

lab2: 定义你的编译器 & 汇编编程

1811431 王鹏

2020 年 10 月 13 日

目录

1 摘要	3
1.1 摘要	3
2 引言	3
3 实验要求 & 个人解析	3
3.1 实验要求	3
3.2 个人解析 (对作业的理解)	3
4 Part1: 定义你的编译器功能	4
4.1 G++ 支持的 C++ 语言特性	4
4.2 C++ 语言特性子集定义 (后续文法设计的对象)	4
4.3 上下文无关文法	5
4.4 CFG 描述 C++ 语言特性子集 (参考 SysY)	5
4.4.1 定义标识符	5
4.4.2 定义数字 (包括整形、浮点型)	5
4.4.3 定义字符和字符串	6
4.4.4 定义注释	6
4.4.5 定义常量	6
4.4.6 定义变量声明	7
4.4.7 定义常量声明	7
4.4.8 定义运算符	7
4.4.9 定义循环语句	10
4.4.10 定义分支语句	10
4.4.11 定义函数	10
4.4.12 定义类与结构体	11
5 Part2: 汇编编程	11
5.1 示例一: 阶乘	11
5.1.1 C 语言代码	11
5.1.2 人脑汇编代码	11
5.1.3 验证运行结果	13
5.1.4 翻译过程	13
5.2 示例二: 斐波那契数列	13
5.2.1 C 语言代码	13
5.2.2 人脑汇编代码	14
5.2.3 验证运行结果	16
5.2.4 翻译过程	16
6 结论	16

1 摘要

1.1 摘要

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, 主要包括标识符、常量、变量声明、表达式、基本语句、函数定义等方面。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/C++

2 引言

计算机发展初期, 所有程序都是由汇编语言写出来的。这种语言对程序员极不友好, 因此人们发明了高级编程语言。编译器也随之产生并担当了把高级语言翻译成机器语言的工作而高级编程语言都具备一组公共的语法特征, 如标识符、常量、表达式、基本语句、函数定义等, 不同的语言仅在特征的细节上有所区别。上下文无关文法是描述语言语法结构的一组形式规则。

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。

3 实验要求 & 个人解析

3.1 实验要求

1. 设计你的源语言: 你所使用的编译器支持哪些主要的 C(C++) 语言特性? 在此基础上定义你的编译器支持的 C 语言子集——学习教材第 2 章以及第 2 章讲义中的 2.2 节, 用上下文无关文法描述你的 C 语言子集。你应该尽可能参考 SysY 的 C 语言定义、在本学期大作业中实现其中特性。

2. 理解你的目标语言: 对某个 C 程序 (如“预备工作 1”给出的阶乘或斐波那契), 编写等价的 (x86 或 arm 的) 汇编程序, 用汇编器生成可执行程序, 调试通过、能正常运行。这个程序也应该包含尽可能全面的语言特性。

3.2 个人解析 (对作业的理解)

Part1: 这一部分作业的主要要求是了解编译器 g++ 所支持的 C++ 语言特性, 如支持的数据类型 `int, float` 等, 支持变量声明, 赋值语句, 分支语句, 循环, 支持运算, 支持函数, 数组指针等等。从中选取重要的部分 (即子集) 定义编译器功能, 使用上下文无关文法描述选取的 C++ 语言子集。

Part2: 这一部分的作业需要温习基础的汇编知识, 如 `mov, add, jump` 等指令的使用, 手动编写与源 C++ 程序等价的汇编程序

4 Part1: 定义你的编译器功能

4.1 G++ 支持的 C++ 语言特性

- 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、枚举类型、数组类型、自定义类型、类、结构体。
- 支持变量声明、赋值语句、复合语句、if/switch 语句，以及 while/do/for/dowhile 循环。
- 支持函数、指针、引用。
- 支持常量类型、类型修饰符、类型转换
- 支持宏定义、命名空间、运算符重载
- 支持模版、异常
- 支持算术运算（加减乘除按位与或等）、逻辑运算（逻辑与或等）、关系运算（不等等于大于小于等），下图是 C++ 的运算优先级

