Производительность языков программирования

Автор: Олег Цилюрик

Редакция 10, от 01.02.2018

Оглавление

Предисловие....1
Прежде всего задача....2
Язык C....4
C++....5
Java....6
Scala....7
Kotlin....8
Python....9
Ruby....10
Perl....10
JavaScript....11
PHP....12
Lua....13
bash....13
Tcl....15
Go....16
Ocaml....17
Scheme....18
Haskell....19
Добавим экзотики….....20
  PureBasic....20
  Euphoria....21
Обсуждение....22
Литература....23

© 2014-2018

Предисловие

Сравнивать скорость выполнения сходных фрагментов кода, записанных на разных языках программирования — дело дурное. Потому что, во-первых, результаты таких экспериментов будут радикально зависеть от множества привходящих факторов, таких, например, как версии компиляторов и интерпретаторов, установленные уровни оптимизации … и другие, которые контролировать во всём их множестве невозможно. С другой стороны, многие языки программирования имеют совершенно другие достоинства, которые нивелируют скорость — здесь имеются в виду такие как: выразительная мощность, лаконичность, прозрачность и понятность кода. Наконец, различные по идеологии языки будут иметь совершенно различающуюся относительную производительность на различных **классах** задач: язык А может в разы превышать скорость языка Б на математических вычислений, и одновременно в десяток раз уступать языку Б на обработке символьных строк.

И, тем не менее, такие сравнения публикуются и публикуются, и продолжается это уже не одно десятилетие. В чём же смысл? А смысл в том, что время выполнения эквивалентных кодов в разных языках могут отличаться в сотни и даже тысячи раз, то есть оценивать можно **порядки** в различиях скорости. Это может определить некоторые резоны для выбора инструментария для вашего будущего проекта.

Ниже показаны сравнения по времени выполнения эквивалентных фрагментов некоторого произвольного алгоритма, записанного на многих (больше 10) языках программирования, да ещё и при некоторых различающихся условиях (используемый компилятор, уровень оптимизации кода). На это можно расценивать как на красивый этюд, позволяющий сравнить разноязыкие коды как по внешнему виду, разнообразию, так и по времени, которое они потянут на выполнении. В классической музыке есть такой жанр как «каприз». Здесь то же самое: игривое сравнение разновеликих вещей в свободное от основных занятий время. Кого не привлекает такой подход могут прекратить чтение ровно на этом месте...

Примечание: Не ищите какого-то скрытого смысла, подтекста в том порядке, в котором представлены различные языки — они описаны в том произвольном порядке, в котором они хронологически тестировались.

Прежде всего задача

Задачу мы хотели бы использовать вычислительного характера, простейшую и в реализации и в понимании, и которая имела бы очень высокую степень роста вычислительной сложности от размерности (например экспоненциальную), чтобы можно было в самых широких пределах изменять интегральную потребность в вычислительных операциях. Для сравнения производительности мало пригодны большинство традиционных реализаций задач, поскольку они нацелены на уменьшение вычислительной сложности, а нам нужно её максимально повысить.

В принципе, если уж замерять **порядки** скорости выполнения, то «по честному» хорошо бы иметь целую **линейку** подготовленных разносортных задач, для которых должны выполняться условия:

- Возможность параметризации размерности задачи N, N может задаваться, например, в командной строке запуска;
- Достаточно высокая степень роста вычислительной сложности: степенная $O(N^M)$, а ещё лучше экспоненциальная $O(M^N)$;
- Задача после запуска не может взаимодействовать с пользователем, ожидать ввода, задача выполняется автономно;
- После выполнения задача должна выводить только единичное (с минимальными затратами на операции вывода) контрольное значение: для контроля за корректностью выполнения, отсутствию переполнений и других ошибок;

Для разного характера вычислительных операций, в качестве таких задач можно бы было предложить: сортировка методом пузырька (как один из самых неэффективных способов сортировки), решение системы линейных уравнений методом Гаусса с выбором главного элемента или методом Гаусса-Жордана...

Для грубых оценок оказывается вполне пригодной задача **рекурсивного** вычисления чисел Фибоначчи (и её часто используют в этом качестве). Эта функция настолько проста, что её строгая формулировка будет просто показана в изложении кода на языке C … и вряд ли требует дополнительных объяснений.

Примечание (для дотошной публики)**:** Существуют 2 определения последовательности чисел Фибоначчи: а). $F_1=0$, $F_2=1$, $F_N=F_{N-1}+F_{N-2}$ и б). $F_1=1$, $F_2=1$, $F_N=F_{N-1}+F_{N-2}$. Как легко видеть, эти последовательности **сдвинуты** на 1 член, так что не стоит ломать копья по этому поводу: можно использовать любую форму. Мы будем использовать 2-ю форму.

Существуют эффективные алгоритмы вычисления последовательности чисел Фибоначчи (циклические, слева направо). Мы же сознательно будем использовать неэффективную рекурсивную реализацию (справа налево), именно в той форме, как в выражениях, записанных выше. При таком алгоритме задача как-раз удовлетворяет требованию высокой степени роста вычислительной сложности, о которой было сказано ранее. В чём убедиться мы можем, написав для этого небольшое приложение (на этот раз на Python):

```
#!/usr/bin/python

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

import sys


c = 0
```

```
def fib( n ) :
    global c
    c += 1
    if n < 2 : return 1
    else: return fib( n - 1 ) + fib( n - 2 )


# тест степени роста вычислительной сложности fib( N ) от N
try:
   n = int( sys.argv[ 1 ] )
except IndexError:
   n = 25
for i in range( 5, n + 1, 5 ):
   c = 0
   f = fib( i )
   print( "{}({})".format( c, i ) )
```

Убеждаемся, что это именно то, что нам и нужно:

```
$ ./Ofibo.py 40
15(5)
177(10)
1973(15)
21891(20)
242785(25)
2692537(30)
29860703(35)
331160281(40)
```

Это характерно экспоненциальная сложность вида O(10ⁿ), одна из самых высоких из достижимых.

Подготовка приложений **к исполнению** очень различается между рассматриваемыми языками: где-то это просто исходный код, который подаётся на вход интерпретатора, в других случаях требуется компиляция в промежуточные байт-коды, или компиляция в исполнимые машинные коды. Все промежуточные фазы подготовки, там где они требуются, сведены в один `Makefile`.

