编译系统原理第二次作业

定义你的编译器功能 & 汇编编程

姓名：马平川[①]

学号：1511442

学院：软件学院

专业：软件工程

2017 年 10 月 14 日

① pika7ma@gmail.com

摘　要

编译器是计算机程序编写执行中十分重要的一环，通过探究编译器的完整工作过程我们能够更加深入地了解程序的始终。本文以编译器工作流程为研究课题，配置了一个简化的 C++ 语言子集并设计了合理的上下文无关文法，随后实验了在该子集内的部分 C++ 语言代码，手动将其转化为汇编语言，由此来进一步理解编译器工作流程和原理。

关键词： 编译器，上下文无关文法，汇编语言，C++ 语言

ABSTRACT

Since the compiler has long been playing a dispensable role in programming field, unpacking the black box can provide us more insights with respect to program from its cradle to its grave. The overall running procedure of compiler is researched in this dissertation, subsequently, a simplified subset of C++ programming language is defined and configured with context-free grammar. Then an instance of program and its manually-translated assembly language program is discussed.

Keywords: compiler, context-free grammar, assembly programming language, C++ programming language

目 录

第一章 绪 论

1.1 研究工作的目的及意义

在计算机发展初期，计算机程序都是由汇编语言（甚至是针孔纸带）直接写成。不难想到，不仅这种语言的编写费时费力，对每种机型都得更改源码，更加不方便的是，这种编程方式使得程序员通常花费大量时间在没必要的架构和修饰问题上，而非解决问题本身。之后人们发明了高级编程语言，使得语言能够独立于机器存在，自此，编译器便担当了把高级语言翻译成机器语言的工作。而作为计算机中不可或缺的一类特殊程序，研究并了解编译器这一黑箱也成为了现代程序员必备的素养。

本文以探究现代编译器工作流程为目的，利用课上所学的上下文无关文法对 C++ 语言进行重新合理且正确的重新定义，并利用汇编知识模拟编译器工作以得到相似结果，而后比较两者异同并从机器的角度思考编译过程的原理。本项目全部源码开源于GitHub。

1.2 本论文的结构安排

本文的章节结构安排如下：

1. 大致总结所用编译器支持的 C++ 语言特性
2. 简单描述性定义使用的 C++ 语言子集
3. 利用上下文无关文法定义 C++ 语言子集
4. 手工编译示例
5. 编译器工作流程思考
6. 总结与回顾

第二章 G++ 编译器语言特性总结

G++ 支持的语言特性大致如下（非常简略地）：

- 支持布尔类型、整型、单精度浮点型、双精度浮点型、字符型、字符串型、宽字符型。
- 支持枚举类型、数组类型。
- 支持自定义类型。
- 支持类、结构体、共同体。
- 支持函数。
- 支持指针、引用。
- 支持类型修饰符。
- 支持常量类型。
- 支持宏定义。
- 支持循环语句。
- 支持循环语句。
- 支持各种表达式。
- 支持类型转换。
- 支持模版。
- 支持异常。
- 支持命名空间。
- 支持运算符重载。

以上仅为主要语法特性，还有更多特性正在更新中，实验主要根据其中常用且必要的特性进行。

第三章 C++ 语言子集定义

基于高效合理的角度，并结合现阶段知识掌握程度，定义支持如下操作的 C++ 语言子集：

1. 数据类型支持 int、float、double、char，及其所有的指针和数组类型，支持字符和字符串。
2. 算数运算符支持首尾 ++、首尾--、+、-、*、/、% 运算。
3. 位运算符支持～、&、|、∧、>>、<< 运算。
4. 逻辑运算符支持 &&、||、！运算。
5. 赋值运算符支持 =、+=、-=、*=、/=、%=、<<=、>>=、&=、∧=、|= 运算。
6. 其他运算符支持取地址、取指针、数组下标、函数调用、类成员、指针指向、逗号、正号、负号运算。
7. 支持合理的运算符优先级计算并支持括号优先级优先计算。
8. 支持函数功能。
9. 支持类和结构体功能。
10. 支持大括号复合语句。
11. 支持预编译语句。
12. 支持注释语句。
13. 支持标签语句。
14. 支持条件判断语句。
15. 支持单变量或多变量的声明和定义语句。
16. 支持循环语句。
17. 支持跳转控制语句。

