &= \sqrt{ \text{real}^2 + \text{image}^2 } \\ \text{arg} &= \text{atan2}(\text{image}, \text{real}) \\ z &= \text{abs} * \exp(i * \text{arg}) = \text{abs} * \\ &\quad (\cos(\text{arg}) + i * \sin(\text{arg})) \end{aligned}$$

Несколько примеров комплексного представления точек 2D пространства:

```
Z1 = 0. + 0. * i
Z2 = -3. + 0 * i
Z3 = 2. + 0 * i
Z5 = 0 - sqrt( 3. ) * i
Z6 = 0 + 4 * i
Z7 = 2. + 3. * i
Z8 = -4 + i
Z9 = -3 - 3 * i
Z10 = sqrt( 2. ) - i
```

Мы используем ту форму из 2-х, которая нам более удобна в данный момент. Математические библиотеки манипуляции с комплексными числами содержат встроенные функции преобразования из одной формы в другую. Например, для показанных на рисунке некоторых чисел (векторов) имеет место (угол \arg показан в радианах, долях π и в угловых градусах для наглядности — это одно и то же значение):

```
z1  = ( +2.0 , + 3.0i ) <=> abs = 3.606 , arg = 0.983 = 0.31*π = 56°
z5  = ( -0.0 , -1.7i ) <=> abs = 1.732 , arg = -1.571 = -0.50*π = -90°
z8  = ( -4.0 , +1.0i ) <=> abs = 4.123 , arg = 2.897 = 0.92*π = 166°
z9  = ( -3.0 , -3.0i ) <=> abs = 4.243 , arg = -2.356 = -0.75*π = -135°
z10 = ( +1.4 , -1.0i ) <=> abs = 1.732 , arg = -0.615 = -0.20*π = -35°
```

Зачем нам такие сложности?

А затем, что дальше вся 2D геометрия становится намного проще:

- вектор, замыкающий точки $z9$ и $z8$, будет вычисляться просто как **разница** ($z8 - z9$);
- его длина (нужная нам как составляющая периметра) — как $\text{abs}(z8 - z9)$;
- а площадь треугольника, построенного на сторонах $z9$ и $z8$ будет вычисляться как:

$$\text{abs}(z8) * \text{abs}(z9) * \sin(\arg(z8) - \arg(z9)) / 2.$$

Тогда, например, на языке C (как наиболее простой и понятный случай) мы опишем тип треугольника как:

```
#define NODES 3 // число вершин
typedef double complex triangle_t [ NODES ]; // тип треугольника
```

А периметр произвольного треугольника в комплексной форме будет вычисляться как сумма $\text{cabs}()$ (длин векторов) для всех его сторон:

```
static double perimeter( double complex triangle_t [ NODES ] ) {
    double summa = 0.0;
    int i, j;
    for( i = 0; i < NODES; i++ ) {
        j = NODES - 1 == i ? 0 : i + 1;
        summa += cabs( pts[ i ] - pts[ j ] );
    }
    return summa;
}
```

А площадь, исходя из векторов (комплексного представления) 2-х его произвольных сторон — как половина произведения длин этих 2-х векторов на $\sin()$ угла между ними:

```
static double square( triangle_t pts ) {
    double complex side1 = pts[ 1 ] - pts[ 0 ],
        side2 = pts[ 2 ] - pts[ 0 ];
    return cabs( side1 ) * cabs( side2 ) *
        fabs( sin( carg( side1 ) - carg( side2 ) ) ) / 2.;
}
```

Нас подкупает простота таких вычислительных процедур, поскольку целью наших реализаций на различных языках являются не вычисления, а создание структуры приложения и структур данных.

Показанного введения в комплексную математику достаточно для понимания всех наших последующих сравнений (даже для тех, кто никогда не сталкивался с комплексной математикой, да и не хочет вникать в её детали).

Дальнейшее расширение постановки задачи (о котором вскользь сказано выше) будет состоять в том, что для языков, допускающих лёгкое изменение размерностей структур данных (массивов), таких как Python, Go, Scala, ... (исключая C, C++, Java, ...) мы можем усложнить задачу. А именно: любой произвольный **выпуклый** N-угольник с вершинами $[1 \dots N]$ может быть представлен как последовательность N-2 треугольников, где K-й треугольник составят вершины $[1, K, K+1]$ исходного многоугольника. Тогда площадь произвольного многоугольника может быть найдена как сумма в цикле площадей N-2 составляющих треугольников (всё это легко видеть далее по коду примеров).

Сравнительные иллюстрации

Специально была подобрана задача, которая будет требовать операторского ввода, реагировать на ошибки ввода (исключительные ситуации), и включать консольный вывод, да ещё в UNICODE и на русском языке...

Примечание: Русскоязычный вывод (или его осуществление специальными опциями) является существенной частью поставленной задачи: для англоязычных авторов, пишущих документацию, поддержка языковых локализация является совершенно второстепенной задачей, не находящей адекватного отображения в описаниях.

Немаловажным вопросом является то, как в каждом из языков выделяются **комментарии**: однострочные (где они есть) и блочные. В каждом из приводимых примеров кода включены комментарии, как те так и другие, в качестве образца, без каких либо дополнительных пояснений.

Язык С

«Существует великое множество языков программирования, которые не уступают или даже превосходят Си по красоте и удобству.

Тем не менее ими никто не пользуется.»

Денис Ритчи

Язык С был и остаётся основным технологическим инструментом разработки для операционных систем класса UNIX, Linux в частности. На нём реализуются как сами операционные системы, так и широченный набор утилит для них, например, GNU утилиты. Другими словами, для UNIX: «язык С — наше всё». Ранние реализации языка (Брайан У. Керниган, Деннис М. Ритчи) относятся к 1972-му году. Это язык из патриархов, и практически все его сверстники либо полностью отошли в прошлое (Algol, Cobol), либо используются, но в очень ограниченной сфере интересов (FORTRAN, LISP).

Поэтому, совершенно естественно, что именно с него мы начнём реализацию, и эту реализацию будем использовать как точку отсчёта в последующих сравнениях.

Примечание: Задача комплексной математики вдвойне интересна ещё и потому, что комплексные переменные в **синтаксис** языка С и их поддержка в стандартной математической **библиотеке** (libm.so) были добавлены относительно недавно (файл <complex.h>), стандартом C99 (1999 год). Само **наличие** таких возможностей в языке С часто вообще **не упоминаются** в распространённой литературе по языку С.

Реализация демонстрационной задачи на языке С:

triangle.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include <string.h>

/* расчёт параметров треугольника */
#define NODES 3 // число вершин
typedef double complex triangle_t [ NODES ]; // тип треугольника

static double perimeter( triangle_t pts ) {
    double summa = 0.0;
    int i, j;
    for( i = 0; i < NODES; i++ ) {
        j = NODES - 1 == i ? 0 : i + 1;
        summa += cabs( pts[ i ] - pts[ j ] );
    }
    return summa;
}
```

```

static double square( triangle_t pts ) {
    double complex side1 = pts[ 1 ] - pts[ 0 ],
               side2 = pts[ 2 ] - pts[ 0 ];
    return cabs( side1 ) * cabs( side2 ) *
           fabs( sin( carg( side1 ) - carg( side2 ) ) ) / 2.;
}

#define INLEN 40
int main( int argc, char **argv, char **envp ) {
    while( 1 ) {
        int i = 0;
        triangle_t polygon;
        printf( "координаты вершин в формате: X Y\n" );
        while( i < NODES ) {
            float x, y;
            printf( "вершина № %d: ", i + 1 );
            fflush( stdout );
            char s[ INLEN ], e[ INLEN ];
            if( !fgets( s, sizeof( s ) - 1, stdin ) ) // строка ввода
                printf( "завершение\n" ), exit( EXIT_SUCCESS );
            s[ strlen( s ) - 1 ] = '\0';           // удалить EOL
            while( ' ' == s[ strlen( s ) - 1 ] )   // удалить хвостовые пробелы
                s[ strlen( s ) - 1 ] = '\0';
            if( sscanf( s, "%f%c%f%s", &x, &y, (char*)&e ) != 2 ) {
                printf( "ошибка ввода!\n" );
                continue;
            }
            polygon[ i++ ] = x + I * y;
        }
        printf( "вершин %d : ", NODES );
        for( i = 0; i < NODES; i++ )
            printf( "[%2f,%2f] ",
                    creal( polygon[ i ] ), cimag( polygon[ i ] ) );
        printf( "\n\nпериметр = %2f\n\nплощадь = %2f\n\n",
                "-----\n",
                perimeter( polygon ), square( polygon ) );
    }
}

```

Код выдержан в духе **структурности**, никакой объектной модели язык С не предлагает. Для описания геометрических фигур «треугольник» определяется новый тип `triangle_t`, но это не более чем синтаксический трюк, позволяющий сократить и упростить запись кода.

Примечание: Отсутствие синтаксической «объектности» совершенно не отвергает возможности объектного построения целевого приложения. Объектность — это способ осмысления моделируемых сущностей, а «объектный» синтаксис языка только облегчает выражение этого осмысления в коде. Лучшим примером сказанного может быть графическая подсистема Photon операционной системы QNX: написанная на чистом С она более объектная, чем иные проекты на Java.

Мы можем одинаково хорошо **собрать** приложение из этого кода как традиционным компилятором GCC, так и новым, активно развивающимся компилятором Clang из проекта LLVM:

```

$ gcc triangle.c -otriangle_c -lm -Wall
$ clang -Wall -lm triangle.c -o triangle_co
$ ls -l | grep rwx
-rwxrwxr-x. 1 0lej 0lej 13220 окт 11 22:05 triangle_c
-rwxrwxr-x. 1 0lej 0lej 13217 окт 11 22:05 triangle_co

```

С определённой степенью вероятности (в меру совместимости используемого API), приложения С (это и другие) могут быть собраны и с помощью MS Visual Studio в Windows.

Примечание: Непереносимость исходных кодов на языке С между операционными системами обусловлена, главным образом, несовместимостью используемых библиотек (API), а не различиями в толкованиях синтаксиса и семантики языка различными компиляторами (которые минимальны). Для ликвидации

такой несовместимости было реализовано несколько проектов API программных обёрток, независимых от платформы. Одним из таких проектов является, например, Apache Portable Runtime (APR) — эти библиотеки использованы в таких крупных проектах Voice-IP программных коммутаторах как Asterisk и FreeSWITCH.

А вот как выполняется только-что собранное нами приложение:

```
$ ./triangle_c
координаты вершин в формате: X Y
вершина № 1: 1 1
вершина № 2: 1 3
вершина № 3: 3 1
вершин 3 : [1.00,1.00] [1.00,3.00] [3.00,1.00]
периметр = 6.83
площадь = 2.00
-----
координаты вершин в формате: X Y
вершина № 1: a1
ошибка ввода!
вершина № 1: a 1
ошибка ввода!
вершина № 1: .5 .5
вершина № 2: .5 1.5
вершина № 3: 1.5 .5
вершин 3 : [0.50,0.50] [0.50,1.50] [1.50,0.50]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y
вершина № 1: ^D завершение

$ ./triangle_co
координаты вершин в формате: X Y
вершина № 1: 1 1
вершина № 2: 1 3
вершина № 3: 3 1
вершин 3 : [1.00,1.00] [1.00,3.00] [3.00,1.00]
периметр = 6.83
площадь = 2.00
-----
координаты вершин в формате: X Y
вершина № 1: ^C
```

Не углубляясь в детали отметим, что характерной сложностью выражения на С является работа с символьными строками и обработка символьной информации (что видно и из предложенного примера кода).

C++



*«Для чего ты стараешься отяготить настоящий день больше, чем уделено ему тяготы?
Для чего возлагаешь на него бремя и наступающего дня?»
Святой Иоанн Златоуст, «Беседы на Матфея», 22-я глава*

Язык C++ (берущий начало от «C with classes» начала 1980-х годов) часто характеризуют как **надмножество** языка С — любой (почти) С-код может быть скомпилирован в режиме C++ с использованием синтаксического анализатора C++ :

```
$ g++ -Wall -lm triangle.c -o triangle_c
$ ls -l *.c
```

```
-rw-r--r--. 1 Olej Olej 2089 окт 11 22:02 triangle.c
-rw-r--r--. 1 Olej Olej 3860 окт 11 21:53 triangle.cc
```

Как легко видеть, даже при самом поверхностном взгляде, результатом компиляции одного и того же текстуально кода будут два различающихся исполнимых файла.

Язык C++ **намного** шире своего прародителя, включает различные новые независимые парадигмы, такие как: строгая статическая **именная** типизация; классы, объекты и наследования; переопределения функций и операций, шаблоны (template), пространства имён... (легче перечислить то что осталось, чем то, что добавилось). Но там, где появляется широта возможностей, возникают и сложность и громоздкость. Реализация описываемой задачи на C++ может выглядеть так:

triangle.cc :

```
#include <stdlib.h>
#include <complex>
#include <iostream>

using namespace std;

/* расчёт параметров класса triangle, представляющего треугольник */

class point : public complex<double> {    // класс вершины наследуемый от complex
private:
    bool bGood;
protected:
    point( void ) : bGood( true ) {
        *(complex<double>*)this = complex<double>( 0.0, 0.0 );
    }
    point( double re, double im ) : bGood( true ) {
        *(complex<double>*)this = complex<double>( re, im );
    }
    point( const complex<double>& c ) : bGood( true ) {
        *(complex<double>*)this = c;
    }
public:
    friend class triangle;
    inline bool bOK( void ) { return bGood; };
    friend ostream& operator << ( ostream& stream, point& obj );
    friend istream& operator >> ( istream& stream, point& obj );
};

inline ostream& operator << ( ostream& stream, point& obj ) {
    stream << "[" << obj.real() << "," << obj.imag() << "];"
    return stream;
};

inline istream& operator >> ( istream& stream, point& obj ) {
    double x, y;
    string s;
    obj.bGood = false;                                // ошибка при неправильном вводе
    if( ( cin >> x ).eof() ) return stream;             // ввод real
    if( cin.rdstate() & ios::failbit ) {
        cerr << "ошибка ввода!" << endl;
        cin.clear();
        getline( cin, s );
        return stream;
    }
    if( ( cin >> y ).eof() ) return stream;             // ввод image
    if( cin.rdstate() & ios::failbit ) {
        cerr << "ошибка ввода!" << endl;
        cin.clear();
        getline( cin, s );
    }
}
```

```

        return stream;
    }
    getline( cin, s );
    if( !s.empty() ) {                                     // если введено больше 2-значений
        basic_string<char>::iterator i;
        for( i = s.begin(); i != s.end() && *i == ' '; i++ );
        if( i != s.end() ) {                               // если там непробельные символы
            cerr << "ошибка ввода: " << s << endl;
            return stream;
        }
    }
    obj = point( complex<double>( x, y ) );
    return stream;
};

class triangle {                                          // класс треугольник производный от point
public:
    static const int NODES = 3;                          // число вершин
protected:
    point pt[ NODES ];                                   // координаты вершин
public:
    double perimeter( void ) {
        double summa = 0.0;
        int i, j;
        for( i = 0; i < NODES; i++ ) {
            j = NODES - 1 == i ? 0 : i + 1;
            summa += abs( pt[ i ] - pt[ j ] );
        }
        return summa;
    }
    double square( void ) {
        complex<double> side1 = pt[ 1 ] - pt[ 0 ],
            side2 = pt[ 2 ] - pt[ 0 ];
        return abs( side1 ) * abs( side2 ) *
            fabs( sin( arg( side1 ) - arg( side2 ) ) ) / 2.;
    }
    inline point& operator [] ( int i ) { return pt[ i ]; }
    friend istream& operator >> ( istream& stream, triangle& obj ) {
        int i = 0;
        while( i < NODES ) {
            cout << "вершина № " << i + 1 << " : " << flush;
            stream >> obj.pt[ i ];
            if( stream.eof() ) return stream;
            if( !obj.pt[ i ].bOK() ) continue;
            i++;
        }
        return stream;
    }
};

int main( int argc, char **argv, char **envp ) {
    int i = 0;
    cout.precision( 3 );
    while( 1 ) {
        triangle polygon;
        cout << "координаты вершин в формате: X Y" << endl;
        if( ( cin >> polygon ).eof() )
            cout << "завершение" << endl, exit( EXIT_SUCCESS );
        cout << "вершин " << triangle::NODES << " : ";
        for( i = 0; i < triangle::NODES; i++ )
            cout << polygon[ i ] << " ";
    }
}

```

```

        cout << endl << "периметр = " << polygon.perimeter() << endl
            << "площадь = " << polygon.square() << endl
            << "-----" << endl;
    }
}

```

Код этой реализации умышленно несколько усложнён (например, при обработке ошибок ввода), но в таком варианте он позволяет увидеть (хотя бы по написанию) основные нововведения C++ относительно классического C: классы и объекты, шаблонные (template) классы, использование итераторов шаблонных классов, потоковые операции ввода и вывода (cin, cout). **Сборка** такого приложения:

```
$ g++ -Wall -lm triangle.cc -o triangle_cc
```

Теперь, после сборки C++ приложения, мы можем кратко коснуться того вопроса, почему C код будет всегда компилироваться и собираться в среде C++. Дело в том, что C++ приложение **всегда** будет компоноваться со стандартной разделяемой библиотекой C (libc.so), в дополнение к своей собственной стандартной библиотеке (libstdc++.so):

