南開大學
Nankai University

南　开　大　学

网 络 空 间 安 全 学 院

编译原理实验报告

第一次作业

段国豪 1911407

年级：2019 级

专业：信息安全、法学

指导教师：王刚

2021 年 9 月 25 日

摘要

本次实验是为了探究语言处理系统的完整工作过程。语言处理系统包括预处理器，编译器，汇编器，链接器。在本次实验中，笔者将分别探究语言处理系统各部分的功能，从而能更好地掌握语言处理系统的工作过程，对本门课程也有望在实验过程中明确学习目标，起到事半功倍的效果。

关键字：预处理器，编译器，汇编器，链接器

目录

一、 概述

本次实验中笔者使用 Ubuntu20.04 操作系统，更换系统配置为阿里源，使用了 gcc、clang、meld、llvm 等工具。下面笔者将简单介绍这几种工具的用途。[1]

1. gcc。GCC 是 GNU 系统的官方编译器（包括 GNU/Linux 家族），在 LLVM、Clang 崛起之前，它也是编译与创建其他操作系统的主要编译器。GCC 通常是跨平台软件的编译器首选。有别于一般局限于特定系统与运行环境的编译器，GCC 在所有平台上都使用同一个前端处理程序，产生一样的中介码，因此此中介码在各个其他平台上使用 GCC 编译，有很大的机会可得到正确无误的输出程序。本次实验中，gcc 将贯穿于实验全过程，主要是实现分阶段结果输出和最终生成可执行程序。

2. meld。Meld 是 GNOME 的可视化 diff 和 merge 工具，其主要针对开发人员。它允许用户直观地比较两个或三个文件或目录，并对不同的行进行颜色编码。Meld 允许你比较文件、目录和版本控制仓库。它提供了文件和目录的双向和三方比较，并支持许多版本控制系统，包括 Git、Mercurial、Baazar、CVS 和 Subversion。Meld 是受 GNU 通用公共许可协议（GPL）条款约束的自由开源软件。本次实验中，meld 主要用来比较编译器在代码优化阶段前后生成的代码，用来观察代码是否进行优化以及进行了什么优化。

3. clang。Clang 是一个 C、C++、Objective-C 和 Objective-C++ 编程语言的编译器前端。它采用了 LLVM 作为其后端，由 LLVM2.6 开始，一起发布新版本。它的目标是提供一个 GNU 编译器套装（GCC）的替代品，支持了 GNU 编译器大多数的编译设置以及非官方语言的扩展。

4. llvm。LLVM 是一套编译器基础设施项目，为自由软件，以 C++ 写成，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。它是为了任意一种编程语言而写成的程序，利用虚拟技术创造出编译时期、链接时期、运行时期以及 “闲置时期” 的优化。它最早以 C/C++ 为实现对象，而目前它已支持包括 ActionScript、Ada、D 语言、Fortran、GLSL、Haskell、Java 字节码、Objective-C、Swift、Python、Ruby、Crystal、Rust、Scala 以及 C 等语言。本实验中 clang 于 llvm 工具主要使用在编译器分析阶段。

本实验的研究思路是创建若干个简单的 C 源代码，通过上述工具，利用不同的指令，获得源代码在语言处理系统中各阶段的输出，研究各阶段输出结果，结合理论分析，从而探究预处理器、编译器、汇编器、链接器的作用，最终尝试总结出语言处理系统的工作过程。

我们大部分功能的测试都将通过 main.c 代码实现，源代码展示如下。

```
#include<stdio.h>
int main()
{
        int i, n, f;
        scanf("%d",&n);
        i = 5;
        f = 6;
        while (i <= n)
        {
                f = f * i;
                i = i + 1;
```

[1]所有工具相关资料均来自维基百科相关词条

```
        }
        printf("%d",f);
        return 0;
}
```

笔者选取这个代码，有以下原因。

1. 代码结构比较简单，输出的结果代码体量较小，便于分析
2. 代码实现的逻辑功能比较全面，包括变量声明、输入输出、循环等功能，使得分析结果不会过于片面
3. 取值比较特别，通过后面的分析反馈，笔者发现数字 1-4 的赋值都容易和寄存器标号混淆，不易分析，在此处将取值改为 5 和 6

