



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

预备工作 1

丁屹

年级：2020 级

专业：计算机科学与技术

2022 年 10 月 2 日

摘要

关键字：预处理器、编译器、汇编器、链接器、LLVM IR

目录

一、 概述	1
二、 预处理器做了什么	1
三、 编译器做了什么	2
(一) 词法分析	2
(二) 语法分析	2
(三) 语义分析	3
(四) 中间代码生成	3
(五) 代码优化	5
(六) 代码生成	6
四、 汇编器做了什么	8
五、 链接器做了什么	10
六、 LLVM IR 编程	14
(一) 变量定义	14
(二) 语句块	15
(三) 函数	16
(四) 数组定义	18
(五) 隐式转换	18
七、 总结	19

一、 概述

本次实验将以编译器 clang 为研究对象，深入地探究语言处理系统的完整工作过程：

1. 预处理器做了什么？
2. 编译器做了什么？
3. 汇编器做了什么？
4. 链接器做了什么？
5. 通过编写 LLVM IR 程序，熟悉 LLVM IR 中间语言。

以一个简单的计算斐波拉契数列的 C 源程序为例，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。

```
1  #include <stdio.h>
2  signed main() {
3      int a = 0;
4      int b = 1;
5      int i = 1;
6      int n;
7      scanf("%d%d%d", &n, &a, &b);
8      while (i < n) {
9          int t = b;
10         b = a + b;
11         printf("%d\n", b);
12         a = t;
13         i = i + 1;
14     }
15 }
```

二、 预处理器做了什么

预处理过程扫描源代码，对其进行初步的转换，产生新的源代码提供给编译器。可见预处理过程先于编译器对源代码进行处理。在 C 语言中，并没有任何内在的机制来完成如下一些功能：在编译时包含其他源文件、定义宏、根据条件决定编译时是否包含某些代码。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对源代码进行响应的转换。预处理过程还会删除程序中的注释和多余的空白字符。预处理指令是以 # 号开头的代码行。# 号必须是该行除了任何空白字符外的第一个字符。# 后是指令关键字，在关键字和 # 号之间允许存在任意个数的空白字符。整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。

对于 clang 编译器，使用命令 `clang fib.c -E -o fib.i`，即可得到预处理后文件。

观察预处理文件，发现文件长度远大于源文件，再对比 `<stdio.h>` 头文件定义内容，发现多出的部分为头文件定义的具体内容和一些编译器内部记号。没有用到的头或注释则会被删去。

三、 编译器做了什么

(一) 词法分析

将源程序转换为单词序列, 把代码切成一个个 token, 比如大小括号、等于号、还有字符串等。对于 LLVM 编译器, 通过以下命令获得 token 序列: `clang -E -Xclang -dump-tokens fib.c`, 部分输出如图1所示。

```
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<fib.c:10:5>
equal '=' [LeadingSpace] Loc=<fib.c:10:11>
identifier 'a' [LeadingSpace] Loc=<fib.c:10:13>
plus '+' [LeadingSpace] Loc=<fib.c:10:15>
identifier 'b' [LeadingSpace] Loc=<fib.c:10:17>
semi ';' Loc=<fib.c:10:18>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<fib.c:11:5>
l_paren '(' Loc=<fib.c:11:11>
string_literal '"%d\n"' Loc=<fib.c:11:12>
comma ',' Loc=<fib.c:11:18>
amp '&' [LeadingSpace] Loc=<fib.c:11:20>
identifier 'b' Loc=<fib.c:11:21>
r_paren ')' Loc=<fib.c:11:22>
semi ';' Loc=<fib.c:11:23>
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<fib.c:12:5>
equal '=' [LeadingSpace] Loc=<fib.c:12:7>
identifier 't' [LeadingSpace] Loc=<fib.c:12:9>
semi ';' Loc=<fib.c:12:10>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<fib.c:13:5>
equal '=' [LeadingSpace] Loc=<fib.c:13:7>
identifier 'i' [LeadingSpace] Loc=<fib.c:13:9>
plus '+' [LeadingSpace] Loc=<fib.c:13:11>
numeric_constant '1' [LeadingSpace] Loc=<fib.c:13:13>
semi ';' Loc=<fib.c:13:14>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<fib.c:14:3>
r_brace '}' [StartOfLine] Loc=<fib.c:15:1>
eof '' Loc=<fib.c:15:2>
```

图 1: token 序列

(二) 语法分析

语法分析, 它的任务是验证语法是否正确, 在词法分析的基础上将单词序列组合成各类此法短语, 如程序、语句、表达式等等, 然后将所有节点组成抽象语法树 (Abstract Syntax Tree AST), 语法分析程序判断程序在结构上是否正确。