优先权	运算符	说明	结合性
1	()	括号	由左至右
2	!, -, ++, --	逻辑运算符NOT、算术运算符负号、递增、递减	由右至左
3	*, /, %	算术运算符的乘法、除法、余数	由左至右
4	+, -	算术运算符加法、减法	由左至右
5	<<, >>, >>>	位操作子左移、右移、无符号右移	由左至右
6	>, >=, <, <=	关系运算符大于、大于等于、小于、小于等于	由左至右
7	==, !=	关系运算符等于、不等于	由左至右
8	&	位操作子AND	由左至右
9	^	位操作子XOR	由左至右
10		位操作子OR	由左至右
11	&&	逻辑运算符AND	由左至右
12		逻辑运算符OR	由左至右
13	?:	条件控制运算符	由右至左
14	=, op=	指定运算符	由右至左

图 4.1: C++ 运算符优先级

4.2 C++ 语言特性子集定义 (后续文法设计的对象)

后续会用上下文无关文法描述以下 C++ 语言子集。此部分参考 SysY 的 C 语言定义

- 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
- 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
- 支持函数、类、结构体
- 支持注释、宏定义
- 支持条件分支语句
- 支持单变量或多变量的声明和定义语句
- 支持循环语句
- 支持算法运算符、位运算符、逻辑运算符、赋值运算符

4.3 上下文无关文法

上下文无关文法（**context-free grammar, CFG**）是一种用于描述程序设计语言语法的表示方式，一个上下文无关文法由四个元素组成：终结符号、非终结符号、一个开始符号和一组产生式。

(1) 终结符号是组成串的基本符号。

(2) 非终结符号是表示串的集合的语法变量。非终结符号表示的串集合用于定义由文法生成的语言。非终结符号给出了语言的层次结构，而这种层次结构是语法分析和翻译的关键。

(3) 在一个文法中，某个非终结符号被指定为开始符号，这个符号表示的串集合就是这个文法生成的语言。按照惯例，首先列出开始符号的产生式。

(4) 一个文法的产生式描述了将终结符号和非终结符号组合成串的方法。每个产生式由下列元素组成：

- 一个被称为产生式头或左部的非终结符号。产生式定义这个头所代表串集合的一部分。
- 符号 \rightarrow
- 一个由零个或多个终结符号与非终结符号组成的产生式体或右部。产生式体中的成分描述了产生式头上的非终结符号所对应的串的某种构造方法。

4.4 CFG 描述 C++ 语言特性子集 (参考 SysY)

4.4.1 定义标识符

在这里定义标识符 **id**

```
ndigit → _ (下划线)
        | a|b|...|z(所有小写字母)
        | A|B|...|Z(所有大写字母)
digit  → 0|nozero_digit
id     → ndigit
        | id digit
        | id ndigit
```

其中，**ndigit** 表示字母和下划线，**digit** 表示数字，**id** 表示标识符，标识符由字母 **a-z**, **A-Z** 或下划线开头，后面跟任意（可为 **0**）个字母、下划线或数字。

4.4.2 定义数字 (包括整形、浮点型)

在这里定义全体实数

```
nozero_digit → 1|2|3|4|5|6|7|8|9
const_int   → nozero_digit
            | const_int digit
nozero_digit 表示非零数字，const_int 表示整型常量。
digit_seq    → digit
            | digit_seq digit
```

```

signed_digit_seq → digit_seq
                  | +digit_seq
                  | -digit_seq
const_frac → .digit_seq
            | digit_seq.
            | digit_seq.digit_seq
exponent → e signed_digit_seq
          | E signed_digit_seq
const_float → const_frac
             | const_frac exponent
             | digit_seq exponent

```

digit_seq 表示数字序列，数字序列的开头可以是 **0**，**signed_digit_seq** 是带符号的数字序列，**const_frac** 是小数部分，**3.1**、**.2**、**0.05** 都是符合规范的小数，**exponent** 表示指数部分，指数部分由 **e** 或 **E** 开头，后面跟带符号的数字序列，**const_float** 表示浮点数常量。

4.4.3 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"

```

4.4.4 定义注释

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

```

4.4.5 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

```

constant → const_int
          | const_float
          | const_char | str

```

constant 表示常量，由整数型常量、浮点数常量、字符型常量和字符串常量构成。

4.4.6 定义变量声明

这里定义 **C++** 语言基本四类型、类或结构体类型、**bool** 类型

```
store → auto | register | static | extern | ε
type  → int | float | char | bool | string | className
idlist → id | idlist, id
decl  → store type idlist;
```

