



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译系统调研报告

预备工作 1——了解编译器及 LLVM IR 编程

吴帅达 宋先锋

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 9 月 17 日

摘要

本文以 GCC、LLVM 为研究对象，通过将阶乘作为源程序，用编译器的命令行选项获得各阶段输出，分析了解编译器各阶段功能。编译器是一种将高级语言源代码转换为目标代码的工具，它可以将源代码转换为可执行的机器代码，以便在目标平台上运行。

LLVM 是一个开源的编译器基础设施项目，它是一个综合的编译器工具链。LLVM 提供了一套通用的工具和库，用于开发编译器、优化器、代码生成器等。

关键字：GCC；LLVM；语言处理；编译器

目录

一、 概述	1
(一) 测试环境	1
(二) 测试的源代码	1
二、 实验分析	2
(一) 完整的编译流程	2
(二) 预处理器	2
1. 预处理器在不同编译器的区别	3
(三) 编译器	3
1. 词法分析	3
2. 语法分析	4
3. 语义分析	5
4. 中间代码生成	5
5. 代码优化	5
6. 代码生成	5
(四) 汇编器	6
(五) 链接器	7
三、 LLVM IR 编程	7
(一) 实例与解析	8
(二) LLVM IR 的结构	9
(三) 编写函数	9
四、 总结	10

一、 概述

(一) 测试环境

基于 Windows10 上 ubuntu 虚拟机提供的 linux 环境 GCC 的版本为 11.4.0

(二) 测试的源代码

阶乘

```
1  #include<iostream>
2  using namespace std;
3  #define new_i i
4  int main()
5  {
6      int i, n, f;
7      cin >> n;
8      new_i = 2; //注释
9      f = 1;
10     while (i <= n)
11     {
12         f = f * i;
13         i = i + 1;
14     }
15     cout << f << endl;
16 }
```

二、 实验分析

(一) 完整的编译流程

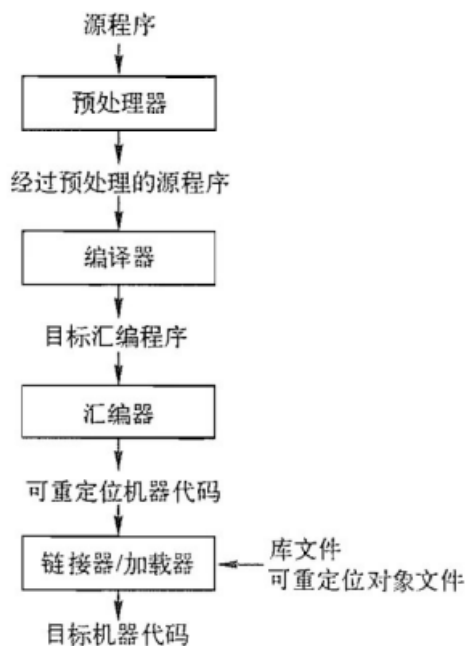


图 1: 编译的基本流程

编译流程包括预处理、编译、汇编、链接四个步骤。

预处理器 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。

编译器 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。

汇编器 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。

链接器 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链接在一起，形成真正能在机器上运行的目标机器代码。

(二) 预处理器

预处理器会执行一些预定义的指令和宏来对源代码进行操作，预处理器的作用是通过预处理的内建功能对一个资源进行等价替换，最常见的预处理器指令有：文件包含、条件编译、布局控制和宏替换 4 种。

在本实验中主要涉及：

头文件包含：将头文件转换为两万多行代码。

宏替换：将 #define 的宏命令会直接替换相应内容。

删除注释：源文件中的注释在预处理后没有出现，这是由于 GCC 将注释内容替换为空。

添加行号和文件标识

```
28638 # 2 "main.cpp"
28639 using namespace std;
28640
28641 int main()
28642 {
28643     int i, n, f;
28644     cin >> n;
28645     i = 2;
28646     f = 1;
28647     while (i <= n)
28648     {
28649         f = f * i;
28650         i = i + 1;
28651     }
28652     cout << f << endl;
28653 }
```

图 2: 预处理后的部分代码

1. 预处理器在不同编译器的区别

通过观察在 clang 和 gcc 编译器下的.i 文件, 可以得到 clang 将注释替换为空格, gcc 替换为空。

(三) 编译器

编译器将一种源语言(通常是高级语言)编写的程序转换成目标语言(通常是机器语言)的程序。其工作主要分为六大部分: 词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成。