对于 gcc, 可以通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出。对于 LLVM 可以通过 `clang -E -Xclang -ast-dump fib.c` 获得相应的 AST。此处我使用后者。部分输出如图2所示, 可见明确树形结构。

```

|-VarDecl 0x55ba2a1b2eb8 <col:5, col:13> col:9 used t 'int' cinit
|-ImplicitCastExpr 0x55ba2a1b2f40 <col:13> 'int' <LValueToRValue>
  |-DeclRefExpr 0x55ba2a1b2f20 <col:13> 'int' lvalue Var 0x55ba2a1b24f0 'b' 'int'
|-BinaryOperator 0x55ba2a1b3020 <line:10:5, col:17> 'int' '='
  |-DeclRefExpr 0x55ba2a1b2f70 <col:5> 'int' lvalue Var 0x55ba2a1b24f0 'b' 'int'
  |-BinaryOperator 0x55ba2a1b3000 <col:13, col:17> 'int' '+'
    |-ImplicitCastExpr 0x55ba2a1b2fd0 <col:13> 'int' <LValueToRValue>
      |-DeclRefExpr 0x55ba2a1b2f90 <col:13> 'int' lvalue Var 0x55ba2a1b2438 'a' 'int'
      |-ImplicitCastExpr 0x55ba2a1b2fe8 <col:17> 'int' <LValueToRValue>
        |-DeclRefExpr 0x55ba2a1b2fb0 <col:17> 'int' lvalue Var 0x55ba2a1b24f0 'b' 'int'
|-CallExpr 0x55ba2a1b30e8 <line:11:5, col:22> 'int'
  |-ImplicitCastExpr 0x55ba2a1b30d0 <col:5> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
  |-DeclRefExpr 0x55ba2a1b3040 <col:5> 'int (const char *, ...)' Function 0x55ba2a197298 'printf' 'int (const cha
r *, ...)'
    |-ImplicitCastExpr 0x55ba2a1b3130 <col:12> 'const char *' <NoOp>
    |-ImplicitCastExpr 0x55ba2a1b3118 <col:12> 'char *' <ArrayToPointerDecay>
      |-StringLiteral 0x55ba2a1b3060 <col:12> 'char[4]' lvalue "%d\n"
    |-UnaryOperator 0x55ba2a1b30a0 <col:20, col:21> 'int *' prefix '&' cannot overflow
    |-DeclRefExpr 0x55ba2a1b3080 <col:21> 'int' lvalue Var 0x55ba2a1b24f0 'b' 'int'
|-BinaryOperator 0x55ba2a1b3538 <line:12:5, col:9> 'int' '='
  |-DeclRefExpr 0x55ba2a1b34e0 <col:5> 'int' lvalue Var 0x55ba2a1b2438 'a' 'int'
  |-ImplicitCastExpr 0x55ba2a1b3520 <col:9> 'int' <LValueToRValue>
    |-DeclRefExpr 0x55ba2a1b3500 <col:9> 'int' lvalue Var 0x55ba2a1b2eb8 't' 'int'
|-BinaryOperator 0x55ba2a1b35f0 <line:13:5, col:13> 'int' '='
  |-DeclRefExpr 0x55ba2a1b3558 <col:5> 'int' lvalue Var 0x55ba2a1b25a8 'i' 'int'
  |-BinaryOperator 0x55ba2a1b35d0 <col:9, col:13> 'int' '+'
    |-ImplicitCastExpr 0x55ba2a1b35b8 <col:9> 'int' <LValueToRValue>
      |-DeclRefExpr 0x55ba2a1b3578 <col:9> 'int' lvalue Var 0x55ba2a1b25a8 'i' 'int'
    |-IntegerLiteral 0x55ba2a1b3598 <col:13> 'int' 1

```

图 2: 生成抽象语法树

(三) 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。

(四) 中间代码生成

完成以上步骤后，就开始生成中间代码 LLVM IR 了，代码生成器会将语法树自顶向下遍历逐步翻译成 LLVM IR，可以通过下面命令可以生成.ll 的文本文件，查看 IR 代码。

对于 GCC，可以通过 `-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。生成的 .dot 文件可以被 graphviz 可视化。

LLVM 的优化级别分别是 -O0、-O1、-O2、-O3、-Os，下面是带优化的生成中间代码 IR 的命令：`clang -O3 -S -fobjc-arc -emit-llvm 源文件路径 -o 输出文件路径`

LLVM 可以通过下面的命令生成 LLVM IR：`clang -S -emit-llvm fib.c`

```

1  ; ModuleID = 'fib.c'
2  source_filename = "fib.c"
3  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
4  target triple = "x86_64-pc-linux-gnu"
5
6  @.str = private unnamed_addr constant [7 x i8] c"%d%d%d\00", align 1
7  @.str.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
8
9  ; Function Attrs: noinline nounwind optnone sspstrong uwtable
10 define dso_local i32 @main() #0 {
11     %1 = alloca i32, align 4
12     %2 = alloca i32, align 4
13     %3 = alloca i32, align 4

```

```

14     %4 = alloca i32, align 4
15     %5 = alloca i32, align 4
16     %6 = alloca i32, align 4
17     store i32 0, i32* %1, align 4
18     store i32 0, i32* %2, align 4
19     store i32 1, i32* %3, align 4
20     store i32 1, i32* %4, align 4
21     %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([7 x i8], [7 x i8]), [7 x i8], [7 x i8])
22     br label %8
23
24 8:                                     ; preds = %12, %0
25     %9 = load i32, i32* %4, align 4
26     %10 = load i32, i32* %5, align 4
27     %11 = icmp slt i32 %9, %10
28     br i1 %11, label %12, label %22
29
30 12:                                    ; preds = %8
31     %13 = load i32, i32* %3, align 4
32     store i32 %13, i32* %6, align 4
33     %14 = load i32, i32* %2, align 4
34     %15 = load i32, i32* %3, align 4
35     %16 = add nsw i32 %14, %15
36     store i32 %16, i32* %3, align 4
37     %17 = load i32, i32* %3, align 4
38     %18 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str, [4 x i8]* @.str), [4 x i8]* @.str, [4 x i8]* @.str)
39     %19 = load i32, i32* %6, align 4
40     store i32 %19, i32* %2, align 4
41     %20 = load i32, i32* %4, align 4
42     %21 = add nsw i32 %20, 1
43     store i32 %21, i32* %4, align 4
44     br label %8, !llvm.loop !6
45
46 22:                                    ; preds = %8
47     %23 = load i32, i32* %1, align 4
48     ret i32 %23
49 }
50
51 declare i32 @__isoc99_scanf(i8* noundef, ...) #1
52
53 declare i32 @printf(i8* noundef, ...) #1
54
55 attributes #0 = { noinline nounwind optnone sspstrong uwtable "frame-pointer"="all" "min-legal-abi"="x86_64" "no-trapping-math"="true" "stack-protector-buffer-size"="1" }
56 attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="1" }
57