```

$ ldd triangle_c
linux-gate.so.1 => (0xb76eb000)
libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xb76a8000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb755a000)
/lib/ld-linux.so.2 (0xb76ec000)
$ ldd triangle_cc
linux-gate.so.1 => (0xb7797000)
libstdc++.so.6 => /usr/lib/i386-linux-gnu/libstdc++.so.6 (0xb768e000)
libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xb7668000)
libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xb764a000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb74fc000)
/lib/ld-linux.so.2 (0xb7798000)

```

Стандартная библиотека C (libc.so) является интерфейсом к **системным вызовам**, предоставляемым операционной системой. А стандартная библиотека C++ является только интерфейсом к **библиотечным вызовам**, предоставляемым библиотекой C. Это будет относиться **ко всем** (или почти всем!) языкам, рассматриваемым далее, что часто опускается из виду. В повседневной практике, зачастую, непонимание этих связей не влечёт последствий. Но в малых и встраиваемых системах, когда библиотеки могут компоноваться к приложениям статически, эквивалентное приложение, скомпилированное в среде C++ может оказаться значительно объёмнее.

И выполнение собранного приложения:

```

$ ./triangle_cc
координаты вершин в формате: X Y
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1,1] [2,1] [1,2]
периметр = 3.41
площадь = 0.5
-----
координаты вершин в формате: X Y
вершина № 1 : завершение работы

```

Неизменно всё тот же код приложения может быть собран и новым компилятором Clang (из проекта LLVM):

```

$ clang++ -Wall -lm triangle.cc -o triangle_cco
$ ls -l *_cc*
-rwxrwxr-x. 1 Olej Olej 20925 окт 11 22:38 triangle_cc
-rwxrwxr-x. 1 Olej Olej 20711 окт 11 22:38 triangle_cco

```

Показанные выше **сборки** приложений (как для языка C, так и для C++) предполагают 2

последовательных фазы: **компиляции** с последующим связыванием с другими объектными файлами или библиотеками. Важнейшей, в контексте нашего текущего рассмотрения, фазой является компиляция. Во всех рассмотренных выше случаях компиляция производится в «нативный» код используемого процессора и с учётом особенностей (форматов) операционной системы. Это означает, что для переноса такого приложения в другую среду (отличный процессор, операционная система) исходный программный код такого приложения должен быть, как минимум, **перекомпилирован** (в предположении совместимости библиотечных API этих различающихся реализаций).

Если говорить в два слова о целевом предназначении, то язык C++ больше подходит, пожалуй, для крупных **целевых** (прикладных) проектов: графика, визуализация, финансы, системы автоматизированного управления и автоматизированного проектирования... Для задач собственно системного программирования (утилиты, библиотеки, протоколы, ...) более подходящим представляется классический C.

Java



*«Если бы мы программировали приложения Java Swing, то могли бы пойти по этому пути. Но в Android это работать не будет.»
Michael Galpin, IBM developerWorks*

Java — объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в настоящее время купленной компанией Oracle). Дата официального выпуска — 23 мая 1995 года. Приложения Java компилируются не в бинарную форму, а в специальный стандартизованный байт-код (файлы .class и их архивы .jar). Поэтому такие программы могут работать на любой виртуальной Java-машине (JVM) вне зависимости от процессорной платформы, операционной системы, локального или удалённого хоста. Примерами самых известных на сегодня JVM, из числа используемых на разных платформах, могут служить:

- Оригинальный JDK (Sun Java Development Kit, на сегодня Oracle JDK) — классическая первоначальная реализация Java, используется в Windows, Solaris, Linux и других системах.
- OpenJDK, входящий в стандартный комплект Linux, и немногим уступающий JDK.
- Dalvik Virtual Machine — Java виртуальная машина для системы Android, байт-код которой отличается от стандартного для JDK, но существуют программы трансформеры для их взаимного преобразования.

Примечание: Как вы могли бы уже заметить из эпиграфа к этой части рассмотрения, слухи о **абсолютной** переносимости кода приложений на Java сильно преувеличены. Тем не менее, это действительно наиболее независимый от платформы языковой инструмент.

Для нашего приложения, в Java, в её стандартных библиотеках, нет комплексной математики. В таких случаях мы можем (здесь и для других языков тоже) поступать на выбор двумя способами:

- Написать достаточно простой набор необходимых приложению комплексных функций. Это очень несложная работа.
- Разыскать (в Интернет) **модуль**, реализующий интересующую нас функциональность (а таких представлено достаточно много). Я выбираю (для примера Java) именно этот вариант от David Eck and Richard Palais: <http://3D-XplorMath.org/j/index.html>. (файл Complex.java — ничего интересного, поэтому его код не приводится, но он необходим и присутствует в архиве примеров).

Теперь всё готово для создания ещё одного приложения:

triangle.java :

```
import java.io.*;
import java.util.StringTokenizer;
import java.lang.Double.*;

/* class Complex заимствуется из отдельного файла */

class Tric {          // класс треугольник
    public static final int nodes = 3;
    Complex [] pt = new Complex[ nodes ];
    public String toString() {
```

```

        String ret = "";
        for( int i = 0; i < nodes; i++ )
            ret += "[" + ( new Double( pt[ i ].re ) ).toString() + "," +
                        ( new Double( pt[ i ].im ) ).toString() + "]" + " ";
        return ret;
    }

    public double perimeter() {
        double summa = 0.0;
        for( int i = 0; i < nodes; i++ )
            summa += pt[ i ].minus( pt[ nodes - 1 == i ? 0 : i + 1 ] ).r();
        return summa;
    }

    public double square() {
        Complex side1 = pt[ 1 ].minus( pt[ 0 ] ),
                side2 = pt[ 2 ].minus( pt[ 0 ] );
        return side1.r() * side2.r() *
            Math.abs( Math.sin( side1.theta() - side2.theta() ) ) / 2.;
    }
}

public class triangle {
    public static void main( String[] args ) {
        Console cons = System.console();
        while( true ) {
            String szStr = "";
            Tric polygon = new Tric();
            System.out.println( "координаты вершин в формате: X Y" );
            for( int i = 0; i < Tric.nodes; ) {
                szStr = cons.readLine( "%s%d%s", "вершина № ", i + 1, " : " );
                if( null == szStr ) { // ^D
                    System.out.println( "завершение работы" );
                    System.exit( 0 );
                }
                StringTokenizer st = new StringTokenizer( szStr, " \r\n" );
                try {
                    double x = ( new Double( (String)st.nextElement() ) ).doubleValue(),
                            y = ( new Double( (String)st.nextElement() ) ).doubleValue();
                    polygon.pt[ i ] = new Complex( x, y );
                }
                catch( java.util.NoSuchElementException ex ) {
                    System.out.println( "ошибка ввода!: " + ex.toString() );
                    continue;
                }
                catch( java.lang.NumberFormatException ex ) {
                    System.out.println( "ошибка ввода!: " + ex.toString() );
                    continue;
                }
                i++;
            }
            System.out.println( "вершин " + Tric.nodes + " : " + polygon.toString() );
            System.out.println( "периметр = " + polygon.perimeter() );
            System.out.println( "площадь = " + polygon.square() );
            System.out.println( "-----" );
        }
    }
}

```

Технология Java предполагает **компиляцию** исходных кодов. Но, в отличие от предыдущих случаев (C и C++), компиляция производится не в непосредственно «нативный» код исполняющей системы, а в абстрактный промежуточный **байт-код** (расширение файлов .class). Позже этот байт-

код будет при исполнении **интерпретироваться** виртуальной Java-машиной (JVM, Java Virtual Machine). Этим достигается полная машинная независимость Java кода.

Имя программы компилятора Java — `javac`. Но во многих дистрибутивах Linux сам компилятор Java (как составная часть среды разработки) не будет установлен по умолчанию, хотя исполняющая система `java` (JVM) будет присутствовать наверняка:

```
$ which javac
/usr/bin/which: no javac in (/usr/lib64/qt-3.3/bin:/usr/libexec/lightdm: ...)
$ which java
/usr/bin/java
$ java -version
java version "1.7.0_65"
OpenJDK Runtime Environment (fedora-2.5.2.5.fc20-x86_64 u65-b17)
OpenJDK 64-Bit Server VM (build 24.65-b04, mixed mode)
```

Вам, возможно, придётся дополнительно установить из репозитория дистрибутива средства разработки Java (соответствующей версии JVM):

```
$ sudo yum install java-1.7.0-openjdk-devel.x86_64
...
Установлено:
  java-1.7.0-openjdk-devel.x86_64 1:1.7.0.65-2.5.2.5.fc20
Выполнено!
$ which javac
/usr/bin/javac
```

Теперь мы готовы выполнять компиляцию своего исходного кода Java в байт-код формата `.class`:

```
$ javac triangle.java
$ ls -l *.class
-rw-rw-r--. 1 Olej Olej 3721 окт 11 23:18 Complex.class
-rw-rw-r--. 1 Olej Olej 2101 окт 11 23:18 triangle.class
-rw-rw-r--. 1 Olej Olej 1192 окт 11 23:18 Tric.class
```

Независимо от того, помещены ли коды классов в один файл (как `triangle` и `Tric`) или импортируются из отдельных файлов как модули (как `Complex`), для каждого **класса** создаётся свой файл байт-кода.

Примечание: Java существенно более объектный язык, чем рассмотренный ранее C++: здесь всё обязано быть классом, или не быть вовсе. Именно поэтому для запуска приложения Java мы обязаны создать фиктивный стартовый класс `triangle`, по существу не нужный.

Выполнение подобно тому, что мы уже видели ранее (но это подобие и есть нашей целью — показана некоторая обработка ошибок данных):

```
$ java triangle
координаты вершин в формате: X Y
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1.0,1.0] [2.0,1.0] [1.0,2.0]
периметр = 3.414213562373095
площадь = 0.5
-----
координаты вершин в формате: X Y
вершина № 1 : 1
ошибка ввода!: java.util.NoSuchElementException
вершина № 1 : 1 2 3
вершина № 2 : 1 ff
ошибка ввода!: java.lang.NumberFormatException: For input string: "ff"
вершина № 2 : ^D завершение работы
```

Python



*«Ведь традиция, как ты понимаешь, это
не сохранение пепла, а поддержание огня.»
Павел Крусанов, «Мёртвый язык»*

Python — **высокоуровневый** язык программирования общего назначения (если в определении уровня отталкиваться от степени структурирования базовых типов данных, что справедливо — в этом смысле Python выше уровнем всех остальных языков, затрагиваемых в обзоре). Тем не менее, синтаксис языка Python минималистичен. Разработчиками утверждается, что язык ориентирован на повышение производительности разработчика и читаемости кода: «язык быстрой разработки». Разработка языка Python была начата в конце 1980-х годов.

Оригинальной «фишкой» синтаксиса является то, что в нём отсутствует понятие блока кода, и отсутствуют операторный скобки ограничивающие блок ({} — в C/C++, begin/end — в Pascal и Modula, и т.д.). Уровни вложенности кода в Python определяются величиной **отступа** в записи кода. Но это означает, что Python (**единственный** из рассматриваемых здесь) не является языком со «свободной» записью кода (неосторожный пробел перед строкой оператора приведёт к суровым синтаксическим сообщениям об ошибке).

Язык в высшей степени элегантный, красивый... Вот как может выглядеть реализация нашей задачи в Python:

triangle.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
import cmath
import math

def inpoint( prompt ):
    while( True ):
        sys.stdout.write( prompt )
        sys.stdout.flush()
        try:
            s = sys.stdin.readline()
        except KeyboardInterrupt:      # ^C - завершение работы
            print( ' завершение работы' )
            quit()
        if s == '':                    # ^D - завершение многоугольника
            sys.stdout.write( '\r' )
            return None
        try:
            ( x, y ) = s.split()
            p = complex( float( x ), float( y ) )
        except ValueError:
            print( 'ошибка ввода!' )
            continue
        return p

class triangle:
    """класс многоугольника"""
    def __init__( self, parm ):
        if isinstance( parm, list ):
            self.pts = parm
        else:
            self.pts = []
    def printt( self ):
        msg = 'вершин {} :'.format( len( self.pts ) )
        for x in self.pts:
            msg += ' [{:.2f},{:.2f}]'.format( x.real, x.imag )
```



```

        print( '{}'.format( msg ) )
def perimeter( self ):
    summa = 0.0;
    for i in range( len( self.pts ) ):
        if i == 0 :
            summa += abs( self.pts[ i ] - self.pts[ len( self.pts ) - 1 ] )
        else :
            summa += abs( self.pts[ i ] - self.pts[ i - 1 ] )
    return summa;
def square( self ):
    summa = 0.0;
    for i in range( len( self.pts ) - 2 ):
        n1, p1 = cmath.polar( self.pts[ i + 1 ] - self.pts[ 0 ] )
        n2, p2 = cmath.polar( self.pts[ i + 2 ] - self.pts[ 0 ] )
        summa += n1 * n2 * abs( math.sin( p2 - p1 ) ) / 2.
    return summa

while( True ):
    print( 'координаты вершин в формате: X Y (^D конец ввода)' )
    parms = []
    i = 1
    while 1:
        z = inpoint( 'вершина № ' + str( i ) + ' : ' )
        if( z == None ): break;
        parms.append( z )
        i += 1
    polygon = triangle( parms )
    polygon.printt()
    print( 'периметр = {:.2f}'.format( polygon.perimeter() ) )
    print( 'площадь = {:.2f}'.format( polygon.square() ) )
    print( '-----' )

```

Для выполнения такого приложения нужна исполняющая система, виртуальная машина Python, выполняющая байт-код (код Python не интерпретируется в чистом виде — он компилируется в промежуточный байт-код, после чего уже этот байт-код выполняется). Таких исполняющих систем предлагается много, наиболее старой и традиционной из которых является Cpython (устанавливаемая во всех операционных системах под именем команды python, в Windows часто используют графическую оболочку IDLE):

\$ python triangle.py

```

координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [1.00,1.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]
периметр = 4.00
площадь = 1.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1.00,1.00] [2.00,1.00] [1.00,2.00]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^C завершение работы

```

Важно!: Поскольку Python является интерпретирующей системой с динамической типизацией

переменных, в нём крайне легко (просто незаметно) реализуется изменение размерности структур (массивов) прямо по ходу выполнения. Неразумно было бы так дёшево не воспользоваться такой возможностью в нашем приложении! Поэтому здесь **и далее** мы изменим формулировку решаемой задачи: мы будем искать параметры (периметр и площадь) не треугольника, а произвольной размерности **выпуклого** многоугольника: 4-х угольника, 5-ти угольника, ... (в примере выше показано выполнение для квадрата). Причём, трансформация кода для этого произойдёт почти незаметно.

Для Python особенностью есть то, что для него одновременно существуют два поколения версий и в настоящее время в мире Python происходит «смена поколений»: на смену версиям 2.x идут версии 3.x. При этом они значительно не совместимы снизу-вверх по синтаксису (назрели изменения слишком радикальные). Продолжающиеся проекты продолжают в Python 2, а новые разработчики ещё не пересели в Python 3. Но, при определённой осторожности и изобретательности, Python-код можно (почти) всегда писать код так (при необходимости — т.к. это ведёт к некоторому усложнению кода), чтобы он **одинаково** исполнялся как в исполняющей системе 2-й версии, так и 3-й :

```
$ python3 triangle.py
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 3
вершина № 3 : 3 3
вершина № 4 : 3 1
вершин 4 : [1.00,1.00] [1.00,3.00] [3.00,3.00] [3.00,1.00]
периметр = 8.00
площадь = 4.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^C завершение работы
```

Ruby



«Я хочу, чтобы компьютер был моим слугой, а не господином, поэтому я должен уметь быстро и эффективно объяснить ему, что делать.»
Юкиhiro Мацумото

Ruby — интерпретируемый язык программирования высокого уровня. Поддерживает много разных парадигм программирования, прежде всего классово-объектную. Ruby был задуман в 1993 году японцем Юкиhiro Мацумото, стремившимся, по его утверждению, создать язык, совмещающий все качества других языков, способствующие облегчению труда программиста. По своим возможностям и выразительной мощности, Ruby примерно соответствует Python, у них много общего и в синтаксических конструкциях, при полной внешней непохожести структуры программы. Здесь, в Ruby, каждый блок кода структурируется так, что он начинается со служебного зарезервированного слова (часто это `do`), а заканчивается каждый блок словом `end`.