第四章 上下文无关文法定义

4.1 定义数字

在这里定义全体实数。

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

int → *digit int* | *digit*

num → *int* | *int* **.** *int*

4.2 定义字符和字符串

为简化操作，这里使用非终结符 `ascii` 表示任一 ASCII 字符，使用 `esc` 表示任一转义序列。

text → *text ascii* | *text esc* | *ε*

character → **'** *ascii* **'** | **'** *esc* **'**

string → **"** *text* **"**

4.3 定义注释语句

为简化操作，这里使用非终结符 `whatever` 表示任意行任意长字符流，使用 `singleLine` 表示单行任意长字符流。

cmtStmt → **/*** *whatever* ***/** | **//** *singleLine*

4.4 定义常量

在这里定义 C++ 语言中的常量，包括数字、字符和字符串。

const → *num* | *character* | *string*

4.5 定义变量

根据 C++ 语言要求，变量需要由下划线或大小写拉丁字母开头，随后为任意位下划线、大小写拉丁字母或数字。

letter → **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

letter → **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

a → *digit a* | *letter a* | *_ a* | *ε*

id → **_** *a* | *letter a*

var → *id* | ***** *var* | *var* **[** *orExpr* **]** | *var* **[]** | **(** *var* **)**

4.6 定义类型

这里定义 C++ 语言基本四类型、类或结构体类型、及其指针类型。

basicTyp → **int** | **char** | **float** | **double** | *className*

typ → *basicTyp* | *typ* ***** | *typ* **[]**

4.7 定义函数

这里定义 C++ 语言中函数的定义和调用语法，支持无参数和多参数调用，支持无返回值类型。

funcName → *id*

funcTyp → *typ* | **void**

noneptDefParam → *typ var* | *typ var* **,** *noneptDefParam*

defParam → *noneptDefParam* | *ε*

noneptCallParam → *expr* **,** *noneptCallParam* | *expr*

callParam → *noneptCallParam* | *ε*

defFunc → *funcTyp funcName* **(** *defParam* **)** *stmtBlock*

callFunc → *funcName* **(** *callParam* **)**

4.8 定义预编译语句

这里定义 C++ 语言中部分预编译语句，如 `define` 和 `include` 的部分功能。

precpStmt → **# define** *id* | **# define** *id text* | **# define** *id* **(** *callParam* **)** *text* | **# include** **<** *text* **>** | **# include** *string*

4.9 定义表达式

在该 C++ 语言子集中，定义如下符号优先级：

优先级	算子	结合性
1	变量 常量（数字、字符、字符串）括号	无
2	[] () . -> 尾 ++ 尾--	由左到右
3	头 ++ 头-- + - ～ ! * & sizeof C 风格转型	由右到左
4	* / %	由左到右
5	+ -	由左到右
6	<< >>	由左到右
7	<<= >>=	由左到右
8	== !=	由左到右
9	&	由左到右
10	^	由左到右
11	|	由左到右
12	&&	由左到右
13	||	由左到右
14	= += -= *= /= %= <<= >>= &= ^= |=	由右到左
15	,	由左到右

表 4-1 C++ 语言子集符号优先级

4.9.1 定义第一优先级表达式

表示最基础类型和括号表达式，使用 `basic` 标志。

basicExpr → *var* | *const* | **(** *expr* **)**

4.9.2 定义第二优先级表达式

表示后缀符号表达式，使用 `post` 标志。

ptrOp → **.** | **->**

selfOp → **++** | **--**

postExpr → *basicExpr* | *postExpr* **[** *expr* **]** | *postExpr ptrOp var* | *postExpr ptrOp callFunc* | *postExpr selfOp* | *callFunc*

4.9.3 定义第三优先级表达式

表示前缀符号表达式，使用 `pre` 标志。

preOp → + | - | **!** | ∼ | ***** | **&**

preExpr → *postExpr* | **sizeof** *preExpr* | **sizeof (** *typ* **)** | **(** *typ* **)** *preExpr* | *preOp preExpr* | *selfOp preExpr*