Многие языковые средства предполагают и предоставляют те или иные способы оптимизации выполнения (например, уровень оптимизации указываемый компилятору). Мы постараемся ограничивать, где это возможно, уровень внутренней оптимизации, чтобы не ставить любой инструмент в неравное положении. Тем не менее, там где известны отчётливые способы управления оптимизации, мы рассмотрим влияние максимального уровня оптимизации.

Запуск программ на хронометраж в многозадачной операционной системе, казалось бы, следовало делать командами вида:

```
# time nice --adjustment=-19 ./fibo.py 30
```

- выполнять команду с максимальным приоритетом (`nice –adjustment=-9 …`), чтобы снизить дисперсию результатов;
- выполнять это от пользователя `root`, поскольку только `root` позволительно повышать приоритет задачи выше нормального (понижайте — сколько угодно);
- хронометраж выполняем системной командой `time` (не будем вмешиваться в процесс временных измерений);
- параметр (30, порядковый номер числа Фибоначчи) определяет размерность задачи, объём вычислений в зависимости от него нарастает экспоненциально.

На самом деле это совершенно не обязательно (опт показывает): если ваша операционная система специально не загружена какими-то параллельными задачами, разброс последовательных запусков по времени практически тот же при запуске тестовых программ и от рядового пользователя со стандартным приоритетом. По каждой реализации ниже будет показываться один запуск, но на самом

деле их делалось достаточно много (серией до 10 и более), а показанный в тексте — это средний, самый устойчивый вариант (при измерении временных интервалов повторяемость всегда является проблемой). Не используем результаты 1-го запуска в серии, чтобы обеспечить для разных запусков серии идентичные условия кэширования.

Всё! Итак, начинаем...

Язык C

Реализация задачи на языке C (файл fibo_c.c):

```
#include <stdio.h>


unsigned long fib( int n ) {
   return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 );
}

int main( int argc, char **argv ) {
   unsigned num = atoi( argv[ 1 ] );
   printf( "%ld\n", fib( num ) );
   return 0;
}
```

Выполнение:

```
$ gcc --version
gcc (GCC) 6.4.1 20170727 (Red Hat 6.4.1-1)
...
$ gcc -O0 fibo_c.c -o fibo_c
$ time ./fibo_c 30
1346269
real    0m0.026s
user    0m0.024s
sys     0m0.001s
$ time ./fibo_c 40
165580141
real    0m1.164s
user    0m1.150s
sys     0m0.001s
```

Можно обоснованно предположить, что приложение, компилированное из C кода, будет самым быстрым из всех. Поэтому именно это число мы станем использовать как базовое значения для последующих сравнений, а при сравнениях использовать максимально достижимые значения, которыми и будут именно результаты GCC, полученные при оптимизации кода компилятором GCC.

Поэтому, попутно посмотрим и зафиксируем что может дать оптимизация в компиляторе GCC:

```
$ gcc -O3 fibo_c.c -o fibo_c
$ time ./fibo_c 30
1346269
real    0m0.012s
user    0m0.011s
sys     0m0.000s
$ time ./fibo_c 40
165580141
real    0m0.475s
```

```
user    0m0.468s
sys     0m0.001s
```

Это мало о чём говорит в абсолютных цифрах, не нужно обольщаться цифрами, но оптимизация в GCC работает, и работает достаточно эффективно, что достаточно общеизвестно по публикациям.

C++

Реализация на языке C++ может выглядеть так (файл fibo_c.cc):

```
#include <iostream>
#include <stdlib.h>
using namespace std;

unsigned long fib( int n ) {
  return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 );
}

int main( int argc, char **argv ) {
   unsigned num = atoi( argv[ 1 ] );
   cout << fib( num ) << endl;
   return 0;
}
```

Из этого единого кода будет создано 2 приложения — компиляцией GCC и компиляцией Clang (Clang вам, возможно, придётся установить, но, на сегодня, это делается настолько стандартными телодвижениями, что не станем на этом останавливаться).

Выполнение приложения, собранного GCC:

```
$ g++ -O0 fibo_cc.cc -o fibo_cc
$ time ./fibo_cc 30
1346269
real    0m0.025s
user    0m0.022s
sys     0m0.003s
$ time ./fibo_cc 40
165580141
real    0m1.160s
user    0m1.144s
sys     0m0.002s
```

Здесь время абсолютно равное случаю реализации C, в пределах статистической погрешности, и трудно было ожидать чего-то иного (один и тот же компилятор, языки одного подмножества). Но для убедительности, и для последующих сравнений, посмотрим что там при оптимизации:

```
$ g++ -O3 fibo_cc.cc -o fibo_cc
$ time ./fibo_cc 40
165580141
real    0m0.471s
user    0m0.461s
sys     0m0.001s
```

Выполнение того же приложения, но собранного компилятором Clang:

```
$ clang --version
clang version 3.9.1 (tags/RELEASE_391/final)
$ clang++ -O0 fibo_cc.cc -o fibo_cl
```

```
$ time ./fibo_cl 30
1346269
real    0m0.025s
user    0m0.023s
sys     0m0.001s
$ time ./fibo_cl 40
165580141
real    0m1.244s
user    0m1.227s
sys     0m0.001s
```

Здесь тоже практически совпадение. И это хорошая новость! Потому что ещё не так давно (2014) для версии Clang 3.3 качество компилированного кода (в смысле скорости) было по цифрам в 2.5-2.7 раз хуже, чем у GCC.

Чуть хуже у Clang обстоит дело с оптимизацией кода:

```
$ clang++ -O3 fibo_cc.cc -o fibo_cl
$ time ./fibo_cl 40
165580141
real    0m0.685s
user    0m0.674s
sys     0m0.000s
```

Это в 1.5 раз хуже, чем у GCC, и это совпадает с рядом публикаций, где оценивается, что Clang всё ещё только приближается по уровню оптимизации к GCC.