Ниже представлена реализация всё той же задачи на языке Ruby:

triangle.rb :

```
#!/usr/bin/ruby
# coding: utf-8

class Shape                                # класс многоугольника
  def initialize
    @points = []
  end
  def clean
    @points = []
  end
  def add val                               # добавление вершины
    @points[ @points.size ] = val
  end
  def size                                 # размерность многоугольника
    @points.size
  end
end
```

```

end
def to_s
  msg = ""
  @points.each do |point|
    msg += "[" + point.real.to_s + "," + point.imag.to_s + "]" "
  end
  msg
end
def perimeter
  summa = 0.0;
  for i in 0..( @points.size - 1 )
    if i == 0 then j = @points.size - 1
    else j = i - 1
    end
    summa += ( @points[ i ] - @points[ j ] ).abs
  end
  summa
end
def square
  summa = 0.0
  for i in 0..( @points.size - 3 )
    s1 = @points[ i + 1 ] - @points[ 0 ]
    s2 = @points[ i + 2 ] - @points[ 0 ]
    summa += 0.5 * s1.abs * s2.abs * Math.sin( s2.arg - s1.arg ).abs
  end
  summa
end
end

shape = Shape.new()
while true do
  puts( 'координаты вершин в формате: X Y (^D конец ввода)' )
  i = 1
  shape.clean
  while true do
    print "вершина № #{i.to_s} : "
    if ( ansv = gets ) == nil then
      print "\r"
      break
    end
    ansv.chomp
    m = ansv.split(/ /)
    if ( n = m.size ) != 2 ||
      m[ 0 ] !~ /\d+\.{0,1}\d*$/ ||
      m[ 1 ] !~ /\d+\.{0,1}\d*$/ then
      puts "ошибка ввода!"
      next
    end
    shape.add( Complex( m[ 0 ].to_f, m[ 1 ].to_f ) )
    i += 1
  end
  printf( "вершин %s : %s\n", shape.size.to_s, shape.to_s )
  printf( "периметр = %.2f\n", shape.perimeter.to_s )
  printf( "площадь = %.2f\n", shape.square.to_s )
  puts "-----"
end

```

Характерной чертой этого синтаксиса есть то, что параметры вызова функций (и методов) могут записываться не в скобках, а через пробел от имени функции. Для функций без параметров, а для методов объектов это наиболее частый случай, последовательные вызовы в такой записи могут порождать цепочки имён, вида: `shape.size.to_s`, `shape.perimeter.to_s`, ...

Выполнение показанного приложения:

\$ ruby triangle.rb

```
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [1.0,1.0] [1.0,2.0] [2.0,2.0] [2.0,1.0]
периметр = 4.00
площадь = 1.00
```

```
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 1
вершин 3 : [1.0,1.0] [1.0,2.0] [2.0,1.0]
периметр = 3.41
площадь = 0.50
```

```
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^Ctriangle.rb:58:in `gets': Interrupt
from triangle.rb:58:in `gets'
from triangle.rb:58:in `<main>'
```

Perl



*«О языке Perl и о том, что он не так уж страшен (если, конечно, не читать чужих программ), мы уже писали...»
Александр Темерев, «Царский путь к Java»*

Язык Perl — это классика в мире UNIX. Основной особенностью языка считаются его богатые возможности для работы с текстом, в том числе работа с регулярными выражениями, встроенная в синтаксис. Работа с регулярными выражениями Perl считается эталоном, на который ссылаются и который заимствуют многие другие проекты. Ларри Уолл начал разработку Perl в 1987 году, версия 1.0 была выпущена и анонсирована 18 декабря 1987 года. Долгое время Perl позиционировался как основной инструмент разработки скриптов и системного программирования UNIX, но в последние годы он заметно разделил эту нишу с Python и Ruby.

Вот как выглядит реализация обсуждаемой задачи в Perl:

triangle.pm :

```
#!/usr/bin/perl
use strict 'subs';
use Math::Complex;

sub perimeter {
    my @par = @_;
    $suma = 0.0;
    for( $i = 0; $i <= $#par; $i++ ) {
        if( $i == 0 ) { $j = $#par; }
        else { $j = $i - 1; }
        $suma += abs( $par[ $i ] - $par[ $j ] );
    }
    return $suma;
}

sub square {
    my @par = @_;
    $suma = 0.0;
    foreach $i ( 0 .. $#par - 2 ) {
```

```

        $s1 = $par[ $i + 1 ] - $par[ 0 ];
        $s2 = $par[ $i + 2 ] - $par[ 0 ];
        $suma += abs( $s1 ) * abs( $s2 ) *
                abs( sin( arg( $s2 ) - arg( $s1 ) ) ) / 2.;
    }
    return $suma;
}

while( "true" ) {
    print "координаты вершин в формате: X Y (^D конец ввода)\n";
    $i = 0;
    @polygon = ();
    while( "true" ) {
        print( "вершина № ", $i + 1, " : " );
        unless( defined( $line = <STDIN> ) ) { # ^D - конец ввода
            print "\r";
            last;
        }
        chop( $line );
        @parm = split( /\s+/, $line );
        if( $#parm != 1 or # только 2 координаты
            $parm[ 0 ] !~ /\d+\.{0,1}\d*$/ or # X - нечисловой
            $parm[ 1 ] !~ /\d+\.{0,1}\d*$/ ) { # Y - нечисловой
            print "ошибка ввода!\n";
            next;
        }
        $c = cplx( $parm[ 0 ], $parm[ 1 ] );
        push( @polygon, $c );
        $i++;
    }
    $msg = "вершин " . ( $#polygon + 1 ) . " : ";
    foreach $c ( @polygon ) {
        $re = Re( $c );
        $im = Im( $c );
        $msg .= "[$re,$im] ";
    }
    print "$msg\n";
    printf( "периметр = %.2f\n", perimeter( @polygon ) );
    printf( "площадь = %.2f\n", square( @polygon ) );
    print "-----\n";
}

```

Обратите внимание на выражение при вычислении площади, которое неумеху может просто свести с ума:

```
$suma += abs( $s1 ) * abs( $s2 ) * abs( sin( arg( $s2 ) - arg( $s1 ) ) ) / 2.;
```

Здесь, в одном выражении используется 3 упоминания функции с именем `abs()`, но это **разные** функции, не имеющие, временами, ничего общего между собой: первые 2 вызова применяются к комплексным аргументам, и означают **длины векторов**, а третий — к разности вещественных углов (величине вещественной), и означают просто **устранение знака** этой величины. Точно та же ситуация и во всех других реализациях, на других языках, но там ситуация смягчается некоторыми дополнительными «окрасками»: имя `fabs()` для вещественной функции, уточняющее имя пакета `Math.abs()` для этой функции и подобные штучки.

Теперь мы имеем право выполнить полученное приложение:

```

$ perl triangle.pm
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1,1] [2,1] [1,2]
периметр = 3.41
площадь = 0.50

```

```

-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [1,1] [1,2] [2,2] [2,1]
периметр = 4.00
площадь = 1.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^C

```

JavaScript



*«Если в стенах дома духу тягостно — постройка неверна.»
Павел Крусанов, «Ворон белый»*

Впервые услышав JavaScript, многие непроизвольно его ассоциируют с Java. Но эти два языка не имеют ничего общего, кроме всего одного слова в наименовании! JavaScript — прототипно-ориентированный сценарный язык программирования. JavaScript берёт начало от разработок компании Nombas 1992 от года и, главным образом, работ компании Netscape от 1995 года.

JavaScript обычно используется как встраиваемый язык для программного доступа к объектам охватывающих приложений. Общеизвестно использование JavaScript в браузерах, но это далеко не единственная область: он используется как инструмент конфигурирования, настройки и быстрого написания управляющих скриптов в VoIP коммутаторах Asterisk, FreeSWITCH. Но JavaScript может использоваться и автономно, как инструмент консольных приложений, что гораздо менее известно.

Если для всех предыдущих языков инструментальные программы у вас будут, скорее всего, уже присутствовать в системе после установки (Python и Perl, например, просто необходимы для работы ряда утилит системы), то консольный JavaScript интерпретатор, со столь же большой вероятностью, в системе будет отсутствовать. Называется такой клиент SpiderMonkey от компании Mozilla. Для экспериментов вам придётся его установить, например, в Debian (в других дистрибутивах это делается аналогично):

```

$ sudo apt-get install spidermonkey-bin
...
Необходимо скачать 1.384 кБ архивов.
...

```

Примечание: В принципе, у вас могут быть в системе Linux другие «родные» интерпретаторы: kjs (KDE), gjs, seed (GNOME), но их синтаксис и библиотеки будут несовместимы, и показанный ниже код придётся править.

Теперь мы можем создать ещё одну реализацию приложения на JavaScript:

triangle.js :

```

#!/usr/bin/js -U

function Complex( re, im ) { // конструктор
    this.x = re; this.y = im;
}

// определение методов экземпляра в объекте-прототипе конструктора ...
Complex.prototype.abs = function() {
    return Math.sqrt( this.x * this.x + this.y * this.y );
}

Complex.prototype.arg = function() {
    return Math.atan2( this.y, this.x );
}

```

```

Complex.prototype.sub = function( that ) {
    return new Complex( this.x - that.x, this.y - that.y );
}

Complex.prototype.toString = function() {
    return " [" + this.x.toFixed( 2 ) + ", " + this.y.toFixed( 2 ) + "]";
}

Complex.sub = function( a, b ) {      // определение методов класса (static, friend)
    return new Complex( a.x - b.x, a.y - b.y );
}

var perimeter = function( triang ) { // функциональные литералы
    var summa = 0.0;
    for( i = 0; i < triang.length; i++ )
        summa += Complex.sub( triang[ i ],
                               triang[ i == 0 ? triang.length - 1 : i - 1 ] ).abs();
    return summa;
},
square = function( triang ) {
    var summa = 0.0;
    for( i = 0; i < triang.length - 2; i++ ) {
        side1 = triang[ i + 1 ].sub( triang[ 0 ] );
        side2 = triang[ i + 2 ].sub( triang[ 0 ] );
        summa += side1.abs() * side2.abs() *
                 Math.abs( Math.sin( side1.arg() - side2.arg() ) ) / 2.;
    }
    return summa
};

while( true ) {
    print( "координаты вершин в формате: X Y (^D конец ввода)" );
    var i = 0;
    var polygon = new Array();
    while( true ) {
        putstr( "вершина №" + ( i + 1 ) + " : " );
        var s = readline();
        if( s === null ) {      // ^D
            putstr( "\r" );
            break;
        }
        var ws = s.split( ' ' );
        if( ws.length !== 2 ||
            isNaN( x = parseFloat( ws[ 0 ] ) ) ||
            isNaN( y = parseFloat( ws[ 1 ] ) ) ) {
            print( "ошибка ввода!" );
            continue;
        }
        polygon[ i++ ] = new Complex( x, y );
        delete x; delete y;
    }
    s = "вершин " + polygon.length + " :";
    for( i in polygon ) s += polygon[ i ];
    print( s );
    print( "периметр = " + perimeter( polygon ).toFixed( 2 ) );
    print( "площадь = " + square( polygon ).toFixed( 2 ) );
    print( "-----" );
}

```

Выполнение приложения (опция -U — **обязательна!**, только в этом случае текстовые строки

программы будут восприниматься в UNICODE, и будет обеспечен вывод русского текста):

```
$ js -U triangle.js
```

```
координаты вершин в формате: X Y (^D конец ввода)
```

```
вершина №1 : 1 1
```

```
вершина №2 : 2 1
```

```
вершина №3 : 1 2
```

```
вершин 3 : [1.00,1.00] [2.00,1.00] [1.00,2.00]
```

```
периметр = 3.41
```

```
площадь = 0.50
```

```
-----  
координаты вершин в формате: X Y (^D конец ввода)
```

```
вершина №1 : 1 1
```

```
вершина №2 : 1 2
```

```
вершина №3 : 2 2
```

```
вершина №4 : 2 1
```

```
вершин 4 : [1.00,1.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]
```

```
периметр = 4.00
```

```
площадь = 1.00
```

```
-----  
координаты вершин в формате: X Y (^D конец ввода)
```

```
вершина №1 : ^C
```

В рамках развивающегося JavaScript было рождено много новых и неординарных решений: техника AJAX (Asynchronous Javascript And Xml), формат JSON (JavaScript Object Notation) и др., которые перешагнули за границы только JavaScript использования. Тем не менее, сам язык в своей консольной (операционной) части является ограниченным, что относится в равной мере и к другим языкам программирования подобного предназначения: PHP, Lua и т.д.

PHP



*«Острый взгляд, цепкая мысль, пружинистый шаг,
разборчивый почерк — всё это в прошлом.»
Павел Крусанов, «Ворон белый»*

PHP (Hypertext Preprocessor) — скриптовый язык программирования **общего назначения**. PHP ведёт родословную от Perl/CGI, и берёт начало развития с 1994 года. Но PHP стал так интенсивно применяться для разработки веб-приложений, что стал одним из лидеров среди языков программирования, применяющихся для создания динамических веб-сайтов. Многие и встречали его только в таком единственном качестве. Но это неверно, и мы его используем для построения консольного приложения. Кроме того, такой способ использования, пожалуй, оптимальный для разбирательства особенностей языка и его тонких возможностей.

Эквивалент приложения, выраженный на языке PHP:

triangle.php :

```
#!/usr/bin/php
```

```
<?php
```

```
class Complex {
```

```
    var $x, $y;
```

```
    function __construct( $re, $im ) {        // конструктор
```

```
        $this->x = $re;
```

```
        $this->y = $im;
```

```
    }
```

```
    function abs() {
```

```
        return sqrt( $this->x * $this->x + $this->y * $this->y );
```

```
    }
```

```
    function arg() {
```

```
        return atan2( $this->y, $this->x );
```



```

    }
    function sub( $that ) {
        return new Complex( $this->x - $that->x, $this->y - $that->y );
    }
}

class Point extends Complex {          // конструктор
    function toString() {
        $s = sprintf( "[%2f,%2f] ", $this->x, $this->y );
        return $s;
    }
}

function perimeter( $triang ) {
    $sum = 0.0;
    for( $i = 0; $i < sizeof( $triang ); $i++ ) {
        $j = $i == 0 ? sizeof( $triang ) - 1 : $i - 1;
        $sum += $triang[ $i ]->sub( $triang[ $j ] )->abs();
    }
    return $sum;
}

function square( $triang ) {
    $sum = 0.0;
    for( $i = 0; $i < sizeof( $triang ) - 2; $i++ ) {
        $side1 = $triang[ $i + 1 ]->sub( $triang[ 0 ] );
        $side2 = $triang[ $i + 2 ]->sub( $triang[ 0 ] );
        $sum += $side1->abs() * $side2->abs() *
            abs( sin( $side1->arg() - $side2->arg() ) ) / 2.;
    }
    return $sum;
}

while( TRUE ) {
    print "координаты вершин в формате: X Y (^D конец ввода)\n";
    $polygon = array();    // динамический массив - образ многоугольника
    $i = 0;
    while( TRUE ) {
        echo "вершина № ", ( $i + 1 ), " : ";
        $line = stream_get_line( STDIN, 1024, PHP_EOL );
        if( strlen( $line ) == 0 ) {          // ^D - конец ввода многоугольника
            echo "\r";
            break;
        }
        $pieces = explode( " ", $line );
        if( sizeof( $pieces ) != 2 ) {
            echo "ошибка ввода!\n";
            continue;
        }
        $fmask = "+-0123456789.Ee";
        if( 0 == strpos( $pieces[ 0 ], $fmask ) || 0 == strpos( $pieces[ 1 ], $fmask ) ) {
            echo "ошибка ввода!\n";
            continue;
        }
        $polygon[] = new Point( (float)$pieces[ 0 ], (float)$pieces[ 1 ] ); // дополнение! в массив
        $i++;
    }
    $s = "вершин " . sizeof( $polygon ) . " : ";
    for( $i = 0; $i < sizeof( $polygon ); $i++ )
        $s .= $polygon[ $i ]->toString();
    echo $s, "\n";
}

```

```

printf( "периметр = %.2f\n", perimeter( $polygon ) );
printf( "площадь = %.2f\n", square( $polygon ) );
print "-----\n";
}
?>

```

Выполнение приложения:

\$ php triangle.php

```

координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1.00,1.00] [2.00,1.00] [1.00,2.00]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [1.00,1.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]
периметр = 4.00
площадь = 1.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^C

```

Lua



«Эти правила, язык и грамматика Игры, представляют собой некую разновидность высокоразвитого тайного языка, в котором участвуют самые разные науки и искусства ..., и который способен выразить и соотнести содержание и выводы чуть ли не всех наук.»
Герман Гессе, «Игра в бисер»

Lua (Луна, исп.) — интерпретируемый язык программирования, разработанный подразделением Tecgraf Католического университета Рио-де-Жанейро, история языка ведёт отсчёт с 1993 года. Изначально Lua создавался как язык программирования баз данных. Фактически, все программирование на Lua сводится к различным манипуляциям с таблицами. Динамические таблицы — это краеугольный камень философии Lua.

В обсуждениях утверждается, что по возможностям, идеологии и реализации язык ближе всего к JavaScript, однако Lua «отличается более мощными и гораздо более гибкими конструкциями». Реализуемая модель объектно-ориентированного программирования — прототипная (как и в JavaScript). Lua — язык чрезвычайно популярный среди разработчиков компьютерных игр.

С инструментарием Lua вас могут возникнуть та же история, что со SpiderMonkey для JavaScript. Проверьте версию Lua, если она даже установлена у вас в системе по умолчанию, запустив программу, например, в режиме интерактивной оболочки:

```

$ lua
Lua 5.0.3 Copyright (C) 1994-2006 Tecgraf, PUC-Rio
>

```

Но это настолько старая версия, что она не отрабатывает даже ряд стандартных синтаксических конструкций, описываемых справочником по языку. Необходимо будет средствами операционной системы (из репозитория) установить свежую версию, например, такую:

```

$ lua
Lua 5.2.2 Copyright (C) 1994-2013 Lua.org, PUC-Rio

```

> ^C

Это уже совсем другое дело — разница в 7 лет!

