网络空间安全学院

编译系统原理实验报告

了解编译器及 LLVM IR 编程

姓名：王天鹏

学号：2013013

专业：信息安全与法学双学位班

2022 年 9 月

目录

1 实验背景介绍

1.1 问题描述

本次实验的基本研究对象为编译器的基本构成和工作原理。笔者以一个简单的 C 语言竭诚的源程序为例，通过调整编译器的程序选项获得各阶段的输出，研究它们与源程序的关系，从而进一步了解一个完整的编译过程，熟悉预处理器、编译器、汇编器、链接器在将程序翻译成汇编语言再到二进制文件过程中各自的分工。本次实验所使用的阶乘程序的 C 语言源代码如下：

```c
#include <stdio.h>
#define N 200
   #ifndef N
     scanf("%d", &n);
 #else
 #pragma message ("N is defined\n")
 #endif
 int main()
 {
     int i, n, f;
     scanf("%d", &i);
     i = 2;
     f = 1;
     while (i <= n)
     {
         f = f * i;
         i = i + 1;
     }
     printf("f= %d\n", f);
     return 0;
 }

```

1.2 实验环境

在本次实验中，笔者使用了 Linux 云服务器，并通过 xshell 进行远程连接，使用的软件版本如下

1. 操作系统：Linux VM-4-12-centos 3.10.0-1160.66.1.el7.x86_64 #1 SMP x86_64 GNU/Linux
2. GCC 编译器：gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
3. CLANG 前端：clang version 3.4.2 (tags/RELEASE_34/dot2-final)
4. LLVM 后端：14.0.0
5. GCC-ARM 编译器：arm-linux-gnueabihf-gcc 11.2.0
6. QEMU-ARM 模拟器：6.2.0

2 预处理器做了什么？

当源程序被编译器处理后，第一个环节是预处理器先处理。预处理器的工作主要是处理源代码中以 开始的预编译指令,例如展开所有宏定义、插入 `include` 指向的文件等,以获得经过预处理的源程序。通过命令 g 始的预编译指令，以及所有宏定义进行插入 include 指向的文件和宏定义语句。其中预处理后文件 main.i 文件有几千行之多，下列仅对 main.i 文件进行了部分展示，省略了大篇幅的对于 stdio.h 头文件的引用部分的展示。

```


 # 1 "main.c"
1 "<built-in>"
1 "<command-line>"
1 "/usr/include/stdc-predef.h" 1 3 4
1 "<command-line>" 2
1 "main.c"
1 "/usr/include/stdio.h" 1 3 4
27 "/usr/include/stdio.h" 3 4
1 "/usr/include/features.h" 1 3 4
375 "/usr/include/features.h" 3 4
1 "/usr/include/sys/cdefs.h" 1 3 4
392 "/usr/include/sys/cdefs.h" 3 4
1 "/usr/include/bits/wordsize.h" 1 3 4
393 "/usr/include/sys/cdefs.h" 2 3 4
376 "/usr/include/features.h" 2 3 4
399 "/usr/include/features.h" 3 4
1 "/usr/include/gnu/stubs.h" 1 3 4
10 "/usr/include/gnu/stubs.h" 3 4
1 "/usr/include/gnu/stubs-64.h" 1 3 4
11 "/usr/include/gnu/stubs.h" 2 3 4
400 "/usr/include/features.h" 2 3 4
28 "/usr/include/stdio.h" 2 3 4
此处省略了对于 stdio.h 头文件的替换部分。

  int main()
  {
      int i, n, f;
      scanf("%d", &i);
      i = 2;
      f = 1;
      while (i <= n)
      {
          f = f * i;
          i = i + 1;
      }
      printf("f= %d\n", f);
      return 0;
  }

```

如上述代码，可以看到，

- 预处理器对源程序中的宏定义N了替换，将N替换为了常量 200。
- 源程序中的注释在这一阶段被删除了。
- 条件编译语句如#idndef、#else被预处理器进行删除替换。
- 预处理器添加了行号和文件名等内容。
- 值得注意的是，gcc 的预处理器并未对于#pragma once进行处理。这是由于在 C/C++ 中，#pragma once是一个非标准但是被广泛支持的方式。#pragma once方式产生于#ifndef之后。#ifndef方式受 C/C++ 语言标准的支持，不受编译器的任何限制；而#pragma once方式有些编译器不支持，兼容性不够好，本次实验的 gcc 编译器预处理器对于#pragma once未作处理。#ifndef可以针对一个文件中的部分代码，而#pragma once只能针对整个文件。相对而言，#ifndef更加灵活，兼容性好，#pragma once操作简单，效率高。