Java

Реализация задачи на Java (файл fibo.java):

```
public class fibo {
   public static long fib( int n ) {
      return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 );
   }

   public static void main( String[] args ) {
      int num = new Integer( args[ 0 ] ).intValue();
      System.out.println( fib( num ) );
   }
}
```

Компиляция приложения выполняется в реализации OpenJDK:

```
$ java -version
openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151-8u151-b12-0ubuntu0.16.04.2-b12)
OpenJDK 64-Bit Server VM (build 25.151-b12, mixed mode)
$ javac fibo.java
$ ls -l *.class
-rw-rw-r-- 1 olej olej 594 янв 30 13:07 fibo.class
```

Выполнение:

```
$ time java fibo 30
1346269
real    0m0.111s
user    0m0.101s
```

```
sys     0m0.023s
$ time java fibo 40
165580141
real    0m0.805s
user    0m0.793s
sys     0m0.015s
```

Выполнение JVM байт-кода Java здесь на 30% даже быстрее, чем компилированный в машинные команды код C без оптимизации, но на 70% хуже (1.69 раз), если сравнивать с кодом, оптимизированным GCC. Это совершенно естественно, поскольку байт-код `fibo.class` выполняется виртуальной языковой машиной (JVM), **интерпретирующей**, хотя и с высокой эффективностью, промежуточный код.

Если то же самое проделать с оригинальном Oracle JDK, то временные результаты могут отличаться, но не очень значительно.

Scala

После Java, чтобы далеко не отвлекаться, перейдём к языку Scala. Scala — относительно новый (2003г.) язык, сочетающий в себе возможности функционального и объектно-ориентированного программирования. Многие считают Scala дальнейшим расширением языковой линии Java и даже называют его как Java++ (Scala вообще, в порядке совместимости, имеет возможность использовать все Java-пакеты, наработанные за десятилетия).

Scala вам ридётся устанавливать дополнительно (но, к счастью) из состава стандартных репозиториев вашего дистрибутива Linux (это потянет много, но очень коротких, пакетов):

```
$ sudo dnf install scala*
...
Установка  58 Пакетов
Объем загрузки: 84 M
Объем изменений: 568 M
Продолжить? [д/N]: y
...
Выполнено!
$ scala -version
Scala code runner version 2.10.4 -- Copyright 2002-2013, LAMP/EPFL
```

Реализация программы вычисления чисел Фибоначчи на языке Scala (файл fibo.scala):

```
object fibo_scala {

   def fib( n: Int ): Long = if( n < 2 ) 1 else fib( n - 1 ) + fib( n - 2 )

   def main( args: Array[ String ] ): Unit = {
      System.out.println( fib( args( 0 ).toInt ) )
   }

}
```

Компиляция такой программы в файл (компиляция происходит заметно продолжительное время):

```
$ scalac fibo.scala
$ ls -l *scala*.class
-rw-rw-r-- 1 olej olej 698 янв 30 13:36 fibo_scala.class
-rw-rw-r-- 1 olej olej 981 янв 30 13:36 'fibo_scala$.class'
```

Хронометраж выполнения этого варианта программы, в сравнении с эталонной реализацией GCC C, и родственной Scala реализацией на Java:

```
$ time scala fibo_scala 30
1346269
real    0m0.381s
user    0m0.455s
sys     0m0.046s
$ time scala fibo_scala 40
165580141
real    0m1.045s
user    0m1.092s
sys     0m0.061s
```

Это очень хороший результат! : всего в 2.2 раз медленнее, чем компилированный бинарный код C, и в 1.3 раза медленнее чем Java. Вполне адекватная плата за функциональность, мощность и возможности нового инструмента.

Kotlin

Ещё один язык, продолжающий линию Java: Kotlin — это совершенно молодой язык от российской компании JetBrains. Появился он только в 2011 году. На конференции Google I/O 2017 команда разработчиков Android сообщила, что Kotlin получил официальную поддержку для разработки Android-приложений.

Самый прямой путь установить себе Kotlin — это установить хорошо известную и ранее свободную среду разработки (IDE) IntelliJ от той же компании JetBrains. Но ещё проще способ установки (и самое главное — последующих обновлений версий!) Kotlin (в UNIX) — это воспользоваться инструментом (скриптом) SdkMan! (The Software Development Kit Manager):

```
$ sdk install kotlin
...
$ kotlin -version
Kotlin version 1.2.21-release-88 (JRE 1.8.0_151-b12)
```

Перепишем наш вариант Java-приложения по Kotlin (файл fibo.kt):

```
fun fib( n: Long ): Long {
   return if( n < 2.toLong() ) 1.toLong() else fib( n - 1 ) + fib( n - 2 );
}

fun main( args: Array<String> ) {
   val x = args[ 0 ].toLong()
   println( fib( x ) )
}
```

Компиляция в архив исполнимого байт-кода (`.jar`)

```
$ kotlinc fibo.kt -include-runtime -d fibo.jar
```

Выполнение и хронометраж:

```
$ time java -jar fibo.jar 30
1346269
real    0m0.114s
user    0m0.108s
sys     0m0.017s
$ time java -jar fibo.jar 40
165580141
real    0m0.714s
user    0m0.704s
sys     0m0.016s
```

Это приложение, как видно, исполняется той же Java виртуальной машиной. И выполнение под той же виртуальной машине Java, байт-кода, созданный компилятором Kotlin, происходит на 13% быстрее.

Python

Python, почти наверняка, установлен в вашем дистрибутиве Linux изначально, по умолчанию. Более того, зачастую установлены 2 параллельных версии: Python 2 и Python 3.