Теперь мы готовы реализовывать вариант приложения на языке Lua:

***triangle.lua* :**

```
#!/usr/bin/lua

-- https://github.com/davidm/lua-matrix/blob/master/lua/complex.lua
local complex = require 'complex'

inpoint = function( prompt )
    io.write( prompt )
    local s = io.read ( )
    if( s == nil ) then
        print '\r'
        return true
    end
    local res = {}
    for w in string.gmatch( s, "%S+" ) do
        table.insert( res, w )
    end
    if( #res ~= 2 ) then return nil; end
    local ret, x = pcall( string.format, "%e", res[ 1 ] )
    if( not ret ) then return nil; end
    local ret, y = pcall( string.format, "%e", res[ 2 ] )
    if( not ret ) then return nil; end
    return x, y -- nil если не число
end

local perimeter = function( triang )
    local function length( p1, p2 )
        n, t = complex.polar( p2 - p1 )
        return n
    end
    local suma = 0.0
    for i, v in ipairs( triang ) do
        if( i == 1 ) then j = #triang
        else j = i - 1
        end
        suma = suma + length( triang[ i ], triang[ j ] )
    end
    return suma
end

local square = function( triang )
    local suma = 0.0
    for i = 1, #triang - 2, 1 do
        -- таблица triang индексируется с 1: n = re, t = im
        local n1, t1 = complex.polar( triang[ i + 1 ] - triang[ 1 ] )
        local n2, t2 = complex.polar( triang[ i + 2 ] - triang[ 1 ] )
        suma = suma + n1 * n2 * math.abs( math.sin( t2 - t1 ) ) / 2.;
    end
    return suma
end

while true do
    print( 'координаты вершин в формате: X Y (^D конец ввода)' )
    local polygon = {}
    local i = 0
    while true do
        x, y = inpoint( 'вершина №' .. ( i + 1 ) .. ' : ' )
```

```

    if( not x ) then print 'ошибка ввода!' -- nil == false
    elseif ( type( x ) == 'boolean' and x ) then break -- EOF - конец ввода
    else
        polygon[ i + 1 ] = complex.new( x, y ) -- добавить (!) вершину
        i = i + 1
    end
end
end
io.write( 'вершин ' .. #polygon .. ' : ' )
for i, v in ipairs( polygon ) do
    local msg = " [" .. string.format( "%.2f", v[ 1 ] )
    msg = msg .. ", " .. string.format( "%.2f", v[ 2 ] ) .. "]"
    io.write( msg )
end
print ''
print( 'периметр = ' .. string.format( "%.2f", perimeter( polygon ) ) )
print( 'площадь = ' .. string.format( "%.2f", square( polygon ) ) )
print "-----"
end
end

```

Для выполнения комплексных вычислений (так же как это было с Java) мы позаимствуем модуль с ресурса, URL которого указан в комментарии внутри кода.

Выполнение такого приложения:

```

$ lua triangle.lua
координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : 1 1
вершина №2 : 2 1
вершина №3 : 1 2
вершина №4 :
вершин 3 : [1.00,1.00] [2.00,1.00] [1.00,2.00]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : 1 1
вершина №2 : 1 2
вершина №3 : 2 2
вершина №4 : 2 1
вершина №5 :
вершин 4 : [1.00,1.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]
периметр = 4.00
площадь = 1.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : ^C^C

```

Командный интерпретатор bash



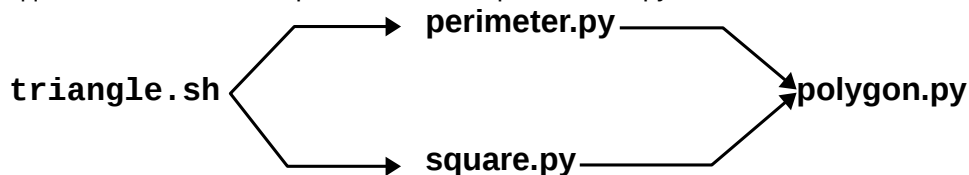
«Неверие Бикса Константина в ад было так же несокрушимо, как и неверие в рай. Тем не менее он довольно четко представлял, чем первое отличается от второго.»
Джеймс Морроу, «Единородная дочь»

Язык оболочки bash (Bourne Again SHell) — усовершенствованная и модернизированная вариация командной оболочки Bourne-shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX. Особенно популярна в среде Linux, где она зачастую используется в качестве предустановленного командного интерпретатора. Bourne-shell — одна из популярных разновидностей командного интерпретатора для UNIX, которая с переменным успехом развивается разными авторами начиная с 1978 года. Язык bash имеет много существенных

расширений, но подобное тому, что будет показано ниже, можно сделать практически на любом командном интерпретаторе UNIX. Часто ошибочно считают, что предназначение `bash` в написании коротких, в 2-3 строчки, командных скриптов операционной системы (эта дурная традиция идёт, скорее всего, от аналогий с командными файлами `.bat` в Windows и ограниченностью их командного языка). Но на сегодня известен уже целый ряд успешных открытых проектов, объёмами до тысяч строк, которые полностью выписаны на `bash` — в каком-то смысле (в определённых кругах) это считается высшим пилотажем.

Самым существенным ограничением для **нашей** эталонной задачи будет то, что в `bash` вообще нет такого понятия как вещественное значение (не говоря уже о комплексном), и что формат вещественного числа он может интерпретировать только как текстовую строку, удовлетворяющую определённому регулярному выражению. Но и это не станет нас смущать: мы можем реализовать всю логику и формирование топологии вершин многоугольника в `bash`, а на расчёт (комплексный) периметра и площади передать эту сформированную топологию внешним приложениям (в файлах), подобным тем, которые обсуждались выше. Эти дочерние процессы и вернут нам в родительскую программу требуемые результаты расчётов. Это несколько витиевато, но наша главная задача — иллюстрация ...

В качестве основы для внешних приложений может быть взят код любого из показанных ранее вариантов, на **любом** языке. Но использовать к этому качеству **компилирующие** реализации C/C++ или Java неразумно, если мы будем компоновать всё это в единое **интерпретируемое** приложение. В качестве инструмента для написания вспомогательных приложений мной выбран Python. Кроме того, поскольку оба расчётных приложения (`perimeter.py` и `square.py`) используют большинство общего кода, то такой код вынесен в отдельный **импортируемый** модуль (`polygon.py`). Схема взаимодействия компонент приложения изображена на грубой схеме:



Вариант такой реализации всё того же эквивалентного приложения на языке командного интерпретатора Linux приведен ниже (подкаталог `bash` в примерах):

triangle.sh :

```
#!/bin/bash

while [ TRUE ]
do
  declare -a polygon_x
  declare -a polygon_y
  do
    i=1
    echo "координаты вершин в формате: X Y (^D конец ввода)"
    while [ TRUE ]
    do
      echo -n "вершина № $i : "
      declare -a parm
      read -a parm
      element_count=${#parm[@]}          # число введенных значений
      if [[ $element_count -eq 0 ]]      # ^D - конец ввода
      then
        echo -en "\r\a"
        break
      elif [[ $element_count -ne 2 ]]    # неправильное число элементов ввода
      then
        echo "ошибка ввода!"
        continue
      fi
      polygon_x[ $i ]=${parm[0]}
      polygon_y[ $i ]=${parm[1]}
      i=`expr $i + 1`
    done
```

```

i=${#polygon_x[@]}
echo -n "вершин $i : "
polystr=""
for (( j=1; j <= i ; j++ ))      # двойные круглые скобки и "j" без "$".
do
    polystr="$polystr [{polygon_x[j]}, {polygon_y[j]}] "
done
echo $polystr
cmd="./perimeter.py \" $polystr \""
ret=`eval $cmd`
echo "периметр = $ret"
cmd="./square.py \" $polystr \""
ret=`eval $cmd`
echo "площадь = $ret"
echo -----
done

```

Код, выполняющийся как отдельный дочерний процесс, рассчитывающий периметр многоугольника:

perimeter.py :

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

from polygon import *

def perimeter( pts ):
    summa = 0.0;
    for i in range( len( pts ) ):
        if i == 0 :
            summa += abs( pts[ i ] - pts[ len( pts ) - 1 ] )
        else :
            summa += abs( pts[ i ] - pts[ i - 1 ] )
    return summa;

print( "{:.2f}".format( perimeter( polygon( instr() ) ) ) ) )
quit( 0 )

```

Код, выполняющийся как отдельный дочерний процесс, рассчитывающий площадь многоугольника:

square.py :

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

import math
from polygon import *

def square( pts ):
    summa = 0.0;
    for i in range( len( pts ) - 2 ):
        n1, p1 = cmath.polar( pts[ i + 1 ] - pts[ 0 ] )
        n2, p2 = cmath.polar( pts[ i + 2 ] - pts[ 0 ] )
        summa += n1 * n2 * abs( math.sin( p2 - p1 ) ) / 2.
    return summa

print( "{:.2f}".format( square( polygon( instr() ) ) ) ) )
quit( 0 )

```

Импортируемый модуль — это библиотека функций, общих для обоих процессов

perimeter.py и square.py:

polygon.py :

```
# -*- coding: utf-8 -*-
import sys
import cmath
import string
import re

def instr():
    arg = ''
    if len( sys.argv ) == 2 :      # ввод из командной строки
        arg = sys.argv[ 1 ]
    elif len( sys.argv ) > 2 :
        print( "ошибка формата команды!" )
        quit( 1 )
    else:                          # ввод из SYSIN
        while( True ):
            s = sys.stdin.readline()
            if s == '': break
            arg += ' ' + s
    return arg

def polygon( s ):
    lst = s.split( ' ' )
    parms = []
    for i in range( len( lst ) ):
        if lst[ i ] == '': continue
        tmatch = re.search( r'\[(.*?),(.*)\]', lst[ i ] )
        if not tmatch: continue
        try:
            p = complex( float( tmatch.group( 1 ) ), float( tmatch.group( 2 ) ) )
        except ValueError:
            continue
        parms.append( p )
    return parms;

if __name__ == '__main__':
    si = instr()
    print( 'ввод: {}'.format( si ) )
    pts = polygon( si )
    msg = 'введено {} : '.format( len( pts ) )
    for i in range( len( pts ) ):
        msg += '{} '.format( pts[ i ] )
    print( msg )
```

Выполнение такого варианта приложения:

\$ sh triangle.sh

```
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 2 1
вершина № 3 : 1 2
вершин 3 : [1,1] [2,1] [1,2]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2
```

```

ошибка ввода!
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [1,1] [1,2] [2,2] [2,1]
периметр = 4.00
площадь = 1.00
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 :

```

В принципе, в приложении на `bash`, этом или подобных ему интерпретаторах, мы могли бы обойтись вообще без каких-либо дополнительных вычислительных скриптов, используя для вычислений стандартный консольный калькулятор Linux `bc`. Вот как может выглядеть элементарный пример команды **вещественных** вычислений для `bc` с терминала:

```
$ echo 'sqrt(4.5 * 2.0)' | bc3.0
```

По аналогии, используя такие команды конвейерных операций, мы могли бы записывать в коде `bash`-приложения вычислительные операции, выполняемые над переменными скрипта. Например, вида:

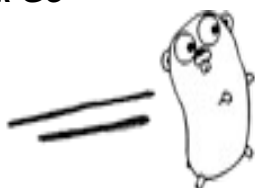
```

side=$(echo "sqrt((${polygon_x[j]}-${polygon_x[k]})^2+\
              (${polygon_y[j]}-${polygon_y[k]})^2)" | bc)
perimeter=$(echo "$perimeter+$side" | bc)

```

Вариант приложения в такой нотации (файл `triangle.bc.sh`) находится (для сравнения) в составе архива примеров (подкаталог `other-variants` в примерах):, но мы не будем детально останавливаться на рассмотрении его кода.

Язык Go



Go — **компилируемый**, многопоточный язык программирования, разрабатываемый компанией Google. Первоначальная разработка Go началась в сентябре 2007 года, а его непосредственным проектированием занимались авторы, непосредственно стоявшие у истоков создания языка C и операционной системы UNIX: Роберт Гризмер, Роб Пайк и Кен Томпсон, занимавшиеся накануне этой работы проектом разработки операционной системы Plan 9, которая должна была прийти на смену UNIX. Официально язык был представлен в ноябре 2009 года.

На данный момент существуют два компилятора Go (собственно, это две согласованные ветви одного процесса развития):

- непосредственно в рамках основного проекта GoLang (6g и 8g для 64-битных платформ и общей архитектуры x86, соответственно, и сопутствующие им инструменты, вместе известные под названием `gc`).
- `gccgo` — ещё один компилятор Go, базирующийся на знакомой всем пользователям Linux системе компиляторов GNU (поддержка Go доступна в GCC начиная с версии 4.6).

Компилятор `gc` (проект GoLang) полагаются полностью на собственный код — создаваемый код не является управляемым, то есть для его работы не нужна никакая виртуальная машина (исполняющая система). По словам Роба Пайка, получаемый после компиляции байт-код совершенно автономен.

Компилятор `gccgo`, естественно (это составная часть проекта GCC) компилирует тот же код в приложения, динамически связанные в Linux со стандартной библиотекой `C libc.so`.

В **2009** Go был признан языком года по версии организации TIOBE.

Обе линии Go доступны в вашей Linux системе:

```

$ aptitude search golang*
p  golang                - Go programming language compiler - metapackage
p  golang-dbg             - Go programming language compiler - debug files

```



```

p  golang-doc          - Go programming language compiler - documentation
p  golang-go           - Go programming language compiler
p  golang-mode         - Go programming language - mode for GNU Emacs
p  golang-src          - Go programming language compiler - source files
v  golang-tools        -
v  golang-weekly       -
v  golang-weekly-dbg   -
v  golang-weekly-doc   -
v  golang-weekly-go    -
v  golang-weekly-src   -
v  golang-weekly-tools -
olej@notebook:~$ aptitude search gccgo*
p  gccgo               - Go compiler, based on the GCC backend
p  gccgo-4.6-doc       - documentation for the GNU Go compiler (gccgo)
p  gccgo-4.7           - GNU Go compiler
p  gccgo-4.7-doc       - documentation for the GNU Go compiler (gccgo)
p  gccgo-4.7-multilib  - GNU Go compiler (multilib files)
p  gccgo-multilib      - Go compiler, based on the GCC backend (multilib files)

```

Вам необходимо установить какой-либо из них ... , а ещё лучше — оба:

```
$ sudo apt-get install gccgo
```

```
...
```

```
Настраивается пакет gccgo (4:4.7.2-1) ...
```

```
$ gccgo --version
```

```
gccgo (Debian 4.7.2-5) 4.7.2
```

```
Copyright (C) 2012 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Язык Go является прямым развитием линии C/C++, с заимствованиями многих «находок» из Oberon, Python и скриптовых языков.

Реализация эталонной задачи на языке Go:

triangle.go :

```

package main

import (
    "fmt"; "os"; "io"; "errors"
    "strings"; "strconv"; "math"; "math/cmplx"
)

// ----- класс точки вершины -----
type point struct {
    xy complex128
}
func (p *point) String() string {                // формат вывода
    return fmt.Sprintf( "[%2f,%2f] ", real( p.xy ), imag( p.xy ) )
}
func (p *point) inpoint() ( ok bool, err error ) { // ввод координат
    buf := make( [] byte, 1024 )
    ok = false
    n, err := os.Stdin.Read( buf )
    if err == io.EOF || n == 0 || buf[ n - 1 ] != '\n' { // конец ввода
        err = io.EOF
        return
    }
    as := strings.Split( string( buf[ : n - 1 ] ), string( " " ) )
    if len( as ) != 2 {
        err = errors.New( "число параметров" )
        return
    }
}

```

```

    }
    x, err := strconv.ParseFloat( as[ 0 ], 64 )
    if err != nil { return } // ошибка преобразования
    y, err := strconv.ParseFloat( as[ 1 ], 64 )
    if err != nil { return } // ошибка преобразования
    p.xy = complex( x, y )
    ok, err = true, nil
    return
}

// ----- класс многоугольника -----
type shape []point
func ( p *shape ) append( data point ) {
    slice := *p
    l := len( slice )
    if l + 1 > cap( slice ) { // недостаточно места
        newSlice := make( [] point, ( l + 1 ) * 2 ) // выделить вдвое большего буфера
        if l > 0 { // скопировать данные
            for i, c := range slice { newSlice[ i ] = c }
        }
        slice = newSlice
    }
    slice = slice[ 0 : l + 1 ]
    slice[ l ] = data
    *p = slice;
}
func ( p *shape ) String() string { // формат вывода
    slice := *p
    var s string = ""
    for _, c := range slice { s += c.String() }
    return s
}
func ( p *shape ) perimeter() float64 {
    summa := 0.0
    slice := *p
    for i, c := range slice {
        if i == 0 {
            summa += cmplx.Abs( c.xy - slice[ len( slice ) - 1 ].xy )
        } else {
            summa += cmplx.Abs( c.xy - slice[ i - 1 ].xy )
        }
    }
    return summa
}
func ( p *shape ) square() float64 {
    summa := 0.0
    slice := *p
    for i := 0; i < len( slice ) - 2; i++ {
        r1, 01 := cmplx.Polar( slice[ i + 1 ].xy - slice[ 0 ].xy )
        r2, 02 := cmplx.Polar( slice[ i + 2 ].xy - slice[ 0 ].xy )
        summa += r1 * r2 * math.Abs( math.Sin( 02 - 01 ) ) / 2.
    }
    return summa
}

func main() {
    for {
        fmt.Println( "координаты вершин в формате: X Y" )
        многоугольник := new( shape )
        i := 0
        точка := new( point )
    }
}

```

