



南開大學
Nankai University

计算机学院
编译原理实验报告

了解编译器及 LLVM IR 编程

姓名：谢子涵
学号：2010507
专业：计算机科学与技术

2022 年 10 月 2 日

摘 要

本实验以 GCC（包含部分 LLVM/Clang）为研究对象，较为深入地探究了语言处理系统的完整的工作过程。主要包括预处理器、编译器、汇编器、链接器/加载器做的具体工作，其中对编译器工作的 6 个阶段（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）进行了较为详细的分析。另外，还与张书睿同学合作，利用 LLVM IR 中间语言对 SysY 编译器的多个语言特性（包括函数、变量/常量声明、语句、表达式）进行了实现，并进行了验证。

关键字：预处理器，编译器，汇编器，链接器，LLVM IR

目录

1 个人工作部分	3
1.1 完整的编译过程	3
1.2 测试 c++ 程序例子	3
1.3 预处理器	4
1.4 编译器	4
1.4.1 词法分析	4
1.4.2 语法分析	5
1.4.3 语义分析	5
1.4.4 中间代码生成	5
1.4.5 代码优化	7
1.4.6 目标代码生成	7
1.5 汇编器	7
1.6 链接器/加载器	8
2 组队工作部分	9
2.1 分工	9
2.2 个人完成部分	9
2.2.1 分支及循环语句	9
2.2.2 其余部分	11
2.2.3 结果验证	13
3 Github 代码链接	13

1 个人工作部分

1.1 完整的编译过程

完整的编译过程：

1. 源程序经过预处理器，处理源代码中的预编译指令，完成宏的替换、头文件的包含等工作。
2. 经过预处理后的源程序通过编译器，由高级语言被翻译成汇编语言，得到目标汇编程序。在这个过程中编译器做的事包括：词法分析（进行分词）、语法分析（构建语法树）、语义分析（进行类型检查和类型转换）、中间代码生成、代码优化和目标代码生成（生成汇编代码）。
3. 目标汇编程序进入汇编器，由汇编语言翻译为机器语言，并被打包成可重定位目标程序。
4. 链接器将可重定位机器码和相应的一些目标文件以及库文件连接在一起。加载器修改可重定位机器码中的地址（加上加载地址），装载到指定内存位置。最终，得到真正可以运行的目标机器代码。

1.2 测试 c++ 程序例子

采用一个计算斐波拉契数列的前 n 项的 c++ 程序 fibo.cpp 来对完整的编译过程进行探究。该程序输入正整数 n ，输出 0 以及斐波拉契数列的前 n 项。

```
1  #include<iostream>
2  using namespace std;
3  #define OUT cout
4  #define MYADD(a,b) (a+b)
5  int globe_n;
6  int myplus(int x,int y){
7      return x+y;
8  }
9  int main()           //斐波拉契数列
10 {
11     int a, b, i, t;
12     a = 0; b = 1;i = 1;
13     const int add=1;
14     cin >> globe_n;
15     OUT << a << endl;
16     OUT << b << endl;
17     while (i < globe_n){
18         t = b;
19         b = myplus(a,b);
20         OUT << b << endl; //cout<<b<<endl
21         a = t;
22         i = MYADD(i,add); //i=i+add
23     }
24     return 0;
25 }
```

1.3 预处理器

预处理器根据以 # 开始的预编译指令，完成在实际编译之前所需的预处理工作。如宏的替换、头文件的包含等工作。通过命令 `g++ fibo.cpp -E -o fibo.i`，获取 `fibo.cpp` 经过预处理后的文件 `fibo.i`（见文末 Github 链接）。对比分析预处理前后的程序，可以看出：

1. 头文件的包含。源程序中的头文件包含指令 `#include<iostream>` 在预处理后的程序中被替换为该文件对应的内容。并且进行的是简单的文本替换。

2. 宏的替换。如图1.1所示，源程序中通过 `#define OUT cout` 定义的宏 `OUT` 在预处理后的程序中全都被替换成了 `cout`。并且经过简单的测试，可以发现宏替换进行的是简单的文本替换，会将程序中所有的与宏名相同的字符串替换为其实参。

3. 带参数的宏的替换。源程序中通过 `#define MYADD(a,b) (a+b)` 定义了带参数宏。如图1.1所示，源程序中的 `MYADD(i,add)` 在预处理后的程序中被替换成了 `(i+add)`。这同样也是简单的文本替换，因此最好加上括号。

4. 其他处理。如图1.1所示，源程序中的注释在预处理后的程序中被删除了。并且预处理后的程序中增加了行号和文件名标识。

```
28660      b = myplus(a,b);
28661      cout << b << endl;
28662      a = t;
28663      i = (i+add);
```