1. 词法分析

编译器将源代码分割成一个个词法单元, 例如关键字、标识符、常数、运算符、分界符、字符串等。词法分析器通常使用正则表达式、有限自动机等方法实现。

```

the 'int' [StartOfLine] [LeadingSpace] Loc=<main.cpp:4:2>
identifier 'main' [LeadingSpace] Loc=<main.cpp:4:6>
l_paren '(' [LeadingSpace] Loc=<main.cpp:4:10>
r_paren ')' [LeadingSpace] Loc=<main.cpp:4:11>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.cpp:5:2>
int 'int' [StartOfLine] [LeadingSpace] Loc=<main.cpp:6:2>
identifier 'i' [LeadingSpace] Loc=<main.cpp:6:6>
comma ',' [LeadingSpace] Loc=<main.cpp:6:7>
identifier 'n' [LeadingSpace] Loc=<main.cpp:6:9>
comma ',' [LeadingSpace] Loc=<main.cpp:6:10>
identifier 'f' [LeadingSpace] Loc=<main.cpp:6:12>
semi ';' [LeadingSpace] Loc=<main.cpp:6:13>
identifier 'cin' [StartOfLine] [LeadingSpace] Loc=<main.cpp:7:2>
greatergreater '>>' [LeadingSpace] Loc=<main.cpp:7:6>
identifier 'n' [LeadingSpace] Loc=<main.cpp:7:9>
semi ';' [LeadingSpace] Loc=<main.cpp:7:10>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.cpp:8:2 <Spe
equal '=' [LeadingSpace] Loc=<main.cpp:8:8>
numeric_constant '2' [LeadingSpace] Loc=<main.cpp:8:10>
semi ';' [LeadingSpace] Loc=<main.cpp:8:11>
slash '/' [LeadingSpace] Loc=<main.cpp:8:12>
identifier 'zhushi' [LeadingSpace] Loc=<main.cpp:8:13>
slash '/' [LeadingSpace] Loc=<main.cpp:8:19>
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<main.cpp:9:2>
equal '=' [LeadingSpace] Loc=<main.cpp:9:4>
numeric_constant '1' [LeadingSpace] Loc=<main.cpp:9:6>
semi ';' [LeadingSpace] Loc=<main.cpp:9:7>
while 'while' [StartOfLine] [LeadingSpace] Loc=<main.cpp:10:2>
l_paren '(' [LeadingSpace] Loc=<main.cpp:10:8>
identifier 'i' [LeadingSpace] Loc=<main.cpp:10:9>
lessequal '<=' [LeadingSpace] Loc=<main.cpp:10:11>
identifier 'n' [LeadingSpace] Loc=<main.cpp:10:14>
r_paren ')' [LeadingSpace] Loc=<main.cpp:10:15>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.cpp:11:2>
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<main.cpp:12:2>

```

图 3: 词法分析

例如识别出标识符: i、n、f; 运算符: +、*、= 等。Loc=<main.cpp:6:9> 的意思是该词项在 test.c 源文件中的第 28 行第 3 个字符开始。

