南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理实验报告

预备工作 1——了解编译器及 LLVM IR 编程

刘芳宜

年级：2022 级

专业：计算机科学与技术

指导教师：李忠伟

2024 年 9 月 12 日

摘要

本实验的主要目的是熟悉编译器，熟悉中间表示（LLVM IR）的基本原理和使用方法，并通过编写和运行小型 SysY 源程序以阶乘程序为例，验证 LLVM IR 与源程序之间的对应关系。我们对比 Clang，gcc，llvm 生成各个阶段的输出，包括语法分析树、控制流图、LLVM IR 代码及汇编代码，深入理解编译器的工作流程。同时，实验中我们探讨了 LLVM IR 如何表达高级语言特性，并通过调试和验证最终生成的目标程序; 以及在代码优化阶段，对 O1 O3 和不同优化 pass 方法对比，并通过程序性能（运行时间）进行优化验证。

关键字：llvm,riscv, 阶乘,

目录

一、 概述

(一) 环境描述

环境	版本
WSL	ubuntu22.02
vscode	2022

表 1: 开发环境信息

二、 实验问题 1 过程

完整的编译过程包括预处理，词法和语法分析，生成 LLVM IR，优化，生成汇编代码，生成目标文件，编译，链接，执行反汇编，使用命令查看完整流 $clang - ccc - print - phasesmul.c$

- 编译
- 链接
- 汇编
- 链接执行

如图11所示

```
lfy@lfy:/mnt/d/2024fall/编译/lab/lab1/CompilingProject/lab1/mul$ clang -ccc-print-phases mul.c
          +- 0: input, "mul.c", c
       +- 1: preprocessor, {0}, cpp-output
    +- 2: compiler, {1}, ir
  +- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
```

图 1: 完整工作过程

阶乘代码 (文件名为 mul.c)

```c
#include <stdio.h>
int main() {
    int i, n, f;
    scanf("%d", &n);
    i = 2;
    f = 1;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    printf("%d\n", f);
}
```

(一) 预处理器的作用

预处理阶段使用命令 $gcc - E\ mul.c - o\ mul.i$ 预处理文件生成 $mul.i$ 此文件包含了宏展开和头文件包含后的源代码如图11所示

```
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 885 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 902 "/usr/include/stdio.h" 3 4

# 2 "mul.c" 2


# 3 "mul.c"
int main() {
    int i, n, f;
    scanf("%d", &n);
    i = 2;
    f = 1;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    printf("%d\n", f);
    printf("%d\n", 3.14);
}       You, 47秒钟前 • Uncommitted changes
```

图 2: mul.i 预处理阶段

预处理阶段负责处理预编译指令，主要包括以 # 开头的指令，如 #include、#define、#if 等。对于 #include 指令，预处理器会将相应的头文件内容替换进来；对于 #define 宏定义，预处理器会用宏的实际内容替换对应的宏调用。此外，预处理阶段还会删除代码中的注释，并添加行号和文件名标识。

在使用 gcc 时，可以通过添加-E 参数让 gcc 只执行预处理过程，再通过-o 参数指定输出文件名。因此，执行命令 $gccmul.c - E - omul.i$ 可以生成预处理后的文件。

