



南開大學
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	3
(三) 语义分析	4
(四) 中间代码生成	4
(五) 代码优化	7
(六) 代码生成	8
四、 汇编器做了什么	10
五、 链接器做了什么	12
六、 LLVM IR 编程	16
(一) 变量定义	16
(二) 语句块	17
(三) 函数	19
(四) 数组定义	20
(五) 隐式转换	20
七、 总结	22

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。

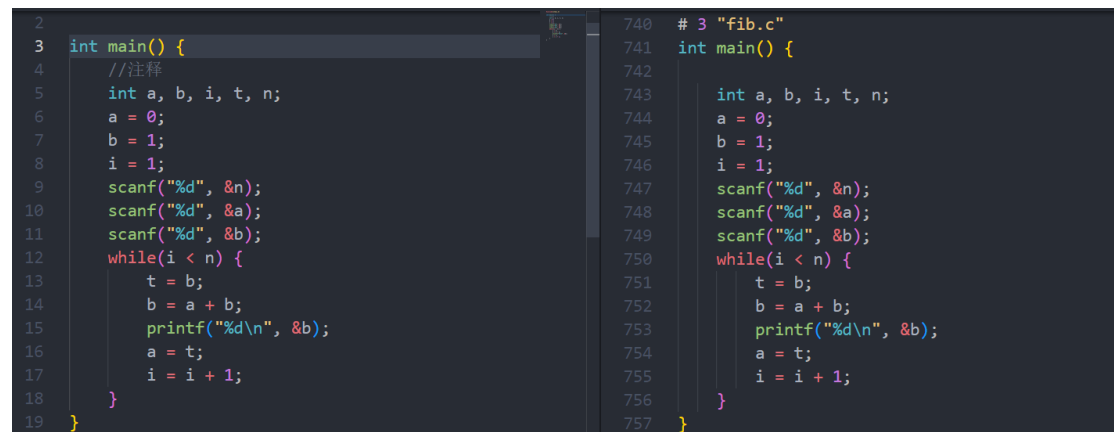
```
1  #include<stdio.h>
2  signed main() {
3      int a, b, i, t, n;
4      a = 0;
5      b = 1;
6      i = 1;
7      scanf("%d%d%d", &n, &a, &b);
8      while(i < n) {
9          t = b;
10         b = a + b;
11         printf("%d\n", &b);
12         a = t;
13         i = i + 1;
14     }
15 }
```

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，发现多出的部分即为头文件定义的具体内容。没有用到的头或注释则会被删去。如对比图1所示。



```
2
3 int main() {
4     //注释
5     int a, b, i, t, n;
6     a = 0;
7     b = 1;
8     i = 1;
9     scanf("%d", &n);
10    scanf("%d", &a);
11    scanf("%d", &b);
12    while(i < n) {
13        t = b;
14        b = a + b;
15        printf("%d\n", &b);
16        a = t;
17        i = i + 1;
18    }
19 }
```

```
740 # 3 "fib.c"
741 int main() {
742
743     int a, b, i, t, n;
744     a = 0;
745     b = 1;
746     i = 1;
747     scanf("%d", &n);
748     scanf("%d", &a);
749     scanf("%d", &b);
750     while(i < n) {
751         t = b;
752         b = a + b;
753         printf("%d\n", &b);
754         a = t;
755         i = i + 1;
756     }
757 }
```