3 编译器做了什么？

3.1 词法分析

经过预处理器处理后，编译器会对预处理后的 main.i 进行词法分析。通过lang -E -Xlang -dump-tokens main.c 选项，编译器会将源程序转换为单词序列，因此我们可以获得文本格式的 token 序列，阶乘程序的部分 tokens 序列如下所示：

```
  typedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:34:1>
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:34:9 <Spelling=<
t 'int'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:34:9 <Spelling=<bu
entifier 'ptrdiff_t'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:34:26
mi ';'^^I^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:34:35>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:1>
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:9 <Spelling=<
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:9 <Sp
t 'int'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:9 <Spelling=<bu
entifier 'size_t'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:23>
mi ';'^^I^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:42:29>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:65:1>
```

```
t 'int'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:65:9 <Spelling=<bu
entifier '__wchar_t'^^I [LeadingSpace]^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:65:24>
mi ';'^^I^^ILoc=</usr/bin/../lib/clang/3.4.2/include/stddef.h:65:31>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:30:1>
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:30:9>
ar 'char'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:30:18>
entifier '__u_char'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:30:23>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:30:31>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:31:1>
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:31:9>
ort 'short'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:31:18>
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:31:24>
entifier '__u_short'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:31:28>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:31:37>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:32:1>
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:32:9>
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:32:18>
entifier '__u_int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:32:22>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:32:29>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:33:1>
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:33:9>
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:33:18>
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:33:23>
entifier '__u_long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:33:27>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:33:35>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:36:1>
gned 'signed'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:36:9>
ar 'char'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:36:16>
entifier '__int8_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:36:21>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:36:29>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:37:1>
pelling=/usr/include/bits/types.h:103:35>>
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:145:12 <Spelling=/usr/include/bits/ty
entifier '__rlim_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:145:26>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:145:34>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:146:1 <Spelling=/usr/include/b
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:146:12 <Spelling=/usr/inclu
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:146:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:146:12 <Spelling=/usr/include/bits/ty
entifier '__rlim64_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:146:28>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:146:38>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:147:1 <Spelling=/usr/include/b
```

```
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:147:12 <Spelling=/usr/inclu
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:147:12 <Spelling=/usr/include/bits/ty
entifier '__id_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:147:24>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:147:30>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:148:1 <Spelling=/usr/include/b
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:148:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:148:12 <Spelling=/usr/include/bits/ty
entifier '__time_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:148:26>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:148:34>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:149:1 <Spelling=/usr/include/b
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:149:12
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:167:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:167:12
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:172:12 <Spelling=/usr/include/bits/ty
entifier '__fsblkcnt64_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:172:32>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:172:46>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:175:1 <Spelling=/usr/include/b
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:175:12 <Spelling=/usr/inclu
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:175:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:175:12 <Spelling=/usr/include/bits/ty
entifier '__fsfilcnt_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:175:30>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:175:42>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:176:1 <Spelling=/usr/include/b
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:176:12 <Spelling=/usr/inclu
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:176:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:176:12 <Spelling=/usr/include/bits/ty
entifier '__fsfilcnt64_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:176:32>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:176:46>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:179:1 <Spelling=/usr/include/b
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:179:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:179:12 <Spelling=/usr/include/bits/ty
entifier '__fsword_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:179:28>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:179:38>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:181:1 <Spelling=/usr/include/b
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:181:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:181:12 <Spelling=/usr/include/bits/ty
entifier '__ssize_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:181:27>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:181:36>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:184:1 <Spelling=/usr/include/b
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:184:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:184:12 <Spelling=/usr/include/bits/ty
entifier '__syscall_slong_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:184:33>
```

```
mi ';'^^I^^ILoc=</usr/include/bits/types.h:184:50>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:186:1 <Spelling=/usr/include/b
signed 'unsigned'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:186:12 <Spelling=/usr/inclu
ng 'long'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:186:12 <Spelling=/usr/include/bits/
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:186:12
t 'int'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:195:12 <Spelling=/usr/include/bits/ty
entifier '__intptr_t'^^I [LeadingSpace]^^ILoc=</usr/include/bits/types.h:195:25>
mi ';'^^I^^ILoc=</usr/include/bits/types.h:195:35>
pedef 'typedef'^^I [StartOfLine]^^ILoc=</usr/include/bits/types.h:198:1 <Spelling=/usr/include/b
中间有省略
```

由于输出的 token 较长，文中仅展示了一小部分的 token 序列。通过上述 token，我们可以看出词法分析的任务是从左至右逐个字符地扫描源程序，产生一个个单词符号，把源程序的字符串改造成为单词符号串。clang 编译器的词法分析器会对词法单元、关键字、空格、缩进、换行、注释以及部分词法错误进行标记处理。通过对程序进行分词和标记，即可以将源代码转换成 tokens，以便于后续的语法处理。