Аналогичный код программы пишем на Python (файл fibo.py):

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

import sys

def fib( n ) :
    if n < 2 : return 1
    else: return fib( n - 1 ) + fib( n - 2 )

n = int( sys.argv[ 1 ] )
print( "{}".format( fib( int( sys.argv[ 1 ] ) ) ) )
```

Для этого кода (он написан в совместимом синтаксисе, между синтаксисом версий Python) мы можем также предложить 2 различных способа исполнения:

– Python версии 2:

```
$ python --version
Python 2.7.13
$ time python fibo.py 30
1346269
real    0m0.632s
user    0m0.614s
sys     0m0.009s
$ time python fibo.py 40
165580141
real    1m9.900s
user    1m9.178s
sys     0m0.005s
```

– Python версии 3:

```
$ python3 --version
Python 3.5.4
$ time python3 fibo.py 30
1346269
real    0m0.682s
user    0m0.670s
sys     0m0.009s
$ time python3 fibo.py 40
165580141
real    1m28.808s
user    1m26.942s
sys     0m0.008s
```

Здесь практически подобные времена выполнения в Python 2 и в Python 3 (разница в 8-25%). И это тоже очень хорошая новость, потому что для более ранней (когда Python 3 только находился в

развитие) версии Python 3.3.2 (2014г.) время выполнения этого же файла кода было в 2.5-3 раза продолжительнее.

А вот в сравнении с нативным компилированным кодом С Python проигрывает 147-186 раз! При особенно больших объёмах повторений всё становится совсем грустно. И это тоже соответствует тому, что звучит в публикациях.

Ruby

Ruby — интерпретируемый язык программирования высокого уровня. Поддерживает много разных парадигм программирования, прежде всего классово-объектную. Ruby был задуман в в 1993 году японцем Юкихиро Мацумото, стремившимся, по его утверждению, создать язык, совмещающий все качества других языков, способствующие облегчению труда программиста.

Ruby, скорее всего, вам придётся устанавливать в вашем дистрибутиве Linux. Но делается это средствами пакетной системы из стандартного репозитория:

```
$ dnf list ruby
...
Установка  12 Пакетов
Объем загрузки: 4.3 M
Объем изменений: 14 M
Продолжить? [д/Н]: у
...
Выполнено!
$ ruby --version
ruby 2.3.4p301 (2017-03-30 revision 58214) [x86_64-linux]
```

Реализация той же задачи на Ruby (файл fibo.rb):

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

def fib( n )
   return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 )
end

puts fib( ARGV[ 0 ].to_i )
```

Выполнение:

```
$ time ruby fibo.rb 30
1346269
real    0m0.277s
user    0m0.262s
sys     0m0.010s
$ time ruby fibo.rb 40
165580141
real    0m40.155s
user    0m38.229s
sys     0m0.021s
```

Это очень здорово для **интерпретирующего** языка: в 2.5-3.2 раза лучше, чем у Python, но медленнее нативного кода С в 85 раз.

Perl

Perl — это патриарх языков программирования, нацеленный, главным образом, на символьную обработку, и не особенно применимый для вычислительных задач. Но всё же...

Реализация задачи на Perl (fibo.pm):

```
#!/usr/bin/perl


sub fib {
  my $n = shift;
  $n < 2 ? 1 : fib( $n - 1 ) + fib( $n - 2 )
}


$f = fib( $ARGV[ 0 ] );
print "$f\n";
```

Выполнение:

```
$ perl --version
This is perl 5, version 24, subversion 3 (v5.24.3) built for x86_64-linux-thread-multi
(with 66 registered patches, see perl -V for more detail)
...
$ time perl fibo.pm 30
1346269
real    0m1.051s
user    0m1.034s
sys     0m0.002s
$ time perl fibo.pm 40
165580141
real    3m15.432s
user    3m11.571s
sys     0m0.011s
```

Здесь проигрыш нативному коду C (оптимизированному) составляет больше 411 раз! Но это совершенно естественно и ожидаемо — Perl не язык для вычислений, и его ниша это текстовая обработка.

JavaScript

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 различных крупныхпроектах. Но JavaScript может использоваться и автономно, как инструмент консольных приложений, что гораздо менее известно. Если у вас ещё не установлена командная оболочка JS shell, то вам нужно её установить — когда-то раньше в составе репозиториев Linux был пакет с именем `spidermonkey-bin` (и возможно есть для вашей системы), с другой стороны вы можете взять свежую бинарную реализацию из проекта:

```
$ ls -l jsshell-linux-*
-rw-rw-r-- 1 olej olej  9045512 янв 30 16:14 jsshell-linux-i686.zip
-rw-rw-r-- 1 olej olej 10711620 янв 30 16:17 jsshell-linux-x86_64.zip
```

Реализация тест-программы на JavaScript (файл fibo.js):

```
#!/usr/bin/js -U


var fib = function( n ) {  // функциональный литерал
       return n < 2 ? 1 : fib( n - 1 ) + fib( n - 2 );
    }
```

```
print( fib( arguments[ 0 ] ) )
```

Выполнение приложения (начиная с уточнения версии):

```
$ js -v
JavaScript-C 1.8.5 2011-03-31
$ time js fibo.js 30
1346269
real    0m0.340s
user    0m0.328s
sys     0m0.007s
$ time js fibo.js 40
165580141
real    1m0.103s
user    0m58.757s
sys     0m0.009s
```

Этот результат удивил: это почти те же цифры, что и у Ruby, и до 2-х раз лучше, чем Python. От нативного кода C здесь отставание в 127 раз.

PHP

Для полноты картины посмотрим PHP. Его тоже, возможно, придётся установить дополнительно, но это такой распространённый инструмент, что ставится он стандартно средствами вашей пакетной системы:

```
$ sudo dnf install php
...
Объем загрузки: 9.5 M
Объем изменений: 34 M
Продолжить? [д/Н]: y
...
Выполнено!
```

Эквивалент нашей задачи, выраженный на языке PHP (файл fibo.php):

```
#!/usr/bin/php
<?php

function fib( $n ) {
   return $n < 2 ? 1 : fib( $n - 1 ) + fib( $n - 2 );
}

echo fib( $argv[ 1 ] ), "\n";
?>
```

Выполнение приложения:

```
$ php --version
PHP 7.0.25 (cli) (built: Oct 28 2017 06:17:04) ( NTS )
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2017 Zend Technologies
$ time php fibo.php 30
1346269
real    0m0.203s
user    0m0.191s
```

```
sys     0m0.008s
$ time php fibo.php 37
165580141
real    0m33.147s
user    0m31.910s
sys     0m0.011s
```

Отставание здесь от C составляет 70 раз.

Lua

Lua (Луна, исп.) — интерпретируемый язык программирования, разработанный подразделением Tecgraf Католического университета Рио-де-Жанейро, история языка ведёт отсчёт с 1993 года. На сегодня это небольшой и модный язык, популярный у разработчиков компьютерных игр … и в других областях там где используется JavaScript.