```

```

58  !llvm.module.flags = !{!0, !1, !2, !3, !4}
59  !llvm.ident = !{!5}
60
61  !0 = !{i32 1, !"wchar_size", i32 4}
62  !1 = !{i32 7, !"PIC Level", i32 2}
63  !2 = !{i32 7, !"PIE Level", i32 2}
64  !3 = !{i32 7, !"uwtable", i32 1}
65  !4 = !{i32 7, !"frame-pointer", i32 2}
66  !5 = !{"clang version 14.0.6"}
67  !6 = distinct !{!6, !7}
68  !7 = !{"llvm.loop.mustprogress"}
69

```

(五) 代码优化

进行与机器无关的代码优化步骤，此处通过 LLVM 现有的优化 pass 优化步骤改进中间代码，生成更好的目标代码。

pass 的分类共分为三种：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。

通过下面的命令生成每个 pass 后生成的 LLVM IR，以观察区别：

```
llc -print-before-all -print-after-all fib.ll >fib.log 2>&1
```

对输出重定向到 fib.log 中，部分生成代码对比如图3所示：

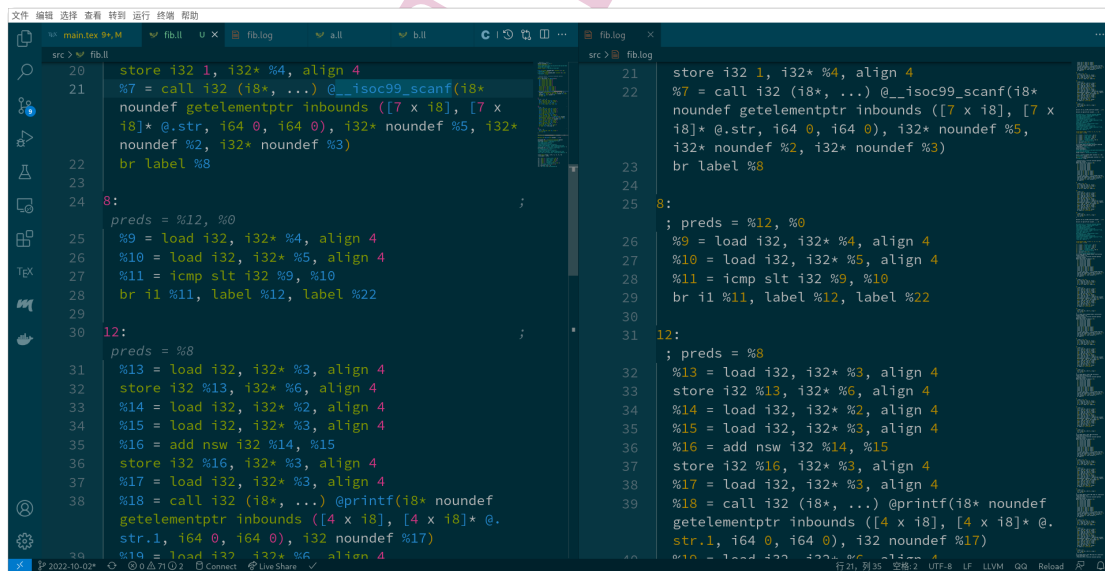


图 3: 左为 fib.ll, 右为 fib.log

对于上述程序，反汇编后代码与先前并无明显区别。

(六) 代码生成

以中间表示形式作为输入，将其映射到目标语言。此处我使用 LLVM 生成。

```

1 llc fib.ll -o fib.S # LLVM 生成目标代码

```

```

1  .text
2  .file  "fib.c"
3  .globl  main                                # -- Begin function main
4  .p2align 4, 0x90
5  .type  main,@function
6  main:                                       # @main
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 subq  $32, %rsp
15 movq  %fs:40, %rax
16 movq  %rax, -8(%rbp)
17 movl  $0, -32(%rbp)
18 movl  $0, -12(%rbp)
19 movl  $1, -16(%rbp)
20 movl  $1, -24(%rbp)
21 movabsq $.L.str, %rdi
22 leaq  -20(%rbp), %rsi
23 leaq  -12(%rbp), %rdx
24 leaq  -16(%rbp), %rcx
25 movb  $0, %al
26 callq  __isoc99_scanf@PLT
27 .LBB0_1:                                    # =>This Inner Loop Header: Depth=1
28 movl  -24(%rbp), %eax
29 cmpl  -20(%rbp), %eax
30 jge  .LBB0_3
31 # %bb.2:                                    #   in Loop: Header=BB0_1 Depth=1
32 movl  -16(%rbp), %eax
33 movl  %eax, -28(%rbp)
34 movl  -12(%rbp), %eax
35 addl  -16(%rbp), %eax
36 movl  %eax, -16(%rbp)
37 movl  -16(%rbp), %esi
38 movabsq $.L.str.1, %rdi

```



```

39     movb  $0, %al
40     callq printf@PLT
41     movl  -28(%rbp), %eax
42     movl  %eax, -12(%rbp)
43     movl  -24(%rbp), %eax
44     addl  $1, %eax
45     movl  %eax, -24(%rbp)
46     jmp   .LBB0_1
47 .LBB0_3:
48     movl  -32(%rbp), %eax
49     movq   %fs:40, %rcx
50     movq  -8(%rbp), %rdx
51     cmpq  %rdx, %rcx
52     jne   .LBB0_5
53 # %bb.4:                                     # %SP_return
54     addq  $32, %rsp
55     popq  %rbp
56     .cfi_def_cfa %rsp, 8
57     retq
58 .LBB0_5:                                     # %CallStackCheckFailBlk
59     .cfi_def_cfa %rbp, 16
60     callq __stack_chk_fail@PLT
61 .Lfunc_end0:
62     .size  main, .Lfunc_end0-main
63     .cfi_endproc
64                                     # -- End function
65     .type  .L.str,@object             # @.str
66     .section .rodata.str1.1,"aMS",@progbits,1
67 .L.str:
68     .asciz  "%d%d%d"
69     .size   .L.str, 7
70
71     .type  .L.str.1,@object           # @.str.1
72 .L.str.1:
73     .asciz  "%d\n"
74     .size   .L.str.1, 4
75
76     .ident  "clang version 14.0.6"
77     .section ".note.GNU-stack","",@progbits
78