3.2 语法分析

编译器会将词法分析生成的词法单元构建抽象语法树 (Abstract Syntax Tree，即 AST)。对于 gcc，笔者使用 gcc -fdump-tree-original-raw flag 获得文本格式的 AST 输出。LLVM 可以通过如下命令通过选项，我们获得了文本格式的 AST。

其中使用命令`cc -fdump-tree-oriinal-raw main.` 获得的 gcc 编译器的文本格式的 AST 输出如下，输出的结果保存在 main.c.003t.original 中。

```
 ;; Function main (null)
enabled by -tree-original

    bind_expr         type: @2       vars: @3       body: @4
    void_type         name: @5       algn: 8
    var_decl          name: @6       type: @7       scpe: @8
                      srcp: main.c:4                size: @9
                      algn: 32       used: 1
    statement_list    0   : @10      1   : @11      2   : @12
                      3   : @13      4   : @14      5   : @15
                      6   : @16      7   : @17      8   : @18
                      9   : @19      10  : @20      11  : @21
                      12  : @22      13  : @23      14  : @24
    type_decl         name: @25      type: @2
    identifier_node   strg: i        lngt: 1
    integer_type      name: @26      size: @9       algn: 32
                      prec: 32       sign: signed   min : @27
```

```
                       max : @28
      function_decl    name: @29      type: @30      srcp: main.c:2
                       link: extern
      integer_cst      type: @31      low : 32
0     decl_expr        type: @2
1     decl_expr        type: @2
2     decl_expr        type: @2
3     call_expr        type: @7       fn  : @32      0   : @33
                       1   : @34
4     modify_expr      type: @7       op 0: @3       op 1: @35
5     modify_expr      type: @7       op 0: @36      op 1: @37
6     goto_expr        type: @2       labl: @38
7     label_expr       type: @2       name: @39
8     modify_expr      type: @7       op 0: @36      op 1: @40
由于篇幅原因，中间 18-81 行内容省略。
1     tree_list        valu: @58
2     type_decl        name: @88      type: @83
3     integer_type     name: @82      size: @51      algn: 8
                       prec: 8        sign: signed   min : @84
                       max : @85
4     integer_cst      type: @83      high: -1       low : -128
5     integer_cst      type: @83      low : 127
6     array_type       size: @89      algn: 8        elts: @83
                       domn: @90
7     array_type       size: @91      algn: 8        elts: @83
                       domn: @92
8     identifier_node  strg: char     lngt: 4
9     integer_cst      type: @31      low : 24
0     integer_type     size: @68      algn: 64       prec: 64
                       sign: signed   min : @93      max : @94
1     integer_cst      type: @31      low : 56
2     integer_type     size: @68      algn: 64       prec: 64
                       sign: signed   min : @93      max : @95
3     integer_cst      type: @96      low : 0
4     integer_cst      type: @96      low : 2
5     integer_cst      type: @96      low : 6
6     integer_type     name: @97      size: @68      algn: 64
                       prec: 64       sign: unsigned min : @93
                       max : @98
7     identifier_node  strg: sizetype lngt: 8
8     integer_cst      type: @96      low : -1