其中 **id** 表示标识符, **type** 表示变量类型, **idlist** 表示标识符列表, **decl** 表示变量声明语句。

4.4.7 定义常量声明

其中包括 **const** 关键字和 **define** 宏定义

```
store → auto | register | static | extern | ε
type  → int | float | char | bool | string | className
idlist → id | idlist, id
decl  → store type idlist;
```

其中 **decl** 表示常量声明语句, **const** 是关键字, **id** 表示标识符, **constant** 是上面说明过的常量。

4.4.8 定义运算符

选取常用的运算符, 按照优先级从低到高的顺序, 用上下文无关文法表达出来:

第十五级: 逗号表达式, 用 **expr** 表示, 结合性从左到右:

```
expr → assign_expr
      | expr, assign_expr
```

第十四级: 表示赋值运算, **assign_op** 表示运算符, **assign_expr** 为同级表达式, 从右到左结合。

```
assign_op → = | += | -= | *= | /= | %= | <<= | >>= | &= | ^= | |=
assign_expr → logic_or_expr
             | logic_or_expr assign_op assign_expr
```

第十三级: 表示逻辑或, 用 **logic_or_expr** 表示, 结合性从左到右

```
logic_or_expr → logic_and_expr
              | logic_or_expr && logic_and_expr
```

第十二级： 表示逻辑与，用 `logic_and_expr` 表示，结合性从左到右

```
logic_and_expr → or_expr  
                | logic_and_expr | or_expr
```

第十一级： 表示逐位或，用 `or_expr` 表示，结合性从左到右

```
or_expr → xor_expr  
         | or_expr | xor_expr
```

第十级： 表示逐位异或，用 `xor_expr` 表示，结合性从左到右：

```
xor_expr → and_expr  
          | xor_expr ^ and_expr
```

第九级： 表示逐位与，用 `and_expr` 表示，结合性从左到右

```
and_expr → equal_expr  
          | and_expr & equal_expr
```

第八级： 表示 `=` 和 `≠`，用 `equal_expr` 表示，结合性从左到右

```
equal_expr → relate_expr  
            | equal_expr == relate_expr  
            | equal_expr != relate_expr
```

第七级： 表示 `>`、`<`、`>=`、`<=`，用 `relate_expr` 表示，结合性从左到右

```
relate_expr → shift_expr  
            | relate_expr >= shift_expr  
            | relate_expr <= shift_expr  
            | relate_expr > shift_expr  
            | relate_expr < shift_expr
```

第六级： 表示左移、右移，用 `shift_expr` 表示，结合性从左到右

```

shift_expr → add_expr
            | shift_expr << add_expr
            | shift_expr >> add_expr

```

第五级： 表示加减运算，用 `add_expr` 表示，结合性从左到右

```

add_expr → mul_expr
          | add_expr + mul_expr
          | add_expr - mul_expr

```

第四级： 乘除、取模运算，用 `mul_expr` 表示，结合性从左到右

```

mul_expr → unary_expr
          | mul_expr * unary_expr
          | mul_expr / unary_expr
          | mul_expr % unary_expr

```

第三级： 单目运算、前缀自增自减、强制类型转换、指针求值、间接取址，用 `unary_expr` 表示，结合性从右到左，`unary_op` 表示这一级中的运算符

```

unary_op → + | - | ! | * | & | ~
unary_expr → postfix_expr
            | unary_op unary_expr

```

第二级： 后缀自增自减、函数调用、数组下标、成员访问，用 `postfix_expr` 表示，结合性从左到右

```

ptr_op → . | ->
self_op → ++ | --
postfix_expr → basic_expr | postfix_expr ++ | postfix_expr -- | postfix_expr [expr]
              | postfix_expr (args) | postfix_expr . id | postfix_expr -> id

```

其中，`postfix_expr[expr]` 表示数组下标访问，`args` 表示函数的参数列表，`args` 为空或者由一个逗号表达式 `expr` 组成，`postfix_expr(args)` 表示函数调用，`postfix_expr.id` 是类的成员访问，`postfix_expr id ->` 是指针成员访问。

第一级：基本表达式，用 `basic_expr` 表示

```
basic_expr → id
           | constant
           | (expr)
```