```

四、 汇编器做了什么

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”。

LLVM 在后端主要是会通过一个个的 Pass 去优化，每个 Pass 做一些事情，最终生成汇编代码。我们通过最终的.bc 或者.ll 代码生成汇编代码：

```
1 clang -S -fobjc-arc fib.bc -o fib.s
2 clang -S -fobjc-arc fib.ll -o fib.s
```

生成代码也可以进行优化：clang -O3 -S -fobjc-arc fib.m -o fib.s

同时也可以直接使用 llc 命令同时汇编和链接 LLVM bitcode：

```
llc fib.bc -filetype=obj -o fib.o
```

上面的指令需要用到 bc 格式，即 LLVM IR 的二进制代码形式，而之前生成的是 LLVM IR 的文本形式。

可以通过下面的命令让 bc 和 ll 这两种 LLVM IR 格式互转，以统一文件格式：

```
1 llvm-dis a.bc -o a.ll # bc 转换为 ll
2 llvm-as a.ll -o a.bc # ll 转换为 bc
```

使用 llc 命令同时汇编和链接 LLVM bitcode，可以得到 fib.o 二进制文件。

llc 指令用于将 LLVM 源输入编译成特定架构的汇编语言，然后汇编语言输出可以通过本机汇编器和链接器来生成本机可执行文件。汇编器具体功能则是把汇编语言源文件翻译成机器语言目标文件，机器语言格式为公用目标格式。链接器用于把多个目标文件组合成单个可执行目标模块。它一边创建可执行模块，一边完成重定位以及决定外部参考。链接器的输入是可重定位的目标文件和目标库文件

对于生成的 fib.o 文件，使用如下命令

```
1 objdump -d fib.o >fib-anti-obj.S
```

对机器码进行反汇编得到 fib-anti-obj.S 文件，得到的反汇编结果如下：

```
1
2  fib.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 20       sub     $0x20,%rsp
11     8:  64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
12    f:  00 00
```

```

13      11: 48 89 45 f8          mov     %rax,-0x8(%rbp)
14      15: c7 45 e0 00 00 00 00 movl    $0x0,-0x20(%rbp)
15      1c: c7 45 f4 00 00 00 00 movl    $0x0,-0xc(%rbp)
16      23: c7 45 f0 01 00 00 00 movl    $0x1,-0x10(%rbp)
17      2a: c7 45 e8 01 00 00 00 movl    $0x1,-0x18(%rbp)
18      31: 48 bf 00 00 00 00 00 movabs  $0x0,%rdi
19      38: 00 00 00
20      3b: 48 8d 75 ec          lea     -0x14(%rbp),%rsi
21      3f: 48 8d 55 f4          lea     -0xc(%rbp),%rdx
22      43: 48 8d 4d f0          lea     -0x10(%rbp),%rcx
23      47: b0 00              mov     $0x0,%al
24      49: e8 00 00 00 00      call    4e <main+0x4e>
25      4e: 8b 45 e8            mov     -0x18(%rbp),%eax
26      51: 3b 45 ec            cmp     -0x14(%rbp),%eax
27      54: 7d 34              jge     8a <main+0x8a>
28      56: 8b 45 f0            mov     -0x10(%rbp),%eax
29      59: 89 45 e4            mov     %eax,-0x1c(%rbp)
30      5c: 8b 45 f4            mov     -0xc(%rbp),%eax
31      5f: 03 45 f0            add     -0x10(%rbp),%eax
32      62: 89 45 f0            mov     %eax,-0x10(%rbp)
33      65: 8b 75 f0            mov     -0x10(%rbp),%esi
34      68: 48 bf 00 00 00 00 00 movabs  $0x0,%rdi
35      6f: 00 00 00
36      72: b0 00              mov     $0x0,%al
37      74: e8 00 00 00 00      call    79 <main+0x79>
38      79: 8b 45 e4            mov     -0x1c(%rbp),%eax
39      7c: 89 45 f4            mov     %eax,-0xc(%rbp)
40      7f: 8b 45 e8            mov     -0x18(%rbp),%eax
41      82: 83 c0 01            add     $0x1,%eax
42      85: 89 45 e8            mov     %eax,-0x18(%rbp)
43      88: eb c4              jmp     4e <main+0x4e>
44      8a: 8b 45 e0            mov     -0x20(%rbp),%eax
45      8d: 64 48 8b 0c 25 28 00 mov     %fs:0x28,%rcx
46      94: 00 00
47      96: 48 8b 55 f8          mov     -0x8(%rbp),%rdx
48      9a: 48 39 d1            cmp     %rdx,%rcx
49      9d: 75 06              jne     a5 <main+0xa5>
50      9f: 48 83 c4 20          add     $0x20,%rsp
51     a3: 5d                pop     %rbp
52     a4: c3                ret
53     a5: e8 00 00 00 00      call    aa <main+0xaa>
54