查看预处理后的文件时，可以发现文件比源文件长很多，这是因为代码中的头文件内容已经被替换进来了。

(二) 编译器的作用

词法分析利用 clang -E -Xclang -dump-tokens mul.c, 将源程序转换为单词序列可以发现，词法分析过程中，对源程序的字符串进行扫描和分解，识别出一个个的单词，对程序进行分词处理，并标明每个 token 的类型。部分词法分析结果如下图所示:

```
char 'char'           Loc=</usr/include/stdio.h:837:23>
star '*'        [LeadingSpace] Loc=</usr/include/stdio.h:837:28>
identifier '__s'              Loc=</usr/include/stdio.h:837:29>
r_paren ')'           Loc=</usr/include/stdio.h:837:32>
__attribute '__attribute__'     [LeadingSpace] Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:19>>
l_paren '('     [LeadingSpace] Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:33>>
l_paren '('           Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:34>>
identifier '__nothrow__'              Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>>
r_paren ')'     [LeadingSpace] Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:53>>
r_paren ')'           Loc=</usr/include/stdio.h:837:34 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54>>
semi ';'        [StartOfLine] [LeadingSpace]  Loc=</usr/include/stdio.h:838:38>
extern 'extern'  [StartOfLine]  Loc=</usr/include/stdio.h:867:1>
void 'void'     [LeadingSpace] Loc=</usr/include/stdio.h:867:8>
identifier 'flockfile'   [LeadingSpace] Loc=</usr/include/stdio.h:867:13>
l_paren '('     [LeadingSpace] Loc=</usr/include/stdio.h:867:23>
identifier 'FILE'             Loc=</usr/include/stdio.h:867:24>
star '*'        [LeadingSpace] Loc=</usr/include/stdio.h:867:29>
identifier '__stream'         Loc=</usr/include/stdio.h:867:30>
r_paren ')'           Loc=</usr/include/stdio.h:867:38>
__attribute '__attribute__'     [LeadingSpace] Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:19>>
l_paren '('     [LeadingSpace] Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:33>>
l_paren '('           Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:34>>
identifier '__nothrow__'              Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>>
r_paren ')'     [LeadingSpace] Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:53>>
r_paren ')'           Loc=</usr/include/stdio.h:867:40 <Spelling=/usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54>>
semi ';'              Loc=</usr/include/stdio.h:867:47>
extern 'extern'  [StartOfLine]  Loc=</usr/include/stdio.h:871:1>
int 'int'       [LeadingSpace] Loc=</usr/include/stdio.h:871:8>
identifier 'ftrylockfile'       [LeadingSpace] Loc=</usr/include/stdio.h:871:12>
l_paren '('     [LeadingSpace] Loc=</usr/include/stdio.h:871:25>
```

图 3: 词法分析

语法分析过程使用命令 $clang - Xclang - ast - dump - fsyntax - only fact.c$ 将代码转化为语法树，分析程序的语法结构。语法分析阶段利用词法分析阶段的单词构成一棵语法分析树，可以看到明显的层次关系。

图 4: 语法分析生成语法树

语义分析与中间代码生成语义分析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、范围检查、数组绑定检查等。利用命令 $clang - S - emit - llvm fact.c - omul.ll$ 生成 mul.ll 中间代码

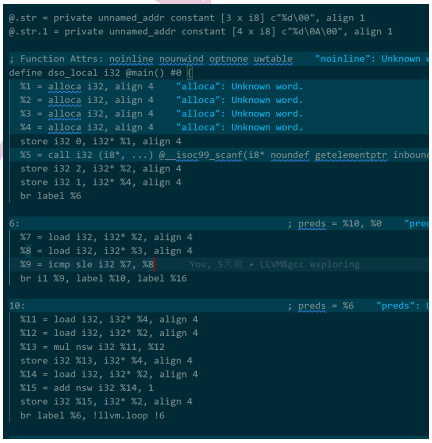

图 5: 中间代码

对比发现，mul.ll 文件中的代码量明显少于前几个生成文件语义分析与中间代码生成阶段生成的.ll 文件中，死代码进行了删除，说明在代码优化这一步之前就已经对死代码进行了优化。

利用 gcc 通过 gcc -fdump-tree-all-graph mul.c 获得中间代码生成的多阶段输出，通过 graphviz 对 CFG 进行可视化，此处选取 mul.c.011t.cfg.dot 和 main.c.086.fixup_cfg4.dot 进行分析：可以很明显看到块与块之间的逻辑关系。包括各部分的跳转、分支等。可以发现控制流图 CFG 的变化是：