4.4.9 定义循环语句

for 循环：

```
opt_expr → expr
          | ε
stmt → for(opt_expr; opt_expr; opt_expr) stmt
```

while 循环和 **do-while** 语句：

```
stmt → while(expr) stmt
stmt → do stmt while(expr);
```

其中，`stmt` 为语句，`expr` 为表达式。

4.4.10 定义分支语句

if 语句：

```
stmt → if(expr) stmt else stmt
      | if(expr) stmt
```

switch-case 语句：

```
single_case → case(constant): stmt
              | case(constant): stmt break;
cases → single_case
       | case single_case
       | ε
```

4.4.11 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。

```
funcdef → type funcname(paralist) stmt
paralist → paralist, paradeft | paradeft | ε
```

```
paradeft→type id  
funcall→funcname(paralist);
```

其中, **paralist** 表示参数列表, **paradeft** 表示参数声明, **funcname** 表示函数名称, **funcdef** 表示函数声明语句, **funcall** 表示函数调用语句, **id** 是在上面已经表示过的标识符。

4.4.12 定义类与结构体

这里定义 C++ 语言类与结构体

```
className→id  
classdef→class className stmtBlock  
         | struct className stmtBlock
```

5 Part2: 汇编编程

5.1 示例一：阶乘

5.1.1 C 语言代码

```
1  #include<stdio.h>  
2  int main() {  
3      int i,n,f;  
4      scanf("%d", &n);  
5      i=2;  
6      f=1;  
7      while(i<=n){  
8          f=f*i;  
9          i=i+1;  
10     }  
11     printf(" 阶乘结果为: %d\n",f);  
12     return 0;  
13 }
```

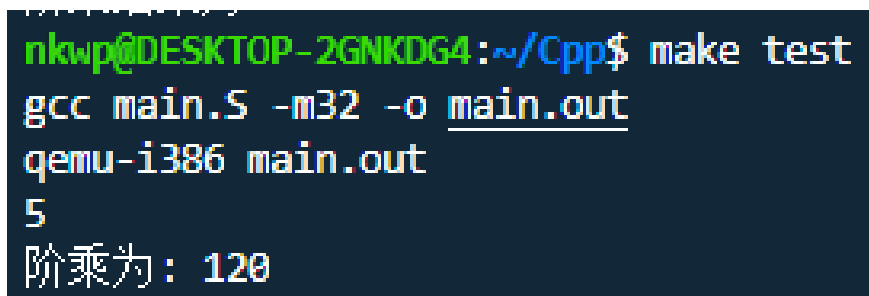
5.1.2 人脑汇编代码

```
1  .data # 全局的未初始化变量存在于.bss 段中, 具体体现为一个占位符;  
2      # 全局的已初始化变量存于.data 段中;  
3  .globl f # 将 f 设为全局变量  
4  .globl i # 将 i 设为全局变量
```

```
5  .align 4 # 令数据地址按 4 对齐
6  .size f, 4    #f 的大小为 4 个字节
7  .size i, 4    #i 的大小为 4 个字节
8  :
9  .long 1 # 给 f 赋初值 1
10 .align 4    # 令数据地址按 4 对齐
11 :
12 .long 1 # 给 i 赋初值 2
13 .align 4    # 令数据地址按 4 对齐
14 .comm n,4   # 未初始化的变量
15 .section    .rodata
16 # rodata 段存储常量
17 STR0:
18 .string "%d"
19 STR1:
20 .string " 阶乘结果为: %d\n"
21 # 主函数
22 .text
23 .globl main
24 .type main, @function
25 main:
26 pushl $n # 从右往左压入参数
27 pushl $.STR0 # 压入参数 STR0
28 call scanf # 调用 scanf 函数
29 addl $8, %esp # 将 n 和 STR0 弹出栈
30
31 jmp L2 # 执行 while 循环
32
33 L3:
34 movl f, %edx # 将 f 的值赋给寄存器 edx
35 Imovl i, %eax # 将 i 的值赋给寄存器 eax
36 imull %edx, %eax # 有符号数乘法指令, 将 f 和 i 相乘
37 movl %eax, f # 相乘的结果赋给 f
38 movl i, %eax # 将 i 的值赋给寄存器 eax
39 addl $1, %eax # 立即数加法, 将 i 的值加 1
40 movl %eax, i # 将 i+1 后的值赋给 i
41
42 L2:
43 movl i, %edx # 将 i 的值赋给寄存器 edx
44 movl n, %eax # 将 n 的值赋给寄存器 eax
45 cmpl %eax, %edx # 将 i 和 n 相减
46 jle L3 # Jump if less or equal 如果小于等于则跳转到 L3
```

```
47
48  movl  f, %eax # 将 f 的值赋给寄存器 eax
49  pushl  %eax # 从右往左压入参数
50  pushl  $.STR1 # 压入常量 STR1
51  call   printf
52  addl   $8, %esp # 将 f 和 STR1 弹出栈
53
54  movl   $0, %eax # return 0
55  ret
56  .section .note.GNU-stack,"",@progbits
```