4.9.4 定义第四优先级表达式

表示乘法、除法和求余，使用 `upper` 标志，使用从左到右结合律。

upperOp → ***** | / | **%**

upperExpr → *preExpr* | *upperExpr upperOp preExpr*

4.9.5 定义第五优先级表达式

表示加法和减法，使用 `lower` 标志，使用从左到右结合律。

lowerOp → + | -

lowerExpr → *upperExpr* | *lowerExpr lowerOp upperExpr*

4.9.6 定义第六优先级表达式

表示移位运算，使用 `shift` 标志，使用从左到右结合律。

shiftOp → << | >>

shiftExpr → *lowerExpr* | *shiftExpr shiftOp lowerExpr*

4.9.7 定义第七优先级表达式

表示除判等外的比较运算，使用 `cmp` 标志，使用从左到右结合律。

cmpOp → < | <= | > | >=

cmpExpr → *shiftExpr* | *cmpExpr cmpOp shiftExpr*

4.9.8 定义第八优先级表达式

表示判等运算，使用 `eql` 标志，使用从左到右结合律。

eqlOp → == | **!=**

eqlExpr → *cmpExpr* | *eqlExpr eqlOp cmpExpr*

4.9.9 定义第九优先级表达式

表示位与运算，使用 `and` 标志，使用从左到右结合律。

andExpr → *eqlExpr* | *andExpr* **&** *eqlExpr*

4.9.10 定义第十优先级表达式

表示位异或运算，使用 `xor` 标志，使用从左到右结合律。

xorExpr → *andExpr* | *xorExpr* ∧ *andExpr*

4.9.11 定义第十一优先级表达式

表示位或运算，使用 `or` 标志，使用从左到右结合律。

orExpr → *xorExpr* | *orExpr* | *xorExpr*

4.9.12 定义第十二优先级表达式

表示逻辑与运算，使用 `lgand` 标志，使用从左到右结合律。

lgandExpr → *orExpr* | *lgandExpr* **&&** *orExpr*

4.9.13 定义第十三优先级表达式

表示逻辑或运算，使用 `lgor` 标志，使用从左到右结合律。

lgorExpr → *lgandExpr* | *lgorExpr* || *lgandExpr*

4.9.14 定义第十四优先级表达式

表示赋值运算，使用 `asg` 标志，使用从右到左结合律。

asgOp → = | += | -= | *= | /= | **%**= | <<= | >>= | **&**= | ∧= | |=

asgExpr → *lgorExpr* | *lgorExpr asgOp asgExpr*

4.9.15 定义第十五优先级表达式

表示逗号运算，至此表达式定义完毕，使用从左到右结合律。

expr → *asgExpr* | *expr* , *asgExpr*

4.10 定义标签语句

这里定义 C++ 语言中的标签语句，包括 `goto`、`switch` 的标签。

gotoName → *id*

labelStmt → *gotoName* : *stmt* | **case** *lgorExpr* : *stmt* | **default** : *stmt* | **public** : | **private** : | **protected** :

4.11 定义条件判断语句

这里定义 C++ 语言中的以条件判断进行的语句，包括 `if`、`switch` 和 `goto` 语句。

jdgStmt → **if (** *expr* **)** *stmt* | **if (** *expr* **)** *stmt* **else** *stmt* | **switch (** *expr* **)** *stmt* | **goto** *id* **;**

4.12 定义声明语句

这里定义 C++ 语言中的声明语句，可以单变量或多变量进行声明。

varList → *var* | *varList* **,** *var* | *asgList* **,** *var*

asgList → *var* **=** *expr* | *varList* **,** *var* **=** *expr* | *asgList* **,** *var* **=** *expr*

dclStmt → *typ varList* **;** | *typ asgList* **;**

4.13 定义跳转控制语句

定义 C++ 语言中代码控制与跳转语句。

jmpStmt → **continue ;** | **break ;** | **return** *expr* **;** | **return ;**

4.14 定义循环语句

定义 C++ 语言中 `for`、`while` 和 `do while` 循环。

eptExpr → *expr* | *ε*

loopStmt → **for (***eptExpr* **;** *eptExpr* **;** *eptExpr* **)** *stmt* | **while (***eptExpr* **)** *stmt* | **do** *stmt* **while (***eptExpr* **)**