二、 预处理器

通过资料查询，预处理器是程序中处理输入数据，产生能用来输入到其他程序的数据的程序。在语言处理系统中，输出被称为输入数据预处理过的形式，常用在之后的程序比如编译器中。一般的预处理器可以执行宏处理、文件包含、“理性”预处理器[2]和语言扩充等功能，高级的预处理器有着完全成熟的编程语言的能力。[3]

运行以下指令，系统将对 main.c 进行预处理，输出结果 main.i，笔者简单观察了 main.i 的代码内容后，选取部分代码进行分析。[4]

```
gcc main.c -E -o main.i
```

下面笔者列出 main.i 中高度相似且高频出现的语句并进行分析。

```
# 1 "/usr/include/stdio.h" 1 3 4
typedef signed char __int8_t;
extern int getw (FILE *__stream);
```

容易发现，第一行代码的作用是导入头文件，实际上，在 main.i 中出现了大量类似语句，但是笔者在源代码中只写了一个 stdio.h 文件，于是笔者打开 stdio.h，发现这个文件中有类似于下面的代码。

```
#include <bits/sys_errlist.h>
```

于是笔者猜测 main.i 中也导入了这个头文件，因此体量庞大，打开 main.i 进行代码查询，果然找到了相同的存在。

```
"/usr/include/x86_64-linux-gnu/bits/sys_errlist.h" 1 3 4
```

从第一行代码我们可以知道预处理器根据源代码导入的文件不止有源文件中出现的头文件，还有出现在出现过的头文件里的头文件。后面两行代码很明显是出现在被导入的头文件里的宏定义，一类是类型定义，另一类是方法定义。这些类型名和方法名组成了语言处理系统分析源代码的词库。由于预处理器未对主代码进行分析，因此它将主代码保留于 main.i 中，同词典一起，由编译器进行分析处理。

[2]这些处理器能把现代控制流和数据结构化机制添加到比较老式的语言中

[3]参考维基百科“预处理器”词条

[4]全部代码多达 900 行，并且多数代码高度相似，因此不在此处一一展示分析

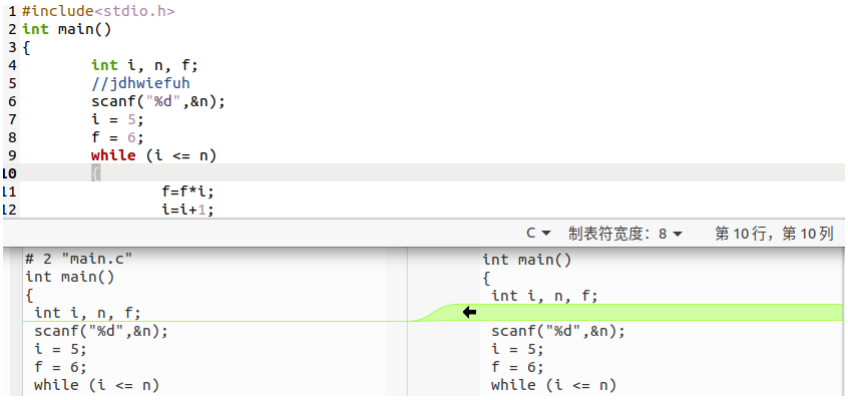

图 1: 注释前后输出代码比较

将 main.c 进行改变，在任意添加任意注释，进行预处理，得到 main1.i，通过 meld 工具比较 main.i 和 main1.i，分析结果如图1。

容易观察到两个文件内容一样，而右半部分空出一行，恰好是笔者添加注释的位置，说明预处理器会忽略所有注释。

于是，通过实验我们验证预处理器的宏处理、文件包含、忽略所有注释等功能。

三、 编译器

编译器的功能是将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。在课程学习中我们知道编译过程分为 6 个阶段，分别是词法分析、语法分析、语义分析、中间代码、代码优化、汇编代码生成，下面笔者将通过分析源代码在各个阶段的输出结果来探究编译器的功能。

(一) 词法分析

词法分析是将字符序列转换为标记序列的过程。预处理器为我们提供了大量的语料，使得我们可以将源代码分解为单词序列。通过运行如下指令我们可以得到词法分析结果如图2。

```
clang -E -Xclang -dump-tokens main.c
```