5.1.3 验证运行结果



```
nkwp@DESKTOP-2GNKDG4:~/Cpp$ make test
gcc main.S -m32 -o main.out
qemu-i386 main.out
5
阶乘为: 120
```

图 5.1: 示例一汇编代码验证

5.1.4 翻译过程

汇编代码中每一句我都写了注释（初次接触汇编语言，真是一把鼻涕一把泪。。。。）

主要依照 `gcc -O0 -o main.S -S -masm=att -m32 -fno-exceptions -fno-asynchronous-unwind-tables -fno-builtin -fno-pie -fverbose-asm main.c` 生成的汇编代码以及实验指导文档中的示例汇编代码撰写

5.2 示例二：斐波那契数列

5.2.1 C 语言代码

```
1  #include<stdio.h>
2  int a=0;
3  int b=1;
4  int i=1;
5  int t;
6  int n;
7  int main(){
8      scanf("%d", &n);
```

```
9     printf("%d\n",a);
10    printf("%d\n",b);
11    while (i < n){
12        t = b;
13        b = a + b;
14        printf("%d\n",b);
15        a = t;
16        i = i + 1;
17    }
18    return 0;
19 }
```

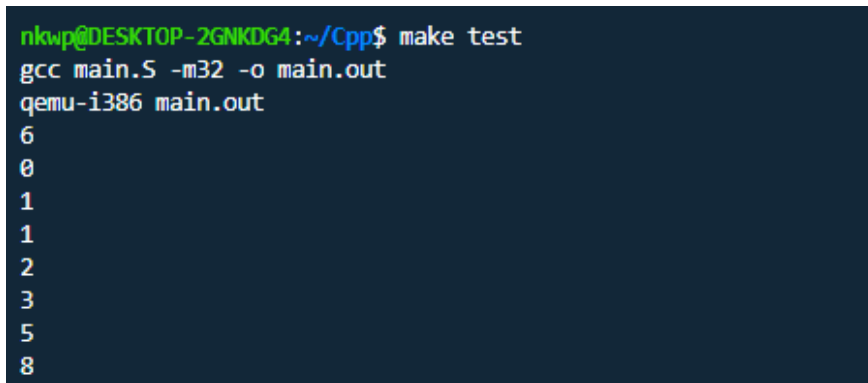
5.2.2 人脑汇编代码

```
1  .data# 全局的已初始化变量存于.data 段中;
2  .globl a # 将 a 设为全局变量
3  .globl b # 将 b 设为全局变量
4  .globl i # 将 i 设为全局变量
5  .align 4 # 令数据地址按 4 对齐
6  .size a, 4 # 大小为 4 个字节
7  .size b, 4 # 大小为 4 个字节
8  .size i, 4 # 大小为 4 个字节
9  :
10 .zero 4 # 给 a 赋初值 0
11 .align 4
12 :
13 .long 1 # 给 b 赋初值 1
14 .align 4 # 令数据地址按 4 对齐
15 :
16 .long 1 # 给 b 赋初值 1
17 .comm t,4 # 未初始化的变量
18 .comm n,4 # 未初始化的变量
19 .section .rodata # rodata 段存储常量
20 STR0:
21 .string "%d"
22 STR1:
23 .string "%d\n"
24 # 主函数
25 .text
26 .globl main
27 .type main, @function
```

```
28 main:
29     pushl    $n    # 从右往左压入参数
30     pushl    $.STR0 # 压入参数 STR0
31     call     scanf  # 调用 scanf 函数
32     addl     $8, %esp # 将 n 和 STR0 弹出栈
33
34     movl     a, %eax # 将 a 的值赋给寄存器 eax
35     pushl    %eax    # 从右往左压入参数
36     pushl    $.STR1  # 压入常量 STR1
37     call     printf  # 调用 printf
38     addl     $8, %esp # 将 a 和 STR1 弹出栈
39
40     movl     b, %eax # 将 b 的值赋给寄存器 eax
41     pushl    %eax    # 从右往左压入参数
42     pushl    $.STR1  # 压入常量 STR1
43     call     printf  # 调用 printf
44     addl     $8, %esp # 将 b 和 STR1 弹出栈
45
46     jmp L2  # 执行 while 循环
47 3:
48     movl     b, %eax # 将 b 的值赋给寄存器 eax
49     movl     %eax, t # 将 b 的值赋给 t
50
51     movl     a, %edx # 将 a 的值赋给寄存器 edx
52     movl     b, %eax # 将 b 的值赋给寄存器 eax
53     addl     %edx, %eax # a 和 b 相加
54
55     movl     %eax, b # a+b 的值赋给 b
56
57     movl     b, %eax # 将 b 的值赋给寄存器 eax
58     pushl    %eax    # 从右往左压入参数
59     pushl    $.STR1  # 压入常量 STR1
60     call     printf  # 调用 printf
61     addl     $8, %esp # 将 b 和 STR1 弹出栈
62
63     movl     t, %eax # 将 t 的值赋给寄存器 eax
64     movl     %eax, a # 将 t 的值赋给 a
65
66     movl     i, %eax # 将 i 的值赋给寄存器 eax
67     addl     $1, %eax # i+1
68     movl     %eax, i # i+1 赋给 i
69 2:
```

```
70  movl    i, %edx # 将 i 的值赋给寄存器 edx
71  movl    n, %eax # 将 n 的值赋给寄存器 eax
72  cmpl    %eax, %edx # 将 i 和 n 相减
73  jl     L3 # Jump if less 如果小于则跳转到 L3
74  movl    $0, %eax # return 0
75  ret
76
77  .section .note.GNU-stack,"",@progbits
```