4.15 定义类与结构体

这里定义 C++ 语言类与结构体，支持其中访问权限控制标签。

className → *id*

classDef → **class** *className stmtBlock* | **struct** *className stmtBlock*

4.16 定义语句

这里定义 C++ 语言语句，其中语句可以为单条语句，也可以是由大括号包裹的复合语句。

eptStmt → *stmt* | *ε*

nobrStmtBlock → *nobrStmtBlock eptStmt* | *eptStmt*

stmtBlock → **{** *nobrStmtBlock* **}** | **{ }**

stmt → *stmtBlock* | *funcDef* | *classDef* | *jmpStmt* | *dclStmt* | *jdgStmt* | *labelStmt* | *cmtStmt* | *precpStmt* | *expr* **;** | **;**

4.17 定义程序

定义了全部子集中的语法项目后，便可以定义整个程序。

program → *stmt program* | *ε*

第五章 人肉编译器

5.1 探究示例一

在示例一中，我们探究（包括但不限于）如下特性：

1. 变量声明
2. 变量定义
3. 变量赋值
4. 变量运算
5. `while` 循环
6. 判断语句
7. `main` 函数书写
8. 输入输出

5.1.1 C++ 语言源码

如下为一个典型的阶乘运算代码，其中使用了许多 C++ 特性：

```cpp
#include <iostream>
using namespace std;
int main() {
    int input, result = 1, index = 1;
    cout << "Tell me the destination: ";
    cin >> input;
    while (index < input) {
        result *= ++index;
    }
    cout << result << endl;
    return 0;
}
```

5.1.2 翻译后的汇编源码

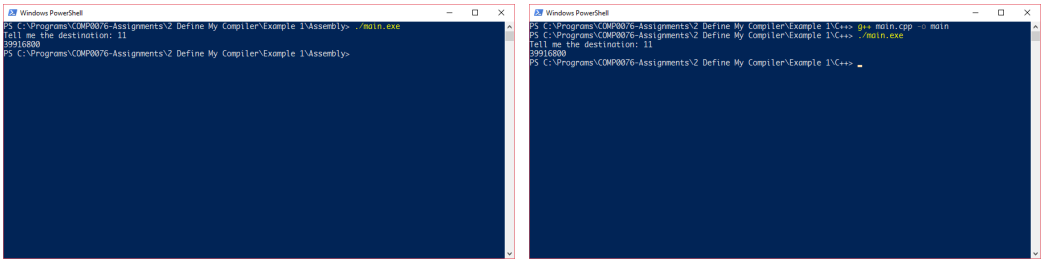
人肉编译该程序后得到如下 `MASM32` 汇编代码：

```
    .486                    ; create 32 bit code
    .model  flat,   stdcall ; 32 bit memory model
    option  casemap :none   ; case sensitive
```

```
4
5       include \masm32\include\windows.inc ; always first
6       include \masm32\macros\macros.asm   ; MASM support macros
7
8     ; -----------------------------------------------
9     ; include files that have MASM format prototypes
10    ; for function calls
11    ; -----------------------------------------------
12      include \masm32\include\masm32.inc
13      include \masm32\include\gdi32.inc
14      include \masm32\include\user32.inc
15      include \masm32\include\kernel32.inc
16      include \masm32\include\msvcrt.inc
17
18    ; -----------------------------------------------
19    ; Library files that have definitions for function
20    ; exports and tested reliable prebuilt code.
21    ; -----------------------------------------------
22      includelib \masm32\lib\masm32.lib
23      includelib \masm32\lib\gdi32.lib
24      includelib \masm32\lib\user32.lib
25      includelib \masm32\lib\kernel32.lib
26      includelib \masm32\lib\msvcrt.lib
27
28      .code       ; Tell MASM where the code starts
29
30  ; «««««««««««««««««««««««««««««««««««««««««««««««««
31
32  start:          ; The CODE entry point to the program
33
34      call main   ; branch to the "main" procedure
35
36      exit
37
38  ; »»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»
39
40  main proc
41
```

```
42      LOCAL   dst     :DWORD
43      LOCAL   result  :DWORD
44      LOCAL   index   :DWORD
45
46      mov     result, 1
47      mov     index,  1
48      mov     dst,    sval(input("Tell me the destination: "))
49
50      mov     eax,    index
51      cmp     eax,    dst
52      jge     label2
53
54    label1:
55      inc     index
56
57      mov     eax,    result
58      imul    eax,    index
59      mov     result, eax
60
61      mov     eax,    index
62      cmp     eax,    dst
63      jl      label1
64
65    label2:
66      print   str$(result)
67      print   chr$(13,10)
68
69      ret
70
71  main endp
72
73  ; «««««««««««««««««««««««««««««««««««««««««««««««««««««
74
75  end start       ; Tell MASM where the program ends
```