```

笔者同时使用命令lang -E -Xlang -ast-dump main.c 获得的 clang 编译器的文本格式的 AST 输出如下。

```
anslationUnitDecl 0x1c64d60 <<invalid sloc>>
TypedefDecl 0x1c65260 <<invalid sloc>> __int128_t '__int128'
TypedefDecl 0x1c652c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
TypedefDecl 0x1c65610 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
TypedefDecl 0x1c65670 </usr/bin/../lib/clang/3.4.2/include/stddef.h:34:1, col:26> ptrdiff_t 'lon
TypedefDecl 0x1c656d0 <line:42:1, col:23> size_t 'unsigned long'
TypedefDecl 0x1c65730 <line:65:1, col:24> wchar_t 'int'
TypedefDecl 0x1c65790 </usr/include/bits/types.h:30:1, col:23> __u_char 'unsigned char'
TypedefDecl 0x1c657f0 <line:31:1, col:28> __u_short 'unsigned short'
TypedefDecl 0x1c65850 <line:32:1, col:22> __u_int 'unsigned int'
TypedefDecl 0x1c658b0 <line:33:1, col:27> __u_long 'unsigned long'
TypedefDecl 0x1c65910 <line:36:1, col:21> __int8_t 'signed char'
TypedefDecl 0x1c65970 <line:37:1, col:23> __uint8_t 'unsigned char'
TypedefDecl 0x1c659d0 <line:38:1, col:26> __int16_t 'short'
TypedefDecl 0x1c65a30 <line:39:1, col:28> __uint16_t 'unsigned short'
TypedefDecl 0x1cd5450 <line:40:1, col:20> __int32_t 'int'

`-NoThrowAttr 0x1d01cd0 </usr/include/sys/cdefs.h:56:35>
FunctionDecl 0x1d01ee0 </usr/include/stdio.h:356:12> fprintf 'int (FILE *, const char *, ...)' e
|-ParmVarDecl 0x1d01f80 <<invalid sloc>> 'FILE *'
|-ParmVarDecl 0x1d01fe0 <<invalid sloc>> 'const char *'
`-FormatAttr 0x1d02050 <col:12> printf 2 3
FunctionDecl 0x1d020a0 prev 0x1d01ee0 <col:1, line:357:43> fprintf 'int (FILE *, const char *, .
|-ParmVarDecl 0x1d01d20 <line:356:21, col:38> __stream 'FILE *restrict'
|-ParmVarDecl 0x1d01d90 <line:357:7, col:30> __format 'const char *restrict'
`-FormatAttr 0x1d02180 <line:356:12> printf 2 3
FunctionDecl 0x1d022c0 <line:362:12> printf 'int (const char *, ...)' extern
|-ParmVarDecl 0x1d02360 <<invalid sloc>> 'const char *'
`-FormatAttr 0x1d023c0 <col:12> printf 1 2
FunctionDecl 0x1d02410 prev 0x1d022c0 <col:1, col:56> printf 'int (const char *, ...)' extern
|-ParmVarDecl 0x1d021c0 <col:20, col:43> __format 'const char *restrict'
`-FormatAttr 0x1d024e0 <col:12> printf 1 2
FunctionDecl 0x1d02740 <line:364:12> sprintf 'int (char *, const char *, ...)' extern
|-ParmVarDecl 0x1d027e0 <<invalid sloc>> 'char *'
|-ParmVarDecl 0x1d02840 <<invalid sloc>> 'const char *'
`-FormatAttr 0x1d028b0 <col:12> printf 2 3
FunctionDecl 0x1d02900 prev 0x1d02740 <col:1, /usr/include/sys/cdefs.h:57:49> sprintf 'int (char
|-ParmVarDecl 0x1d02520 </usr/include/stdio.h:364:21, col:38> __s 'char *restrict'
|-ParmVarDecl 0x1d02590 <line:365:7, col:30> __format 'const char *restrict'
|-NoThrowAttr 0x1d029b0 </usr/include/sys/cdefs.h:57:37>
`-FormatAttr 0x1d029f0 </usr/include/stdio.h:364:12> printf 2 3
FunctionDecl 0x1d02c20 <line:371:12> vfprintf 'int ()' extern
`-FormatAttr 0x1d02cc0 <col:12> printf 2 0
FunctionDecl 0x1d02d10 prev 0x1d02c20 <col:1, line:372:24> vfprintf 'int (FILE *restrict, const 
|-ParmVarDecl 0x1d02a30 <line:371:22, col:39> __s 'FILE *restrict'
|-ParmVarDecl 0x1d02aa0 <col:44, col:67> __format 'const char *restrict'
|-ParmVarDecl 0x1d02b10 </usr/include/_G_config.h:46:20, /usr/include/stdio.h:372:19> __arg '__v
`-FormatAttr 0x1d02df0 <line:371:12> printf 2 0
FunctionDecl 0x1d02ff0 <line:377:12> vprintf 'int (const char *, __va_list_tag *)' extern


FunctionDecl 0x1d04440 prev 0x1d041b0 <col:1, line:392:62> vsnprintf 'int (char *, unsigned long
|-ParmVarDecl 0x1d03e80 <line:390:23, col:40> __s 'char *restrict'
|-ParmVarDecl 0x1d03ef0 <col:45, col:52> __maxlen 'size_t':'unsigned long'
|-ParmVarDecl 0x1d03f60 <line:391:9, col:32> __format 'const char *restrict'
|-ParmVarDecl 0x1d03fd0 </usr/include/_G_config.h:46:20, /usr/include/stdio.h:391:53> __arg '__v
|-FormatAttr 0x1d04500 <line:392:32, col:60> printf 3 0
`-NoThrowAttr 0x1d04550 </usr/include/sys/cdefs.h:57:37>
中间进行了省略
|-ParmVarDecl 0x1d05130 <<invalid sloc>> 'const char *restrict'
`-FormatAttr 0x1d05190 <col:12> scanf 1 2
FunctionDecl 0x1d051e0 prev 0x1d05090 <col:1, col:55> scanf 'int (const char *restrict, ...)' ex
|-ParmVarDecl 0x1d05010 <col:19, col:42> __format 'const char *restrict'
`-FormatAttr 0x1d052b0 <col:12> scanf 1 2
FunctionDecl 0x1d05470 <line:433:12> sscanf 'int (const char *restrict, const char *restrict, ..
|-ParmVarDecl 0x1d05510 <<invalid sloc>> 'const char *restrict'
|-ParmVarDecl 0x1d05570 <<invalid sloc>> 'const char *restrict'
`-FormatAttr 0x1d055e0 <col:12> scanf 2 3
```