在输出结果中，笔者发现有大量相似句式。

```
long 'long' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/
    include/stddef.h:46:9 <Spelling=<built-in>:79:23>>
identifier 'size_t' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang
    /10.0.0/include/stddef.h:46:23>
semi ';' Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef
    .h:46:29>
```

每一行前面就是我们的单词，后面就是单词的来源。分析完成后我们可以便可以得到单词序列，下面我们展示分析的结尾部分。

```
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:13:2>
l_paren '(' Loc=<main.c:13:8>
string_literal '"%d"' Loc=<main.c:13:9>
comma ',' Loc=<main.c:13:13>
```

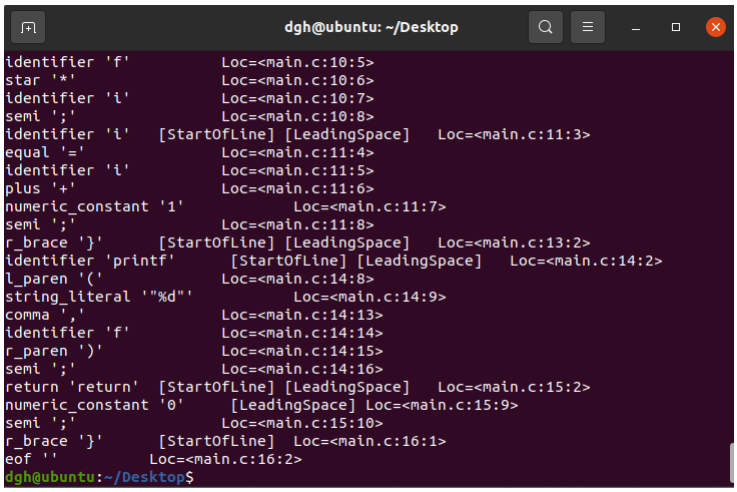

图 2: 词法分析结果部分代码

```
identifier 'f' Loc=<main.c:13:14>
r_paren ')' Loc=<main.c:13:15>
semi ';' Loc=<main.c:13:16>
return 'return' [StartOfLine] [LeadingSpace] Loc=<main.c:14:2>
numeric_constant '0' [LeadingSpace] Loc=<main.c:14:9>
semi ';' Loc=<main.c:14:10>
r_brace '}' [StartOfLine] Loc=<main.c:15:1>
eof '' Loc=<main.c:15:2>
```

很明显这些单词是 printf(”%d”,f);return 0; 由此可知除去空格的所有单词都已经被进行分析。为了进一步论证，笔者将代码片段中的标识符 f 改为 inti, 因为我们源代码里有 “int i” 的声明，笔者试图将两者进行混淆，观察词法分析是否会出现错误。改变后对”printf(”%d”,inti)” 的分析结果如下。

```
identifier 'inti' Loc=<main.c:13:14>
```

可见，词法分析会将字符串拆分为有意义的词，但如果仅止步于此，那么分析为”int i” 与 inti 都将是合理形式，并且笔者认为根据语法分析器的拆分规则，前者更应作为分析结果，但此处仅出现了 inti，说明词法分析在一定程度上也考虑了语法结构而进行了较为合理的词的拆分。

(二) 语法分析

语法分析是根据某种给定的形式文法对由单词序列构成的输入文本进行分析并确定其语法结构的一种过程。语法分析器可以帮助程序构造语法分析树。[5]llvm 可以通过如下指令获得语法分析树。

```
clang -E -Xclang -ast-dump main.c
```

笔者从输出结果中找到两句代码进行分析，来探究 llvm 工具生成语法分析树的结构。

```
c0 'printf' 'int (const char *, ...)'
    | |-ImplicitCastExpr 0xa9e1e8 <col:9> 'const char *' <NoOp>
```

[5]参考编译原理第一版

```
3 |   | | -ImplicitCastExpr 0xa9e1d0 <col:9> 'char *' <ArrayToPointerDecay>| |
      -StringLiteral 0xa9e130 <col:9> 'char [3]' lvalue "%d"
4 |   | -ImplicitCastExpr 0xa9e200 <col:14> 'int' <LValueToRValue>|
      -DeclRefExpr 0xa9e150 <col:14> 'int' lvalue Var 0xa9d7f8 'f' 'int'
5 |   -ReturnStmt 0xa9e238 <line:14:2, col:9>-IntegerLiteral 0xa9e218 <col:9> 'int' 0
```