2. 语法分析

编译器将词法单元组织成语法树, 并检查语法是否符合语言规范, 构建抽象语法树。AST 用抽象语法的方式描绘了记号流的语法情况。在生成树时, 解析器会删除一些没必要的标识词项。语法分析器通常使用上下文无关文法、语法分析器生成器等方法实现。

```

-WhileStmt 0x19900c8 <line:10:2, line:14:2>
|-BinaryOperator 0x198ff00 <line:10:9, col:14> 'bool' '<='
| |
| |-ImplicitCastExpr 0x198fea8 <col:9> 'int' <LValueToRValue>
| | |
| | |-DeclRefExpr 0x198fe68 <col:9> 'int' lvalue Var 0x198e278 'i' 'int'
| | |
| | |-ImplicitCastExpr 0x198fec0 <col:14> 'int' <LValueToRValue>
| | |
| | |-DeclRefExpr 0x198fe88 <col:14> 'int' lvalue Var 0x198e2f8 'n' 'int'
| | |
| -CompoundStmt 0x19900a8 <line:11:2, line:14:2>
| |-BinaryOperator 0x198ffd0 <line:12:2, col:10> 'int' lvalue '='
| |
| | |-DeclRefExpr 0x198ff20 <col:2> 'int' lvalue Var 0x198e378 'f' 'int'
| | |
| | |-BinaryOperator 0x198ffb0 <col:6, col:10> 'int' '*'
| | |
| | |-ImplicitCastExpr 0x198ff80 <col:6> 'int' <LValueToRValue>
| | |
| | | |-DeclRefExpr 0x198ff40 <col:6> 'int' lvalue Var 0x198e378 'f' 'int'
| | |
| | | |-ImplicitCastExpr 0x198ff98 <col:10> 'int' <LValueToRValue>
| | |
| | | |-DeclRefExpr 0x198ff60 <col:10> 'int' lvalue Var 0x198e278 'i' 'int'
| | |
| -BinaryOperator 0x1990088 <line:13:2, col:10> 'int' lvalue '='
| |
| | |-DeclRefExpr 0x198fff0 <col:2> 'int' lvalue Var 0x198e278 'i' 'int'
| | |
| | |-BinaryOperator 0x1990068 <col:6, col:10> 'int' '+'
| | |
| | |-ImplicitCastExpr 0x1990050 <col:6> 'int' <LValueToRValue>
| | |
| | | |-DeclRefExpr 0x1990010 <col:6> 'int' lvalue Var 0x198e278 'i' 'int'
| | |
| | -IntegerLiteral 0x1990030 <col:10> 'int' 1

```

图 4: 语法分析

图四显示的是 while 循环语句部分对应的语法分析，前 6 行为 while 判断条件，其余部分为两个执行语句。

[2]

3. 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，进行类型检查等。

4. 中间代码生成

完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。可以使用下面的命令生成 LLVM IR 中间代码：clang -S -emit-llvm main.c

LLVM IR 是 LLVM 架构中一个重要的组成成分，编译器前端将抽象语法树转变为 LLVM IR，而编译器后端则根据 LLVM IR 进行优化，生成可执行程序。LLVM IR 包含了源代码中的基本块、变量、函数以及对应的操作指令等。通过将源代码转换为 LLVM IR，编译器可以利用 LLVM 提供的优化器对中间表示进行各种优化操作，以提高程序的性能和效率。

5. 代码优化

进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。在 LLVM 官网对所有 pass 的分类 3 中，共分为三种：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

-O1:

这两个命令的效果是一样的，目的都是在不影响编译速度的前提下，尽量采用一些优化算法降低代码大小和可执行代码的运行速度。

-O2

该优化选项会牺牲部分编译速度，除了执行-O1 所执行的所有优化之外，还会采用几乎所有的目标配置支持的优化算法，用以提高目标代码的运行速度。

-O3

该选项除了执行-O2 所有的优化选项之外，一般都是采取很多向量化算法，提高代码的并行执行程度，利用现代 CPU 中的流水线，Cache 等。

[3]

6. 代码生成

以中间表示形式作为输入，将其映射到目标语言。我们使用以下命令生成了 LLVM 的目标代码：llc main.ll -o main.S

```

1 | .text
2 | .file "main.cpp"
3 | .section .text.startup,"ax",@progbits
4 | .p2align 4, 0x90 # -- Begin function __cxx_global_var_init
5 | .type __cxx_global_var_init,@function
6 | __cxx_global_var_init: # @__cxx_global_var_init
7 | .cfi_startproc
8 | # %bb.0:
9 | pushq %rbp
10 | .cfi_def_cfa_offset 16
11 | .cfi_offset %rbp, -16
12 | movq %rsp, %rbp
13 | .cfi_def_cfa_register %rbp
14 | movl $ZStL8__ioinit, %edi
15 | callq _ZNSt8ios_base4InitC1Ev
16 | movl $ZNSt8ios_base4InitD1Ev, %edi
17 | movl $ZStL8__ioinit, %esi
18 | movl $_dso_handle, %edx
19 | callq __cxa_atexit
20 | popq %rbp
21 | .cfi_def_cfa %rsp, 8
22 | retq
23 | .Lfunc_end0:
24 | .size __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
25 | .cfi_endproc
26 | # -- End function
27 | .text
28 | .globl main # -- Begin function main
29 | .p2align 4, 0x90
30 | .type main,@function
31 | main: # @main
32 | .cfi_startproc
33 | # %bb.0:
34 | pushq %rbp
35 | .cfi_def_cfa_offset 16
36 | .cfi_offset %rbp, -16
37 | movq %rsp, %rbp
38 | .cfi_def_cfa_register %rbp
39 | subq $16, %rsp

