****

程序语言认知实验

班 级： 07111505

姓 名： 兰天

学 号： 1120151828

目录

[一. 实验目的 1](#_Toc508551502)

[二. 实验内容 1](#_Toc508551503)

[三. 实验环境 1](#_Toc508551504)

[3.1. 计算机硬件配置 1](#_Toc508551508)

[3.2. 语言环境配置 2](#_Toc508551509)

[四. 实验过程 2](#_Toc508551510)

[4.1. 实验思路 2](#_Toc508551512)

[4.2. Intel x86 – Nasm 2](#_Toc508551513)

[4.3. C++ 3](#_Toc508551514)

[4.4. Java 3](#_Toc508551515)

[4.5. Haskell 4](#_Toc508551516)

[4.6. Python 4](#_Toc508551517)

[五. 语言易用性及程序规模对比 5](#_Toc508551518)

[5.1. 代码总行数对比如图5–1 5](#_Toc508551520)

[5.2. 运行时间对比如图5-3 7](#_Toc508551521)

[六. 心得体会 7](#_Toc508551522)

# 实验目的

感受编译执行和解释执行的两种不同执行程序的方式。初步体验语言对编译器设计的影响，为后续编译程序的设计和开发奠定良好的基础。了解不同的语言对编译的效果的影响。体会不同的语言的编译或者解释的方式和过程。

# 实验内容

利用计算机语言 C/C++, Java, Python, Haskell, X86 Assembly实现对大矩阵的乘法计算，对不同的语言的编程效率，程序的规模，程序的运行效率进行对比分析，得出相应的实验效果

# 实验环境



## 计算机硬件配置

|  |  |
| --- | --- |
| **Model** | **Intel(R) Core(TM)i7-6500U** |
| **Frequency** | 2.5GHz |
| **Cores** | 4 |
| **L1 cache** | 32KB |
| **L2 cache** | 256KB |
| **L3 cache** | 3072KB |

## 语言环境配置

|  |  |
| --- | --- |
| **语言** | **编译器** |
| **X86 NASM** | nasm |
| **C** | GCC 4.8 |
| **Java** | jdk 1.8.0\_92 |
| **Haskell** | GHC 7.10.3 |
| **Python** | Python 3.6.1 |

# 实验过程

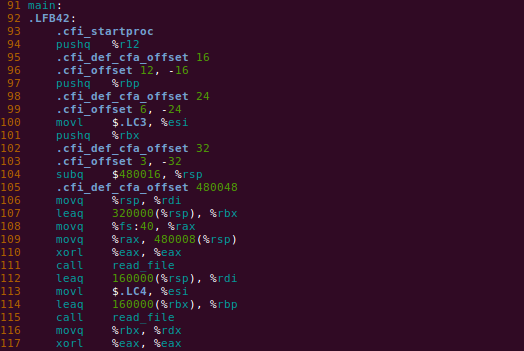


## 实验思路

* + - * + 使用 Python 随机生成两个 200 x 200 的矩阵然后保存，并将计算结果保存至另外的文件中用于验证
        + 编写每一个语言的对应的大矩阵乘法的程序然后在运算结果正确的基础上计算核心代码的执行时间
        + 比对程序的计算的时间效率等等性能分析

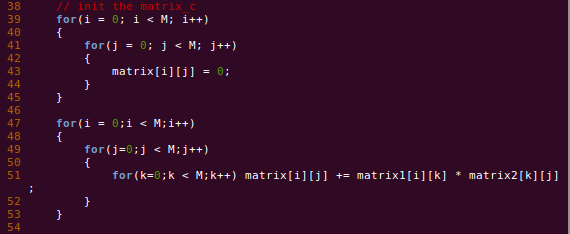
## Intel x86 – Nasm

矩阵乘法



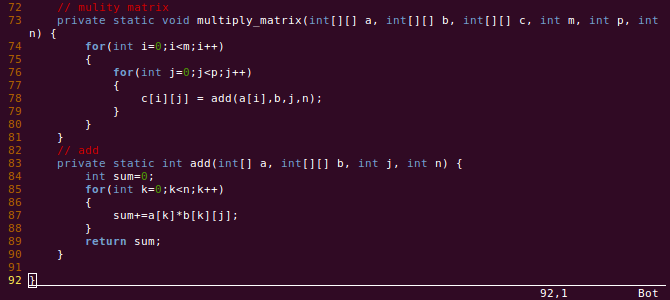
## C++

矩阵乘法



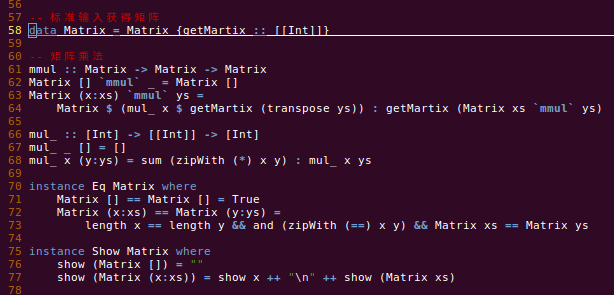
## Java

矩阵乘法



## Haskell

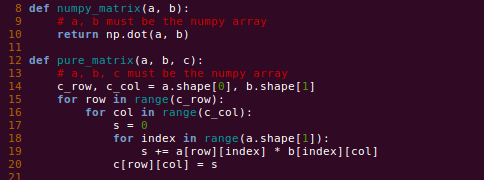
矩阵乘法



## Python

矩阵乘法

Python实现了两种矩阵乘法的格式，一种是采用纯Python语言实现的三重循环，另一种是采用numpy库实现的三重循环，numpy的实现版本可以作为矩阵乘法的基准实现