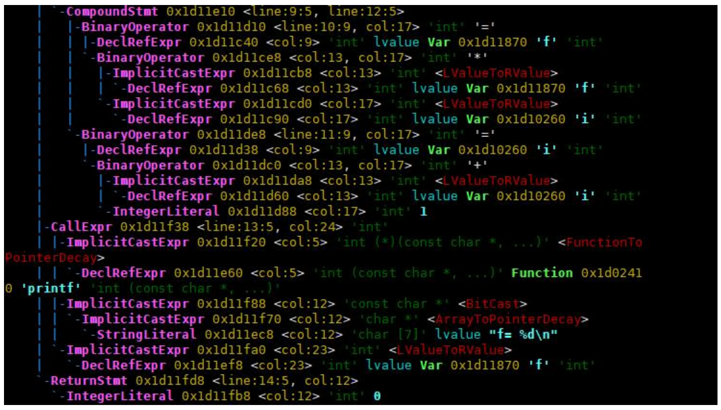
clang 编译器的终端使用用颜色区分语法树。我们可以用图形显示如上：

LLVM IR 有 3 种表示形式：text: 便于阅读的文本格式，类似于汇编语言，拓展名.ll，$clang - S - emit - llvmmain.mmemory : \quad bitcode : \quad .bc$ clang -c -emit-llvm main.m

本次实验以 text 形式编译查看。编译器同时会使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。在这一步，编译器同时进行了语法、词法相关的错误检查。若 mian.c 源程序中存在语法错误，clang 在输出语法树的同时也会报错。