С инструментарием Lua вас могут возникнуть проблемы связанные с версией. Проверьте версию Lua, если она даже установлена у вас в системе по умолчанию — версия должна быть хотя бы не ниже 5.2 (в более старых версиях просто даже не отрабатывает ряд стандартных синтаксических конструкций, описываемых справочником по языку):

```
$ lua -v
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
```

Теперь мы можем реализовать нашу задачу на языке Lua (файл fibo.lua):

```
#!/usr/bin/lua

fib = function( n ) -- функциональный литерал
         if( n < 2 ) then
            return 1
         else
            return fib( n - 1 ) + fib( n - 2 )
         end
      end

print( fib( arg[ 1 ] + 0 ) )
```

Выполнение такого приложения:

```
$ time lua fibo.lua 30
1346269
real    0m0.266s
user    0m0.263s
sys     0m0.002s
$ time lua fibo.lua 40
165580141
real    1m12.525s
user    1m10.412s
sys     0m0.006s
```

Это примерно те же результаты, что и у JavaScript. Отставание от C составляет 152 раза.

bash

Можно ли организовать подобные вычисления в интерпретаторе bash, учитывая, что функции bash могут возвращать только значения кода завершения в пределах [0...255], то есть в нашем смысле — не имеющие возвращаемых вычисленных значений? Прежде всего, можно организовать подобные вычисления, если сам скрипт будет **рекурсивно** вызывать свои копии (но при этом каждый раз

вызываемая копия считывается с диска, пусть даже и из кэша диска).

Вот только то и всего — в реализация задачи в bash (файл fido.sh):

```
#!/bin/bash

if [ "$1" -lt "2" ]
then
   echo "1"
else
   f1=$($0 `expr $1 - 1`)
   f2=$($0 `expr $1 - 2`)
   echo `expr $f1 + $f2`
fi
```

Я не рискну вызывать такое решение с аргументом 30 (как остальные варианты) — я просто никогда не дождусь такого решения... Но выполняется такого скрипт вполне успешно:

```
$ bash --version
GNU bash, version 4.3.43(1)-release (x86_64-redhat-linux-gnu)
...
$ time ./fibo.sh 10
89
real    0m2.955s
user    0m1.223s
sys     0m1.742s
$ time ./fibo.sh 15
987
real    0m12.551s
user    0m4.932s
sys     0m7.645s
```

Получается, что скрипт bash вычисляет функцию от 10 примерно столько же, сколько не очень «спешному» Perl требуется для вычисления функции от 32! Практического смысла показанная реализация bash не имеет, но сама такая возможность интересна.

Другой возможностью может быть искусственно организованная рекурсия (с очередью, стеком возвратов … со всем что положено) при вызове функции внутри скрипта.

Внутренняя рекурсия в bash (файл fido_f.sh):

```
#!/bin/bash

declare -a res

fib () {
   if [ "$1" -lt 2 ]
   then
      res[ $1 ]=1.
   else.
      fib `expr $1 - 1`
      let s=${res[ `expr $1 - 1` ]}+${res[ `expr $1 - 2` ]}
      res[ $1 ]=$s
   fi
}
```

```
res[ 0 ]=1
fib $1
echo ${res[ $1 ]}
```

Здесь уже совсем другие результаты:

```
$ time ./fibo_f.sh 30
1346269
real    0m0.161s
user    0m0.056s
sys     0m0.108s
$ time ./fibo_f.sh 37
39088169
real    0m0.203s
user    0m0.064s
sys     0m0.141s
```

Результаты даже значительно превосходят результаты, полученные при выполнении нативного C кода. Но здесь мы просто наблюдаем результат обмана: при вычислениях сделана «оптимизация» и фактически (действительно) **рекурсивное** вычисление выродилось в циклическое, не порождая 2-х деревьев рекурсивных вызовов.

Tcl

Tcl (Tool Command Language — командный язык инструментов) — это старый (1993-1994г.) добротный скриптовый язык высокого уровня, очень популярный во всех UNIX-подобных операционных системах. До сих пор достаточно используемый, особенно когда нужно быстро и просто создать графический интерфейс пользователя (виджеты), и тогда Tcl используется совместно с графической библиотекой Tk (Tool Kit), и тогда всё это вместе называется Tcl/Tk.

Наше тестовое приложение на Tcl будет выглядеть так (файл fibo.tcl):

```
proc tcl::mathfunc::fib { n } {
   if { $n < 2 } {
      return 1
   } else {
      return [ expr { fib( $n - 1 ) + fib( $n - 2 ) } ]
   }
}

puts [ tcl::mathfunc::fib [ lindex $argv 0 ] ]
```

Тестируем:

```
$ time tclsh fibo.tcl 30
1346269
real    0m1.426s
user    0m1.246s
sys     0m0.006s
$ time tclsh fibo.tcl 40
165580141
real    4m11.922s
user    4m7.899s
sys     0m0.055s
```

Здесь отставание от C составляет 530 раз! Но в этом ничего нет удивительного, никто и не может ожидать скорости от Tcl, у него другая функция — быстрое создание простых GUI-приложений.

Go

Go — относительно новый **компилируемый** язык программирования, разрабатываемый компанией Google, ориентированный на высокопроизводительную параллельную обработку (но в наших целях его параллелизм никак не будет использоваться … чтобы сохранить равные условия для всех языков). 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. В заметном смысле Go является по многим признакам современным продолжателем языковой линии C/C++. Официально язык был представлен в ноябре 2009 года.

Реализация теста на языке Go (файл fido_go.go):

```
package main


import (
   "fmt"; "os"; "strconv"
)


func fib( n int ) int {
   if n < 2 {
      return 1
   } else { return fib( n - 1 ) + fib( n - 2 ) }
}


func main(){
   n, _ := strconv.Atoi( os.Args[ 1 ] )
   fmt.Println( fib( n ) )
}
```

На данный момент существуют две основных линии развития 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).