在分析过程中笔者发现这些代码中有很多类型名称，于是笔者在 clang 说明文档中找到这些名称的含义，结果如下。

```
1 ImplicitCastExpr - Allows us to explicitly represent implicit type
    conversions, which have no direct representation in the original source
    code. In C, implicit casts always produce rvalues. 表示隐式强制转换，在C
    语言中，总是强制转换为左值。
2 StringLiteral - This represents a string literal expression.是一种字符序列的
    表达。
3 DeclRefExpr - A reference to a declared variable, function, enum, etc.对已声
    明变量、函数、枚举等的引用。
```

由此，可以知道了第一句话分析的是 printf("%d",n)。我们知道，printf 是格式化输出符号，会识别后面的单词来决定输出格式，下一个单词是%d，代表的是 int 格式，接着识别下一个单词来决定输出目标，下一个单词是 f，这里会对 f 进行隐式强制转换为 int 进行输出。

第二句话是返回声明，由于主函数声明是 “int main”，因此，返回值类型是 int，后面的单词是我们的返回值 0。

(三) 语义分析

在语言学中，语义分析是将句法结构联系起来的过程，从短语、从句、句子和段落的层次到作为一个整体的写作层次，再到它们独立于语言的意义。在分析过程中，会考虑习语、比喻等特定语言和文化背景。在计算机科学中，笔者认为语义分析的定义可以由语言学演化而来，语义分析就是核查语法分析中生成的语法分析树是否符合逻辑，比如检查类型是否匹配，是否具有合法结构，同时也会考虑特定语言背景，比如说变量声明为 “int=int+float” 时 float 会被强制转换为 int，结构依然合法。

(四) 中间代码

编译器经过词法分析、语法分析、语义分析后，源程序的代码格式便会得到规范，至少从表面上看代码的内容都是合法的，接下来编译器生成中间代码——源代码的低级或类机器语言的中间表示。通过如下指令，我们可以获得中间代码的多阶段的输出，直观地看到中间代码的生成逻辑。

```
1 gcc -O0 -fdump-tree-all-graph main.c
```

生成了图3中的多个逻辑分析文件，分析结果可以在 vscode 中以图像形式直观表示，由于代码逻辑较为简单，多阶段的输出文件内容相同，如图4所示。

于是我们看到了中间代码的运行逻辑，下面我们对中间代码进行解读，通过 clang 和 llvm 工具，我们使用下面指令得到中间代码 (llvm IR)main.ll。

```
1 clang -S -emit-llvm main.c
```