3.3 中间代码生成与优化

具体命令如下：



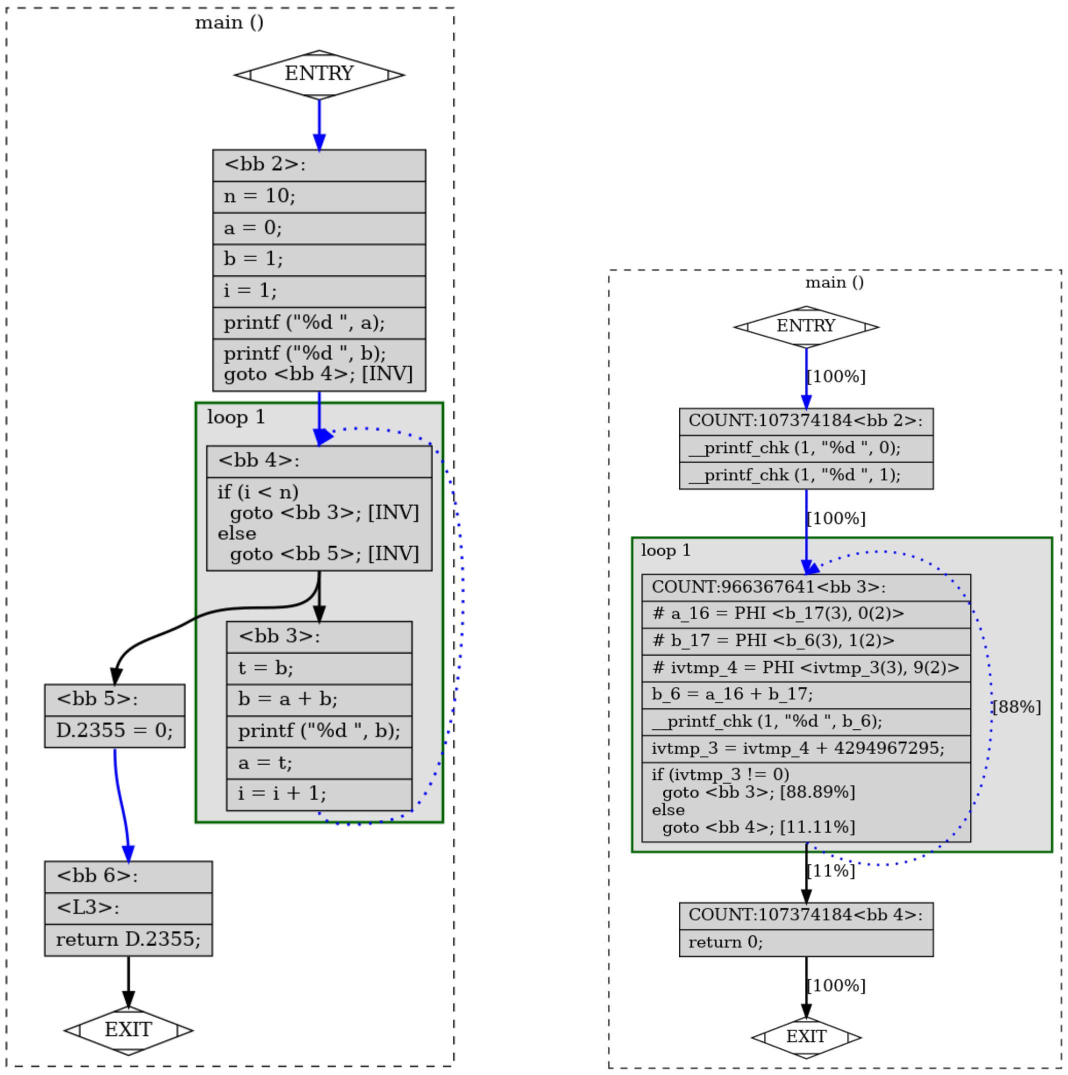
```
  | `-CompoundStmt 0x1d11e10 <line:9:5, line:12:5>
  |   |-BinaryOperator 0x1d11d10 <line:10:9, col:17> 'int' '='
  |   | |-DeclRefExpr 0x1d11c40 <col:9> 'int' lvalue Var 0x1d11870 'f' 'int'
  |   | `-BinaryOperator 0x1d11ce8 <col:13, col:17> 'int' '*'
  |   |   |-ImplicitCastExpr 0x1d11cb8 <col:13> 'int' <LValueToRValue>
  |   |   | `-DeclRefExpr 0x1d11c68 <col:13> 'int' lvalue Var 0x1d11870 'f' 'int'
  |   |   `-ImplicitCastExpr 0x1d11cd0 <col:17> 'int' <LValueToRValue>
  |   |     `-DeclRefExpr 0x1d11c90 <col:17> 'int' lvalue Var 0x1d10260 'i' 'int'
  |   `-BinaryOperator 0x1d11de8 <line:11:9, col:17> 'int' '='
  |     |-DeclRefExpr 0x1d11d38 <col:9> 'int' lvalue Var 0x1d10260 'i' 'int'
  |     `-BinaryOperator 0x1d11dc0 <col:13, col:17> 'int' '+'
  |       |-ImplicitCastExpr 0x1d11da8 <col:13> 'int' <LValueToRValue>
  |       | `-DeclRefExpr 0x1d11d60 <col:13> 'int' lvalue Var 0x1d10260 'i' 'int'
  |       `-IntegerLiteral 0x1d11d88 <col:17> 'int' 1
  |-CallExpr 0x1d11f38 <line:13:5, col:24> 'int'
  | |-ImplicitCastExpr 0x1d11f20 <col:5> 'int (*)(const char *, ...)' <FunctionTo
PointerDecay>
  | | `-DeclRefExpr 0x1d11e60 <col:5> 'int (const char *, ...)' Function 0x1d0241
0 'printf' 'int (const char *, ...)'
  | |-ImplicitCastExpr 0x1d11f88 <col:12> 'const char *' <BitCast>
  | | `-ImplicitCastExpr 0x1d11f70 <col:12> 'char *' <ArrayToPointerDecay>
  | |   `-StringLiteral 0x1d11ec8 <col:12> 'char [7]' lvalue "f= %d\n"
  | `-ImplicitCastExpr 0x1d11fa0 <col:23> 'int' <LValueToRValue>
  |   `-DeclRefExpr 0x1d11ef8 <col:23> 'int' lvalue Var 0x1d11870 'f' 'int'
  `-ReturnStmt 0x1d11fd8 <line:14:5, col:12>
    `-IntegerLiteral 0x1d11fb8 <col:12> 'int' 0
```

图 3.1: clang 编译器语法树

```
gcc -fdump-tree-all-graph -fdump-rtl-all-graph main.c
dot -Tpng a-main.c.015t.cfg.dot > a-main.c.015t.cfg.dot.png
dot -Tpng a-main.c.244t.optimized.dot > a-main.c.244t.optimized.dot.png
```

在经过语法分析之后，编译器会生成一个明确的类机器语言的中间表示。实验中笔者通过-fdump-tree-all-graph 和-fdump-rtl-all-graph 两个 gcc flag 获得中间代码生成的多阶段的输出。由于生成了较多阶段的.dot 文件。笔者选择了部分生成的.dot 文件，利用 graphviz 进行可视化。可视化后我们可以看到程序的控制流图（CFG）以及各阶段处理中 CFG 的变化。

如图下图所示，`a-main.c.015t.cfg.dot`和`a-main.c.244t.optimized.dot`这两个文件分别是程序的控制流图和优化后的控制流图。我们不难发现，在优化后的程序中，程序的变量和相关罗杰逻辑被改变了，相对未优化的版本，优化的版本流程更加简短，减少了执行的时间。