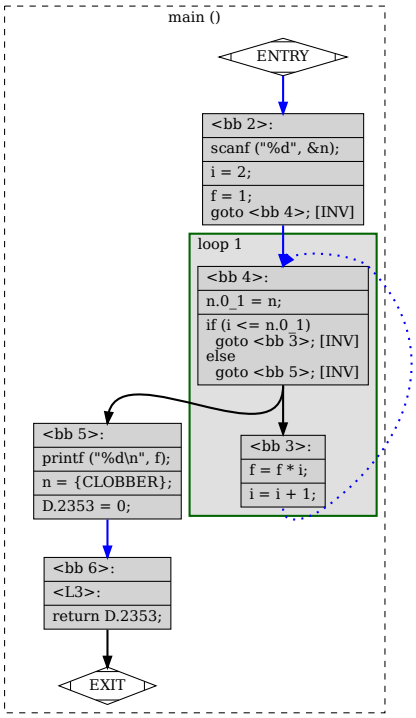

图 6: a-mul.c.022t.fixup_cfg1

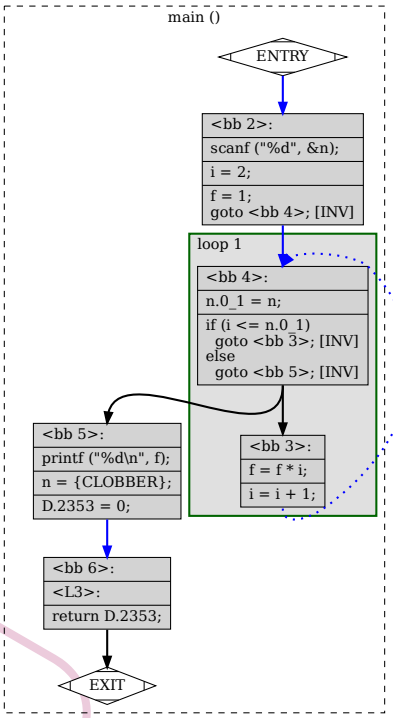

图 7: a-mul.c.015t.cfg

图 8: 中间代码的比较

优化分析

相同的控制流：在两个图中，函数的基本结构和控制流几乎完全相同，主要的块（例如：ENTRY、EXIT、循环体和条件跳转）保持不变。优化集中在特定区域的改进上。**循环优化：** loop 1 结构在两个图中保持一致，主要包含变量 i_2 和 f_3 的循环计算。虽然表面看似没有明显变化，但图中的一些边缘（例如从 <bb 4> 到 <bb 3> 的边缘）可能涉及优化，例如循环展开或循环的增强，以减少循环体内的冗余计算。**块间的边缘权重优化：**在图中，连接基本块的边缘有权重，例如 <bb 3> 到 <bb 4> 的权重是”dotted,bold” 和”solid,bold” 的组合。权重优化可以表明编译器已优化了条件跳转，减少了不必要的分支预测失误。**内存存取优化：**在两个图中的 <bb 5> 块内，使用了 CLOBBER 语法来表示变量 n 的状态变化。这表明编译器已优化了一些与内存有关的操作，如减少多余的内存读写或者将变量存储优化为寄存器变量。**冗余代码消除：**从这两个图来看，可能存在一些冗余代码已被消除。尽管控制流图显示的基本块大致相同，但某些不必要的中间变量和计算可能已被编译器去除，最终在优化后的程序中消耗更少的资源。

优化特点总结

- 循环优化：编译器可以通过循环展开、循环合并或减少循环内不必要的操作来提升循环执行效率。这使得程序的运行时间更短，尤其是在重复操作的情况下。
- 条件跳转优化：在图中看到条件分支的优化迹象，这可以通过减少不必要的条件判断、优化条件预测路径或合并分支来实现。对于高效代码执行，减少不必要的条件跳转尤为重要。
- 内存优化：CLOBBER 表明编译器对内存操作进行了优化，可能是减少对内存的读写频率，或者将频繁使用的变量存放在寄存器中，从而提升程序性能。