5.1.3 运行结果比对



(a) (b)

图 5-1 示例一运行结果对比

5.1.4 翻译过程

汇编程序的翻译基本上经历如下几个阶段：

- 首先需要确定硬件构架，并确定是否大小写敏感。
- 随后需要根据操作中使用到的操作和函数等来确定包括的库和定义等。
- 之后需要插入代码起始端。
- 在开始汇编代码编写后，我们需要将 C++ 中的变量声明定义于汇编中——这里需要注意汇编与 C++ 的关键字并不一样，标准库中的函数名也有可能与我们已经设置的变量有冲突，如 `input` 已经是汇编中的一个函数了，这里我们将其改为 `dst`。随后依次将赋初值和输入操作翻译为汇编码。
- 之后需要进行判断操作，这里需要注意在操作数中至少有一个需要存在于寄存器中，所以我们需要将其中一个数赋给寄存器并进行比较。
- 随后进入循环部分，这里需要注意，在 C++ 中的 `while` 循环在最开始会进行一次判断，如果不符合则跳出，而在汇编中我们需要使用跳转命令来实现循环，所以需要提前进行一次判断。
- 随后进行内部，同样我们需要注意比较时的寄存器问题并依次将运算内容写入。
- 最后我们需要输出运算结果并输出一个回车。

5.2 探究示例二

在示例二中，我们进一步探究（包括但不限于）如下特性：

1. 指针
2. 数组

3. for 循环

5.2.1 C++ 语言源码

如下为一个数组元素遍历输出代码：

```
#include <iostream>
using namespace std;
int main() {
    int temp[5] = {1, 2, 3, 4, 5};
    int *ptr = temp;
    for (int i = 0; i < 5; ++i) {
        cout << *ptr;
        ++ptr;
    }
    return 0;
}
```

5.2.2 翻译后的汇编源码

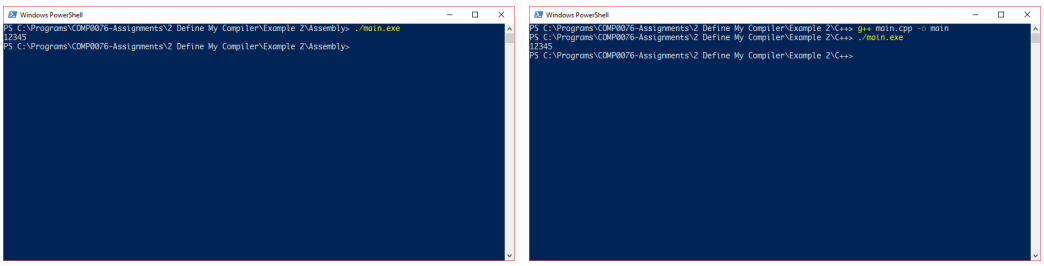
人肉编译该程序后得到如下 MASM32 汇编代码：

```
    .486                    ; create 32 bit code
    .model  flat,   stdcall ; 32 bit memory model
    option  casemap :none   ; case sensitive

    include \masm32\include\windows.inc ; always first
    include \masm32\macros\macros.asm   ; MASM support macros

  ; -----------------------------------------------
  ; include files that have MASM format prototypes
  ; for function calls
  ; -----------------------------------------------
    include \masm32\include\masm32.inc
    include \masm32\include\gdi32.inc
    include \masm32\include\user32.inc
    include \masm32\include\kernel32.inc

  ; -----------------------------------------------
  ; Library files that have definitions for function
  ; exports and tested reliable prebuilt code.
```

```
20    ; ------------------------------------------------
21      includelib \masm32\lib\masm32.lib
22      includelib \masm32\lib\gdi32.lib
23      includelib \masm32\lib\user32.lib
24      includelib \masm32\lib\kernel32.lib
25
26      .data
27
28      num0    dd  1
29      num1    dd  2
30      num2    dd  3
31      num3    dd  4
32      num4    dd  5
33
34      array   dd  num0, num1, num2, num3, num4
35
36      .code       ; Tell MASM where the code starts
37
38  ; «««««««««««««««««««««««««««««««««««««««««««««««««««
39
40  start:          ; The CODE entry point to the program
41
42      call main   ; branch to the "main" procedure
43
44      exit
45
46  ; «««««««««««««««««««««««««««««««««««««««««««««««««««
47
48  main proc
49
50      LOCAL   cnt     :DWORD
51
52      push    esi
53      push    edi
54
55      mov     cnt,    0
56      mov     esi,    array
57
```

```
58    label1:
59
60      mov     edi,    [esi]
61      print   str$(edi)
62      add     esi,    4
63      inc     cnt
64      cmp     cnt,    5
65      jl      label1
66
67      pop     edi
68      pop     esi
69
70      ret
71
72  main endp
73
74  ; «««««««««««««««««««««««««««««««««««««««««««««««««««««««
75
76  end start        ; Tell MASM where the program ends
```