# 语言易用性及程序规模对比



## 代码总行数对比如图5–1

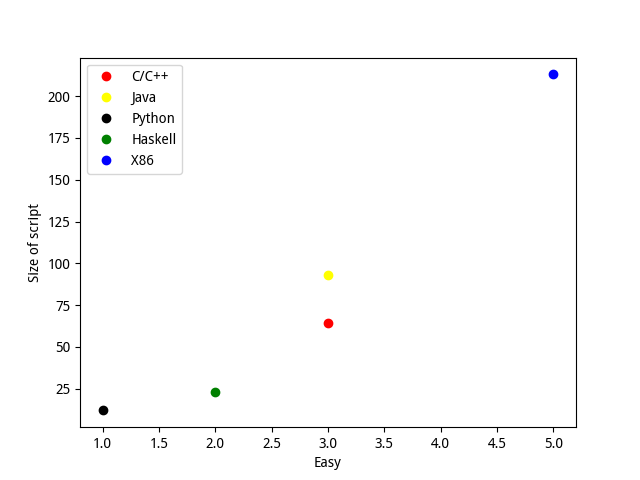


图5 – 1

程序的代码规模与语言的易用性成反比，语言越高级，代码规模越小。就Python和Nasm汇编的代码相差几乎达到百倍。语言越高级，编码方式越精简，更加适合作为项目开发的工具。语言的精简度直接决定了开发的速度，这一点也是为什么高级语言近几年来不断受到欢迎的原因

但是上图显示的仅仅是整体的程序的规模，但是考虑到任务的性质，实际中的代码的核心部分可能并不存在这样的特质。实际上对于大矩阵乘法来说基本上的算法都是三重循环，所以在实际的核心代码量比对上，每一个语言的相差度并不大，但是Intel汇编会比其他语言使用的核心代码量变得更多，由于寄存器的使用限制导致在进行每一层循环时都要反复进行读写内存，更改寄存器内容等操作导致代码量急剧增加。

## 运行时间对比如图5-3

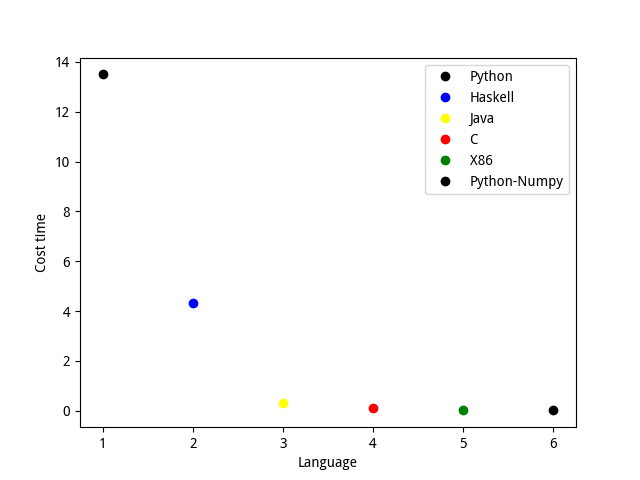


图 5 – 3

语言越高级，解释器或者是编译器内部做的工作越多，运行时间可能也越长。运行速度也差别非常巨大。在速度方面，越接近底层的语言速度越快，适合用在对计算速度有极致要求的地方。比如说内部的硬件的驱动等等，但是在高级程序开发或者说是数据处理等任务中，高级语言的使用可以避免让程序员陷入到繁琐的底层细节中

# 心得体会

本次试验中，通过对 C,Java,Python,Haskell,X86汇编的编程使用，了解了大量的有关其他语言的知识并且认识到了语言对程序的编译和执行的效率差别以及高级语言和低级语言相比的优缺点。

首先，并不能以偏概全的认为高级语言或者说低级语言相对的优秀或者说是不优秀，因为在某种层面上，不同的语言的工作环境和使用的方式其实也是不尽相同的。这里实际上包含了我们对程序编写的两种主要的诉求。

一方面，我们希望编程这门工作可以进行的简单有趣而不是枯燥乏味，而另一方面，我们又希望实现的语言具有优秀的时间复杂度和空间复杂度，换句话说我们希望编写的程序具有良好的执行效率。

但是这两者不可能兼得，高级语言反映了我们对底层语言的封装和简化，希望可以通过更接近自然语言的形式表达数理逻辑。但是相对的为了实现这种目的高级语言在向机器语言的转化的同时不可避免的会引入其他的我们没有在高级语言编程的时候考虑的细节优化，这一定会相对比低级语言更复杂。但是对于某些灵活度或者说是应用性非常强大的功能来说，高级语言相比低级语言来说会为我们带来巨大的程序实现上的便利，这也是高级语言不断的流行的原因之一。