5.2.3 验证运行结果



```
nkwp@DESKTOP-2GNKDG4:~/Cpp$ make test
gcc main.S -m32 -o main.out
qemu-i386 main.out
6
0
1
1
2
3
5
8
```

图 5.2: 示例二汇编代码验证

5.2.4 翻译过程

有了示例一的训练，感觉斐波那契数列的汇编程序还算上手了，编写耗费的时间也不是很长。。。

仍然是主要依照 `gcc -O0 -o main.S -S -masm=att -m32 -fno-exceptions -fno-asynchronous-unwind-tables -fno-builtin -fno-pie -fverbose-asm main.c` 生成的汇编代码以及实验指导文档中的示例汇编代码撰写。写到这里，对汇编代码中的变量声明、变量定义、变量赋值、变量运算、`while` 循环、判断语句、`main` 函数书写、输入输出都有了一定的了解

6 结论

编译器要有能力翻译所有合法的 C++ 程序。但穷举所有 C++ 程序是不可能的，因此我们需要提炼语言特性，用符号化的方法进行翻译。

在上下文无关文法中有许许多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。

另外，分别编写阶乘和斐波那契数列的 C++ 程序的汇编程序，并用 32 位处理器 i386 生成可执行程序，调试运行。这次编写的程序比较简单，主要是对跳转语句的使用，没有过多地考虑寄存器分配、数据溢出等问题，实际的汇编编程中会有更多要考虑的因素。

个人感悟 写到这里是个漫长曲折的过程。我不禁感叹道：在伟大的编译器面前，我要学会谦卑。

参考文献

- [1] https://zh.cppreference.com/w/cpp/language/operator_precedence
- [2] Alfred V.Aho 等. 编译原理 [M]. 北京：机械工业出版社，2014. 114-115.
- [3] 编译原理 [M]. 机械工业出版社，(美) 阿霍 (Alfred, 2009
- [4] D. Kusswurm. Modern x86 assembly language programming: 32-bit, 64-bit, sse, and avx[M]. Berkely, CA, USA: Apress, 2014
- [5] <http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>