图 1.1: 预处理后的部分程序

1.4 编译器

经过预处理后的源程序通过编译器，由高级语言被翻译成汇编语言，得到目标汇编程序。在这个过程中编译器做的事包括：词法分析（进行分词）、语法分析（构建语法树）、语义分析（进行类型检查和类型转换）、中间代码生成、代码优化和目标代码生成（生成汇编代码）。

1.4.1 词法分析

词法分析将源程序切分为基本组成单元——单词，输入为字符流，输出为单词序列。由于头文件代码太多，暂时将头文件包含语句注释掉。如图1.2所示，通过命令 `clang -E -Xclang -dump-tokens fibo.cpp` 得到了词法分析后的单词序列。可以看出，词法分析器从左至右地对源程序进行扫描，按照语言的词法规则识别各类单词，并产生相应单词的属性字，还标出了该单词在源程序中的位置。结合后续语法分析结果可以看出，词法分析器通常不会关心单词之间的关系，举例来说：尽管没有包含头文件 `iostream`，词法分析器仍然会将 `cin`、`cout` 等语句单词化，而不会报错。另外，词法分析器能够将括号识别为单词，但并不保证括号是否匹配。

```
● zihan@ubuntu:~/workspace/compile_h1$ clang -E -Xclang -dump-tokens fibo.cpp
int 'int' [StartOfLine] Loc=<fibo.cpp:5:1>
identifier 'globe_n' [LeadingSpace] Loc=<fibo.cpp:5:5>
semi ';' Loc=<fibo.cpp:5:12>
int 'int' [StartOfLine] Loc=<fibo.cpp:6:1>
identifier 'myplus' [LeadingSpace] Loc=<fibo.cpp:6:5>
l_paren '(' Loc=<fibo.cpp:6:11>
int 'int' Loc=<fibo.cpp:6:12>
identifier 'x' [LeadingSpace] Loc=<fibo.cpp:6:16>
comma ',' Loc=<fibo.cpp:6:17>
int 'int' Loc=<fibo.cpp:6:18>
identifier 'y' [LeadingSpace] Loc=<fibo.cpp:6:22>
r_paren ')' Loc=<fibo.cpp:6:23>
```

图 1.2: 词法分析后的部分 token 序列

1.4.2 语法分析

语法分析在词法分析的基础上抽象出语法概念，体现出程序要干什么。语法分析器以词法分析出的单词流作为输入，按照源语言的语法规则，在词法分析的基础上将单词序列组合成各类语法短语，输出是单词间的层次关系——语法树（从词法分析的结果中识别出相应的语法范畴），同时还会进行语法检查，判断源程序在结构上是否正确。如1.3所示，展示了函数 myplus 部分对应的 AST 输出。可以看出，这描述了一个名字为 myplus，参数为两个 int 型变量 x 和 y，返回值为 int 型变量，返回内容是 x+y 的函数。

```

-FunctionDecl 0x1090630 <line:6:1, line:9:1> line:6:5 used myplus 'int (int, int)'
| -ParmVarDecl 0x10904d0 <col:12, col:16> col:16 used x 'int'
| -ParmVarDecl 0x1090550 <col:18, col:22> col:22 used y 'int'
| -CompoundStmt 0x1090780 <line:7:1, line:9:1>
| | -ReturnStmt 0x1090770 <line:8:5, col:14>
| | | -BinaryOperator 0x1090750 <col:12, col:14> 'int' '+'
| | | | -ImplicitCastExpr 0x1090720 <col:12> 'int' <LValueToRValue>
| | | | | -DeclRefExpr 0x10906e0 <col:12> 'int' lvalue ParmVar 0x10904d0 'x' 'int'
| | | | | -ImplicitCastExpr 0x1090738 <col:14> 'int' <LValueToRValue>
| | | | | -DeclRefExpr 0x1090700 <col:14> 'int' lvalue ParmVar 0x1090550 'y' 'int'

```

图 1.3: 语法分析后的部分 AST 输出

由于头文件代码太多，暂时将头文件包含语句注释掉。发现语法分析结果产生了如图1.4所示的报错。虽然仍然输出了 AST，但该 AST 不包括头文件的内容，也不包括 cin、cout 等语句的内容。这说明语法分析器会关心单词之间的关系，进行了语法检查。

```

zihan@ubuntu:~/workspace/compile_h1$ clang -E -Xclang -ast-dump fibo.cpp
fibo.cpp:17:5: error: use of undeclared identifier 'cin'
    cin >> globe_n;
    ^
fibo.cpp:18:5: error: use of undeclared identifier 'cout'
    OUT << a << endl;
    ^

```

图 1.4: 语法分析对 cin、cout 的报错

1.4.3 语义分析

语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，审查源程序有无语义错误，为代码生成阶段收集类型信息，进行类型审查以及类型转换。语义分析会审查语法关系中的每一个组成成分是否符合运算符对其的要求，并进行需要进行的类型转换。当不符合语言规范时，编译程序应报告错误。