图 1: 左为源代码, 右为经预处理器处理后的代码, 可见行数增加且注释被删去。

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2所示。

```

while 'while'      [StartOfLine] [LeadingSpace]  Loc=<fib.c:12:5>
l_paren '('        Loc=<fib.c:12:10>
identifier 'i'     Loc=<fib.c:12:11>
less '<'          [LeadingSpace] Loc=<fib.c:12:13>
identifier 'n'     [LeadingSpace] Loc=<fib.c:12:15>
r_paren ')'        Loc=<fib.c:12:16>
l_brace '{'        [LeadingSpace] Loc=<fib.c:12:18>
identifier 't'     [StartOfLine] [LeadingSpace]  Loc=<fib.c:13:9>
equal '='          [LeadingSpace] Loc=<fib.c:13:11>
identifier 'b'     [LeadingSpace] Loc=<fib.c:13:13>
semi ';'          Loc=<fib.c:13:14>
identifier 'b'     [StartOfLine] [LeadingSpace]  Loc=<fib.c:14:9>
equal '='          [LeadingSpace] Loc=<fib.c:14:11>
identifier 'a'     [LeadingSpace] Loc=<fib.c:14:13>
plus '+'          [LeadingSpace] Loc=<fib.c:14:15>
identifier 'b'     [LeadingSpace] Loc=<fib.c:14:17>
semi ';'          Loc=<fib.c:14:18>
identifier 'printf' [StartOfLine] [LeadingSpace]  Loc=<fib.c:15:9>
l_paren '('        Loc=<fib.c:15:15>
string_literal '%"d\n"' Loc=<fib.c:15:16>
comma ','          Loc=<fib.c:15:22>
amp '&'          [LeadingSpace] Loc=<fib.c:15:24>
identifier 'b'     Loc=<fib.c:15:25>
r_paren ')'        Loc=<fib.c:15:26>
semi ';'          Loc=<fib.c:15:27>
identifier 'a'     [StartOfLine] [LeadingSpace]  Loc=<fib.c:16:9>
equal '='          [LeadingSpace] Loc=<fib.c:16:11>
identifier 't'     [LeadingSpace] Loc=<fib.c:16:13>
semi ';'          Loc=<fib.c:16:14>
identifier 'i'     [StartOfLine] [LeadingSpace]  Loc=<fib.c:17:9>
equal '='          [LeadingSpace] Loc=<fib.c:17:11>
identifier 'i'     [LeadingSpace] Loc=<fib.c:17:13>
plus '+'          [LeadingSpace] Loc=<fib.c:17:15>
numeric_constant '1' [LeadingSpace] Loc=<fib.c:17:17>
semi ';'          Loc=<fib.c:17:18>
r_brace '}'        [StartOfLine] [LeadingSpace]  Loc=<fib.c:18:5>
r_brace '}'        [StartOfLine] Loc=<fib.c:19:1>
eof ''            Loc=<fib.c:19:2>

```

图 2: 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3所示，可见明确树形结构。

```

| ~DeclRefExpr 0x1007438 <col:18> 'int' lvalue Var 0x10068d8 'b' 'int'
| ~WhileStmt 0x1007cd8 <line:12:5, line:18:5>
| ~BinaryOperator 0x1007570 <line:12:11, col:15> 'int' '<'