```

对比原汇编代码 fib.S 文件可见，主体部分的代码完全一致，原文件中 label、无用的标记符

号等被删去，得到了更为简单的汇编代码。

五、 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。进而链接器对该机器代码进行执行生成可执行文件。clang fib.o -o fib 对可执行文件进行反汇编：objdump -d fib >fib-anti-exe.S，得到的反汇编结果如下：

```

1
2  fib:      文件格式 elf64-x86-64
3
4
5  Disassembly of section .init:
6
7  0000000000001000 <_init>:
8      1000:  f3 0f 1e fa      endbr64
9      1004:  48 83 ec 08      sub    $0x8,%rsp
10     1008:  48 8b 05 c1 2f 00 00  mov    0x2fc1(%rip),%rax      # 3fd0 <__gmon_start__@Base>
11     100f:  48 85 c0          test   %rax,%rax
12     1012:  74 02            je     1016 <_init+0x16>
13     1014:  ff d0            call   *%rax
14     1016:  48 83 c4 08      add    $0x8,%rsp
15     101a:  c3              ret
16
17  Disassembly of section .plt:
18
19  0000000000001020 <__stack_chk_fail@plt-0x10>:
20     1020:  ff 35 ca 2f 00 00  push   0x2fca(%rip)          # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
21     1026:  ff 25 cc 2f 00 00  jmp     *0x2fcc(%rip)        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x1>
22     102c:  0f 1f 40 00      nopl   0x0(%rax)
23
24  0000000000001030 <__stack_chk_fail@plt>:
25     1030:  ff 25 ca 2f 00 00  jmp     *0x2fca(%rip)        # 4000 <__stack_chk_fail@GLIBC_2.
26     1036:  68 00 00 00 00  push   $0x0
27     103b:  e9 e0 ff ff ff  jmp     1020 <_init+0x20>
28
29  0000000000001040 <printf@plt>:
30     1040:  ff 25 c2 2f 00 00  jmp     *0x2fc2(%rip)        # 4008 <printf@GLIBC_2.2.5>
31     1046:  68 01 00 00 00  push   $0x1
32     104b:  e9 d0 ff ff ff  jmp     1020 <_init+0x20>
33
34  0000000000001050 <__isoc99_scanf@plt>:
35     1050:  ff 25 ba 2f 00 00  jmp     *0x2fba(%rip)        # 4010 <__isoc99_scanf@GLIBC_2.7>

```

```

36      1056: 68 02 00 00 00      push    $0x2
37      105b: e9 c0 ff ff ff      jmp     1020 <_init+0x20>
38
39      Disassembly of section .text:
40
41      0000000000001060 <_start>:
42      1060: f3 0f 1e fa          endbr64
43      1064: 31 ed               xor     %ebp,%ebp
44      1066: 49 89 d1            mov     %rdx,%r9
45      1069: 5e                 pop     %rsi
46      106a: 48 89 e2            mov     %rsp,%rdx
47      106d: 48 83 e4 f0         and     $0xfffffffffffffff0,%rsp
48      1071: 50                 push    %rax
49      1072: 54                 push    %rsp
50      1073: 45 31 c0            xor     %r8d,%r8d
51      1076: 31 c9              xor     %ecx,%ecx
52      1078: 48 8d 3d e1 00 00 00 lea     0xe1(%rip),%rdi      # 1160 <main>
53      107f: ff 15 3b 2f 00 00   call   *0x2f3b(%rip)      # 3fc0 <__libc_start_main@GLIBC_2
54      1085: f4                 hlt
55      1086: 66 2e 0f 1f 84 00 00 cs nopw 0x0(%rax,%rax,1)
56      108d: 00 00 00
57
58      0000000000001090 <deregister_tm_clones>:
59      1090: 48 8d 3d 91 2f 00 00 lea     0x2f91(%rip),%rdi      # 4028 <__TMC_END__>
60      1097: 48 8d 05 8a 2f 00 00 lea     0x2f8a(%rip),%rax      # 4028 <__TMC_END__>
61      109e: 48 39 f8            cmp     %rdi,%rax
62      10a1: 74 15              je      10b8 <deregister_tm_clones+0x28>
63      10a3: 48 8b 05 1e 2f 00 00 mov     0x2f1e(%rip),%rax      # 3fc8 <_ITM_deregisterTMClon
64      10aa: 48 85 c0            test    %rax,%rax
65      10ad: 74 09              je      10b8 <deregister_tm_clones+0x28>
66      10af: ff e0              jmp     *%rax
67      10b1: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
68      10b8: c3                 ret
69      10b9: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
70
71      00000000000010c0 <register_tm_clones>:
72      10c0: 48 8d 3d 61 2f 00 00 lea     0x2f61(%rip),%rdi      # 4028 <__TMC_END__>
73      10c7: 48 8d 35 5a 2f 00 00 lea     0x2f5a(%rip),%rsi      # 4028 <__TMC_END__>
74      10ce: 48 29 fe            sub     %rdi,%rsi
75      10d1: 48 89 f0            mov     %rsi,%rax
76      10d4: 48 c1 ee 3f         shr     $0x3f,%rsi
77      10d8: 48 c1 f8 03         sar     $0x3,%rax
78      10dc: 48 01 c6            add     %rax,%rsi
79      10df: 48 d1 fe            sar     %rsi