1.4.4 中间代码生成

为了翻译成目标语言（汇编语言），首先要将程序转换为中间代码。中间代码是一种结构简单、含义明确的记号系统，由其构成的程序是一种虚拟机程序，并不是实际存在的目标程序代码。中间代码的复杂性介于源程序语言和机器语言之间，容易被翻译为目标代码。另外，还可以在中间代码一级进行与机器无关的优化工作使得代码优化比较容易实现。

采用 -fdump-tree-all-graph 和 -fdump-rtl-all-graph 两个 g++ flag 获得中间代码生成的多阶段的输出，并观察各阶段处理中的控制流图 (CFG) 及其变化。若采用 -fdump-tree-all-graph, 图1.5为 fibo.cpp.012t.cfg.dot 生成的 CFG, 图1.6为 fibo.cpp.231t.optimized.dot 生成的 CFG。对比可以发现，经过多个阶段后，CFG 的内容发生了变化。由于 fdump-rtl-all-graph 生成的 CFG 截图后很模糊，此处不再展示结果。

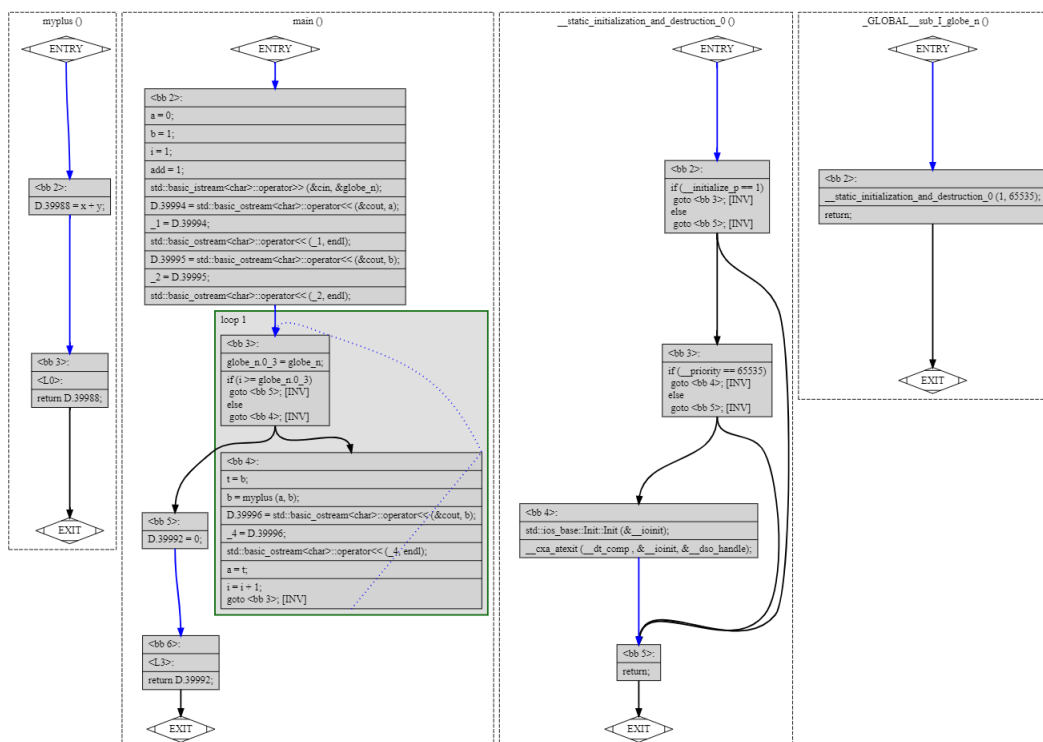


图 1.5: 中间代码生成过程中的控制流图 1

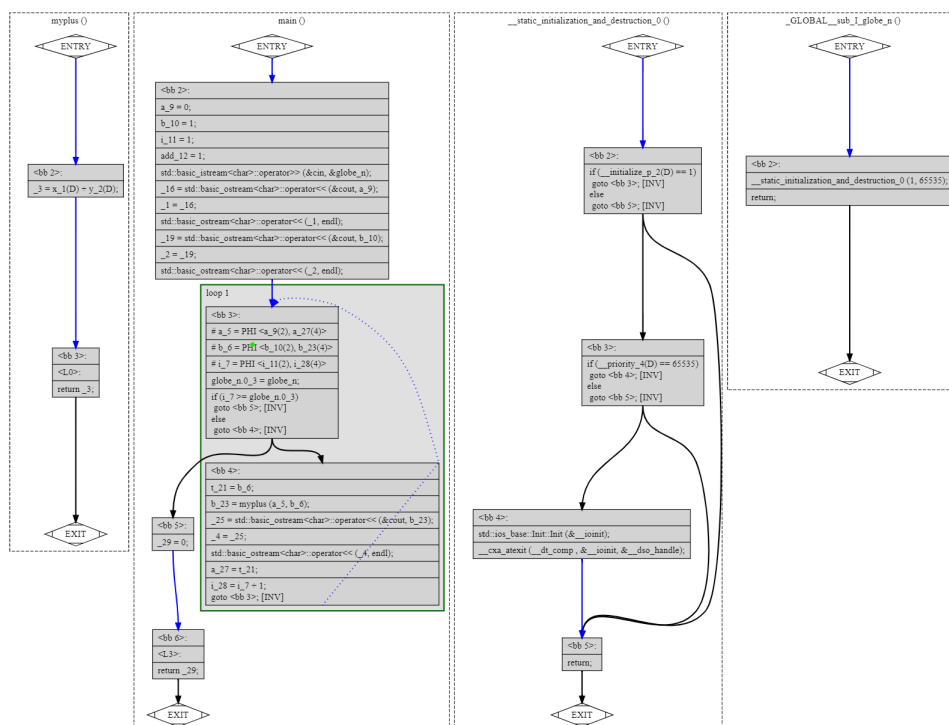


图 1.6: 中间代码生成过程中的控制流图 2

1.4.5 代码优化

在中间代码一级，可以采用更通用的优化技术，进行机器无关的代码优化，从而使最终生成的目标代码运行时间更短、占用空间更小。LLVM 提供的优化特性被作为 Passes 实现，它们遍历程序的某些部分以收集信息或转换程序。所有的 pass 分为三类：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。Utility Passes 提供了一些实用程序。

采用指令 `opt -O1 -S fibo.ll -time-passes`，可以显示出所有的优化和最终的结果。如图1.7所示，可以看到各个 pass 的执行时间。

```

=====
... Pass execution timing report ...
=====
Total Execution Time: 0.0156 seconds (0.0418 wall clock)

--User Time-- --System Time-- --User+System-- --Wall Time-- -- Name --
0.0007 ( 6.7%) 0.0002 ( 3.2%) 0.0009 ( 5.5%) 0.0137 (32.8%) Simplify the CFG