```

图 5: LLVM 生成目标代码

(四) 汇编器

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。汇编器将汇编语言程序转换成目标文件（object file）。目标文件通常包含以下 6 个不同的部分：

目标文件头：描述目标文件其他部分的大小和位置。

代码段：包含机器语言代码。

静态数据段：包含在程序生命周期内分配的数据

重定位信息：标记了一些在程序加载进内存时依赖于绝对地址的指令和数据。

符号表：包含未定义的剩余标记，如外部引用。

调试信息：包含一份说明目标模块如何编译的简明描述，这样，调试器能够将机器指令关联到 C 源文件，并时数据结构也变得可读。


```

1
2 main.o: 文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8   0: 55                push    %rbp
9   1: 48 89 e5          mov     %rsp,%rbp
10  4: 48 83 ec 10       sub     $0x10,%rsp
11  8: c7 45 f0 00 00 00 movl    $0x0,-0x10(%rbp)
12  f: 48 bf 00 00 00 00 movabs  $0x0,%rdi
13 16: 00 00 00
14 19: 48 8d 75 f4       lea     -0xc(%rbp),%rsi
15 1d: e8 00 00 00 00    callq  22 <main+0x22>
16 22: c7 45 fc 02 00 00 movl    $0x2,-0x4(%rbp)
17 29: c7 45 f8 01 00 00 movl    $0x1,-0x8(%rbp)
18 30: 8b 45 fc          mov     -0x4(%rbp),%eax
19 33: 3b 45 f4          cmp     -0xc(%rbp),%eax
20 36: 7f 15            jg      4d <main+0x4d>
21 38: 8b 45 f8          mov     -0x8(%rbp),%eax
22 3b: 0f af 45 fc       imul    -0x4(%rbp),%eax
23 3f: 89 45 f8          mov     %eax,-0x8(%rbp)
24 42: 8b 45 fc          mov     -0x4(%rbp),%eax
25 45: 83 c0 01          add     $0x1,%eax
26 48: 89 45 fc          mov     %eax,-0x4(%rbp)
27 4b: eb e3            jmp     30 <main+0x30>
28 4d: 8b 75 f8          mov     -0x8(%rbp),%esi
29 50: 48 bf 00 00 00 00 movabs  $0x0,%rdi
30 57: 00 00 00
31 5a: e8 00 00 00 00    callq  5f <main+0x5f>
32 5f: 48 89 c7          mov     %rax,%rdi
33 62: 48 be 00 00 00 00 movabs  $0x0,%rsi
34 69: 00 00 00
35 6c: e8 00 00 00 00    callq  71 <main+0x71>

```

图 6: 反汇编代码

(五) 链接器

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。链接器：也称链接编译器。它是一个系统程序，把各个独立汇编的机器语言程序组合起来并且解决所有未定义的标记，最后生成可执行文件。

可执行文件：一个具有目标文件格式的功能程序，不包含未解决的引用。它可以包含符号表和调试信息。

链接器的工作分 3 个步骤：

- 1) 将代码和数据模块象征性地放入内存。
- 2) 决定数据和指令标签的地址。
- 3) 修补内部和外部引用。

三、 LLVM IR 编程

在开发编译器时，通常的做法是将源代码编译到某种中间表示 (Intermediate Representation, 一般称为 IR)，然后再将 IR 翻译为目标体系结构的汇编（比如 MIPS 或 X86）。LLVM IR 具有三种表示形式，这三种中间格式是完全等价的：

- 在内存中的编译中间语言（我们无法通过文件的形式得到）
- 在硬盘上存储的二进制中间语言（格式为.bc）
- 人类可读的代码语言（格式为.ll）

本节将结合实例介绍 llvm ir 的基本语法并编写 llvm ir 程序

(一) 实例与解析

```

1 int foo(int first, int second) {
2     return first + second;
3 }
4
5 int a = 5;
6
7 int main() {
8     int b = 4;
9     return foo(a, b);
10 }