Поэтому любопытно взглянуть и на сборку и на выполнение в каждой из этих 2-х линеек:

```
$ go build fibo_go.go
$ go version
go version go1.7.6 linux/amd64
$ gccgo fibo_go.go -g -O0 -o fibo_go_gcc
$ ls -l *_go*
-rwxrwxr-x 1 olej olej 1646458 янв 30 18:59 fibo_go
-rwxrwxr-x 1 olej olej   33704 янв 30 18:59 fibo_go_gcc
-rw-r--r-- 1 olej olej     242 янв 29 23:49 fibo_go.go
```

(Обращаем внимание на разительную разницу в размерах созданных исполнимых файлов. Связано это с тем, что GoLang по умолчанию создаёт статически скомпонованное приложение, независимое от операционной системы и её разделяемых библиотек. Но начиная с версии 1.5 GoLang, по желанию, может собирать и приложение, связываемое с динамическими библиотеками системы.)

Сборка и выполнения этой программы (с демонстрацией используемой версии):

- компилятор GCC

```
$ time ./fibo_go_gcc 30
```

```
1346269
real    0m0.175s
user    0m0.154s
sys     0m0.018s
$ time ./fibo_go_gcc 40
165580141
real    0m1.770s
user    0m1.736s
sys     0m0.014s
```

- компилятор проекта GoLang :

```
$ time ./fibo_go 30
1346269
real    0m0.020s
user    0m0.019s
sys     0m0.002s
$ time ./fibo_go 40
165580141
real    0m1.088s
user    0m1.077s
sys     0m0.001s
```

И вот здесь нас ожидает сюрприз: маленький и не очень пока отточенный из-за своей новизны GoLang, показывает замечательные результаты, **лучше** чем код на C, откомпилированный GCC! А на очень больших объёмах Go (оба компилятора) уступает оптимизированному C до 3-х раз.

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.

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,

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 подобных). Ocaml вам придётся устанавливать, но это делается пакетным менеджером из репозитория дистрибутива, для наших целей достаточно иметь хотя бы:

```
$ dnf list installed ocaml*
Установленные пакеты
ocaml.x86_64                    4.02.3-3.fc25                   @fedora
ocaml-compiler-libs.x86_64      4.02.3-3.fc25                   @fedora
ocaml-runtime.x86_64            4.02.3-3.fc25                   @fedora
```

Реализация теста на языке Ocaml (файл fido_ml.ml):

```
let rec fib n =
  if n < 2 then 1 else fib( n - 1 ) + fib( n - 2 );;


let main () =
  let arg = int_of_string Sys.argv.( 1 ) in
```

```
  print_int( fib arg );
  print_newline();
  exit 0;;

main ();;
```

Но этот код можно Ocaml можно выполнять двояким способом … посредством его интерпретации:

```
$ ocaml -version
The OCaml toplevel, version 4.02.3
$ time ocaml fibo_ml.ml 30
1346269
real    0m0.214s
user    0m0.089s
sys     0m0.007s
$ time ocaml fibo_ml.ml 40
165580141
real    0m7.288s
user    0m7.152s
sys     0m0.006s
```

Другой вариант — выполнение кода, предварительно откомпилированного в машинные команды, бинарную форму:

```
$ ocamlc -o fibo_ml fibo_ml.ml
$ time ./fibo_ml 30
1346269
real    0m0.084s
user    0m0.082s
sys     0m0.001s
$ time ./fibo_ml 40
165580141
real    0m7.203s
user    0m7.187s
sys     0m0.002s
```

Вообще то, по времени выполнения различия между способами запуска не столь существенные. Это наталкивает на мысль, что интерпретатор Ocaml работает с предкомпиляцией (JIT), а компилированная форма — это тот же байт-код с прикомпонованной к нему исполняющей системой.

Звучащие утверждения об очень высокой производительности Ocaml этими экспериментами никак не подтверждаются. Скорость в сравнении с кодом C меньше в 15 раз.

Scheme

Scheme — один из двух наиболее популярных в наши дни диалектов языка Lisp. Одна из доступных реализаций Scheme под Linux (есть и другие) называется Guile (присутствует в стандартных репозиториях):

```
$ guile --version
guile (GNU Guile) 2.0.13
...
```

Реализация того же приложения на чисто функциональном языке Scheme (файл fibo.scm):

```
;; функция Фибоначчи – требует параллельной рекурсии
(define (fib n)
  (cond ((= n 0) 1)
```

```
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))


(begin (write ( fib (string->number (car (cdr (command-line)))))) (newline))
```

Выполнение:

```
$ time guile -q --no-auto-compile fibo.scm 30
1346269
real    0m0.319s
user    0m0.311s
sys     0m0.012s
$ time guile -q --no-auto-compile fibo.scm 40
165580141
real    0m29.391s
user    0m29.160s
sys     0m0.011s
```

Разница с оптимизированным C здесь составляет 62 раза. Но Lisp никогда и не предназначался для массированных численных вычислений.

Haskell

Ещё один стандартизованный чистый функциональный язык программирования общего назначения Haskell. В 1990 г. была предложена первая версия языка, Haskell 1.0. Непосредственно на него оказал очень сильное влияние язык Miranda, разработанный в 1985г. Дэвидом Тёрнером (Miranda 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 начального вхождения в программирование на Haskell высок.

Haskell вам, скорее всего, придётся устанавливать отдельно, но это опять же делается средствами пакетной системы:

```
$ sudo dnf install ghc
...
Установка  49 Пакетов
Объем загрузки: 88 M
Объем изменений: 985 M
Продолжить? [д/Н]: y
...
Выполнено!
```

Реализация той же функциональности на Haskell (файл fibo.hs):

```
module Main where
import System.Environment

main :: IO ()

-- определение функции
fibo :: Integer -> Integer
fibo n | n == 0    = 1
       | n == 1    = 1 {- вложенный комментарий -}
       | otherwise = fibo( n - 1 ) + fibo( n - 2 )

-- сама программа
```

```
main = do
     args <- getArgs
     print( fibo( read( args !! 0 ) :: Integer ) )
```

Компиляция:

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

Выполнение:

```
$ time ./fibo_hs  30
1346269
real    0m0.271s
user    0m0.263s
sys     0m0.005s
$ time ./fibo_hs  40
165580141
real    0m29.087s
user    0m28.635s
sys     0m0.081s
```

Цифры здесь очень близкие к Scheme, а проигрыш в скорости C — 65 раз.