0.0048 (46.1%) 0.0000 ( 0.0%) 0.0048 (30.4%) 0.0096 (23.0%) Print Module IR
0.0000 ( 0.1%) 0.0010 (17.9%) 0.0010 ( 6.1%) 0.0048 (11.6%) Combine redundant instructions
0.0000 ( 0.0%) 0.0015 (27.9%) 0.0015 ( 9.5%) 0.0036 ( 8.6%) Early CSE
0.0000 ( 0.0%) 0.0008 (14.2%) 0.0008 ( 4.8%) 0.0012 ( 2.9%) SROA
0.0008 ( 7.7%) 0.0000 ( 0.0%) 0.0008 ( 5.1%) 0.0012 ( 2.8%) Induction Variable Simplification
0.0001 ( 0.9%) 0.0001 ( 2.4%) 0.0002 ( 1.4%) 0.0011 ( 2.7%) Globals Alias Analysis
0.0001 ( 0.5%) 0.0001 ( 1.4%) 0.0001 ( 0.8%) 0.0008 ( 2.0%) PGOMemOPSize
0.0002 ( 1.5%) 0.0002 ( 4.3%) 0.0004 ( 2.4%) 0.0004 ( 0.9%) Rotate Loops
0.0003 ( 3.1%) 0.0000 ( 0.1%) 0.0003 ( 2.1%) 0.0003 ( 0.8%) Combine redundant instructions #4
0.0002 ( 1.5%) 0.0002 ( 3.0%) 0.0003 ( 2.0%) 0.0003 ( 0.8%) Inliner for always_inline functions
0.0001 ( 1.3%) 0.0002 ( 2.9%) 0.0003 ( 1.8%) 0.0003 ( 0.7%) Deduce function attributes
0.0003 ( 2.5%) 0.0000 ( 0.0%) 0.0003 ( 1.6%) 0.0003 ( 0.6%) Loop Invariant Code Motion
0.0002 ( 1.7%) 0.0000 ( 0.0%) 0.0002 ( 1.2%) 0.0002 ( 0.4%) Loop Vectorization
0.0000 ( 0.2%) 0.0001 ( 2.3%) 0.0001 ( 0.9%) 0.0001 ( 0.4%) Module Verifier