- 冗余消除：图中一些变量的初始化和使用，可能通过静态分析被优化掉。这减少了程序中的无用指令，使代码更加简洁高效。

通过这些优化措施，编译后的程序可以减少执行时间，降低内存占用，并提升整体运行效率。

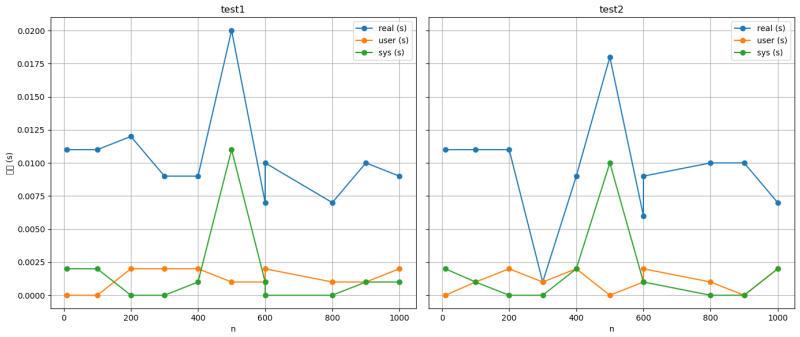

图 9: 性能运行比较

无优化 (-O0):$gcc-O0mul.c-omul\_O0$ 基本优化 (-O1):$gcc-O1mul.c-omul\_O1$ 更高优化 (-O2):$gcc-O2mul.c-omul\_O2$ 最高优化 (-O3): $gcc-O3mul.c-omul\_O3$ 去除符号表和调试信息 $gcc-O2-smul.c-omul\_O2\_strip$ 使用静态链接 $gcc-O2-staticmul.c-omul\_O2\_static$

在比较 O1 优化与未优化情况下程序性能，将中间代码进行汇编、链接之后生成可执行文件，分别运行阶乘程序并带入不同的 n 进行比较，为增加稳定性，每组（阶乘）循环并取平均值，如下表所示

n	real (s)	user (s)	sys (s)
10	0m0.011s	0m0.000s	0m0.002s
100	0m0.011s	0m0.000s	0m0.002s
200	0m0.012s	0m0.002s	0m0.000s
300	0m0.009s	0m0.002s	0m0.000s
400	0m0.009s	0m0.002s	0m0.001s
500	0m0.020s	0m0.001s	0m0.011s
600	0m0.007s	0m0.001s	0m0.001s
600	0m0.010s	0m0.002s	0m0.000s
800	0m0.007s	0m0.001s	0m0.000s
900	0m0.010s	0m0.001s	0m0.001s
1000	0m0.009s	0m0.002s	0m0.001s

表 2: 不同 n 值下程序的平凡性能测试结果

n	real (s)	user (s)	sys (s)
10	0m0.011s	0m0.000s	0m0.002s
100	0m0.011s	0m0.001s	0m0.001s
200	0m0.011s	0m0.002s	0m0.000s
300	0m0.001s	0m0.001s	0m0.000s
400	0m0.009s	0m0.002s	0m0.002s
500	0m0.018s	0m0.000s	0m0.010s
600	0m0.006s	0m0.001s	0m0.001s
600	0m0.009s	0m0.002s	0m0.001s
800	0m0.010s	0m0.001s	0m0.000s
900	0m0.010s	0m0.000s	0m0.000s
1000	0m0.007s	0m0.002s	0m0.002s