| ~ImplicitCastExpr 0x1007540 <col:11> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007500 <col:11> 'int' lvalue Var 0x1006958 'i' 'int'
| ~ImplicitCastExpr 0x1007558 <col:15> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007520 <col:15> 'int' lvalue Var 0x1006a58 'n' 'int'
| ~CompoundStmt 0x1007ca0 <col:18, line:18:5>
| ~BinaryOperator 0x10075e8 <line:13:9, col:13> 'int' '='
| ~DeclRefExpr 0x1007590 <col:9> 'int' lvalue Var 0x10069d8 't' 'int'
| ~ImplicitCastExpr 0x10075d0 <col:13> 'int' <LValueToRValue>
| ~DeclRefExpr 0x10075b0 <col:13> 'int' lvalue Var 0x10068d8 'b' 'int'
| ~BinaryOperator 0x10076b8 <line:14:9, col:17> 'int' '='
| ~DeclRefExpr 0x1007608 <col:9> 'int' lvalue Var 0x10068d8 'b' 'int'
| ~BinaryOperator 0x1007698 <col:13, col:17> 'int' '+'
| ~ImplicitCastExpr 0x1007668 <col:13> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007628 <col:13> 'int' lvalue Var 0x1006858 'a' 'int'
| ~ImplicitCastExpr 0x1007680 <col:17> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007648 <col:17> 'int' lvalue Var 0x10068d8 'b' 'int'
| ~CallExpr 0x1007780 <line:15:9, col:26> 'int'
| ~ImplicitCastExpr 0x1007768 <col:9> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
| ~DeclRefExpr 0x10076d8 <col:9> 'int (const char *, ...)' Function 0xfef798 'printf' 'int (const char *, ...)'
| ~ImplicitCastExpr 0x10077c8 <col:16> 'const char *' <NoOp>
| ~ImplicitCastExpr 0x10077b0 <col:16> 'char *' <ArrayToPointerDecay>
| ~StringLiteral 0x10076f8 <col:16> 'char[4]' lvalue "%d\n"
| ~UnaryOperator 0x1007738 <col:24, col:25> 'int *' prefix '&' cannot overflow
| ~DeclRefExpr 0x1007718 <col:25> 'int' lvalue Var 0x10068d8 'b' 'int'
| ~BinaryOperator 0x1007bc8 <line:16:9, col:13> 'int' '='
| ~DeclRefExpr 0x1007b70 <col:9> 'int' lvalue Var 0x1006858 'a' 'int'
| ~ImplicitCastExpr 0x1007bb0 <col:13> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007b90 <col:13> 'int' lvalue Var 0x10069d8 't' 'int'
| ~BinaryOperator 0x1007c80 <line:17:9, col:17> 'int' '='
| ~DeclRefExpr 0x1007be8 <col:9> 'int' lvalue Var 0x1006958 'i' 'int'
| ~BinaryOperator 0x1007c60 <col:13, col:17> 'int' '+'
| ~ImplicitCastExpr 0x1007c48 <col:13> 'int' <LValueToRValue>
| ~DeclRefExpr 0x1007c08 <col:13> 'int' lvalue Var 0x1006958 'i' 'int'
| ~IntegerLiteral 0x1007c28 <col:17> 'int' 1