```

图 1.7: 各个 Pass 执行时间

1.4.6 目标代码生成

该阶段将中间代码转换成目标代码，即相应平台等价的汇编码。这个过程主要包括指令集的选择和寄存器的分配。可采用指令 `g++ fibo.i -S -o fibo_x86.S` 生成 x86 格式的目标代码；采用指令 `arm-linux-gnueabi-g++ fibo.cpp -S -o fibo_arm.S` 生成 arm 格式目标代码；还可以采用指令 `llc fibo.ll -o fibo_fromll.S` 将前面步骤中生成的 llvm 代码转换成目标代码。生成的各个文件均见文末 github 链接。

1.5 汇编器

汇编器将编译器生成的汇编代码翻译为目标机器指令，最终生成可重定位的机器代码。汇编语言以处理器指令系统为基础，采用助记符表达指令操作码，采用标识符表示指令操作数。使用汇编语言编写的程序，机器不能直接识别，需要汇编器将汇编语言翻译成机器语言。而在机器代码中只能用内存地址来表示变量（不像汇编代码中能用符号表示），因此，需要给每一个变量分配地址。

编译器后端的代码生成常分为四个步骤：指令选择、寄存器分配、指令调度和指令编码。汇编器生成可重定位机器代码的过程可以通过两次扫描汇编实现。每次读取一遍输入文件，第一次扫描时，识别标识符，将其存入符号表并指定相对内存地址；第二次扫描时，将指令转换为对应的二进制序列，同时将标识符转换为第一次扫描时指定的相对内存地址。这样就产生了可重定位的机器代码。可以看出，这个可重定位机器代码中的地址为相对地址（相对于这段程序起始地址的偏移量），从而使得该程序之后可以在不同的位置运行。

以 x86 格式汇编代码为例，可以通过指令 `gcc -O0 -c -o fibo_x86.o fibo_x86.S` 生成其对应的可重定位机器码。可以用反编译工具 `nm` 查看二进制码中名字，用 `objdump` 查看二进制码中的段，并与经

过链接器后的结果进行对比分析。如图1.8和1.10所示，是两个阶段的 `myplus` 函数部分的反汇编结果，可以看出，经过链接器后的地址整体比经过汇编器后的地址增加了 `11c9`。这说明汇编器产生的可重定位机器码采用的是相对地址。当然，从图1.9中 `nm` 产生的结果也可以看出这一点。另外，通过 `nm` 和 `objdump` 的结果可以看出，汇编器产生的可重定位机器码的结果只针对了这个程序本身，并未包括库文件。

```

7  ▾ 0000000000000000 <_Z6myplusii>
8      0: f3 0f 1e fa                endbr64
9      4: 55                          push    %rbp
10     5: 48 89 e5                    mov     %rsp,%rbp
11     8: 89 7d fc                    mov     %edi,-0x4(%rbp)
12     b: 89 75 f8                    mov     %esi,-0x8(%rbp)
13     e: 8b 55 fc                    mov     -0x4(%rbp),%edx
14    11: 8b 45 f8                    mov     -0x8(%rbp),%eax
15    14: 01 d0                      add     %edx,%eax
16    16: 5d                          pop     %rbp
17    17: c3                          retq

```

图 1.8: objdump 查看汇编器产生的可重定位机器码中的片段













```
workspace > compile_h1 >  fibo-nm-obj.txt
1       U __cxa_atexit
2       U __dso_handle
3       U _GLOBAL_OFFSET_TABLE_
4      0000000000000148 t _GLOBAL_sub_I_globe_n
5      0000000000000000 B globe_n
6      0000000000000018 T main
7      0000000000000fb t _Z41__static_initialization_and_destruction_0ii
8      0000000000000000 T _Z6mplusplusi
9       U _ZN5sirsEri
10      U _ZN5solsEi
11      U _ZN5solsEPFRSoS_E
12      U _ZN5t8ios_base4InitC1Ev
13      U _ZN5t8ios_base4InitD1Ev
14      U _ZSt3cin
15      U _ZSt4cout
16      U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamM
17     0000000000000000 r _ZStL19piecewise_construct
18     0000000000000004 b _ZStL8_ioinit
19
```

图 1.9: nm 查看汇编器产生的可重定位机器码中的名字

1.6 链接器/加载器

可重定位的机器代码需要和其他可重定位的目标文件以及库文件链接到一起，才能形成真正在机器上运行的代码。链接器会将多个可重定位机器码文件组合为单一程序。同时，加载器能修改可重定位机器码中的地址（加上加载地址），装载到指定内存位置。这样才能将可重定位机器代码转换为目标机器代码（可执行文件）。

程序调用了一些库函数，若为动态链接库，则其地址也是不确定的。在可重定位机器码中有一些符号表，指出程序中的某些地址对应的是库函数的地址（哪个库、哪个函数都明确指出）。当 Loader 加载后，链接器就会检查这个表，将里面函数的调用正确地用指向当前动态链接库里该函数对应的地址来填写，这样在运行时就会真正调用这个函数。