(a) a-main.c.015t.cfg.dot (b) a-main.c.244t.optimized.dot

图 3.2: GCC 中间结果的可视化

LLVM 的中间结果是一种类似汇编、却又不受限于平台的特殊语言 LLVM IR。笔者通过`-emit-llvm`参数来获取它。

```
clang -S -emit-llvm -O2 ./main.c -o main.O2.ll
clang -S -emit-llvm -Xclang -disable-O0-optnone ./main.c -o main.ll

opt -S main.ll -O1 -print-before-all > fin.log.ll 2>&1
```

本次实验通过`-O2`和`-Xclang -disable-O0-optnone`两个选项，输出流阶乘在 O2 优化和无优化选项下的 LLVM IR 代码，由于笔者有意使用了常量N作为斐波那契数列的长度，因此在 O2 优化下，编译器会在编译时就计算出数列的值，之后在运行时程序就可以直接将其输出。其中 clang -S -emit-llvm -O2 ./main.c -o main.O2.ll 为 O2 优化级别，clang -S -emit-llvm -Xclang -disable-O0-optnone ./main.c -o main.ll 为禁用优化级别，同时使用 opt -S main.ll -O1 -print-before-all > fin.log.ll 2>1 来输出优化

结果，编译器将会根据优化的结果生成汇编代码。

4 汇编器做了什么？

从本质上讲，汇编器也是编译器的一种，只是汇编器处理的“高级语言”是汇编语言，输出的是机器语言二进制形式。查阅资料发现，汇编器的构造和编译器相似，都需要进行词法分析、语法分析、语义处理、符号表管理和代码生成（机器代码）等阶段。

对于编译器来说，代码生成阶段只需要将解析的语法树映射到汇编语言子模块，而对于汇编器，将解析出的指令简洁的映射到正确机器代码相对比较复杂。由于本汇编器处理的输入文件为编译器生成的汇编文件，因此汇编器不需要考虑源文件会产生错误，因此它的语法分析的目的是识别出输入语言的语法结构并进行解析引导机器代码生成，因此不需要进行语法检查。汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”。可以使用 gcc 命令`gcc main.c -c -o main.o`将汇编代码汇编为二进制文件。

通过`readelf`来解析这个文件的相关信息，如下图所示。

```
  ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              REL (Relocatable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x0
Start of program headers:          0 (bytes into file)
Start of section headers:          832 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           0 (bytes)
Number of program headers:         0
Size of section headers:           64 (bytes)
Number of section headers:         13
Section header string table index: 12

ction Headers:
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[ 0]                   NULL             0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
[ 1] .text             PROGBITS         0000000000000000  00000040
```

```
     0000000000000066  0000000000000000  AX       0     0     1
[ 2] .rela.text        RELA              0000000000000000  00000260
     0000000000000060  0000000000000018   I      10     1     8
[ 3] .data             PROGBITS          0000000000000000  000000a6
     0000000000000000  0000000000000000  WA       0     0     1
[ 4] .bss              NOBITS            0000000000000000  000000a6
     0000000000000000  0000000000000000  WA       0     0     1
[ 5] .rodata           PROGBITS          0000000000000000  000000a6
     000000000000000a  0000000000000000   A       0     0     1
[ 6] .comment          PROGBITS          0000000000000000  000000b0
     000000000000002e  0000000000000001  MS       0     0     1
[ 7] .note.GNU-stack   PROGBITS          0000000000000000  000000de
     0000000000000000  0000000000000000           0     0     1
[ 8] .eh_frame         PROGBITS          0000000000000000  000000e0
     0000000000000038  0000000000000000   A       0     0     8
[ 9] .rela.eh_frame    RELA              0000000000000000  000002c0
     0000000000000018  0000000000000018   I      10     8     8
[10] .symtab           SYMTAB            0000000000000000  00000118
     0000000000000120  0000000000000018          11     9     8
[11] .strtab           STRTAB            0000000000000000  00000238
     0000000000000023  0000000000000000           0     0     1
[12] .shstrtab         STRTAB            0000000000000000  000002d8
     0000000000000061  0000000000000000           0     0     1
y to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

ere are no section groups in this file.

ere are no program headers in this file.

location section '.rela.text' at offset 0x260 contains 4 entries:
Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000000010  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000001a  000a00000002 R_X86_64_PC32     0000000000000000 __isoc99_scanf - 4
0000000051  00050000000a R_X86_64_32       0000000000000000 .rodata + 3
00000005b  000b00000002 R_X86_64_PC32     0000000000000000 printf - 4

location section '.rela.eh_frame' at offset 0x2c0 contains 1 entries:
Offset          Info           Type           Sym. Value    Sym. Name + Addend
```

```
0000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0

e decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently su

mbol table '.symtab' contains 12 entries:
 Num:    Value          Size Type    Bind   Vis      Ndx Name
   0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
   2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
   3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
   4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
   5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
   6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
   7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
   8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
   9: 0000000000000000   102 FUNC    GLOBAL DEFAULT    1 main
  10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __isoc99_scanf
  11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf

 version information found in this file.