```

图 3: 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 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。如图4可以看到控制流图，以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。

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`

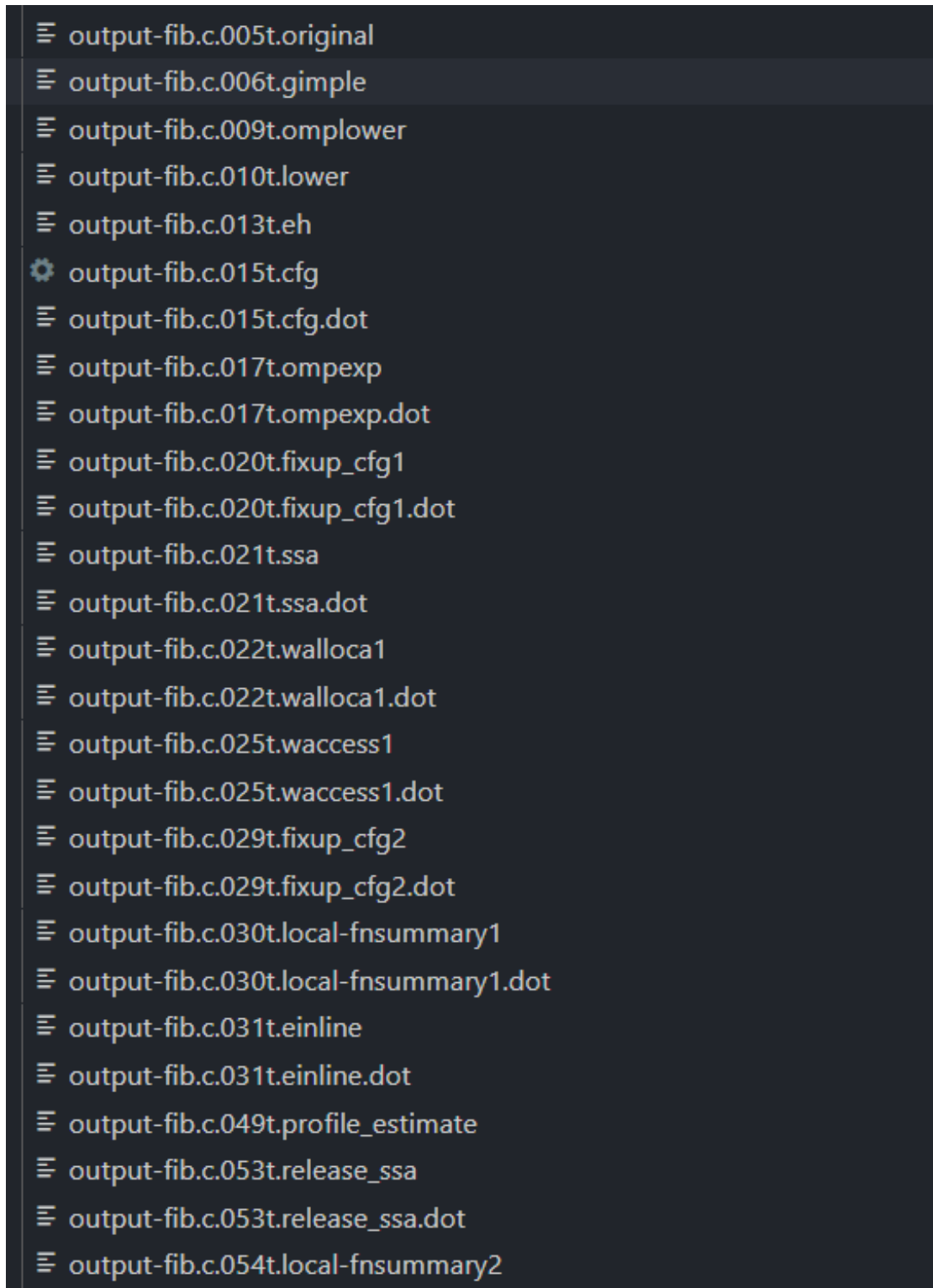


图 4: 控制流图

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 [3 x i8] c"%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 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 ([3 x i8], [3 x i8]
22     %8 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]
23     %9 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]
24     br label %10
25
26 10:                                     ; preds = %14, %0
27     %11 = load i32, i32* %4, align 4
28     %12 = load i32, i32* %6, align 4
29     %13 = icmp slt i32 %11, %12
30     br i1 %13, label %14, label %23
31
32 14:                                     ; preds = %10
33     %15 = load i32, i32* %3, align 4
34     store i32 %15, i32* %5, align 4
35     %16 = load i32, i32* %2, align 4
36     %17 = load i32, i32* %3, align 4
37     %18 = add nsw i32 %16, %17
38     store i32 %18, i32* %3, align 4
39     %19 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str
40     %20 = load i32, i32* %5, align 4
41     store i32 %20, i32* %2, align 4
42     %21 = load i32, i32* %4, align 4
43     %22 = add nsw i32 %21, 1
44     store i32 %22, i32* %4, align 4
45     br label %10, !llvm.loop !6
46

```