用 `objdump` 和 `nm` 对可执行文件反汇编的结果如图1.10、1.11所示，与图1.8和1.9中结果进行对比，可以看出：可执行程序 `nm` 的结果中的名字数量增加，并且相同名字的地址也增加了；可执行程序中 `objdump` 的结果中内容大量增加（加入了库文件等内容），并且指令的地址也增加了 11c9。这说明在该阶段进行了可重定位的机器代码与其他可重定位的目标文件以及库文件的链接，并形成真正在机器上运行的代码。

```

150 0000000000011c9 <_Z6myplusii>:
151 11c9: f3 0f 1e fa      endbr64
152 11cd: 55              push    %rbp
153 11ce: 48 89 e5        mov     %rsp,%rbp
154 11d1: 89 7d fc        mov     %edi,-0x4(%rbp)
155 11d4: 89 75 f8        mov     %esi,-0x8(%rbp)
156 11d7: 8b 55 fc        mov     -0x4(%rbp),%edx
157 11da: 8b 45 f8        mov     -0x8(%rbp),%eax
158 11dd: 01 d0          add     %edx,%eax
159 11df: 5d              pop     %rbp
160 11e0: c3              retq

```

图 1.10: objdump 查看链接器产生的可执行文件中的片段

```

workspace > compile_h1 > fibo-nm-exe.txt
31 | | | | | U __libc_start_main@@GLIBC_2.2.5
32 | 00000000000011e1 T main
33 | 0000000000001140 t register_tm_clones
34 | 00000000000010e0 T _start
35 | 0000000000004010 D __TMC_END__
36 | 00000000000012c4 t _Z41__static_initialization_and_destruction_0ii
37 | 00000000000011c9 T _Z6myplusii
38 | | | | | U _ZN5SirsERi@@GLIBCXX_3.4
39 | | | | | U _ZN5SolsEi@@GLIBCXX_3.4
40 | | | | | U _ZN5SolsEPFRSoS_E@@GLIBCXX_3.4
41 | | | | | U _ZN5St8ios_base4InitC1Ev@@GLIBCXX_3.4
42 | | | | | U _ZN5St8ios_base4InitD1Ev@@GLIBCXX_3.4
43 | 0000000000004160 B _ZSt3cin@@GLIBCXX_3.4
44 | 0000000000004040 B _ZSt4cout@@GLIBCXX_3.4
45 | | | | | U _ZSt4endlC1St11char_traitsIcEERSt13basic_ostreamI
46 | 0000000000002004 r _ZStL19piecewise_construct
47 | 0000000000004280 b _ZStL8__ioinit
48 |

```

图 1.11: nm 查看链接器产生的可执行文件中的名字

2 组队工作部分

2.1 分工

该部分由我和张书睿同学合作完成，具体分工为：

张书睿：负责函数和变量/常量声明部分。函数部分包括函数的定义、类型、参数表、返回等内容。变量/常量声明部分包括不同数据类型、局部/全局的变量/常量的定义与初始化。

谢子涵：负责语句和表达式部分。语句部分包括赋值语句、表达式语句、语句块、if 语句、while 语句、continue 语句、return 语句。表达式部分包括基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&）。

2.2 个人完成部分

2.2.1 分支及循环语句

采用了 LLVM IR 中间代码实现了 while 循环以及 if-else 语句。其中包含了关系运算、逻辑运算、continue 语句以及单目运算符 ++ 和 -- 的实现。实现的具体功能如以下 c 代码片段所示。a 的初始值为 4，b 的初始值为 0。可以通过在 LLVM IR 代码中该片段结束后调用 printf 函数来输出 a 和 b 的值，从而对编写的 LLVM IR 代码进行验证。理论上，会执行 4 次 while 循环，其中有两次进入 if 语句块执行 continue 语句，有两次进入 else 语句块执行 b++。因此执行完该部分代码后，a 的值应为 0，b 的值应为 2。

```

1  while(a>0){
2      a--;
3      if(a>1&&b<3){
4          continue;
5      }else
6          b++;
7  }