表 3: 不同 n 值下程序的性能优化测试结果

三、 中间代码生成 llvm

以该实验以斐波那契实验为例子进行实验验证，根据实验指导编写了 makefile 文件对工程文件进行实验

llvm(文件名为 fib.c)

```
.PHONY: pre, lexer, ast-gcc, ast-llvm, cfg, ir-gcc, ir-llvm, asm, obj, exe,
    antiobj, antiexe, clean, clean-all
# 预处理阶段
pre:
	gcc fib.c -E -o fib.i
# 词法分析阶段, 使用 Clang 查看词法 token
lexer:
	clang -E -Xclang -dump-tokens fib.c
# 生成 AST 树, 使用 GCC 生成原始抽象语法树
ast-gcc:
	gcc -fdump-tree-original-raw fib.c
# 生成 LLVM AST 树
ast-llvm:
	clang -E -Xclang -ast-dump fib.c
# 生成 CFG 文件, 可以用 Graphviz 可视化
cfg:
	gcc -O0 -fdump-tree-all-graph fib.c
# 生成 GCC 的 RTL 文件
ir-gcc:
	gcc -O0 -fdump-rtl-all-graph fib.c
# 生成 LLVM IR
ir-llvm:
	clang -S -emit-llvm fib.c
# 生成汇编代码 (.S)
asm:
	gcc -O0 -o fib.S -S -masm=att fib.i
# 生成目标文件 (.o)
obj:
	gcc -O0 -c -o fib.o fib.S
# 反汇编目标文件, 查看机器代码
antiobj:
	objdump -d fib.o > fib-anti-obj.S
	nm fib.o > fib-nm-obj.txt
# 链接生成可执行文件
exe:
	gcc -O0 -o fib fib.o
# 反汇编可执行文件
antiexe:
	objdump -d fib > fib-anti-exe.S
	nm fib > fib-nm-exe.txt
# 清理生成的临时文件
clean:
	rm -rf *.c.*
# 清理所有生成的文件
clean-all:
	rm -rf *.c.* *.o *.S *.dot *.out *.txt *.ll *.i fib
```

(一) 编译源文件生成目标文件

从汇编生成目标文件、链接生成可执行文件的过程。目标文件不能直接执行，必须与其他目标文件或库链接，生成最终的可执行文件。链接器的作用是将这些机器代码结合起来，处理外部引用，分配地址空间，最终生成可执行的二进制文件。使用命令 $gcc - cmain.c - omain.o$ 生成可重定位的机器代码。

使用链接器将目标文件链接为可执行文件。

- 使用静态链接所有库的代码都会被复制到最终的可执行文件中，生成的可执行文件体积较大，可以独立运行。使用命令 $gcc - staticmain.o - omain$
- 使用动态链接将库的引用保留在最终的可执行文件中，运行时需要操作系统提供相应的动态链接库。使用命令 $gccmain.o - omain$

(二) 反汇编可执行文件

使用 objdump 反汇编工具来查看生成的目标文件或可执行文件的汇编代码，比较不同阶段

- 反汇编目标文件执行命令 objdump -d main.o
- 反汇编可执行文件执行命令 objdump -d main

.init 段

```
0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3
        fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret
```

.plt 段

```
0000000000001020 <printf@plt-0x10>:
    1020:       ff 35 e2 2f 00 00       push   0x2fe2(%rip)        # 4008 <
        _GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 e4 2f 00 00       jmp    *0x2fe4(%rip)       # 4010 <
        _GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <printf@plt>:
    1030:       ff 25 e2 2f 00 00       jmp    *0x2fe2(%rip)       # 4018 <
        printf@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   $0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>
```

.plt.got 段

```
0000000000001040 <__cxa_finalize@plt>:
    1040:       ff 25 b2 2f 00 00       jmp    *0x2fb2(%rip)        # 3ff8 <
        __cxa_finalize@GLIBC_2.2.5>
    1046:       66 90                   xchg   %ax,%ax
```

.text 段

```
0000000000001050 <_start>:
    1050:       f3 0f 1e fa             endbr64
    1054:       31 ed                   xor    %ebp,%ebp
    1056:       49 89 d1                mov    %rdx,%r9
    1059:       5e                      pop    %rsi
    105a:       48 89 e2                mov    %rsp,%rdx
    105d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1061:       50                      push   %rax
    1062:       54                      push   %rsp
    1063:       45 31 c0                xor    %r8d,%r8d
    1066:       31 c9                   xor    %ecx,%ecx
    1068:       48 8d 3d 01 01 00 00    lea    0x101(%rip),%rdi        # 1170
        <main>
    106f:       ff 15 63 2f 00 00       call   *0x2f63(%rip)        # 3fd8 <
        __libc_start_main@GLIBC_2.34>
    1075:       f4                      hlt
```

.fini 段

```
0000000000001190 <_fini>:
    1190:       f3 0f 1e fa             endbr64
    1194:       48 83 ec 08             sub    $0x8,%rsp
    1198:       48 83 c4 08             add    $0x8,%rsp
    119c:       c3                      ret
```