```

    for {
        fmt.Printf( "вершина № %v: ", i + 1 )
        ok, err := точка.inpoint() // ввод координат вершины
        if !ok {
            if err == io.EOF { fmt.Printf( "\r" ); break } // конец ввода вершин
            fmt.Printf( "ошибка ввода: %s!\n", err )
            continue
        }
        многоугольник.append( *точка )
        i++
    }
    fmt.Printf( "вершин %d : %v\n", len( *многоугольник ), многоугольник )
    fmt.Printf( "периметр = %.2f\n", многоугольник.perimeter() )
    fmt.Printf( "площадь = %.2f\n", многоугольник.square() )
    fmt.Println( "-----" )
}
}

```

Сборка 2-мя различными компиляторами:

```

$ gccgo -g triangle.go -o triangle_go
...
$ go build -o triangle_gc -compiler gc triangle.go
...
$ ls -l *_g*
-rwxrwxr-x. 1 Olej Olej 2031360 окт 13 19:58 triangle_gc
-rwxrwxr-x. 1 Olej Olej 48597 окт 13 19:58 triangle_go

```

В иллюстрацию сказанного ранее о различиях двух компиляторов, посмотрим те разделяемые библиотеки, которые используют каждое из созданных приложений:

```

$ ldd triangle_go
linux-vdso.so.1 => (0x00007fffe87c5000)
libgo.so.4 => /lib64/libgo.so.4 (0x00007f9819299000)
libm.so.6 => /lib64/libm.so.6 (0x000000308f400000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003090800000)
libc.so.6 => /lib64/libc.so.6 (0x000000308e800000)
/lib64/ld-linux-x86-64.so.2 (0x000000308e400000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x000000308f000000)
$ ldd triangle_gc
не является динамическим исполняемым файлом

```

Здесь 2-е (GoLang) приложение (triangle_gc) является полностью автономным, и не использует стандартную библиотеку языка C в Linux.

И вот как выполняется только-что собранное нами приложение и, в частности, обрабатывает ошибки ввода пользователем:

```

$ ./triangle_go
координаты вершин в формате: X Y
вершина № 1: 1. 1.
вершина № 2: 1. 2.
вершина № 3: 2. 1.
вершин 3 : [1.00,1.00] [1.00,2.00] [2.00,1.00]
периметр = 3.41
площадь = 0.50
-----
координаты вершин в формате: X Y
вершина № 1: 1. 1.
вершина № 2: 1. 2.
вершина № 3: 2. 2.
вершина № 4: 2. 1.
вершин 4 : [1.00,1.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]
периметр = 4.00

```

```

площадь = 1.00
-----
координаты вершин в формате: X Y
вершина № 1: 3.3
ошибка ввода: число параметров!
вершина № 1: 1. 2. 3. 4.
ошибка ввода: число параметров!
вершина № 1: 2.2 4.r
ошибка ввода: strconv.ParseFloat: parsing "4.r": invalid syntax!
вершина № 1: k 5
ошибка ввода: strconv.ParseFloat: parsing "k": invalid syntax!
вершина № 1: ^C

```

Язык чрезвычайно изящный: синтаксис во многом повторяющий С (без необходимости лишних разделителей ';', завершающих каждый оператор), дополненный своеобразными механизмами классов и объектов (хотя и названных другими терминами), но без громоздкости и тяжеловесности языка С++. Из интересных особенностей обратим внимание для начала на операторы (повторим их специально):

```

r1, θ1 := cmplx.Polar( slice[ i + 1 ].xy - slice[ 0 ].xy )
r2, θ2 := cmplx.Polar( slice[ i + 2 ].xy - slice[ 0 ].xy )
summa += r1 * r2 * math.Abs( math.Sin( θ2 - θ1 ) ) / 2.

...
многоугольник := new( shape )

...
точка := new( point )

...
ok, err := точка.inpoint()                                // ввод координат вершины

...
многоугольник.append( *точка )

```

Ничего удивительного — просто последовательная поддержка UNICODE в кодировке UTF-8 в языке, ведь Роб Пайк, один из идеологов Go, и был разработчиком системы кодирования UTF-8. В качестве идентификаторов допускаются символы **алфавита любого языка**, или **математические символы**.

Язык Go приятно сочетает в себе лаконизм и ясность С, с такими вещами (например из Python), как множественные возвраты из функций, простота представления и лёгкость работы со строками и др. Но главная «фишка» Go не в этом. Язык предназначен для поддержания параллельного выполнения (реального, а не квази) на нескольких процессорах (ядрах). Для этого **любую** функцию Go можно запустить выполняться в отдельном потоке (оператором go). Параллельно выполняющиеся ветви выполняются как **сопрограммы**, и могут обмениваться между собой **синхронными** сообщениями через двунаправленные каналы. Через каналы могут передаваться данные любых типов. Но чем 10 раз это рассказывать, лучше 1 раз это показать:

multy.go :

```

package main

import (
    "fmt"
    "time"
    "os"
)

func child( num int, in, out chan string ) {
    str1 := fmt.Sprintf( "%v : ", num )
    for {
        str2 := <- in                // строка полученная из входного канала
        fmt.Println( str1 + str2 )
        if out != nil { out <- str2 } // ретранслируется в выходной канал
    }
}

```

```

func ввод( ch chan string ) {
    const per = 300000000
    buf := make( [] byte, 1024 )
    for {
        fmt.Printf( "> " )
        n, _ := os.Stdin.Read( buf )
        str := string( buf[ : n - 1 ] )
        fmt.Println( str )
        ch <- str
        time.Sleep( per )
    }
}

func main(){
    канал := [...] chan string { make( chan string ), make( chan string ),
                                make( chan string ), make( chan string ) }

    for i := range канал {
        if i != len( канал ) - 1 {
            go child( i, канал[ i ], канал[ i + 1 ] )
        } else {
            go child( i, канал[ i ], nil )
        }
    }
    ввод( канал[ 0 ] )
}

```

Вот как происходит выполнение такого приложения, в котором, как вы понимаете, произвольное число параллельных ветвей (определяется размерностью заданного нами массива), и каждая ветвь выполняется на отдельном процессоре:

```

$ ./multy
> ввод
ввод
0 : ввод
1 : ввод
2 : ввод
3 : ввод
> srting from terminal
srting from terminal
0 : srting from terminal
1 : srting from terminal
2 : srting from terminal
3 : srting from terminal
> 123 456 789.000
123 456 789.000
0 : 123 456 789.000
1 : 123 456 789.000
2 : 123 456 789.000
3 : 123 456 789.000
> ^C

```

Таким образом, язык Go **предвосхитил** (к началу разработки в 2007г. это ещё не было очевидным) тотальный переход всего компьютерного железа на многоядерность (многопроцессорность) и возможности параллельной обработки на многих процессорах. И то, что в ближайшее время совершенно ординарной архитектурой станет даже не 2 ядра, а 16, 32, 64. В этом язык Go в чём то наследует процедурному языку параллельного программирования Оссам, разработанному в начале 1980-х годов для программирования транспьютеров.

Scheme

Scheme — это функциональный язык программирования, один из двух наиболее популярных в наши дни диалектов языка Лисп (Lisp, другой популярный диалект — это Common Lisp). Авторы языка Scheme — Гай Стил (англ. Guy L. Steele) и Джеральд Сассмен (англ. Gerald Jay Sussman) из Массачусетского технологического института — создали его в середине 1970-х годов. Scheme, как это не покажется странным, достаточно широко используется или использовался **в промышленности**, такими компаниями, как Texas Instruments, Tektronix, Hewlett Packard, Sun Microsystems.

Scheme придётся устанавливать в системе дополнительно:

```
$ yum list guile
Доступные пакеты
guile.i686                    5:2.0.9-4.fc20                    fedora
$ sudo yum install guile
...
$ guile --version
guile (GNU Guile) 2.0.9
Copyright (C) 2013 Free Software Foundation, Inc.
...
```

Интерпретатор Scheme может использоваться в режиме интерактивного консольного калькулятора:

```
$ guile
GNU Guile 2.0.9
Copyright (C) 1995-2013 Free Software Foundation, Inc.
Guile comes with ABSOLUTELY NO WARRANTY; for details type `,show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type `,show c' for details.
Enter `,help' for help.
scheme@(guile-user)> (* 3 7)
$1 = 21
scheme@(guile-user)> (sqrt $1)
$2 = 4.58257569495584
scheme@(guile-user)> ^D
```

Это оказывается очень удобно а). при изучении языка и б). для уточнения его синтаксических конструкций.

Примечание: Из этого режима сложно выйти — никакие ^C, quit и exit не помогут. Это делается вводом символа EOF во входном потоке: ^D (ну и, наверное, ^Z в Windows варианте).

От Scheme оказывается довольно трудно добиться вывода русского текста (UTF-8) в терминал — это обычная практика старых языков, в которые поддержка Unicode добавлялась позже и какими-то «странными» конструкциями. Научить Scheme понимать кодировку UTF-8 можно, например, так:

```
#!/usr/bin/guile -s
!#
(define stdout (current-output-port))
(set-port-encoding! stdout "utf-8")

;; пробная программа на Scheme - guile :
(begin (write "Привет из Scheme, ") (write (car (cdr (command-line)))) (newline))
(display "... вот так выводится русская строка\n" )
```

Две первые (комментарии) строки позволяют запускать Scheme-скрипты не набирая постоянно: guile ... (естественно, при этом нужно сделать chmod a+x для файла скрипта). Следующие две строки обеспечивают вывод UTF-8 функциями (write ...) и (display ...):

```
$ ./hello.scm Вася
"Привет из Scheme, ""Вася"
... вот так выводится русская строка
```

Теперь мы можем создать приложение, эквивалентное предыдущим:
Эталонная задача на функциональном языке Scheme:

triangle.scm :

```
#!/usr/bin/guile -s
!#
(define stdout (current-output-port))
(set-port-encoding! stdout "utf-8")

(define d2 (lambda (f) (/ (round (* f 100.0)) 100.0) ))

(define (point n)          ; ввод и создание точки вершины
  (define x 0) (define y 0) (define z 0+0i)
  (display "вершина № ") (display n) (display " : ")
  (set! x (read))
  (cond ((eof-object? x) x)
        (else
         (set! y (read))
         (cond ((eof-object? y) y)
               (else (set! z (+ x (* y 0+1i)))) z)
         )
  )
)

(define (put_point p)      ; вывод точки
  (display "[" ) (display (d2 (real-part p)))
  (display "," ) (display (d2 (imag-part p))) (display "]" " ")
)

(define show              ; вывод вершин многоугольника
  (lambda ( lst )
    (map (lambda (x) (put_point x)) lst)))

(define (leng x)
  (let loop ((z x) (n 0))
    (if (null? z) n
        (loop (cdr z) (+ 1 n)))))

(define poligon          ; функция ввода координат вершин
  (lambda ( n lst )
    (define p 0+0i)
    (set! p (point n)) ; ввод новой вершины
    (cond ((eof-object? p) (display "\r"))
          (else (set! lst (poligon (+ n 1 ) (cons p lst)))))
    lst
  )
)

(define perimeter        ; периметр по списку вершин
  (lambda ( lst )
    (define size (lambda (p1 p2) (magnitude (- p1 p2))))
    (define head (car lst))
    (let ploop( (z lst) (per 0.0))
      (if (null? (cdr z ))
          (+ (size (car z) head) per)
          (ploop (cdr z) (+ (size (car z) (car (cdr z ))) per))))
    )
)

(define square          ; площадь по списку вершин
```

```

(lambda (lst)
  (define shead (car lst))
  (define triag (lambda (p1 p2)
    (set! s1 (- p1 shead))
    (set! s2 (- p2 shead))
    (* (* (abs (sin (- (angle s2) (angle s1)))) 0.5)
      (* (magnitude s1) (magnitude s2))))
  ))
  (define s1 0+0i )
  (define s2 0+0i )
  (let sloop( (z (cdr lst)) (squ 0.0))
    (if (null? (cdr z )) squ
      (sloop (cdr z) (+ (triag (car z) (car (cdr z ))) squ))))
  )
)

(define next ; цикл по фигурам
  (lambda ()
    (define shape '() )
    (display "координаты вершин в формате: X Y\n" )
    (set! shape (poligon 1 '() ) )
    (display "вершин ") (display (leng shape))
    (display " : ") (show shape) (newline)
    (display "периметр = ") (display (d2 (perimeter shape))) (newline)
    (display "площадь = ") (display (d2 (square shape))) (newline)
    (display "-----\n")
    (next) ; рекурсия обеспечивает бесконечный цикл
  ))

(next) ; запуск всей программы

```

Громоздкость этой показанной программы обусловлена, главным образом, множеством отдельных **операций вывода** (`display ...`) для придания внешнего поведения функциональной программе подобного её императивным эквивалентам. Родная функция (`format ...`) языка Scheme, предназначенная, в частности, для объединения фрагментированных выводов в единый оператор, похоже, просто сходит с ума, когда в выводе появляются UTF-8 коды (хоть и учили мы Scheme понимать UTF-8, но так до конца и не научили). В англоязычной форме диалога, за счёт использования вызова (`format ...`), всё становится намного лаконичнее.

Результат выполнения этой программы:

```

$ ./triangle.scm
координаты вершин в формате: X Y
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 1
вершин 3 : [2.0,1.0] [1.0,2.0] [1.0,1.0]
периметр = 3.41
площадь = 0.5
-----
координаты вершин в формате: X Y
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [2.0,1.0] [2.0,2.0] [1.0,2.0] [1.0,1.0]
периметр = 4.0
площадь = 1.0
-----
координаты вершин в формате: X Y
вершина № 1 : ^C

```


Как уже упоминалось ранее, Scheme — это один из клонов старого, из числа патриархов, и хорошо известного языка программирования Lisp. Поэтому отдельно на Lisp вариант приложения мы рассматривать не станем.

Scala

«Попросту говоря, в программировании на Java протупает возраст.»

Тед Ньювард, «Путеводитель по Scala для Java-разработчиков»

Scala — мультипарадигмальный язык программирования, спроектированный кратким и типобезопасным для простого и быстрого создания компонентного программного обеспечения. Мультипарадигмный потому, что сочетает в себе возможности функционального и объектно-ориентированного программирования. Первые версии языка созданы в 2003 году коллективом лаборатории методов программирования Федеральной политехнической школы Лозанны под руководством Мартина Одерски.

Scala включает единообразную объектную модель — в том смысле, что любое значение является объектом, а любая операция — вызовом метода. Многие считают Scala дальнейшим расширением языка Java и даже называют его как Java++.

Прежде всего, нам предстоит установить Scala, это может быть подобный набор пакетов для начала (показано на примере RPM-дистрибутива Fedora 20):

```
$ yum list scala*
```

```
Загружены модули: langpacks, refresh-packagekit
```

```
Доступные пакеты
```

```
...
```

```
scala.noarch                               2.10.3-8.fc20
```

```
...
```

```
scalaz.noarch                             7.0.0-2.fc20
```

```
...
```

```
$ sudo yum install scala scalaz
```

```
...
```

```
Установить 2 пакета
```

```
Объем загрузки: 32 М
```

```
Объем изменений: 37 М
```

```
...
```

```
Выполнено!
```

Как видно, это потянет достаточно большой объём изменений. Здесь scalaz — библиотека функционального программирования, которая сильно расширяет возможности Scala (но можете её и не устанавливать):

```
$ yum info scalaz.noarch
```

```
...
```

```
Аннотация: extension to the core Scala library for functional programming
```

```
Ссылка: http://typelevel.org
```

```
Лицензия: BSD
```

```
Описание:
```

```
: Scalaz is a Scala library for functional programming. It provides
: purely functional data structures to complement those from the Scala
: standard library. It defines a set of foundational type classes
: (e.g. Functor, Monad) and corresponding instances for a large number
: of data structures.
```

В итоге получаем:

```
$ scalac -version
```

```
Scala compiler version 2.10.3-20130923-e2fec6b28dfd73482945ffab85d9b582d0cb9f17 -- Copyright 2002-2013, LAMP/EPFL
```

```
$ scala -version
```

```
Scala code runner version 2.10.3-20130923-e2fec6b28dfd73482945ffab85d9b582d0cb9f17 -- Copyright 2002-2013, LAMP/EPFL
```

Исполняющую систему Scala (интерпретатор байт-кодов) можно напрямую использовать и в режиме интерактивного калькулятора для выражений любой степени сложности:

```
$ scala
```

```
Welcome to Scala version 2.10.3-20130923-e2fec6b28dfd73482945ffab85d9b582d0cb9f17 (OpenJDK Server VM, Java 1.7.0_51).
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala> def square(x: Double) = x * x
```

```
square: (x: Double)Double
```

```
scala> square(5 + 3)
```

```
res0: Double = 64.0
```

```
scala>^C
```

Это очень удобно а). при изучении языка и б). для уточнения синтаксических конструкций (в отдельном терминале) непосредственно при написании Scala приложений.

Расширения имён файлов кода Scala (приложений, сценариев): .scb, .scala. Scala отходит от принятое в Java соглашения соответствия имён классов содержащих им файлов: не требует в названии файла, содержащего определение класса, отражать имя этого класса (как это и будет происходить в показанном примере).

Пример, аналогичный предыдущим, в Scala может выглядеть так:

triangle.scala :

```
import scala.math._ /* символ _ в Scala - "групповой" символ */
import java.util.StringTokenizer
```