```

该部分对应的 LLVM IR 代码如下所示，具体详细解释见代码中的注释。

```

1  br label %8    ; 此处 br 为无条件分支
2
3  8:    ;while 语句起始位置
4      %9 = load i32, i32* %2, align 4    ; 此处 br 为无条件分支
5      %10 = icmp sgt i32 %9, 0
6      ;sgt: 有符号大于。当%9 (即 a) 大于 0 时, icmp 的结果%10(i1 类型) 为 1, 否则为 0
7      br i1 %10, label %11, label %24
8      ; 此处 br 为有条件分支, 当 i1 类型的变量%10 (icmp 的结果) 的值为真时, 执行 label%11,
9      ; 进入 while 循环; 否则执行 label%21 退出 while 循环
10
11  11:
12      %12 = load i32, i32* %2, align 4    ; 从%2 中取出 a 的值存在%12 中
13      %13 = add nsw i32 %12, -1          ; 将%12 加上-1 并存入%13, 即 a--
14      store i32 %13, i32* %2, align 4    ; 将%13 中的 a--后的值存回%2, 完成 a 的更新
15      ;if 语句起始位置
16      %14 = load i32, i32* %2, align 4    ; 从%2 中取出 a 的值存在%14 中
17      %15 = icmp sgt i32 %14, 1
18      ; 当%14 (即 a) 大于 1 时, icmp 的结果%15(i1 类型) 为 1, 否则为 0
19      br i1 %15, label %16, label %20
20      ; 此处 br 为有条件分支, 当 i1 类型的变量%15 (icmp 的结果) 的值为真时, 执行 label%16,
21      ; 进行关系运算 <<> 的右部表达式的计算; 否则执行 label%20 进入 else 语句
22
23  16:    ; 关系运算符 <<> 的右部表达式的判断
24      %17 = load i32, i32* %3, align 4    ; 从%3 中取出 b 的值存在%17 中
25      %18 = icmp slt i32 %17, 3
26      ;slt: 有符号小于。当%17 (即 b) 小于 3 时, icmp 的结果%18(i1 类型) 为 1, 否则为 0
27      br i1 %18, label %19, label %20
28      ; 此处 br 为有条件分支, 当 i1 类型的变量%18 (icmp 的结果) 的值为真时, 执行 label%19
29      ; 进入 if 语句块; 否则执行 label%20 进入 else 语句
30
31  19:    ;if 语句块内容

```

```

32     br label %8
33     ;if 语句块的内容为 continue, 因此直接跳转到 while 语句开始的位置, 即 label%8
34
35 20:    ;else 语句块内容
36     %21 = load i32, i32* %3, align 4    ; 从%3 中取出 b 的值存在%21 中
37     %22 = add nsw i32 %21, 1            ; 将%21 加上 1 并存入%22, 即 b++
38     store i32 %22, i32* %3, align 4    ; 将%22 中的 b++ 后的值存回%3, 完成 b 的更新
39     br label %23                        ; 退出 else 语句块
40
41 23:
42     br label %8
43     ; 本轮 while 循环结束, 无条件跳转回 while 语句起始位置 (即 label%8) 进行下一轮判断

```

2.2.2 其余部分

采用了 LLVM 中间代码实现了赋值语句、表达式语句、return 语句以及算术运算 (包括 +、-、*、/、%)。具体实现的功能如以下 c 代码片段所示。c 代码中注释标出了每个操作后理论上的值。可以在编写的 LLVM IR 程序中调用 printf 函数输出对应的值, 从而验证程序编写是否正确。

```

1  a=b+const_a;    //a=12
2  a=21-b;         //a=19
3  a=b*globe_a;    //a=10
4  a=const_ga/globe_a;    //a=1
5  a=const_ga%globe_a;    //a=4
6  globe_a=test_int(a,globe_a,f1);    //globe_a=11
7  return 0;

```

该部分对应的 LLVM IR 代码如下所示, 具体详细解释见代码中的注释。

```

1 24:    ;while 循环结束位置
2     %25 = load i32, i32* %2, align 4    ; 从%2 中取出 a 的值存在%25 中
3     %26 = call i32 @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
4     i64 0, i64 0), i32 %25)            ; 调用 printf 函数, 输出%25 中 a 的值
5     %27 = load i32, i32* %3, align 4    ; 从%3 中取出 b 的值存在%27 中
6     %28 = call i32 @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
7     i64 0, i64 0), i32 %27)            ; 调用 printf 函数, 输出%27 中 b 的值
8     %29 = load i32, i32* %3, align 4    ; 从%3 中取出 b 的值存在%29 中
9     %30 = add nsw i32 %29, 10           ; 将%29(即 b) 加上 10(即 const_a) 并存入临时寄存器%30 中
10    store i32 %30, i32* %2, align 4    ; 将%30 中的 b+const_a 的结果存入%2, 对 a 进行赋值
11    %31 = load i32, i32* %2, align 4    ; 从%2 中取出 a 的值存在%31 中
12    %32 = call i32 @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
13    i64 0, i64 0), i32 %31)            ; 调用 printf 函数, 输出%31 中 a 的值