这里笔者依然截取部分代码进行分析，主要是观察中间代码的逻辑结构，以此对中间代码有更深一步地了解。

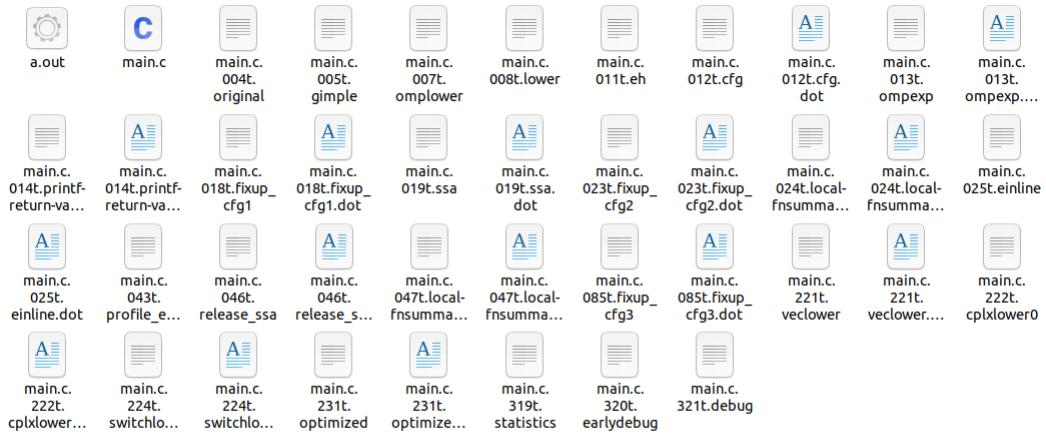

图 3: 生成文件

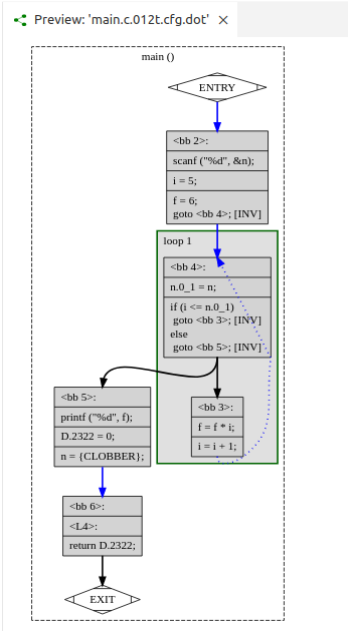

图 4: 代码逻辑图

```
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %5 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x
      i8], [3 x i8]* @.str, i64 0, i64 0), i32* %3)
  store i32 5, i32* %2, align 4
  store i32 6, i32* %4, align 4
  br label %6

6:                                                ; preds = %10, %0
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %3, align 4
  %9 = icmp sle i32 %7, %8
  br i1 %9, label %10, label %16

10:                                               ; preds = %6
  %11 = load i32, i32* %4, align 4
  %12 = load i32, i32* %2, align 4
  %13 = mul nsw i32 %11, %12
  store i32 %13, i32* %4, align 4
  %14 = load i32, i32* %2, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %2, align 4
  br label %6

16:                                               ; preds = %6
  %17 = load i32, i32* %4, align 4
  %18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3
      x i8]* @.str, i64 0, i64 0), i32 %17)
  ret i32 0
```

查阅llvm 文档可以知道。

- % 代表局部变量。
- alloca 指令用于分配内存给局部变量, 函数生命周期结束时自动释放。
- i 代表位数，i32 代表占 32 位。
- label% 代表标签，代表目的函数地址。
- store、load 代表存储、加载数据。

分析得出%2,%4 分别代表 i 和 f，%6 代表循环语句判断，%10 代表循环成立分支，%16 代表结束循环。由此可见中间代码的逻辑和图4相同。

（五） 代码优化

完成以上工作，编译器会对代码逻辑结构进行优化，来提高程序的运行速度，生成更好的目标代码。

为了是创造更大的代码优化空间，获得更明显的实验效果，笔者将主代码进行改变，但是改变只应用于对这一步的分析，到” 汇编代码生成” 模块以后仍然使用原来的主代码。

使用如下指令可以对中间代码进行优化，并将优化结果存储于 main1.ll 中。

```
opt -<module> -S main.ll -o main1.ll
```

llvm 为我们提供了多种优化策略，这些策略即是上述指令中的 <module>，我们可以在llvm中的优化 pass中进行探索，获得自己想要分析的优化策略。

这里笔者整理了一些常见的优化策略。

- adce：激进的死代码删除。
- bb-vectorize：基本块向量化。
- constprop：常量传播优化。
- dce：死代码消除。
- globaldce：全局死代码消除。
- deadargelim：死变量消除。
- globalopt：全局优化。
- inline: 内联函数优化。
- lowerswitch: 降低 switch 语句，转化成分支语句。
- simplifycfg: 控制流简化。
- tailcallelim: 尾调用消除。

当笔者将主代码改变如下时，可以分析出来”b” 是一个无用的宏定义，使用”-globaldce” 进行优化，meld 工具比较 main.ll 和 main1.11，得到图5结果。

```
#include<stdio.h>
const int a=3;
#define b 3
int main()
{
  int i, n, f;
  i = 5;
  f = 6;
  n=0;
  f=i*a;
  printf("%d",f);
  return 0;
}
```