5.2.3 运行结果比对



(a) (b)

图 5-2 示例二运行结果对比

5.2.4 翻译过程

除示例一显示外，该汇编程序在翻译过程中还有如下注意点：

- 可以发现在汇编中由于语法的限制，循环都是通过跳转来进行的，同样的还有 `goto` 和 `switch` 等操作。
- 对数组进行操作需要在知道其长度的情况下。

- 指针在将其指向下一项时需要得到数组内的类型以便进行操作。

第六章 总结与回顾

6.1 编译器的思考

本文以编译器编译流程为主要研究课题，对编译器的思考过程和上下文无关文法的撰写进行了研究，并在研究中产生了自己的思考。

首先我们可以看到在上下文无关文法中有许多许多在平时编程时并没有考虑到的问题，比如符号优先级的思考、终结符的分类、定义与声明的不同等。通过这次作业可以为我们更加方便地完成期末大作业打下了基础，并给了我一些新的思考，比如编程风格对编译速度的影响。在此基础上，我发现通过对指针、函数、类等处的定义的分析，我能更加灵活并快速地使用并掌握 C++ 语言了。

其次通过对 C++ 语言的翻译，我发现实际上编译器做的工作远不只把单条语句翻译成汇编，不仅要把各种高级语法——比如类型定义、循环语句、类继承多态——外，编译器还负责考虑寄存器的分配、多种语法的匹配、链接库的转换等琐碎的工作。通过研究编译器自动生成的汇编代码我发现虽然凌乱但整个过程井井有条，可见编译器的强大。

6.2 作业问题解答

对于作业中的问题，在我的思考后得到了如下不成熟的答案：

编译器的数据结构与算法设计：数据结构需要使用栈来实现上下文无关文法中的递归，为了免去大量回溯，需要使用提前预判等优化算法。

6.3 后续工作展望

在研究过程中发现了很多作者自身值得改善的地方，仍有以下几点值得进一步学习：

1. 汇编语言实在不熟悉，两个汇编程序研究了好几天才写出来并调试通，还需要进一步对其进行训练。
2. 对上下文无关文法不够熟练，对递归定义的理解不到位导致在写作过程中出现许多二义性文法，花了大量时间校对修改，还需进一步训练。

参考文献

[1] A. V. Aho, M. S. Lam, R. Sethi, et al. Compilers: Principles, techniques, and tools (2nd edition)[M]. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006

[2] D. Kusswurm. Modern x86 assembly language programming: 32-bit, 64-bit, sse, and avx[M]. Berkely, CA, USA: Apress, 2014

[3] A. Ferrari. x86 assembly guide[EB/OL]. http://www.cs.virginia.edu/~evans/cs216/guides/x86.html, January 1, 2017