观察可得：可执行文件中的地址是绝对的，而目标文件中的地址是相对的，搜索资料可知链接器会负责地址分配。外部函数调用在可执行文件中已经被解析为真实的地址，而在目标文件中仍是未解析的符号。

通过 $nmfact$ 查看符号表

```
lfy@lfy:/mnt/d/2024fall/编译/lab/lab1/CompilingProject/lab1/lab1_2$ nm factoria
0000000000003df8 d _DYNAMIC
0000000000004000 d _GLOBAL_OFFSET_TABLE_
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000000020e8 r __FRAME_END__
0000000000002008 r __GNU_EH_FRAME_HDR
0000000000004030 D __TMC_END__
000000000000037c r __abi_tag
0000000000004030 B __bss_start
                 w __cxa_finalize@GLIBC_2.2.5
0000000000004020 D __data_start
00000000000010f0 t __do_global_dtors_aux
0000000000003df0 d __do_global_dtors_aux_fini_array_entry
0000000000004028 D __dso_handle
0000000000003de8 d __frame_dummy_init_array_entry
                 w __gmon_start__
                 U __libc_start_main@GLIBC_2.34
0000000000004030 D _edata
0000000000004038 B _end
0000000000001190 T _fini
0000000000001000 T _init
0000000000001050 T _start
0000000000004030 b completed.0
```

图 10: 查看符号表

(三) 加载和执行

执行可执行文件时，操作系统的加载器会将可执行文件的内容载入内存，设置必要的内存布局，初始化堆栈和堆，并将程序的控制权交给入口点。这部分过程是由操作系统和硬件管理的，超出了编译和链接的范围。

- 编译生成的目标文件不能直接执行，必须通过链接器与其他文件和库结合，生成最终的可执行文件。
- 反汇编目标文件和可执行文件可以看到链接后的地址分配和符号解析的不同。
- 可以通过调整链接选项比如静态链接来改变可执行文件的生成方式，从而影响反汇编后的结构。

llvmir

```
; ModuleID = 'factorial_module'
source_filename = "factorial_module"

@.format_str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; 阶乘函数 factorial
define i32 @factorial(i32 %n) {
entry:
    ; 判断 n 是否小于等于1
    %cmp = icmp sle i32 %n, 1
    br i1 %cmp, label %return_one, label %recursive_case

return_one: ; n <= 1 的情况，返回 1
    ret i32 1

recursive_case: ; 递归计算 n * factorial(n - 1)
    ; 计算 n - 1
    %n_minus_1 = sub i32 %n, 1
```

```
19
20      ; 递归调用 factorial(n - 1)
21      %recursive_result = call i32 @factorial(i32 %n_minus_1)
22
23      ; 计算 n * factorial(n - 1)
24      %result = mul i32 %n, %recursive_result
25
26      ; 返回结果
27      ret i32 %result
28  }
29
30  ; main 函数, 调用 factorial(5) 并输出结果
31  define i32 @main() {
32  entry:
33      ; 调用 factorial(5)
34      %result = call i32 @factorial(i32 5)
35
36      ; 调用 printf 函数输出结果
37      %fmt = getelementptr [4 x i8], [4 x i8]* @.format_str, i32 0, i32 0
38      call i32 (i8*, ...) @printf(i8* %fmt, i32 %result)
39
40      ; 返回 0
41      ret i32 0
42  }
43
44  ; 外部声明 printf 函数
45  declare i32 @printf(i8*, ...)
```

```
lfy@lfy:/mnt/d/2024fall/编译/lab/lab1/CompilingProject/lab1/lab1_2$ clang factorial.ll -o factorial
warning: overriding the module target triple with x86_64-pc-linux-gnu [-Woverride-module]
1 warning generated.
lfy@lfy:/mnt/d/2024fall/编译/lab/lab1/CompilingProject/lab1/lab1_2$ ./factorial
120
```

图 11: 编译运行

四、 总结

GitHub Github