Добавим экзотики…

PureBasic

Ещё один вариант для любителей простых и лёгких решений — PureBasic (простые и лёгкие решения позже оказываются очень сложными при дальнейшем развитии проекта). PureBasic, вообще то говоря, коммерческий продукт, но его демонстрационная версия, полностью работоспособная, позволяет создавать приложения не более 800 строк. Для небольших утилит и системных скриптов этого вполне достаточно, а для крупных проектов использовать что-либо из клонов Basic я бы просто не рискнул.

Как язык программирования всё, что есть клонами Basic — вообще неинтересно … по мнению автора. Но в контексте нашего рассмотрения PureBasic интересен только как ещё один сравнительный вариант реализаций с **компиляцией** в исполнимый машинный код не требующий исполняемой системы, которых на сегодня в природе почти не осталось (к этому вопросу мы ещё отдельно вернёмся в заключительной части). Если вы пожелаете воспроизвести показываемые далее результаты, вам будет необходимо а). скачать архив purebasic-demo_x64.tar.gz (см. библиографию), б). разархивироать его и в). запустить инсталляцию.

Реализация в системе PureBasic (файл fido.pb):

```
rocedure fib(n)
   If n<2
      ProcedureReturn 1
   Else
      ProcedureReturn fib(n-1) + fib(n-2)
      EndIf
EndProcedure

OpenConsole()
PrintN(Str(fib(Val(ProgramParameter(0)))))
CloseConsole()
```

Если вы успешно установили PureBasic, то можете компилировать этот код:

```
$ pbcompiler fido.pb -e fibo_pb
```

И то, что из этого получилось в итоге:

```
$ time ./fibo_pb 30
1346269
real    0m0.196s
user    0m0.026s
sys     0m0.003s
$ time ./fibo_pb 40
165580141
real    0m1.482s
user    0m1.478s
sys     0m0.001s
```

Это компилирующая реализация. Но она проигрывает C в более чем 3 раза.

Euphoria

Euphoria – простой, гибкий, легкий в изучении язык программирования (в русскоязычном сегменте почти не известный). Он позволит быстро и без затруднений разрабатывать программы для Windows, DOS, Linux и FreeBSD. Начало Euphoria было положено в 1993.

Euphoria вам практически наверняка придётся установить самостоятельно, в репозиториях Linux вы её не найдёте. Вы можете скачать (см. библиографию) архив, разархивировать его в любое место, но лучше в /usr/share/euphoria на который настроены конфигурации, и затем править конфигурации в каталоге установки: bin/eu.cfg. Но ещё проще путь: скачать инсталляционный скрипт с GitHub (он же есть и на указанной выше странице) и запустить его — это действие полностью проведёт установку (а позже, если нужно, и удаление):

```
$ ./geuphoria.sh
    [SYNOPSIS]
    ./geuphoria.sh <option>
    [OPTIONS]
    install | i     Install euphoria-4.1.0
    remove  | r     Uninstall euphoria-4.1.0
    clean   | c     Clear out euphoria downloads in /tmp
$ sudo ./geuphoria.sh i
...
'/usr/bin/eui' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eui'
'/usr/bin/euc' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/euc'
'/usr/bin/eutest' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eutest'
'/usr/bin/eudist' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eudist'
'/usr/bin/eudoc' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eudoc'
'/usr/bin/eushroud' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eushroud'
'/usr/bin/euloc' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/euloc'
'/usr/bin/eudis' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eudis'
'/usr/bin/eubind' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eubind'
'/usr/bin/eub' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eub'
'/usr/bin/echoversion' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/echoversion'
'/usr/bin/creole' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/creole'
'/usr/bin/eucoverage' -> '/usr/local/euphoria-4.1.0-Linux-x64/bin/eucoverage'
$ which eui
/usr/bin/eui
$ eui -v
```

```
Euphoria Interpreter v4.1.0 development
   64-bit Linux, Using System Memory
   Revision Date: 2015-02-02 14:18:53, Id: 5861:57179171dbed
```

Сборка исполнимого приложения (исполняющая система прикомпонована к приложению):

```
$ eubind -out fibo_eu fibo.ex
deleted 239 unused routines and 1017 unused variables.
You may now run ./fibo_eu
```

И выполнение:

```
$ time ./fibo_eu 30
1346269
real    0m0.409s
user    0m0.396s
sys     0m0.008s
$ time ./fibo_eu 40
165580141
real    0m47.612s
user    0m47.006s
sys     0m0.014s
```

Но можете исполнять программы Euphoria и в режиме интерпретации кода исполняющей системой:

```
$ time eui fibo.ex 30
1346269
real    0m0.515s
user    0m0.463s
sys     0m0.043s
$ time eui fibo.ex 40
165580141
real    0m56.494s
user    0m55.908s
sys     0m0.009s
```

Обещанные скорости Euphoria явно преувеличены — оптимизированному C она уступает в 118 раз.

Обсуждение

Прежде всего, отмечаем, что эффективность исполняемого кода зависит не только от языка, на котором код написан и от технологии используемой в языке (компиляция, интерпретация, компиляция в промежуточный байт-код), но и от конкретного компилятора и заказанных уровней его оптимизации. Разница в зависимости от этих факторов может составлять порядка 2-3-х раз.

С другой стороны, понятно, что интерпретируемые языки, более гибкие в смысле динамической типизации и, особенно, в операциях с функциями высших порядков (элементы функционального программирования), будут уступать в скорости компилируемым.

Хронометрировать скорость выполнения — неблагодарное занятие, а полученные результаты будут весьма и весьма относительными. Чтобы не сравнивать численные значения, разложим полученные значения для разных языков по логарифмическим кластерам, по фактору скорости относительно самого быстрого GCC C варианта (взяты цифры на очень больших числах повторений вычислений):

Фактор скорости	Язык
1 ... 2	GCC C/C++, Clang C++, Java, Kotlin
2 ... 5	GCC Go, GoLang, Scala, PureBasic
5 ... 10	-
10 ... 20	Ocaml

Фактор скорости	Язык
20 ... 50	Scheme
50 ... 100	Ruby, PHP, Haskell
100 ... 200	JavaScript, Python 2/3, Lua, Euphoria
200 ... 500	Tcl,
> 500	bash

Интуитивно понятно, что скорость выполнения кода, записанного на разных языках, будет различаться. Но проведенное тестирование (по 23 языкам и диалектам) показывает, что эти отличия могут составлять не десятки процентов или разы, и даже не десятки раз, а **многие сотни раз**.