```

用 clang -emit-llvm -S main.c -o main.ll -O0 命令将其转化为 llvm ir 形式。得到如下代码：

```

1     ; ModuleID = 'test.c'
2 source_filename = "test.c"
3 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
4     :16:32:64-S128"
5 target triple = "x86_64-pc-linux-gnu"
6 @a = dso_local global i32 @5, align 4
7
8 ; Function Attrs: noinline nounwind optnone uwtable
9 define dso_local i32 @foo(i32 %0, i32 %1) #0 {
10     %3 = alloca i32, align 4
11     %4 = alloca i32, align 4
12     store i32 %0, i32* %3, align 4
13     store i32 %1, i32* %4, align 4
14     %5 = load i32, i32* %3, align 4
15     %6 = load i32, i32* %4, align 4
16     %7 = add nsw i32 %5, %6
17     ret i32 %7
18 }
19
20 ; Function Attrs: noinline nounwind optnone uwtable
21 define dso_local i32 @main() #0 {
22     %1 = alloca i32, align 4
23     %2 = alloca i32, align 4
24     store i32 0, i32* %1, align 4
25     store i32 4, i32* %2, align 4
26     %3 = load i32, i32* @a, align 4
27     %4 = load i32, i32* %2, align 4
28     %5 = call i32 @foo(i32 %3, i32 %4)
29     ret i32 %5
30 }
31

```

```

32 attributes #0 = { noline nounwind optnone uwtable "correctly-rounded-divide
    -sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="all"
    "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-
    math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-
    signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector
    -buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+
    mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
33
34 !llvm.module.flags = !{!0}
35 !llvm.ident = !{!1}
36
37 !0 = !{i32 1, !"wchar_size", i32 4}
38 !1 = !{"clang version 10.0.0-4ubuntu1 "}

```

接下来解析每行的语法

- ModuleID,source_filename 是文件名
- target datalayout,triple 是文件标签。
- @ 是声明全局变量, dso_local 是变量和函数的运行时抢占说明符
- define 是函数定义, % 声明临时变量, alloca 分配空间, align 表示对齐。
- store a, b 表示将 a 存入 b。
- load 表示复制、装载。a=load type, type b 表示将 b 赋值给 a
- call 表示使用函数 foo
- 要注意的是, 临时变量的作用域是函数。

(二) LLVM IR 的结构

LLVM IR 文件的基本单位称为 module, 一个 module 中可以拥有多个顶层实体, 比如 function 和 global variavle, 一个 function define 中至少有一个 basicblock, 每个 basicblock 中有若干 instruction, 并且都以 terminator instruction 结尾。

(三) 编写函数

由上一小节, 可以仿写出一段简单程序:

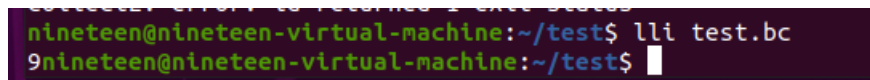
```

1      @.str = private unnamed_addr constant [3 x i8] c"%d\00"
2
3  define dso_local i32 @main() #0 {
4      %1 = alloca i32
5      %2 = alloca i32
6      %3 = alloca i32
7      %4 = alloca i32
8      store i32 0, i32* %1
9      store i32 4, i32* %2
10     store i32 5, i32* %3

```

```
11 %5 = load i32, i32* %2
12 %6 = load i32, i32* %3
13 %7 = add nsw i32 %5, %6
14 store i32 %7, i32* %4
15 %8 = load i32, i32* %4
16 %9 = call i32 @printf(i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x
    i8]* @.str, i64 0, i64 0), i32 %8)
17 ret i32 0
18 }
19
20 declare dso_local i32 @printf(i8*, ...)
```

执行结果如下：



```
nineteen@nineteen-virtual-machine:~/test$ lli test.bc
9nineteen@nineteen-virtual-machine:~/test$
```

图 7: 执行结果

[1]

四、 总结

本文以 GCC、LLVM 为研究对象，通过将阶乘作为源程序，用编译器的命令行选项获得各阶段输出，对编译器各阶段的功能有了更深入的理解，知道了编译器主要有词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成过程。研究了 LLVM IR 的基本语法，并采用了 LLVM IR 语言来编写程序。

小组分工：

第二部分：实验分析由宋先锋 2113427 完成

第三部分：LLVM IR 编程由吴帅达 2110301 完成

参考文献

- [1] minisysy 编译实验. <https://buaa-se-compiling.github.io/miniSysY-tutorial/>.
- [2] 编译原理概论. <https://blog.csdn.net/cprimesplus/article/details/105724168>.
- [3] 编译器、汇编器、链接器. https://blog.csdn.net/qq_39918677/article/details/120372053.

NIKU