```

```

80      10e2: 74 14                                je      10f8 <register_tm_clones+0x38>
81      10e4: 48 8b 05 ed 2e 00 00                mov     0x2eed(%rip),%rax          # 3fd8 <_ITM_registerTMCloneT
82      10eb: 48 85 c0                            test    %rax,%rax
83      10ee: 74 08                                je      10f8 <register_tm_clones+0x38>
84      10f0: ff e0                                jmp     *%rax
85      10f2: 66 0f 1f 44 00 00                  nopw    0x0(%rax,%rax,1)
86      10f8: c3                                  ret
87      10f9: 0f 1f 80 00 00 00 00              nopl    0x0(%rax)
88
89      00000000000001100 <__do_global_dtors_aux>:
90      1100: f3 0f 1e fa                        endbr64
91      1104: 80 3d 1d 2f 00 00 00              cmpb    $0x0,0x2f1d(%rip)          # 4028 <__TMC_END__>
92      110b: 75 33                              jne     1140 <__do_global_dtors_aux+0x40>
93      110d: 55                                  push    %rbp
94      110e: 48 83 3d ca 2e 00 00              cmpq    $0x0,0x2eca(%rip)          # 3fe0 <__cxa_finalize@GLIBC_
95      1115: 00
96      1116: 48 89 e5                          mov     %rsp,%rbp
97      1119: 74 0d                              je      1128 <__do_global_dtors_aux+0x28>
98      111b: 48 8b 3d fe 2e 00 00              mov     0x2efe(%rip),%rdi          # 4020 <__dso_handle>
99      1122: ff 15 b8 2e 00 00                call    *0x2eb8(%rip)              # 3fe0 <__cxa_finalize@GLIBC_2.2.
100     1128: e8 63 ff ff ff                  call    1090 <deregister_tm_clones>
101     112d: c6 05 f4 2e 00 00 01              movb    $0x1,0x2ef4(%rip)          # 4028 <__TMC_END__>
102     1134: 5d                                  pop     %rbp
103     1135: c3                                  ret
104     1136: 66 2e 0f 1f 84 00 00              cs nopw 0x0(%rax,%rax,1)
105     113d: 00 00 00
106     1140: c3                                  ret
107     1141: 66 66 2e 0f 1f 84 00              data16  cs nopw 0x0(%rax,%rax,1)
108     1148: 00 00 00 00
109     114c: 0f 1f 40 00                      nopl    0x0(%rax)
110
111     00000000000001150 <frame_dummy>:
112     1150: f3 0f 1e fa                        endbr64
113     1154: e9 67 ff ff ff                  jmp     10c0 <register_tm_clones>
114     1159: 0f 1f 80 00 00 00 00              nopl    0x0(%rax)
115
116     00000000000001160 <main>:
117     1160: 55                                  push    %rbp
118     1161: 48 89 e5                          mov     %rsp,%rbp
119     1164: 48 83 ec 20                      sub     $0x20,%rsp
120     1168: 64 48 8b 04 25 28 00              mov     %fs:0x28,%rax
121     116f: 00 00
122     1171: 48 89 45 f8                      mov     %rax,-0x8(%rbp)
123     1175: c7 45 e0 00 00 00 00              movl    $0x0,-0x20(%rbp)

```

```

124      117c: c7 45 f4 00 00 00 00    movl    $0x0,-0xc(%rbp)
125      1183: c7 45 f0 01 00 00 00    movl    $0x1,-0x10(%rbp)
126      118a: c7 45 e8 01 00 00 00    movl    $0x1,-0x18(%rbp)
127      1191: 48 bf 04 20 00 00 00    movabs  $0x2004,%rdi
128      1198: 00 00 00
129      119b: 48 8d 75 ec            lea     -0x14(%rbp),%rsi
130      119f: 48 8d 55 f4            lea     -0xc(%rbp),%rdx
131      11a3: 48 8d 4d f0            lea     -0x10(%rbp),%rcx
132      11a7: b0 00                  mov     $0x0,%al
133      11a9: e8 a2 fe ff ff        call    1050 <__isoc99_scanf@plt>
134      11ae: 8b 45 e8                mov     -0x18(%rbp),%eax
135      11b1: 3b 45 ec                cmp     -0x14(%rbp),%eax
136      11b4: 7d 34                  jge     11ea <main+0x8a>
137      11b6: 8b 45 f0                mov     -0x10(%rbp),%eax
138      11b9: 89 45 e4                mov     %eax,-0x1c(%rbp)
139      11bc: 8b 45 f4                mov     -0xc(%rbp),%eax
140      11bf: 03 45 f0                add     -0x10(%rbp),%eax
141      11c2: 89 45 f0                mov     %eax,-0x10(%rbp)
142      11c5: 8b 75 f0                mov     -0x10(%rbp),%esi
143      11c8: 48 bf 0b 20 00 00 00    movabs  $0x200b,%rdi
144      11cf: 00 00 00
145      11d2: b0 00                  mov     $0x0,%al
146      11d4: e8 67 fe ff ff        call    1040 <printf@plt>
147      11d9: 8b 45 e4                mov     -0x1c(%rbp),%eax
148      11dc: 89 45 f4                mov     %eax,-0xc(%rbp)
149      11df: 8b 45 e8                mov     -0x18(%rbp),%eax
150      11e2: 83 c0 01                add     $0x1,%eax
151      11e5: 89 45 e8                mov     %eax,-0x18(%rbp)
152      11e8: eb c4                  jmp     11ae <main+0x4e>
153      11ea: 8b 45 e0                mov     -0x20(%rbp),%eax
154      11ed: 64 48 8b 0c 25 28 00    mov     %fs:0x28,%rcx
155      11f4: 00 00
156      11f6: 48 8b 55 f8            mov     -0x8(%rbp),%rdx
157      11fa: 48 39 d1                cmp     %rdx,%rcx
158      11fd: 75 06                  jne     1205 <main+0xa5>
159      11ff: 48 83 c4 20            add     $0x20,%rsp
160      1203: 5d                      pop     %rbp
161      1204: c3                      ret
162      1205: e8 26 fe ff ff        call    1030 <__stack_chk_fail@plt>
163
164      Disassembly of section .fini:
165
166      000000000000120c <_fini>:
167      120c: f3 0f 1e fa            endbr64

```

```

168      1210:  48 83 ec 08          sub    $0x8,%rsp
169      1214:  48 83 c4 08          add    $0x8,%rsp
170      1218:  c3                  ret
171