```
class Complex( re: Double, im: Double ) extends AnyRef {
  private val r: Double = re
  private val i: Double = im
  def -(that: Complex) =
    new Complex( r - that.r, i - that.i )
  def real: Double = r
  def imag: Double = i
  override def toString = "[%2f,%2f] ".format( r, i )
  def abs(): Double = sqrt( r * r + i * i )
  def arg(): Double = atan2( i, r )
}

object triangle {

  def perimeter( l: List[ Complex ] ): Double = {
    val h: Complex = l.head
    def distance( p1: Complex, p2: Complex ): Double = ( p1 - p2 ).abs()
    def sides( l: List[ Complex ] ): Double = {
      if( Nil == l.tail ) distance( l.head, h )
      else distance( l.head, l.tail.head ) + sides( l.tail )
    }
    sides( l )
  }

  def square( l: List[ Complex ] ): Double = {
    val h: Complex = l.head
    def trisq( l: List[ Complex ] ): Double = {
      val s1 = l.head - h
      val s2 = l.tail.head - h
      s1.abs * s2.abs * abs( sin( s1.arg - s2.arg ) ) / 2.0
    }
    def addsq( l: List[ Complex ] ): Double = {
      if( Nil == l.tail ) 0.0
      else trisq( l ) + addsq( l.tail )
    }
  }
}
```

```

    }
    addsq( l.tail )
}

def next(): Unit = {
    Console.println( "координаты вершин в формате: X Y (^D конец ввода)" )
    var i: Int = 0
    var eof: Boolean = false
    var shape: List[ Complex ] = List()
    while( !eof ) {
        var s: String = Console.readLine( "%s%d%s", "вершина № ", i + 1, " : " )
        if( null == s ) {
            eof = true // ^D - конец ввода
            Console.print( "\r" )
        }
        else { // в блоке используется Java API:
            try {
                val st: StringTokenizer = new StringTokenizer( s, " \r\n" )
                var x: Double = st.nextElement.toString.toDouble
                var y: Double = st.nextElement.toString.toDouble
                try { // попытка чтения сверх x, y
                    if( null != st.nextElement.toString )
                        throw new Exception( "Сгенерированное исключения" )
                }
                catch {
                    case ex: NoSuchElementException =>
                        shape = new Complex( x, y ) :: shape
                        i = i + 1
                }
            }
            catch { // все! ошибки ловятся здесь
                case ex: Exception =>
                    Console.println( "ошибка ввода!" )
            }
        }
    }
    Console.print( "вершин " + shape.length + " : " )
    shape.map( x => Console.print( x ) )
    Console.println
    Console.println( "периметр = %.2f".format( perimeter( shape ) ) )
    Console.println( "площадь = %.2f".format( square( shape ) ) )
    Console.println( "-----" )
    next()
}

def main( args: Array[ String ] ): Unit = next() // главная стартовая подпрограмма
}

```

Здесь есть и классы-объекты, и функции как объекты данных (функциональное программирование), показана и обработка исключений (даже структурированная обработка исключений). Специально показано (StringTokenizer), как Scala может напрямую использовать все возможности Java API.

Созданное приложение нужно откомпилировать в файлы классов (*.class). Компиляция Scala происходит до неприятного долго (в сравнении с Java, например):

```

$ scalac -deprecation triangle.scala
$ ls *.class
Complex.class triangle_sc$$anonfun$next$1.class triangle_sc.class triangle_sc$.class

```

Выполнение полученного приложения:

\$ scala triangle

```
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 1
вершин 3 : [2.00,1.00] [1.00,2.00] [1.00,1.00]
периметр = 3.41
площадь = 0.5 0
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 1 1
вершина № 2 : 1 2
вершина № 3 : 2 2
вершина № 4 : 2 1
вершин 4 : [2.00,1.00] [2.00,2.00] [1.00,2.00] [1.00,1.00]
периметр = 4.0 0
площадь = 1.0 0
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : 0 0
вершина № 2 : 0 1
вершина № 3 : 3 1
вершина № 4 : 3 0
вершин 4 : [3.00,0.00] [3.00,1.00] [0.00,1.00] [0.00,0.00]
периметр = 8.0 0
площадь = 3.0 0
-----
координаты вершин в формате: X Y (^D конец ввода)
вершина № 1 : ^C
```

Вот как срабатывает обработка ошибок ввода в коде программы:

\$ scala triangle

```
вершина № 1 : 4 а
ошибка ввода!
вершина № 1 : 1 2 3
ошибка ввода!
вершина № 1 : 3ю5 2
ошибка ввода!
вершина № 1 :
ошибка ввода!
вершина № 1 : 1
ошибка ввода!
вершина № 1 : ^C
```

Scala, за счёт использования техники функционального программирования, очень сильно расширяет возможности Java, а иногда и значительно укорачивает и упрощает код. Но рассмотрение всех замысловатостей Scala увело бы нас далеко в сторону.

Ocaml

Ocaml — ещё один объектно-ориентированный язык функционального программирования общего назначения (странное такое сочетание, но та же характеристика может быть отнесена и к обсуждавшемуся выше Scala, и, в принципе, и относительно Python тоже можно так сказать). Язык разрабатывался с ориентацией на безопасность исполнения и надёжность программ. Утверждается, что этот язык имеет высокую степень выразительности, что позволяет его легко выучить и использовать. Язык CamL поддерживает функциональную, императивную и объектно-ориентированную парадигмы программирования. Ocaml разработан в 1996 году во французском институте INRIA, который занимается исследованиями в области информатики (авторы: Xavier Leroy, Jérôme Vouillon, Damien Doligez и Didier Rémy). Разработка сделана основываясь на языке Caml, существующем с 1985 года. Это самые распространённые в практической работе диалект языка ML,

одного из самых старых и известных функциональных языков, который, вообще то говоря, имеет много самых разных реализаций.

Инструментарий Ocaml включает в себя интерпретатор, компилятор в байт-код, и оптимизирующий компилятор в машинный код (авторами утверждается, что превосходящий по своим параметрам аналогичные компиляторы C/C++ для многих задач, особенно связанных с синтаксическим анализом и т. п.).

В дистрибутивах Linux представлено действительно весьма много инструментальных средств, связанных со средствами Ocaml, для самых разных целей применения:

```
$ yum list ocaml* | wc -l
270
```

Для минимальных целей нам достаточно:

```
$ sudo yum install ocaml
...
Установить 1 пакет (+6 зависимых)
Обновить (4 зависимых)
Общий размер: 10 М
Объем загрузки: 8.5 М
Is this ok [y/d/N]: y
...
Установлено:
  ocaml.x86_64 0:4.00.1-3.fc20
Установлены зависимости:
  ... ocaml-compiler-libs.x86_64 0:4.00.1-3.fc20 ... ocaml-runtime.x86_64 0:4.00.1-3.fc20
```

Как видим, даже такая минимальная установка тянет довольно много по зависимостям.

Ocaml, как и предыдущие рассмотренные языки, позволяет вести тестирование или отладку в режиме интерактивного консольного калькулятора:

```
$ ocaml
OCaml version 4.00.1
# let pi=4.0*.atan 1.0;;
val pi : float = 3.14159265358979312
# let square x = x*.x;;
val square : float -> float = <fun>
# square(sin(pi))+.square(cos(pi));;
- : float = 1.
# ^D
```

(Операции «с точкой»: *. , +. — указывают на вещественный тип операции, целочисленные операции будут выглядеть по-другому: * , +). Во 2-м утверждении мы определили функцию square() как объект, что характерно для функционального программирования.

Специфической «фишкой» Ocaml является заточенность его средств на реализацию лексических анализаторов: с одной стороны, есть Ocaml-версии лексического анализатора Lex и синтаксического анализатора YACC, обрабатывающие LALR-языки с помощью автоматов с магазинной памятью, с другой — предопределенные типы потоков (символов и токенов) и операции сопоставления с образцом для потоков, облегчающие написание рекурсивных нисходящих анализаторов для LL-языков. Вычислительный класс приложений, которые мы сравниваем, не подпадает под специфику Ocaml, но и они с успехом реализуются в языке.

В составе пакета Ocaml очень большое количество прикладных библиотек, посмотрим их в /usr/lib64/ocaml (у вас вполне эти библиотеки могут оказаться в /usr/lib/ocaml):

```
$ ls -l /usr/lib64/ocaml/*.mli | wc -l
60
```

Это интерфейсы модулей (.mli), доступные в текстовом виде для изучения. Важность этого источника информации заключается ещё в том, что **вся** документация, описания, книги по Ocaml — **отвратительного качества**. Объясняется это, скорее всего, неблагоприятным наложением целого ряда факторов:

1. Документация Ocaml писалась на французском языке и давно, с неё делались не очень умелые переводы на английский, а а уже потом с него — те рваные переводы и описания, которые доступны на русском.

2. Ещё одна черта информационных источников по Ocaml, которая явно прослеживается, состоит в том, что написано это всё в академических, университетских кругах: много замысловатых конструкций и примеров, но нет систематического изложения и мало пригодно для практического использования.
3. Время жизни языка (начиная с клона ML) достаточно велико, совместимостью между версиями авторы не озадачивались, а в документации не указываются ни версии, ни сроки написания — большинство примеров из документации и учебников просто даже не компилируется.

Из-за всех этих особенностей, порог начального вхождения в программирование Ocaml — **высокий**, и к этому нужно быть готовым. И именно из-за этого в этой части описания мы уделяем некоторое излишнее внимание таким рутинным вещам как инсталляция, библиотеки и подобные им.

Разные способы использования стандартных модулей поставки (показанных выше) можно наблюдать в приведенном ниже листинге на примере использования API из модулей `Complex` и `Format`.

Реализация задачи на функциональном языке Ocaml:

triangle_ml.ml :

```
open Complex;;
```

```
let print_point pt = (* вывод комплексного числа - координаты *)
  print_string "[";
  print_float pt.re;
  print_string ",";
  print_float pt.im;
  print_string "]" ;;
```

```

let rec poligon shape n =
    print_string "вершина №";
    print_int n;
    print_string " : ";
    Format.print_flush();
    try
        let str = read_line() in
        try
            let pt = Scanf.sscanf str "%f %f" (fun x y -> { re=x; im=y }) in
            let lst = pt :: shape in
            poligon lst ( n + 1 );
        with float_of_string ->
            print_string "ошибка ввода!\n";
            poligon shape n;
    with End_of_file ->
        shape;;
(* ввод координат многоугольника *)
(* ошибка формата ввода*)
(* ^D - конец списка вершин *)

```

```
let rec show shape = (* вывод координат многоугольника *)
  match shape with
  | [] ->
    print_newline();
  | hd :: tl ->
    print_point hd;
    show tl;;
```

```

let perimeter shape =
  let dist p1 p2 = norm( sub p1 p2 ) in          (* расстояние между 2-мя точками*)
  let rec add_line lst sum =
    if List.tl( lst ) = [] then
      dist (List.hd lst) (List.hd shape)         (* последняя точка *)
    else
      dist (List.hd lst) (List.hd( List.tl lst )) +.
      add_line (List.tl lst ) sum                 (* промежуточные точки *)
  in                                              (* end add_line *)
  add_line shape 0.0;;                          (* накапливающая сумма *)

```

```

let square shape =
  let triang p1 p2 =
    let s1 = sub p1 ( List.hd shape ) in
    let s2 = sub p2 ( List.hd shape ) in
    norm( s1 ) *. norm( s2 ) *. abs_float( sin( ( arg( s1 ) -. arg( s2 ) ) ) ) /. 2.0
  in
  let rec add_squa lst sum =
    let squa = triang (List.hd lst) (List.hd( List.tl lst )) in
    if List.tl( List.tl lst ) = [] then squa
    else
      squa +. add_squa (List.tl lst ) sum
  in
  add_squa (List.tl shape) 0.0;;

let rec next() =
  print_string "координаты вершин в формате: X Y (^D конец ввода)\n";
  let shape = polygon [] 1 in
  begin
    print_string "\rвершин ";
    print_int( List.length shape );
    print_string " : ";
    show shape ;
    print_string "периметр = ";
    print_float( perimeter shape );
    print_newline();
    print_string "площадь = ";
    print_float( square shape );
    print_newline();
    print_string "-----\n";
    next();
  end;;

next();;

```

Логика приложения подобна тому, как оно выглядит и на других языках: там даже есть достаточно полная обработка ошибок ввода, но нет форматирования вывода вещественных значений. Выполнять код Ocaml можно разнообразным образом:

- непосредственной интерпретацией кода:

```

$ ocaml triangle.ml
координаты вершин в формате: X Y (^D конец ввода)
...

```

- компиляцией в байт-код с последующим его выполнением:

```

$ ocamlc triangle.ml -o triangle_ml
$ ocamlrun triangle_ml
координаты вершин в формате: X Y (^D конец ввода)
...

```

- оптимизирующей компиляцией в бинарный исполнимый формат (ELF в случае Linux) и его выполнение:

```

$ ocamlpt triangle.ml -o triangle_mlo
$ ./triangle_mlo
координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : 1 1
вершина №2 : 1 2
вершина №3 : 2 1
вершин 3 : [2.,1.] [1.,2.] [1.,1.]
периметр = 3.41421356237
площадь = 0.5
-----

```

```

координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : 1 1
вершина №2 : 1 2
вершина №3 : 2 2
вершина №4 : 2 1
вершин 4 : [2.,1.] [2.,2.] [1.,2.] [1.,1.]
периметр = 4.
площадь = 1.
-----
координаты вершин в формате: X Y (^D конец ввода)
вершин 0 : ^C

```

Как и следует предполагать, 3 исполнимые формы файлов различаются как по размерам, так и по форматам:

```

$ ls -l triangle_ml*
-rwxrwxr-x. 1 Olej Olej 130156 окт 20 20:37 triangle_ml
...
-rw-rw-r--. 1 Olej Olej 3046 окт 20 18:50 triangle_ml.ml
-rwxrwxr-x. 1 Olej Olej 605333 окт 20 20:46 triangle_mlo
...
$ file triangle_ml.ml
triangle_ml.ml: UTF-8 Unicode text
$ file triangle_ml
triangle_ml: a /usr/bin/ocamlrun script executable (binary data)
$ file triangle_mlo
triangle_mlo: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.32, BuildID[sha1]=6f4e0599f50605658e244a9fc34a2f79d3f3cefb, not stripped

```

Обработка ошибок ввода — вы должны видеть что-то подобное следующему:

```

$ ocaml triangle_ml.ml
координаты вершин в формате: X Y (^D конец ввода)
вершина №1 : 1 2
вершина №2 : 2 r
ошибка ввода!
вершина №2 : 5,5 6
ошибка ввода!
вершина №2 : 3 4
...

```

Haskell

Haskell — стандартизованный **чистый** функциональный язык программирования общего назначения. Выше обсуждавшиеся языки Ocaml или Scala — считаются смешанными языками, помимо функционального поддерживающие и императивный стиль вычислений, Haskell не допускает императивного программирования. Является одним из самых распространённых языков программирования с поддержкой отложенных вычислений. Типизация языка Haskell **строгая, статическая**, с автоматическим выводом типов. Серьёзное отношение к типизации — ещё одна отличительная черта Haskell (что не характерно, вообще то говоря, для функциональных языков). Поскольку язык функциональный, то основная управляющая структура — это функция.

В 1990 г. была предложена первая версия языка, Haskell 1.0. Непосредственно на него оказал очень сильное влияние язык Miranda, разработанный в 1985 г. Дэвидом Тёрнером (Миранда была первым чистым функциональным языком). Но выход Haskell в «широкий свет» начался только в 2003г. — таким образом, в течение 13 лет этот язык был уделом лабораторий, главным образом математически ориентированных.

Порог вхождения в программирование на Haskell высок. Во-первых, из-за его происхождения из кругов абстрактных математиков и из-за формулирования его понятий в терминах понятий из абстрактной математики (теории категорий). Другая причина, связанная с предыдущей, из-за которых и сложилось устойчивое ложное представление о колоссальной сложности языка Haskell — это **отсутствие** внятных описаний и руководств. А официальная документация Haskell выкладывается также в виде строгих **формальных определений**, на изучение которых могут уйти месяцы.

Например, одно из важных понятий и терминов в языке — «монада» (от греческого μονάς, «единица»), пришедшие в Haskell именно из теории категорий. Вслушаемся, как звучит его определение в официальной документации (даже в переводе на русский язык):

Монада может быть определена через общее понятие моноида в моноидальной категории. Монада над категорией K — это моноид в моноидальной категории эндифункторов $End(K)$.

С таким же успехом это определение можно было бы перевести на китайский! Тем не менее, описать монады «на пальцах» можно достаточно просто, а пользоваться ними может любой средний практик, слегка освоившись с их применением. На сегодня есть уже некоторое количество руководств, относительно пригодных для начального освоения языка (см. указатель ресурсов в конце текста).

С другой стороны, Haskell является языком, строго организующим мышление программиста. В ряде ведущих университетов мира именно Haskell выбран как первый язык обучения «искусству программирования» (по определению Д.Кнутта) студентов 1-го курса.

Существует **несколько** реализаций Haskell доступных в Linux, но компилятор GHC стал фактическим стандартом в отношении новых возможностей языка:

```
$ yum list ghc* | wc -l
620
$ sudo yum install ghc
...
Установить 1 пакет (+46 зависимых)
Объем загрузки: 85 М
Объем изменений: 809 М
Is this ok [y/d/N]: y
...
Выполнено!
New leaves:
  ghc.x86_64
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.3
```

Как легко видеть, объём изменений эта инсталляция потянет значительный. В пакете будет установлен одновременно диалоговый интерпретатор Haskell, полезный для отработки конструкций языка, он же может выполнять отдельные приложения:

```
# ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 2+2
4
Prelude> ^D
Leaving GHCi.
```

Но Haskell — это целая своеобразная технология, и в этой технологии есть ещё такая штука как cabal (**C**ommon **A**rchitecture for **B**uilding **A**pplications and **L**ibraries). Говорят, что это нечто типа make в мире C/C++ (я бы сравнил скорее с Сmake) — построитель проектов Haskell. Но cabal придётся устанавливать отдельно:

```
$ yum list cabal*
...
Доступные пакеты
cabal-dev.x86_64                                0.9.2-2.fc20                                fedora
cabal-install.x86_64                            1.16.0.2-34.fc20                            updates
cabal-rpm.x86_64                                0.9.1-1.fc20                                updates
$ sudo yum install cabal*
...
Установить 3 пакета (+10 зависимых)
Объем загрузки: 1.7 М
Объем изменений: 11 М
Is this ok [y/d/N]: y
...
```

```
$ sudo yum install cabal*
```

```
...
```

```
Установить 3 пакета (+13 зависимых)
```

```
Объем загрузки: 1.7 М
```

```
Объем изменений: 9.1 М
```

```
...
```

```
Выполнено!
```

Кроме того, что это инструмент построения собственных модульных проектов, это мощнейшее средство управления модулями (библиотеками) Haskell:

```
$ cabal --help
```

```
...
```

```
Usage: cabal COMMAND [FLAGS]
```

```
or: cabal [GLOBAL FLAGS]
```

```
...
```

```
Commands:
```

install	Installs a list of packages.
update	Updates list of known packages
list	List packages matching a search string.
info	Display detailed information about a particular package.
fetch	Downloads packages for later installation.
unpack	Unpacks packages for user inspection.
check	Check the package for common mistakes
sdist	Generate a source distribution file (.tar.gz).
upload	Uploads source packages to Hackage
report	Upload build reports to a remote server.
init	Interactively create a .cabal file.
configure	Prepare to build the package.
build	Make this package ready for installation.
copy	Copy the files into the install locations.
haddock	Generate Haddock HTML documentation.
clean	Clean up after a build.
hsccolour	Generate HsColour colourised code, in HTML format.
register	Register this package with the compiler.
test	Run the test suite, if any (configure with UserHooks).
bench	Run the benchmark, if any (configure with UserHooks).
upgrade	(command disabled, use install instead)
help	Help about commands

```
For more information about a command use:
```

```
cabal COMMAND -help
```

```
...
```

```
$ cabal update
```

```
Config file path source is default config file.
```

```
Config file /home/Olej/.cabal/config not found.
```

```
Writing default configuration to /home/Olej/.cabal/config
```

```
Downloading the latest package list from hackage.haskell.org
```

С помощью cabal вы можете найти, выбрать и установить любой модуль (библиотеку) из главного мирового репозитория Haskell (его называют Hackage):

```
$ cabal list complex
```

```
* complex-generic
```

```
Synopsis: complex numbers with non-mandatory RealFloat
```

```
Default available version: 0.1.1
```

```
Installed versions: [ Not installed ]
```

```
Homepage: https://gitorious.org/complex-generic
```

```
License: BSD3
```

```
* complex-integrate
```

```
Synopsis: A simple integration function to integrate a complex-valued
         complex functions
Default available version: 1.0.0
Installed versions: [ Not installed ]
Homepage: https://github.com/hijarian/complex-integrate
License: PublicDomain
```

```
* complexity
  Synopsis: Empirical algorithmic complexity
  Default available version: 0.1.3
  Installed versions: [ Not installed ]
  License: BSD3
```

```
* storable-complex
  Synopsis: Storable instance for Complex
  Default available version: 0.2.1
  Installed versions: [ Not installed ]
  License: BSD3
```

Библиотека Haskell очень обширна (1-я команда выведет полный листинг библиотеки):

```
$ cabal list >> cabal.lst
$ ls -l cabal.lst
-rw-rw-r--. 1 Olej Olej 1273606 map  9 11:40 cabal.lst
$ wc -l cabal.lst
41520 cabal.lst
```

Учитывая, что информация о каждом модуле выводится в 6 строк (см. выше) — это даёт объём библиотеки в 6920 единиц компиляции.

Файлы исходного кода Haskell имеют расширения `.hs` или `.lhs`. Одним из фундаментальных свойств языка Haskell, которым программисты пугают друг друга, является полное отсутствие в нём **оператора** присваивания. В написании реализации нашей тестовой задачи, в отношении Haskell мы пойдём другим путём, отличающимся от того, как это делалось в других языках: мы не станем вручную компилировать из командной строки файл кода `triangle.hs`, а создадим проект, пользуясь возможностями `cabal` по созданию и управлению проектами (так, как это именно и делают в технологии Haskell).

Примечание: Конечно, вы можете откомпилировать полученный ниже файл кода задачи и вручную, предварительно переименовав его в `triangle.hs`.

Создадим для начала ручную файловую инфраструктуру проекта, например так:

```
$ mkdir triangle
$ cd triangle
$ mkdir src
$ cd src
$ touch Main.hs
$ cd ..
$ tree
.
├── src
│   └── Main.hs
```

Теперь, находясь в корне дерева проекта (каталог `triangle`), выполним построение проекта командой, которая проведёт диалог настройки проекта, вопросы которого достаточно понятны:

```
$ cabal init
Package name? [default: triangle]
Package version? [default: 0.1.0.0]
Please choose a license:
* 1) (none)
  2) GPL-2
  3) GPL-3
  4) LGPL-2.1
```

```

5) LGPL-3
6) BSD3
7) MIT
8) Apache-2.0
9) PublicDomain
10) AllRightsReserved
11) Other (specify)
Your choice? [default: (none)] 7
Author name?
Maintainer email?
Project homepage URL?
Project synopsis?
Project category:
* 1) (none)
  2) Codec
  3) Concurrency
  4) Control
  5) Data
  6) Database
  7) Development
  8) Distribution
  9) Game
 10) Graphics
 11) Language
 12) Math
 13) Network
 14) Sound
 15) System
 15) System
 16) Testing
 17) Text
 18) Web
 19) Other (specify)
Your choice? [default: (none)]
What does the package build:
  1) Library
  2) Executable
Your choice? 2
Include documentation on what each field means (y/n)? [default: n]

```

Guessing dependencies...

Generating LICENSE...

Warning: unknown license type, you must put a copy in LICENSE yourself.

Generating Setup.hs...

Generating triangle.cabal...

Warning: no synopsis given. You should edit the .cabal file and add one.

You may want to edit the .cabal file and add a Description field.

\$ tree

```

.
├── Setup.hs
├── src
│   └── Main.hs
└── triangle.cabal

```

В каталоге src могут быть созданы дополнительные файлы кода к проекту (.hs), или каталоги (например Utils), содержащие такие файлы. Дополнительные файлы кода будут содержать код **модулей**, которые компонуется в проект. Имена файлов и каталогов в src лучше именовать **с заглавной буквы** — это связано с именованием и импортом модулей в Haskell.

После генерации в файловой иерархии появилось 2 файла: Setup.hs нас не интересует, и

файл конфигурации проекта `triangle.cabal`, в котором мы будем неоднократно редактировать строки по ходу развития проекта (сами параметры строк уже записаны в файл в виде комментариев, нам предстоит раскомментировать их вписать им значения). Прежде всего, нужно (обязательно) определить файл кода с которого стартует приложение (`Main.hs`, он может иметь произвольное имя):

```
...
executable triangle
  ghc-options:      -W
  main-is:          Main.hs
  build-depends:    haskell98 >=2.0.0.2 , exceptions
...
```

Здесь показаны только строки, подвергшиеся изменению в том исходном файле, который был создан при построении проекта (в порядке как они показаны):

- определить включить вывод предупреждений компиляции, не только ошибок;
- определить файл `Main.hs` как стартовый (на самом деле имена файлов кода могут быть произвольными);
- описать импорт дополнительных стандартных пакетов (библиотек): в данном случае пакет `haskell98` содержит модуль `Complex` для работы с комплексными числами, а пакет `exceptions` — обработку исключений;

Теперь нам осталось сконфигурировать проект под наши правки (конфигурацию лучше делать **каждый раз** после редактирования `triangle.cabal`):

```
$ cabal configure
Resolving dependencies...
Configuring triangle-0.1.0.0...
Warning: The 'license-file' field refers to the file 'LICENSE' which does not exist.
```

Всё! Проект готов. Дальше нам предстоит наполнять смыслом файлы исходного кода (`Main.hs`) и **компилировать** проект. Вот как может выглядеть код сравниваемой задачи в упрощённой реализации на Haskell (упрощение касается только отсутствия обработки ошибок ввода пользователя — чтобы не перегружать код):

Реализация задачи на языке Haskell (файл `Main.hs` каталог `triangle`):

```
module Main where
import Complex
import Numeric
import IO

{- код проверен для версии:
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.3
-}

type Point = Complex Double      -- синоним координатной точки

get_coord :: String -> Point      -- декодирование строки ввода в координаты x % y
get_coord str =
  f( words str )
  where
    f :: [String] -> Point
    f lst = ( ( read( lst !! 0 ) :: Double ) :+ ( read( lst !! 1 ) :: Double ) )

try_to_input :: IO String         -- ввод строки с ожиданием ^D
try_to_input = do
  line <- hGetLine stdin `catch` (\e -> if IO.isEOFError e then return [] else ioError e)
  return line

get_shape :: [Point] -> IO [Point]
get_shape shape = do
  let pos = length( shape ) + 1
  putStr( "вершина № " ++ show( pos ) ++ " : " )
```

```

hFlush stdout
line <- try_to_input
if length( line ) == 0 then return shape else get_shape( get_coord( line ) : shape )

showP = \p -> "[" ++ ( showFFloat ( Just 2 ) ( realPart( p ) ) "" ) ++
    "," ++ ( showFFloat ( Just 2 ) ( imagPart( p ) ) "" ) ++ "]"

show_shape :: [Point] -> String
show_shape (x:xs) =
    if length xs == 0 then showP x else ( showP x ) ++ show_shape( xs )

perimeter :: [Point] -> Double
perimeter shape =
    summa shape 0.0
    where distance = \ p1 p2 -> magnitude( p1 - p2 )
          summa :: [Point] -> Double -> Double
          summa (y:ys) perim      -- локальная функция накопления длин сторон
          | length( ys ) == 0 = ( distance y $ head shape ) + perim
          | otherwise = ( distance y $ head ys ) + summa ys perim

square :: [Point] -> Double
square (y:ys) =
    summa y ys 0.0
    where summa :: Point -> [Point] -> Double -> Double
          summa top shape squa    -- локальная функция накопления площади
          | length( tail shape ) == 0 = squa
          | otherwise = ( triang top shape ) + summa top ( tail shape ) squa
          triang :: Point -> [Point] -> Double
          triang top (z:zs) =      -- локальная функция площадь треугольника
              ( magnitude side1 ) * ( magnitude side2 ) *
              ( abs $ sin( phase side1 - phase side2 ) ) * 0.5
              where side1 = z - top
                    side2 = z - head zs

next_shape :: IO ()              -- цикл расчёта
next_shape = do
    putStrLn( "координаты вершин в формате: X Y" )
    shape <- get_shape []
    putStrLn $ "\rвершин " ++ show( length shape ) ++ " : " ++ show_shape shape
    putStrLn( "периметр = " ++ showFFloat ( Just 2 ) ( perimeter shape ) "" )
    putStrLn( "площадь = " ++ showFFloat ( Just 2 ) ( square shape ) "" )
    putStrLn $ "-----"
    next_shape

main :: IO ()
main = do next_shape            -- запуск цикла программы

```

Исходный код на Haskell **не является форматно независимым**: его смысл зависит от отступов новых строк, переносов строк и других вещей, связанных с размещением кода. Это достаточно редкий случай для языков программирования, и здесь (только в написании) Haskell близок с Python.

В показанном фрагменте кода есть достаточно много: и лямбда-определения функций, и сопоставления с образцом, и обработка исключений (в определении конца ввода, ситуации EOF), и в использовании рекурсии.

Теперь мы готовы компилировать полученный **проект**:

```

$ cabal build
Building triangle-0.1.0.0...
Preprocessing executable 'triangle' for triangle-0.1.0.0...
[1 of 1] Compiling Main             ( src/Main.hs, dist/build/triangle/triangle-tmp/Main.o )
src/Main.hs:3:1: Warning:
    The import of `Numeric' is redundant

```

```

    except perhaps to import instances from `Numeric'
    To import instances alone, use: import Numeric()
src/Main.hs:42:1: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `show_shape': Patterns not matched: []
src/Main.hs:50:10: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `summa': Patterns not matched: [] _
src/Main.hs:55:1: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `square': Patterns not matched: []
src/Main.hs:62:10: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `triang': Patterns not matched: _ []
Linking dist/build/triangle/triangle ...

```

Почему так много предупреждений? Не знаю... Могу предположить по смыслу, что все они (связанные с сопоставлением с образцом) используют синтаксические конструкции, заимствованные из описаний прежних версий (Haskell 98). А очень свежий компилятор (Haskell 2010) хотел бы более современных определений образцов. Предоставим читателям возможность и право самостоятельно улучшить код в этом направлении...

После правок, конфигураций и компиляций дерево проекта имеет структуру:

```

$ tree
.
├── dist
│   ├── build
│   │   ├── autogen
│   │   │   ├── cabal_macros.h
│   │   │   └── Paths_triangle.hs
│   │   └── triangle
│   │       ├── triangle
│   │       └── triangle-tmp
│   │           ├── Main.hi
│   │           └── Main.o
│   ├── package.conf.inplace
│   └── setup-config
├── Setup.hs
├── src
│   └── Main.hs
└── triangle.cabal

```

Здесь поддерево `dist` и есть, собственно, каталогом сборки. Запускать откомпилированное приложение на тестирование мы можем прямо из каталога проекта:

```

$ ./dist/build/triangle/triangle
координаты вершин в формате: X Y
вершина № 1 : 1.00001 1.00003
вершина № 2 : 1.00003 2.0003
вершина № 3 : 2.0004 1.00005
вершин 3 : [2.00,1.00] [1.00,2.00] [1.00,1.00]
периметр = 3.42
площадь = 0.50
-----
координаты вершин в формате: X Y
вершина № 1 : ^C

```

Как видим, оно не сильно отличается от того, что мы видели в реализациях на других языках программирования.

После построения проекта, симметрично, очищаем следы его создания:

```

$ cabal clean

```

cleaning...

P.S. Для того, чтобы не быть голословным относительно возможности и ручной компиляции отдельных файлов Haskell кода, о которой упоминалось выше, продемонстрируем его на простейшем приложении, которое заодно проверит как реализация языка ведёт себя с Unicode, кодировкой UTF-8 и русским текстом (это всегда нужно делать для начала). Само «приложение»:

Простейшее приложение на языке Haskell:

hello_hs.hs :

```
module Main where
import System.Environment

main :: IO ()
main = do
    -- сама программа
    args <- getArgs {- вложенный комментарий -}
    putStrLn( "Привет от Haskell, " ++ args !! 0 )
```

Его ручная компиляция

```
$ ghc -o hello_hs hello_hs.hs
[1 of 1] Compiling Main           ( hello_hs.hs, hello_hs.o )
Linking hello_hs ...
```

Ручное выполнение:

```
$ ./hello_hs Вася
Привет от Haskell, Вася
```

Другие языки...

X10

X10 — язык программирования, разработанный корпорацией IBM как часть проекта PERCS, спонсируемого в рамках программы «Высокопродуктивные компьютерные системы» (High Productivity Computing Systems, HPCS) Агентства по перспективным оборонным научно-исследовательским разработкам США.

X10 был разработан с учетом требований параллельного программирования. По сути, это «расширенное подмножество» языка программирования Java, сильно схожее с ним во многих аспектах, но обладающее особой дополнительной поддержкой массивов и процессов.

X10, как легко догадаться из анонсов, вы не найдёте в репозиториях Linux, его придётся устанавливать с сайтов проекта. Загрузить архивы X10 в последних редакциях можно отсюда: <http://x10-lang.org/software/download-x10/latest-release.html>. Там же представлены RPM-пакеты для инсталляции, например x10-2.4.2-1.linux_x86.rpm. Последняя представленная версия X10 2.4.2 датирована февралём 2014 года.

Chapel

Chapel — новый язык программирования с поддержкой распараллеливания, разработанный корпорацией Cray. Язык был разработан в рамках проекта Cascade, для участия в программе DARPA Высокопродуктивные компьютерные системы (High Productivity Computing Systems, HPCS), целью которой являлось увеличение производительности суперкомпьютеров к 2010 году.

Chapel также именуемый как «Каскадный высокопроизводительный язык» (Cascade High Productivity Language), поддерживает модель высокоуровневого многопоточного параллельного программирования за счёт поддержки абстрагирования распараллеливания данных, задач и вложенных подзадач.

Пакет Chapel отсутствует в пакетных репозиториях Linux. Его нужно устанавливать с сайта проекта, взять архив для установки можно, например, здесь: <http://chapel.cray.com/download.html>. Последняя представленная версия Chapel 1.8.0 на сайте проекта (см. Ресурсы) датирована 17 октября 2013 года.

HaXe

HaXe — универсальный объектно-ориентированный язык программирования высокого уровня. Автором и основным разработчиком платформы является французская медийная компания Motion-Twin, которая так же известна как разработчик предшественника HaXe — mtsac (компилятор ActionScript 2). На данный момент существует общественный фонд HaXe Foundation, занимающийся поддержкой и развитием языка.

Это мультиплатформенный язык, который может использоваться в различных операционных средах, начиная от встроенных двоичных систем до интерпретаторов и виртуальных машин. На данный момент разработчики могут писать программы на haXe, затем компилировать их в объектный код, JavaScript, PHP, Flash/ActionScript или байткод NekoVM. Дополнительные модули для компилирования C# и Java находятся в разработке.

HaXe, так же как и X10 и Chapel не представлен в пакетных репозиториях многих дистрибутивов Linux. Но он представлен в репозитории Ubuntu (и, нужно думать, Debian):