Из всех охваченных в сравнении языков только очень немногие (C, C++, Go, PureBasic) используют технику «нативной» компиляции в машинный код используемой платформы. 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).

Такая тенденция связана, в первую очередь, с лавинным ростом скорости современных процессоров: такие скорости, которую предлагают производители процессоров, но которую не могут потребить и которая не нужна львиной доле потребителей компьютерного оборудования. Интерпретирующая реализация приложений канализирует избыточную производительность, чем удовлетворяет потребности рынков электронных компонент.

Но есть ещё одна объяснимая причина пристрастия к интерпретирующей реализации: она позволяет локализовать ошибки выполняемого программного кода (ошибки программиста) внутри исполняющей системы (виртуальной машины), не позволяя им распространиться на операционную систему. Это вполне можно принять как актуальный способ для повышения жизнестойкости... «настольных» приложений: учебных, математических вычислений, развлекательных, мультимедийных, бытовых... К сожалению, такой способ неприемлем для «промышленных» приложений: управляющих систем, электронных платежей, систем связи и телекоммуникаций — падение самого приложения там эквивалентно аварийной ситуации даже если операционная среда останется не затронутой, и это приложение будет повторно поднято.

Очень интересно сравнить (см. тесты выше) отношения времён выполнения на небольших (N=30) и очень больших (N=40) для разных языков программирования, и сравнения их с отношениями оценок фактических объёмов вычислений из скрипта Ofibo.py : K = O(40) / O(30) = 331160281 / 2692537 = 123. Для чисто **интерпретируемых** реализаций (Python, Ruby, JavaScript, PHP, Lua и другие) отношения времён T_{40}/T_{30} будет приближаться к этому значению K — затраченное время растёт пропорционально фактическому объёму вычислений. Для других же языков, для которых выполняется **компиляция** либо в машинный код (C, C++, Go), либо в промежуточный байт-код (Java, Scala, Kotlin — где, предположительно, всегда выполняется компиляция времени выполнения JIT, Just In Time) — отношения затрачиваемых времён T_{40}/T_{30} будет **меньше** K: C/C++ — 40, Go — 10, Java — 7.25, Scala — 2.7, Kotlin — 6.2, … Конечно, здесь смазывает картину достаточно заметное время загрузки, подготовки и запуска компилируемых приложений … но не до такой же степени.

Можно предположить, что для интерпретирующих реализаций, в силу локальности последовательно интерпретируемых операторов, невозможны а). оптимизация загружаемого кода перед выполнением и б). кэширование промежуточных результатов в ходе выполнения. А для компилируемых (в любой степени) языков доступно и то и другое.

Но проверить **точно** эти зависимости можно отказавшись от системного способа вычислений временных интервалов командой `time`, и используя временные «засечки» в самом коде тестовых приложений (разными способами для каждого языка теста). Это достаточно трудоёмкая задача.

Но различная реакция приложения на O(N) может влиять на **характер** поведения приложения в зависимости от массированной повторяемости вычислений, степени итерационности. И влиять на выбор языкового инструментария для тех или иных приложений.

Литература

- А. Гриффитс, GCC. Полное руководство. Platinum Edition, М.: «ДиаСофт», 2004, стр.624, ISBN 966-7992-33-0

  http://www.books.ru/books/gcc-polnoe-rukovodstvo-platinum-edition-190067/?show=1
- Java™ Platform, Standard Edition 6 API Specification

  http://docs.oracle.com/javase/6/docs/api/overview-summary.html
- 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
- The Software Development Kit Manager

  http://sdkman.io/index.html
- SDKMAN! - тул менеджмента SDK для разработчика на Java/Groovy/Kotlin/Scala

  https://ryadov.livejournal.com/2840.html
- Руководство по языку Kotlin

  http://kotlinlang.ru/
- Среда разработки IntelliJ IDEA, Java и Kotlin (отсюда можно свободно скачать)

  https://www.jetbrains.com/idea/
- Учебник Python 3.1

  http://ru.wikibooks.org/wiki/%D0%A3%D1%87%D0%B5%D0%B1%D0%BDD0%B8%D0%BA_Python_3.1#.D0.9A.D0.BB.D0.B0.D1.81.D1.81.D1.8B
- Ruby

  http://ru.wikibooks.org/wiki/Ruby
- Юкихиро Мацумото, Программирование на языке Ruby. Идеология языка, теория и практика применения

  http://lib.rus.ec/b/353387/read
- Том Кристиансен, Брайан Де Фой, Ларри Уолл, Джон Орвант, Программирование на Perl, 4-е издание, Сп-Б.: «Символ-Плюс», 2013, стр. 1048, ISBN 978-5-93286-214-8

  http://www.books.ru/books/programmirovanie-na-perl-4-e-izdanie-3135461/?show=1
- latest-mozilla-central (свежие реализации проекта Mozilla / SpiderMonkey): https://archive.mozilla.org/pub/firefox/nightly/latest-mozilla-central/
- Introduction to the JavaScript shell

  https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Introduction_to_the_JavaScript_shell
- Руководство по PHP

  http://ru2.php.net/manual/ru/index.php
- Lua 5.1 Reference Manual

  http://www.lua.org/manual/5.1/manual.html
- Справочное руководство по языку Lua 5.1

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

- Lua programming language information and resources

  http://lua-users.org/wiki/
- 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/
- 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
- Revised(5) Report on the Algorithmic Language Scheme

  http://groups.csail.mit.edu/mac/ftpdir/scheme-reports/r5rs-html/r5rs_toc.html#TOC1
- Язык и библиотеки Haskell 98

  http://www.haskell.ru/
- PureBasic (здесь можно скачать демоверсию для ознакомления)

  http://www.purebasic.com/download.php
- openEuphoria Linux Downloads

  http://openeuphoria.org/wiki/view/linux_download.wc
- Русскоязычный ресурс Euphoria

  http://www.rapideuphoria.com/russian/index_r.htm