```

可以发现所得结果长度大大增加，相较上一层新增了链接所得的内容。

六、 LLVM IR 编程

本次 SysY 语言特性研究，涵盖了函数，语句块，变量定义特性的程序例子编写和验证。

(一) 变量定义

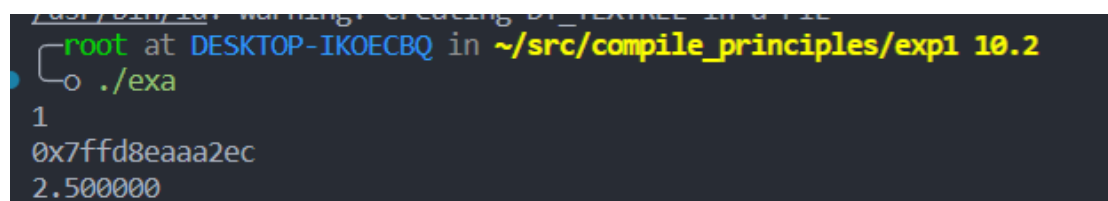
此处定义了一个 int 类型变量、一个 int 类型指针、一个 float 类型变量，并分别定义时赋值、赋值地址、定义后赋值。

```

1  @.str = private unnamed_addr constant [7 x i8] c"%d%p%f\00", align 1
2
3  define dso_local i32 @main() #0 {
4      %1 = alloca i32, align 4
5      %2 = alloca i32*, align 8
6      %3 = alloca float, align 4
7      store i32 1, i32* %1, align 4
8      store i32* %1, i32** %2, align 8
9      store float 2.500000e+00, float* %3, align 4
10     %4 = load i32, i32* %1, align 4
11     %5 = load i32*, i32** %2, align 8
12     %6 = load float, float* %3, align 4
13     %7 = fpext float %6 to double
14     %8 = call i32 @i8*, ... @printf(i8* noundef getelementptr inbounds ([7 x i8], [7 x i8]* @.str,
15     ret i32 0
16 }
17
18 declare i32 @printf(i8* noundef, ...) #1
19

```

经过格式转换、汇编、链接，运行可执行程序可得正确输出结果如图4:



```

root at DESKTOP-IKOECBQ in ~/src/compile_principles/exp1 10.2
└─o ./exa
1
0x7ffd8eaaa2ec
2.500000

```

图 4: 变量定义输出结果

(二) 语句块

此处设计了一个普通语句块对变量进行修改和查看，并且设置了一个条件分支语句块和一个循环分支语句块。

```

1
2 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
3
4 define dso_local i32 @main() #0 {
5     %1 = alloca i32, align 4
6     %2 = alloca i32, align 4
7     %3 = alloca i32, align 4
8     %4 = alloca i32, align 4
9     %5 = alloca i32, align 4
10    %6 = alloca i32, align 4
11    store i32 0, i32* %1, align 4
12    store i32 1, i32* %3, align 4
13    %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]
14    store i32 0, i32* %5, align 4
15    %8 = load i32, i32* %5, align 4
16    %9 = call i32 @printf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str,
17    %10 = load i32, i32* %3, align 4
18    %11 = call i32 @printf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str,
19    %12 = load i32, i32* %2, align 4
20    %13 = icmp ne i32 %12, 0
21    br i1 %13, label %14, label %27
22
23    14:                                     ; preds = %0
24    store i32 0, i32* %4, align 4
25    store i32 1, i32* %6, align 4
26    br label %15
27
28    15:                                     ; preds = %23, %14
29    %16 = load i32, i32* %6, align 4
30    %17 = load i32, i32* %2, align 4
31    %18 = icmp sle i32 %16, %17
32    br i1 %18, label %19, label %26
33
34    19:                                     ; preds = %15
35    %20 = load i32, i32* %3, align 4
36    %21 = load i32, i32* %4, align 4
37    %22 = add nsw i32 %20, %21
38    store i32 %22, i32* %4, align 4
39    br label %23
40

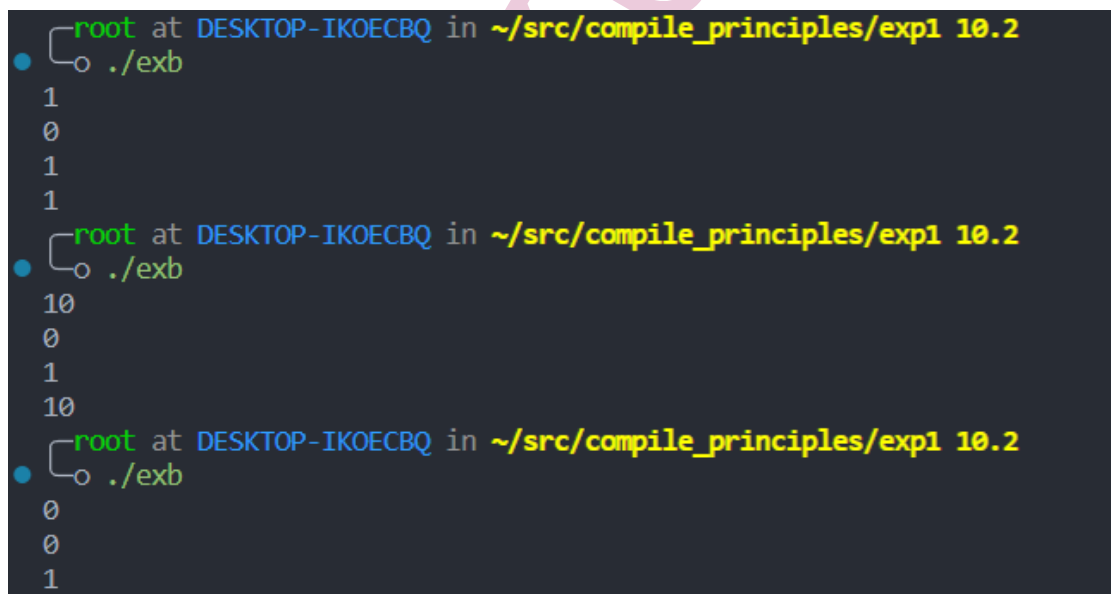
```

```

41      23:                                     ; preds = %19
42      %24 = load i32, i32* %6, align 4
43      %25 = add nsw i32 %24, 1
44      store i32 %25, i32* %6, align 4
45      br label %15, !llvm.loop !6
46
47      26:                                     ; preds = %15
48      br label %27
49
50      27:                                     ; preds = %26, %0
51      %28 = load i32, i32* %1, align 4
52      ret i32 %28
53  }
54
55  declare i32 @__isoc99_scanf(i8* noundef, ...) #1
56
57  declare i32 @printf(i8* noundef, ...) #1
58