```

47     23:                                     ; preds = %10
48     %24 = load i32, i32* %1, align 4
49     ret i32 %24
50 }
51
52 declare i32 @__isoc99_scanf(i8* noundef, ...) #1
53
54 declare i32 @printf(i8* noundef, ...) #1
55
56 attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="default" "no-trapping-math"="true" "stack-protector-buffer-size"="default" "target-cpu"="x86_64" "target-features"="" "unwind-tables"="none" }
57 attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="default" "target-cpu"="x86_64" "target-features"="" "unwind-tables"="none" }
58
59 !llvm.module.flags = !{!0, !1, !2, !3, !4}
60 !llvm.ident = !{!5}
61
62 !0 = !{i32 1, !"wchar_size", i32 4}
63 !1 = !{i32 7, !"PIC Level", i32 2}
64 !2 = !{i32 7, !"PIE Level", i32 2}
65 !3 = !{i32 7, !"uwtable", i32 1}
66 !4 = !{i32 7, !"frame-pointer", i32 2}
67 !5 = !{"Debian clang version 14.0.6-2"}
68 !6 = distinct !{!6, !7}
69 !7 = !{"llvm.loop.mustprogress"}
70

```

(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5所示：

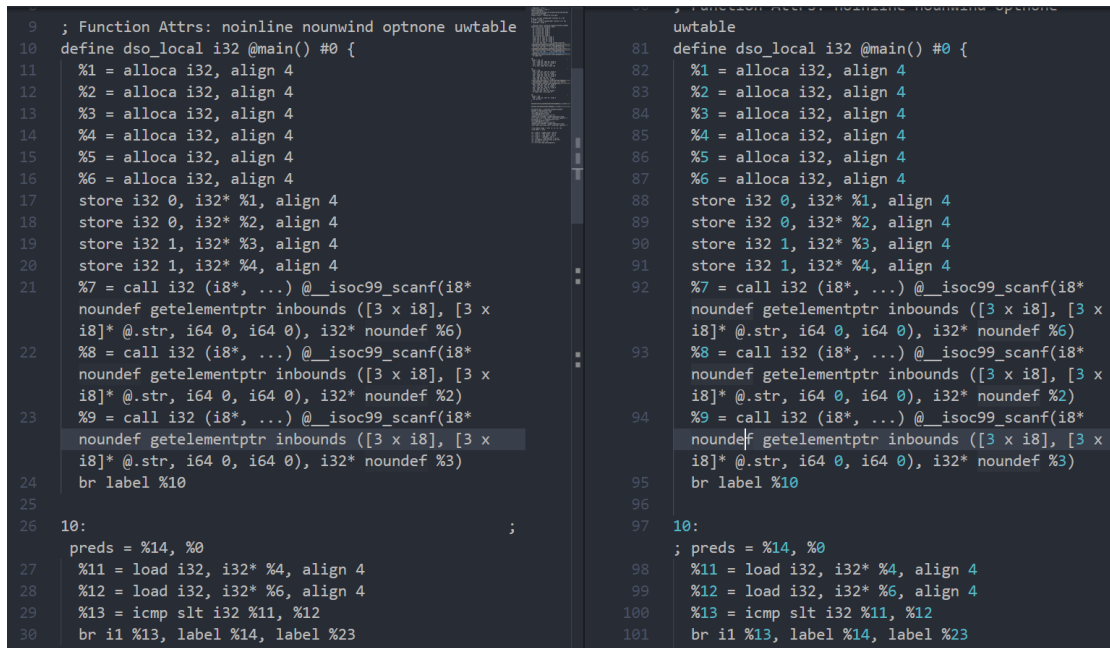


图 5: 左为 fib.ll, 右为 fib.log

(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 gcc fib.i -S -o fib.S # 生成 x86 格式目标代码
2 arm-linux-gnueabi-gcc fib.i -S -o fib.S # 生成 arm 格式目标代码
3 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 movl $0, -24(%rbp)
16 movl $0, -12(%rbp)
17 movl $1, -4(%rbp)
```

```

18  movl  $1, -8(%rbp)
19      movabsq  $.L.str, %rdi
20  leaq  -16(%rbp), %rsi
21  movb  $0, %al
22      callq  __isoc99_scanf@PLT
23      movabsq  $.L.str, %rdi
24  leaq  -12(%rbp), %rsi
25  movb  $0, %al
26      callq  __isoc99_scanf@PLT
27      movabsq  $.L.str, %rdi
28  leaq  -4(%rbp), %rsi
29  movb  $0, %al
30      callq  __isoc99_scanf@PLT
31      .LBB0_1:                                # =>This Inner Loop Header: Depth=1
32      movl  -8(%rbp), %eax
33      cmpl  -16(%rbp), %eax
34      jge  .LBB0_3
35      # %bb.2:                                # in Loop: Header=BB0_1 Depth=1
36      movl  -4(%rbp), %eax
37      movl  %eax, -20(%rbp)
38      movl  -12(%rbp), %eax
39      addl  -4(%rbp), %eax
40      movl  %eax, -4(%rbp)
41      movabsq  $.L.str.1, %rdi
42  leaq  -4(%rbp), %rsi
43  movb  $0, %al
44      callq  printf@PLT
45      movl  -20(%rbp), %eax
46      movl  %eax, -12(%rbp)
47      movl  -8(%rbp), %eax
48      addl  $1, %eax
49  movl  %eax, -8(%rbp)
50  jmp  .LBB0_1
51  .LBB0_3:
52  movl  -24(%rbp), %eax
53  addq  $32, %rsp
54  popq  %rbp
55      .cfi_def_cfa %rsp, 8
56  retq
57  .Lfunc_end0:
58      .size  main, .Lfunc_end0-main
59      .cfi_endproc
60      # -- End function
61      .type  .L.str,@object                    # @.str