```
$ aptitude search haxe
p   haxe                - web-oriented universal programming language
```

Но его также можно устанавливать (в любом дистрибутиве) с сайта проекта, из архива, который можно взять здесь: <http://haxe.org/download>. Последняя представленная там версия HaXe 3.1.0 датирована 4 марта 2014 года.

Erlang

Erlang — функциональный язык программирования, позволяющий писать программы для разного рода распределённых систем. Разработан и поддерживается компанией Ericsson. Язык включает в себя средства порождения параллельных процессов и их коммуникации с помощью послылки асинхронных сообщений. Программа транслируется в байт-код, исполняемый виртуальной машиной, что обеспечивает переносимость.

Erlang на сегодня язык в программистской среде модный, поэтому практически в любом дистрибутиве Linux вам не придётся устанавливать его из исходных кодов, в пакетной системе дистрибутива вы найдёте множество пакетов, связанных с Erlang:

```
$ yum list erlang* | wc -l
149
```

Установка, даже минимальная, достаточно объёмная:

```
$ sudo yum install erlang
...
Установить 1 пакет (+63 зависимых)
Объем загрузки: 40 М
Объем изменений: 77 М
...
```

Обсуждение

Характерно, что из всех охваченных в обзоре языков только три языка используют технику «нативной» компиляции в машинный код используемой платформы: С, С++ и Go (а точнее даже два, потому что С и С++ в реализации GCC, по крайней мере, это одна реализация). Все же остальные, в той или иной мере и технике, используют виртуальную исполняющую машину (среду выполнения). Это, очевидно, становится тенденцией последнего десятилетия. Все ранние проекты языков программирования (Algol, FORTRAN, COBOL, Pascal) предполагали компиляцию именно в исполнимый машинный код. Требование наличия исполняющей виртуальной машины может показаться, на первый взгляд, некоторым ограничивающим фактором. Но и чисто компилирующие реализации (С, С++) давно уже не исполняются как абсолютный бинарный код — невозможно перенести файл .exe из Windows в Linux, или наоборот, файл ELF-формата из Linux в Windows. Так что, с некоторой натяжкой, ядро операционной системы можно также считать исполняющей виртуальной машиной для бинарных форматов (ещё до недавнего времени ядро Linux содержало код для непосредственного запуска и выполнения файлов .class бит-кода Java, что было позже устранено за ненадобностью).

Всё, что упоминалось при обсуждении Python, относительно гибкости динамической типизации (или почти полной её отсутствия как в Lua), относится ко всем обсуждаемым **интерпретирующим** языкам... Конечно, возможность изменять размерности структур данных (массивов) по ходу выполнения — это большой плюс системы программирования. В нашей иллюстрационной задаче изменение размерности от 3-х, скажем, до 10-ти — это не так существенно, и можно зарезервировать размеры данных по максимуму. Но существуют целые классы задач, например, комбинаторного характера (что часто свойственно задачам распознавания разнообразных образов), когда размерность структур данных, в зависимости от исходных условий, может изменяться и на 3-4 порядка (в тысячи и десятки тысяч раз), и тогда это становится серьёзной проблемой.

То, что в языках со строгой **статической** типизацией (С, С++, Java) невозможно изменять размерности так прямолинейно, вовсе не означает, что они принципиально ограничены в этом смысле. Но там потребности в данных переменной размерности реализуются за счёт некоторых потерь или усложнения кода. Как альтернативные варианты решения подобных проблем можно вспомнить:

- Резервирование под структуры данных объёмов исходя из оценок **максимально возможных потребностей**, так как это делалось ещё в FORTRAN в 1958 году (и там это был единственный способ).
- Использование специальных программных техник, основанных на **ссылочных динамических структурах**. В С это могут быть линейные или циклические (как в ядре Linux) списки. В С++ это решается тем же образом, но скрыто, за счёт использования пакетов STL или Boost.
- Стандартом C99 для языка С разрешено использование массивов с динамически изменяемыми границами (VLA), но только объявленных локально в рамках функций, в которых они определены (но это не сильно ограничивающее условие).

Новый компилирующий язык Go позиционирован его авторами как язык, в том числе, и системного программирования и разработки операционных систем. В подтверждение этому есть то, что на Go выполнено довольно много программного обеспечения в операционной системе Plan 9, в рамках проекта которой он и начал развиваться.

Некоторые из рассмотренных интерпретирующих языков (Scheme, Scala) демонстрируют тот рост интереса (или возврат интереса после большого затишья), который наметился в последние годы к парадигме функционального программирования. Для реализаций исполняющих систем таких языков уже стало характерным обеспечивать возможность использования их в режиме интерактивного консольного калькулятора. Это сильно снижает «порог начального вхождения» в инструмент, что, возможно, особенно важно для инструментов функционального программирования (в виду их специфичности). Кроме того, такой режим позволяет использовать интерпретатор (в отдельном терминале) для тонкой отработки и тестирования синтаксических конструкций непосредственно по ходу разработки основного проекта.

Заключение

Рассмотрены реализации **одной и той же** задачи (расчёта параметров выпуклого многоугольника через представление в комплексной плоскости) в совершенно различных языках программирования. В обзор включены практически все широко используемые языки, по крайней мере, на платформе операционной системы Linux. Сюда вошли и 10 языков, во всех рейтингах называемые в числе наиболее используемых в реальных проектах: C, C++, Java, Perl, Python, Ruby, JavaScript, PHP, Lua, bash.

Но помимо рассмотренных, в природе существует множество интересных языков. За рамками рассмотрения остались такие языки как Pascal, Modula 2/3, Oberon, NewSqueak, Limbo, Rust и другие. Это связано с двумя обстоятельствами: а). они мало применяются в развитии реальных проектов и представляют больше академический интерес и б). из-за ограниченности времени и объёма публикации.

Начальные сведения о них, и ещё о многих десятках существующих и использующихся языков прогнозирования вы можете найти на сайте энциклопедии языков программирования, который перечислен в перечне источников информации в конце текста. Начальные сведения о языках там сопровождаются простейшими демонстрирующими программами (такими как вычисление чисел Фибоначчи), которые позволяют составить представление о том или ином языке.

В обзоре не рассматривались языковые средства технологии .NET (C# в первую очередь) или VisualBasic (VBA, VBScript) — только потому, что они являются исключительно прерогативой единственной операционной системы Windows ... пусть даже и самой широко используемой. Из тех же соображений отдельно не выделена реализация Objective-C.

Указатель источников информации

Язык C:

- Брайан У. Керниган, Деннис М. Ритчи : «Язык программирования C».
<http://www.books.ru/books/yazyk-programmirovaniya-c-3583364/?show=1>
- А. Гриффитс : «GCC. Полное руководство. Platinum Edition», М.: «ДиаСофт», 2004, ISBN 966-7992-33-0, стр. 624
<http://www.books.ru/books/gcc-polnoe-rukovodstvo-platinum-edition-190067/?show=1>

Язык C++:

- Бьерн Страуструп : «Язык программирования C++. Специальное издание», М.: «Бином» 2010, ISBN: 978-5-9518-0425-9, стр. 1136
<http://www.codpro.ru/content/yazyk-programmirovaniya-s-b-strastrup-spetsialnoe-izdanie>

Язык Java:

- «Java™ Platform, Standard Edition 6 API Specification»
<http://docs.oracle.com/javase/6/docs/api/overview-summary.html>

Язык Python:

- «Учебник Python 3.1»
http://ru.wikibooks.org/wiki/%D0%A3%D1%87%D0%B5%D0%B1%D0%BD%D0%B8%D0%BA_Python_3.1#.D0.9A.D0.BB.D0.B0.D1.81.D1.81.D1.8B
- Олег Цилюрик : «Тонкости использования языка Python: Часть 1. Версии и совместимость»
http://www.ibm.com/developerworks/ru/library/l-python_details_01/index.html

Язык Ruby:

- «Ruby»
<http://ru.wikibooks.org/wiki/Ruby>
- Юкиhiro Мацумото : «Программирование на языке Ruby. Идеология языка, теория и практика применения»
<http://lib.rus.ec/b/353387/read>
- Крис Пайн : «Учебник по языку Ruby - Учись программировать», перевод М.Шохирев
http://www.opennet.ru/docs/RUS/ruby_learn/

Язык Perl:

- Том Кристиансен, Брайан Де Фой, Ларри Уолл, Джон Орвант : «Программирование на Perl», 4-е издание, Сп-Б.: «Символ-Плюс», 2013, ISBN: 978-5-93286-214-8, стр. 1048
<http://www.books.ru/books/programmirovanie-na-perl-4-e-izdanie-3135461/?show=1>
- Math::Complex
<http://perldoc.perl.org/Math/Complex.html>

Язык JavaScript:

- «Introduction to the JavaScript shell»
https://developer.mozilla.org/en-US/docs/SpiderMonkey/Introduction_to_the_JavaScript_shell#getpda.28object.29
- «Справочник по современному javascript!»
<http://javascript.ru/manual>

Язык PHP:

- «Using PHP from the command line»
<http://www.php.net/manual/en/features.commandline.php>
- «Использование PHP в командной строке»
<http://www.php.net/manual/ru/features.commandline.php>
- «Руководство по PHP»
<http://ru2.php.net/manual/ru/index.php>

Язык Lua:

- «Lua 5.1 Reference Manual»

<http://www.lua.org/manual/5.1/manual.html>

- «Программирование Lua, 4 части»

<http://symmetrica.net/lua/lua1.pdf>

<http://symmetrica.net/lua/lua2.pdf>

<http://symmetrica.net/lua/lua3.pdf>

<http://symmetrica.net/lua/lua4.pdf>

- lua-matrix / lua / complex.lua

<https://github.com/davidm/lua-matrix/blob/master/lua/complex.lua>

- «Справочное руководство по языку Lua 5.1»

<http://www.lua.ru/doc/>

- «Про Lua»

http://ilovelua.narod.ru/about_lua.html

- Lua programming language information and resources

<http://lua-users.org/wiki/>

Командный интерпретатор bash:

- Mendel Cooper : «Advanced Bash-Scripting Guide» — «Искусство программирования на языке сценариев командной оболочки», перевод: Андрей Киселев

http://www.opennet.ru/docs/RUS/bash_scripting_guide/

Язык Go:

- Go

<http://ru.wikipedia.org/wiki/Go>

- «Effective Go»

http://golang.org/doc/effective_go.html#examples

- «Miek Gieben : Learning Go (PDF)»

<http://archive.miek.nl/files/go/Learning-Go-latest.pdf>

- Directory src/pkg/

<http://golang.org/src/pkg/>

- Евгений Охотников : «Краткий пересказ Effective Go на русском языке»

http://eao197.narod.ru/desc/short_effective_go.html

- «Go для программистов C++»

<http://netsago.org/ru/docs/1/16/>

- go-wiki

<https://code.google.com/p/go-wiki/w/list>

- Directory src/pkg/

<http://golang.org/src/pkg/>

- Mark Summerfield : «Programming in Go. Creating Applications for the 21st Century», ISBN-10: 0321774639

<http://www.qtrac.eu/gobook.html>

- Jan Newmarch : «Network programming with Go», v1.0, 27 April 2012

<http://jan.newmarch.name/go/>

- AstaXie : «Build Web Application with Golang»

<https://docs.google.com/file/d/0B2GBHFyTK2N8TzM4dEtIWjBJdEk/edit>

http://www.luminpdf.com/files/5854580/build-web-application-with-golang_EN.pdf

- Matt Aimonetti : «Go Bootcamp. Everything you need to know to get started with Go.», Last updated: 2014/08/21/.

<http://www.golangbootcamp.com/book/>

- Kyle Isom : «Practical Cryptography With Go»

<https://leanpub.com/gocrypto/read>

- Yigal Duppen : «Test-driven development with Go»

<https://dl.dropboxusercontent.com/u/750049/4gophers.com/books/golang-tdd.zip>

- Русскоязычный сайт «Язык программирования Go» (большое число публикаций)

<http://4gophers.com/>

Язык Scheme:

- Scheme

<http://ru.wikipedia.org/wiki/Scheme>

- Кен Дики : «Язык программирования Scheme», перевод Алексея Десятника

<http://alexey.tamb.ru/scheme/intro-to-scheme.html>

- «Revised(5) Report on the Algorithmic Language Scheme»

http://groups.csail.mit.edu/mac/ftplib/scheme-reports/r5rs-html/r5rs_toc.html#TOC1

- «Преобразование программ на языке Scheme для облегчения компиляции в язык C»

<http://citforum.ru/programming/digest/scheme/>

- «Write Yourself a Scheme in 48 Hours»

http://ru.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours

- «Создание скриптов при помощи Guile»

<http://www.ibm.com/developerworks/ru/library/l-guile/index.html>

- «The Guile Reference Manual ... This manual documents Guile version 2.0.9.»

<http://www.gnu.org/software/guile/manual/guile.html#Character-Encoding-of-Source-Files>

Язык Scala:

- Серия публикаций по Scala на IBM developerWorks

http://www.ibm.com/developerworks/ru/views/java/libraryview.jsp?search_by=%20scala+Java

- «Обзор языка программирования Scala»

<http://www.rsdn.ru/article/philosophy/Scala.xml>

- Scala API Docs

<http://www.scala-lang.org/api/2.10.3/#package>

- Michel Schinz, Philipp Haller: «Руководство по Scala для Java программистов». Версия 1.3, декабрь 2010, пер. Ржевский Дмитрий

http://www.scala-lang.org/docu/files/ScalaTutorial-ru_RU.pdf

Язык Ocaml:

- Yaron Minsky, Anil Madhavapeddy, Jason Hickey: Real World Ocaml. «Functional programming for the masses», O'Reilly Media, November 2013, pages 510

<https://realworldocaml.org/v1/en/html/index.html>

- Package List

<https://ocaml.janestreet.com/ocaml-core/latest/doc/>

- Ксавье Лерой и др. : «Система Objective Caml, релиз 3.10». Документация и руководство пользователя

<http://ocaml.spb.ru/>

- Emmanuel Chailloux, Pascal Manoury, Bruno Pagano : «Разработка программ с помощью Objective Caml»

<http://shamil.free.fr/comp/ocaml/html/index.html>

Язык Haskell:

- Haskell

<http://ru.wikipedia.org/wiki/Haskell>

- «O Haskell по-человечески», март 2014

<http://ohaskell.ru/>

- «Язык Haskell: O пользе и вреде лени»

[http://ru.wikibooks.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_Haskell:_%D0%9E_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%B5_%D0%B8_%D0%B2%D1%80%D0%B5%D0%B4%D0%B5_%D0%BB%D0%B5%D0%BD%D0%B8]

- «Язык и библиотеки Haskell 98»

<http://www.haskell.ru/>

- «Haskell 2010. Language Report»

<http://www.haskell.org/onlinereport/haskell2010/>

- Пол Хьюдак, Джон Петерсон, Джозеф Фасел: «Мягкое введение в Haskell», пер. Денис Москвин

Часть 1: http://rsdn.ru/article/haskell/haskell_part1.xml#EKC

Часть 2: http://rsdn.ru/article/haskell/haskell_part2.xml#EOD

- «Основы функционального программирования / Основы языка Haskell»

http://ru.wikibooks.org/wiki/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B_%D1%8F

[%D0%B7%D1%8B%D0%BA%D0%B0_Haskell](#)

Язык X10:

- X10

[http://ru.wikipedia.org/wiki/X10_\(%FF%E7%FB%EA_%EF%F0%EE%E3%F0%E0%EC%EC%E8%F0%EE%E2%E0%ED%E8%FF\)](http://ru.wikipedia.org/wiki/X10_(%FF%E7%FB%EA_%EF%F0%EE%E3%F0%E0%EC%EC%E8%F0%EE%E2%E0%ED%E8%FF))

- «X10: Performance and Productivity at Scale»

<http://x10-lang.org/>

- IBM Research – официальный сайт проекта X10

<http://www.research.ibm.com/x10/>

Язык Chapel:

- Chapel (язык программирования)

[http://ru.wikipedia.org/wiki/Chapel_\(язык_программирования\)](http://ru.wikipedia.org/wiki/Chapel_(язык_программирования))

- The Chapel Parallel Programming Language – сайт проекта

<http://chapel.cray.com/>

Язык Haxe:

- Haxe Language Reference

<http://haxe.org/ref>

- Documentation

<http://haxe.org/doc>

- Haxe API – Standard Library

<http://api.haxe.org/>

Язык Erlang:

- Erlang

<http://progopedia.ru/language/erlang/>

- Build massively scalable soft real-time systems – сайт проекта

<http://www.erlang.org/>

Разное:

- «Энциклопедия языков программирования» - начальные сведения о нескольких десятках (практически всех существующих) языках программирования

<http://progopedia.ru/>