```

经过格式转换、汇编、链接，运行可执行程序可得正确输出结果如图5:



```

root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
└─o ./exb
1
0
1
1
root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
└─o ./exb
10
0
1
10
root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
└─o ./exb
0
0
1

```

图 5: 语句块输出结果

(三) 函数

这里设置了三个函数，一个返回 `int` 类型且不含参数，一个空类型且不含参数，一个返回 `float` 类型且含一个 `int` 类型参数。分别验证返回结果。

```

1
2  @b = dso_local global i32 0, align 4

```

```

3  @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
4  @.str.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
5
6  define dso_local i32 @f1() #0 {
7      %1 = load i32, i32* @b, align 4
8      ret i32 %1
9  }
10
11 define dso_local void @f2() #0 {
12     store i32 1, i32* @b, align 4
13     ret void
14 }
15
16 define dso_local float @f3(i32 noundef %0) #0 {
17     %2 = alloca i32, align 4
18     store i32 %0, i32* %2, align 4
19     %3 = load i32, i32* %2, align 4
20     %4 = sdiv i32 %3, 2
21     %5 = sitofp i32 %4 to float
22     ret float %5
23 }
24
25 define dso_local i32 @main() #0 {
26     store i32 0, i32* @b, align 4
27     %1 = call i32 @f1()
28     %2 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
29     call void @f2()
30     %3 = load i32, i32* @b, align 4
31     %4 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
32     %5 = load i32, i32* @b, align 4
33     %6 = call float @f3(i32 noundef %5)
34     %7 = fpext float %6 to double
35     %8 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str,
36     ret i32 0
37 }
38
39 declare i32 @printf(i8* noundef, ...) #1
40

```

经过格式转换、汇编、链接，运行可执行程序可得正确输出结果如图6:

```

root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
├─o ./exc
├─0
├─1
└─0.000000

```

图 6: 函数输出结果

(四) 数组定义

这里设计了不同维度, int 和 float 类型的数组定义。

```

1 define dso_local i32 @main() #0 {
2     %1 = alloca [2 x i32], align 4
3     %2 = alloca [3 x [5 x i32]], align 16
4     %3 = alloca [7 x [11 x [2 x i32]]], align 16
5     %4 = alloca [13 x float], align 16
6     %5 = alloca [2 x [4 x [6 x [10 x float]]]], align 16
7     ret i32 0
8 }

```

(五) 隐式转换

这里设计了 int to float 和 float to int 两种隐式类型转换, 等价 SysY 代码如下

```

1 int    i1 = 3.141592654;
2 int    i2 = 0.499999;
3 int    i3 = 0.500000;
4 int    i4 = 0.500001;
5 int    i5 = -0.499999;
6 int    i6 = -0.500000;
7 int    i7 = -0.500001;
8 float  f1 = 3141592654;
9 float  f2 = -314;
10 int    i8 = f1 / f2;
11 float  f3 = i1 + i8;

```

隐式转换中, 字面常量会直接被编译器计算, float to int 过程中会被截断小数部分, 因此上述代码 i2 i7 均为 0。int to float 过程中按照 IEEE 754 规范转换, 如果数字过大或过小会转换成 inf。如果是没有在编译期确定的常量, 会使用 fptosi 指令将浮点数转换成整数、sitofp 指令将整数转换成浮点数。

```

1 define dso_local i32 @main() #0 {
2     %1 = alloca i32, align 4
3     %2 = alloca i32, align 4

```

```
4    %3 = alloca i32, align 4
5    %4 = alloca i32, align 4
6    %5 = alloca i32, align 4
7    %6 = alloca i32, align 4
8    %7 = alloca i32, align 4
9    %8 = alloca float, align 4
10   %9 = alloca float, align 4
11   %10 = alloca i32, align 4
12   %11 = alloca float, align 4
13   store i32 3, i32* %1, align 4
14   store i32 0, i32* %2, align 4
15   store i32 0, i32* %3, align 4
16   store i32 0, i32* %4, align 4
17   store i32 0, i32* %5, align 4
18   store i32 0, i32* %6, align 4
19   store i32 0, i32* %7, align 4
20   store float 0x41E7681CC0000000, float* %8, align 4
21   store float -3.140000e+02, float* %9, align 4
22   %12 = load float, float* %8, align 4
23   %13 = load float, float* %9, align 4
24   %14 = fdiv float %12, %13
25   %15 = fptosi float %14 to i32
26   store i32 %15, i32* %10, align 4
27   %16 = load i32, i32* %1, align 4
28   %17 = load i32, i32* %10, align 4
29   %18 = add nsw i32 %16, %17
30   %19 = sitofp i32 %18 to float
31   store float %19, float* %11, align 4
32   ret i32 0
33 }
```

七、 总结

通过实战编写 LLVM/IR 程序，更深入地感受了语言处理系统各项完整的工作过程，熟悉了 LLVM IR 中间语言，并对其实现方式有了一定了解，为今后编写完整编译器打下良好基础。