```

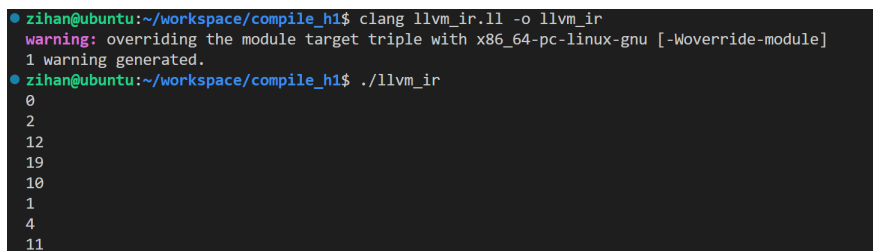
```

14  %33 = load i32, i32* %3, align 4 ; 从%3 中取出 b 的值存在%33 中
15  %34 = sub nsw i32 21, %33 ; 用 21 减去%33(即 b) 并存入%34 中
16  store i32 %34, i32* %2, align 4 ; 将%34 中的 21-b 的结果存入%2, 对 a 进行赋值
17  %35 = load i32, i32* %2, align 4 ; 从%2 中取出 a 的值存在%35 中
18  %36 = call i32 @i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
19  i64 0, i64 0), i32 %35) ; 调用 printf 函数, 输出%35 中 a 的值
20  %37 = load i32, i32* %3, align 4 ; 从%3 中取出 b 的值存在%37 中
21  %38 = load i32, i32* @globe_a, align 4 ; 取出全局变量 globe_a 的值并存入%38
22  %39 = mul nsw i32 %37, %38
23  ; 计算%37(即 b) 乘以%38(即 globe_a) 的结果并存入临时寄存器%39 中
24  store i32 %39, i32* %2, align 4 ; 将%39 中的 b*globe_a 的结果存入%2, 对 a 进行赋值
25  %40 = load i32, i32* %2, align 4 ; 从%2 中取出 a 的值存在%40 中
26  %41 = call i32 @i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
27  i64 0, i64 0), i32 %40) ; 调用 printf 函数, 输出%40 中 a 的值
28  %42 = load i32, i32* @globe_a, align 4 ; 取出全局变量 globe_a 的值并存入%42
29  %43 = sdiv i32 9, %42
30  ; 计算 9(即全局常量 const_ga) 除以%42(即 globe_a) 的结果并存入临时寄存器%43 中
31  store i32 %43, i32* %2, align 4
32  ; 将%43 中的 const_ga/globe_a 的结果存入%2, 对 a 进行赋值
33  %44 = load i32, i32* %2, align 4 ; 从%2 中取出 a 的值存在%44 中
34  %45 = call i32 @i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
35  i64 0, i64 0), i32 %44) ; 调用 printf 函数, 输出%44 中 a 的值
36  %46 = load i32, i32* @globe_a, align 4 ; 取出全局变量 globe_a 的值并存入%46
37  %47 = srem i32 9, %46
38  ; 计算 9(即全局常量 const_ga) 对%46(即 globe_a) 取模的结果并存入临时寄存器%47 中
39  store i32 %47, i32* %2, align 4
40  ; 将%47 中的 const_ga%globe_a 的结果存入%2, 对 a 进行赋值
41  %48 = load i32, i32* %2, align 4 ; 从%2 中取出 a 的值存在%48 中
42  %49 = call i32 @i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
43  i64 0, i64 0), i32 %48) ; 调用 printf 函数, 输出%48 中 a 的值
44  %50 = load i32, i32* %2, align 4 ; 从%2 中取出 a 的值存在%50 中
45  %51 = load i32, i32* @globe_a, align 4 ; 取出全局变量 globe_a 的值并存入%51
46  %52 = load float, float* %4, align 4 ; 从%4 中取出 f1 的值存在%52 中
47  %53 = call i32 @test_int(i32 %50, i32 %51, float %52)
48  ; 调用函数 test_int, 参数为 (%50,%51,%51), 即 (a,globe_a,f1), 返回值存入%53
49  store i32 %53, i32* @globe_a, align 4 ; 将%53 中返回值赋值给全局变量 globe_a
50  %54 = load i32, i32* @globe_a, align 4 ; 取出全局变量 globe_a 的值并存入%54
51  %55 = call i32 @i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
52  i64 0, i64 0), i32 %54) ; 调用 printf 函数, 输出%54 中 globe_a 的值
53  ret i32 0 ;return 0

```

2.2.3 结果验证

用 Clang 将编写的 LLVM IR 程序编译成目标程序并执行，从而进行验证。如图2.12所示，每个输出都符合上述分析的理论结果，说明 LLVM IR 程序编写正确。



```
zihan@ubuntu:~/workspace/compile_h1$ clang llvm_ir.ll -o llvm_ir
warning: overriding the module target triple with x86_64-pc-linux-gnu [-Woverride-module]
1 warning generated.
zihan@ubuntu:~/workspace/compile_h1$ ./llvm_ir
0
2
12
19
10
1
4
11
```

图 2.12: LLVM IR 程序编译并执行后结果

3 Github 代码链接

[Github 连接](#)