```

5 链接器做了什么？

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而连接器对该机器代码进行执行生成可执行文件。

使用`gcc main.o -o main`命令，链接器根据汇编器中的信息将可重定位文件组合在一起，由于汇编器产生的可重定位文件中函数和其他标识符仍然有其名称定义，并不绑定到任何特定的地址，因此在将程序编译为可重定位 ELF 文件后，链接器生成可以链接可重定位文件会与库文件为一个整体，最生成可执行 ELF 文件。

6 LLVM IR 编程

6.1 小组分工

小组分工为：熊宇轩同学负责变量声明、赋值、运算符、数组等 SysY 特性，王天鹏同学负责循环、分支和函数调用等 SysY 特性，由于小组成员目前对编译相关知识了解还不多，后期相关任务分配可能会有调整。

但本次的 LLVM IR 编程作业任务较少，故使用王天鹏编写的 C 语言阶乘代码，由熊宇轩将其翻译成 LLVM IR 代码，此外，两人都参与了代码的调试和修饰工作。

6.2 C 语言阶乘源码

C 语言的阶乘程序源码如下所示：

```c
#include <stdio.h>

int factorial(int n)
{
    int i = 2, f = 1;
    if (n < 0)
        return -1;
    while(i <= n)
    {
        f = f*i;
        i = i+1;
    }
    return f;
}

int main()
{
    int n;
    scanf("%d", &n);
    n = factorial(n);
    printf("%d", n);
    return 0;
}
```

6.3 LLVM IR 代码

LLVM IR 的阶乘程序源码如下所示：

```llvm
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @factorial(i32 noundef %0) #0 {
  ; ***BEGIN OF STUDENT CODE***
  %2 = alloca i32, align 4 ; i
  %3 = alloca i32, align 4 ; f
  %4 = alloca i32, align 4
  ; ^没有用，但也删不掉，问就是非得按顺序来，
  ;  不然就'%5' is expected to be numbered '%4'
  ;  编译器级别的“这句没用，但是别改，删了跑不了”
  ;  重构时都是先用英语命名标识符，然后从上到下一个个分配数字
  ;  后来才发现label不一定要用数字。。。
  store i32 2, i32* %2, align 4 ; i = 2
  store i32 1, i32* %3, align 4 ; f = 1

  %5 = icmp slt i32 %0, 0 ; n < 0
  br i1 %5, label %6, label %7

6:
  store i32 -1, i32* %3, align 4 ; ret = -1
  br label %15

7:
  %8 = load i32, i32* %2, align 4 ; i
  %9 = icmp sle i32 %8, %0 ; i <= n
  ; ^ 这个i<=n之前写反了导致程序跑不了, 瞪了好久, 不知道怎么调试LLVM IR
  ;   后来想起用Cutter/Ghidra反编译成C代码, 一眼发现。。。
  br i1 %9, label %10, label %15

10:
  %11 = load i32, i32* %2, align 4 ; i
  %12 = load i32, i32* %3, align 4 ; f
  %13 = mul nsw i32 %11, %12  ; f*i
  store i32 %13, i32* %3, align 4 ; f = f*i
  %14 = add nsw i32 %11, 1 ; i+1
  store i32 %14, i32* %2, align 4 ; i = i+1
  br label %7

15:
  %16 = load i32, i32* %3, align 4 ; ret
  ret i32 %16
  ; ***END OF STUDENT CODE***
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  ; ***BEGIN OF STUDENT CODE***
  %1 = alloca i32, align 4 ; n
  %2 = alloca i32, align 4 ; 13

  %3 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x
      i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %1)

  %4 = load i32, i32* %1, align 4
  %5 = call i32 @factorial(i32 noundef %4); 13 = factorial(n)

  %6 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([3 x i8], [3 x
      i8]* @.str, i64 0, i64 0), i32 noundef %5)

  ret i32 0

  ; ***END OF STUDENT CODE***
}
```

上文中的代码仅包含了函数实现的部分，其他外围 LLVM IR 的代码则是照搬 clang 编译 C 语言版阶乘程序时所生成结果，完整的手工翻译的及 clang 生成的 LLVM IR 代码GitHub Gist找到。

通过`llc ./mytest.ll -filetype=obj -o test.o`和`clang ./test.o -static -o ./test`，我们成功将以上 LLVM IR 代码编译成了目标程序，经过测试，其行为与直接编译 C 语言版本的源码一致。