我们发现优化效果并不明显，于是笔者将代码和优化策略再次进行改变。

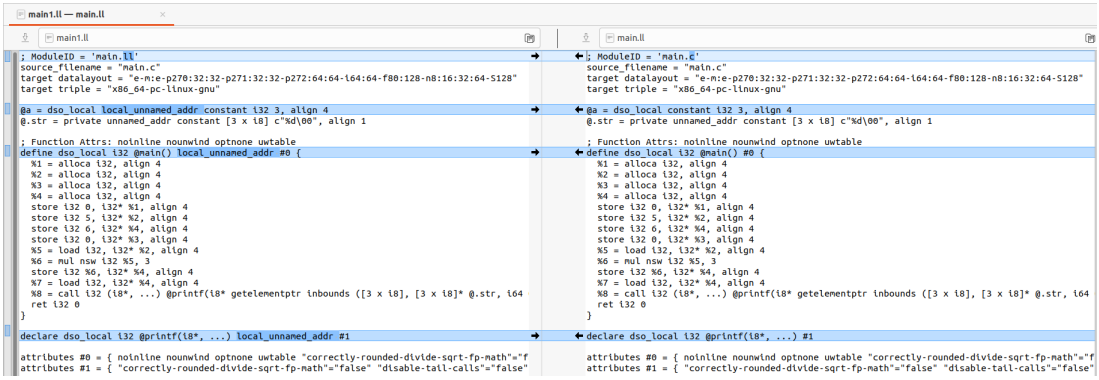

图 5: 第一种优化策略

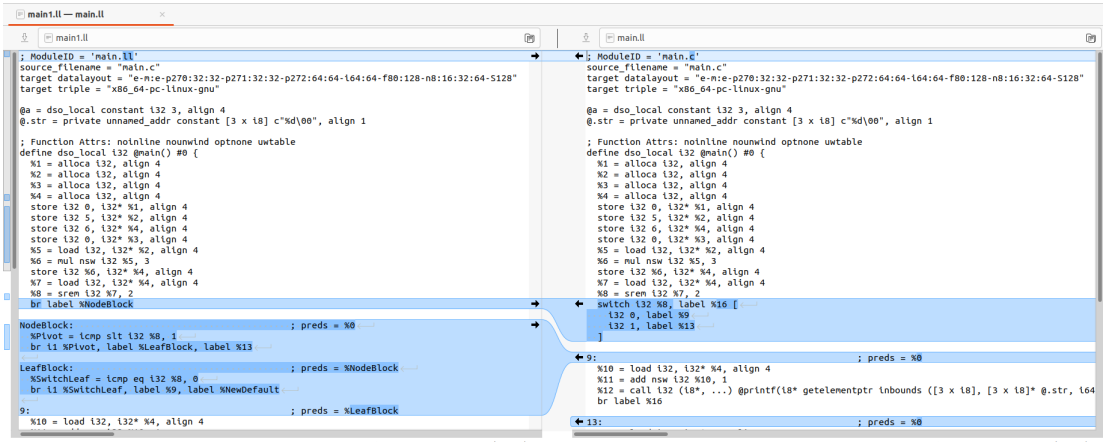

图 6: 第二种优化策略

```
优化策略: opt -lowerswitch -S main.ll -o main1.ll
#include<stdio.h>
const int a=3;
#define b 3
int main()
{
        int i, n, f;
        i = 5;
        f = 6;
        n=0;
        f=i*a;
        switch(f%2){
        case 0:printf("%d",f+1);
        break;
        case 1:printf("%d",f);
        break;}
        return 0;
}
```

这一次比较结果如图6所示。

可以看到出现了 main1.ll 出现了明显变化，展现出来了优化器的活动。中间代码生成以后，优化器可以根据内置的优化策略对中间代码的逻辑进行优化。

（六） 汇编代码生成

经过逻辑优化的中间代码通过如下指令会转变为汇编代码 main.S，至此编译工作完成。

```
gcc main.i -S -o main.S
```

main.S:

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq       %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq        %rsp, %rbp
    .cfi_def_cfa_register 6
    subq        $32, %rsp
    movq        %fs:40, %rax
    movq        %rax, -8(%rbp)
    xorl        %eax, %eax
    leaq        -20(%rbp), %rax
    movq        %rax, %rsi
    leaq        .LC0(%rip), %rdi
    movl        $0, %eax
    call        __isoc99_scanf@PLT
    movl        $5, -16(%rbp)
    movl        $6, -12(%rbp)
    jmp .L2
.L3:
    movl        -12(%rbp), %eax
    imull       -16(%rbp), %eax
    movl        %eax, -12(%rbp)
    addl        $1, -16(%rbp)
.L2:
    movl        -20(%rbp), %eax
    cmpl        %eax, -16(%rbp)
    jle .L3
    movl        -12(%rbp), %eax
    movl        %eax, %esi
    leaq        .LC0(%rip), %rdi
    movl        $0, %eax
    call        printf@PLT
    movl        $0, %eax
    movq        -8(%rbp), %rdx
    xorq        %fs:40, %rdx
    je  .L5
```

```
    call    __stack_chk_fail@PLT
.L5:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

四、 汇编器

汇编器的功能是将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。使用汇编语言编写的源代码，然后通过相应的汇编程序将它们转换成可执行的机器代码，这一过程被称为汇编过程。实验中我们可以使用如下指令将汇编代码转化为机器码。

```
gcc main.S -c -o main.o
```

使用如下指令，便可以查看生成的.o 文件，同时在代码右侧看到机器码所对应的汇编代码。

```
objdump -d main.o > main2.S
objdump实用程序读取二进制或可执行文件，并将汇编语言指令转储到屏幕上。
```

```
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  13:   00 00
  15:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  19:   31 c0                   xor    %eax,%eax
  1b:   48 8d 45 ec             lea    -0x14(%rbp),%rax
  1f:   48 89 c6                mov    %rax,%rsi
  22:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 29 <main+0x29>
  29:   b8 00 00 00 00          mov    $0x0,%eax
  2e:   e8 00 00 00 00          callq  33 <main+0x33>
  33:   c7 45 f0 05 00 00 00    movl   $0x5,-0x10(%rbp)
  3a:   c7 45 f4 06 00 00 00    movl   $0x6,-0xc(%rbp)
  41:   eb 0e                   jmp    51 <main+0x51>
  43:   8b 45 f4                mov    -0xc(%rbp),%eax
  46:   0f af 45 f0             imul   -0x10(%rbp),%eax
  4a:   89 45 f4                mov    %eax,-0xc(%rbp)
  4d:   83 45 f0 01             addl   $0x1,-0x10(%rbp)
  51:   8b 45 ec                mov    -0x14(%rbp),%eax
  54:   39 45 f0                cmp    %eax,-0x10(%rbp)
  57:   7e ea                   jle    43 <main+0x43>
  59:   8b 45 f4                mov    -0xc(%rbp),%eax
  5c:   89 c6                   mov    %eax,%esi
  5e:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 65 <main+0x65>
  65:   b8 00 00 00 00          mov    $0x0,%eax
```

```
29  6a:   e8 00 00 00 00          callq  6f <main+0x6f>
30  6f:   b8 00 00 00 00          mov    $0x0,%eax
31  74:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
32  78:   64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
33  7f:   00 00
34  81:   74 05                   je     88 <main+0x88>
35  83:   e8 00 00 00 00          callq  88 <main+0x88>
36  88:   c9                      leaveq
37  89:   c3                      retq
```

五、 链接器

链接器的功能是将可重定位的机器代码和相应的一些目标文件以及库文件连接在一起，形成真正能在机器上运行的目标机器代码。我们可以使用如下指令对汇编指令进行链接。

```
1  gcc main.o -o main
```

链接完成以后就完成了从源代码到可执行程序的过程。对程序进行简单测试，结果如表1。

测试样例	1	5	8
理论值	6	30	10080
实际值	6	30	10080

表 1: 测试结果

程序运行成功，源代码处理正确。

六、 总结

通过以上分析，笔者更加深入地了解语言处理系统的工作过程。这个过程可以概括为源代码在预处理器、编译器、汇编器和链接器的依次处理下变成可执行文件的过程。

在预处理过程中，预处理器会进行文件包含工作，文件包括我们引用的头文件以及在头文件中出现的其他头文件，将头文件和我们的宏定义存入词典库，删除所有注释，得到一个只有宏定义和主代码的文件，然后交给编译器处理。

编译器首先会根据词典库将主代码拆解为单词并且每个单词都有意义，这中间也会进行一些抉择来把主代码拆解成合适的字符序列，比如 inti、int、i 都有宏定义，inti 拆分结果就是 inti，而非 int 和 i，然后在语法分析中根据展开式规则生成语法分析树，在语义分析中验证语法分析树结构是否合理，接着将逻辑结构输出为中间代码，进行逻辑优化后转变为汇编代码。

汇编器将汇编代码转变为机器码，机器码通过链接器与链接库函数链接从而转变为可执行程序。

这样，源代码通过语言处理系统最终转变为可执行程序。