```

```

62  .section  .rodata.str1.1,"aMS",@progbits,1
63  .L.str:
64  .asciz  "%d"
65  .size  .L.str, 3
66
67  .type  .L.str.1,@object          # @.str.1
68  .L.str.1:
69  .asciz  "%d\n"
70  .size  .L.str.1, 4
71
72  .ident  "Debian clang version 14.0.6-2"
73  .section  ".note.GNU-stack","",@progbits

```

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

```

我使用 llc 命令同时汇编和链接 LLVM bitcode，成功得到 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。如图6所示：

```

root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
├─o llvm-as fib.ll -o fib.bc
├─root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
│   └─o llc fib.bc -filetype=obj -o fib.o
└─root at DESKTOP-IKOEBCQ in ~/src/compile_principles/exp1 10.2
    └─o 1
total 1.4M
drwxr-xr-x 3 root root 4.0K Oct  2 12:11 .
drwxr-xr-x 4 root root 4.0K Oct  1 22:30 ..
-rw-r--r-- 1 root root  83 Oct  2 11:50 a.log
-rw-r--r-- 1 root root 2.7K Oct  2 12:11 fib.bc
-rw-r--r-- 1 root root 304 Oct  1 22:30 fib.c
-rwxr-xr-x 1 root root 18K Oct  2 11:48 fib.i
-rw-r--r-- 1 root root 2.9K Oct  2 11:35 fib.ll
-rw-r--r-- 1 root root 857K Oct  2 11:50 fib.log
-rw-r--r-- 1 root root 1.5K Oct  2 12:11 fib.o

```

图 6: 汇编和链接 LLVM bitcode

对于生成的 fib.o 文件, 使用如下命令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 fib.o:      file format elf64-x86-64
2
3 Disassembly of section .text:
4
5 0000000000000000 <main>:
6 0: 55                push    %rbp
7 1: 48 89 e5          mov     %rsp,%rbp
8 4: 48 83 ec 20       sub     $0x20,%rsp
9 8: c7 45 e8 00 00 00 00 movl    $0x0,-0x18(%rbp)
10 f: c7 45 f4 00 00 00 00 movl    $0x0,-0xc(%rbp)
11 16: c7 45 fc 01 00 00 00 movl    $0x1,-0x4(%rbp)
12 1d: c7 45 f8 01 00 00 00 movl    $0x1,-0x8(%rbp)
13 24: 48 bf 00 00 00 00 00 movabs  $0x0,%rdi
14 2b: 00 00 00
15 2e: 48 8d 75 f0       lea     -0x10(%rbp),%rsi
16 32: b0 00            mov     $0x0,%al
17 34: e8 00 00 00 00    call   39 <main+0x39>
18 39: 48 bf 00 00 00 00 00 movabs  $0x0,%rdi
19 40: 00 00 00
20 43: 48 8d 75 f4       lea     -0xc(%rbp),%rsi
21 47: b0 00            mov     $0x0,%al
22 49: e8 00 00 00 00    call   4e <main+0x4e>
23 4e: 48 bf 00 00 00 00 00 movabs  $0x0,%rdi
24 55: 00 00 00
25 58: 48 8d 75 fc       lea     -0x4(%rbp),%rsi
26 5c: b0 00            mov     $0x0,%al

```

```

27  5e: e8 00 00 00 00      call  63 <main+0x63>
28  63: 8b 45 f8            mov    -0x8(%rbp),%eax
29  66: 3b 45 f0            cmp    -0x10(%rbp),%eax
30  69: 7d 35              jge    a0 <main+0xa0>
31  6b: 8b 45 fc            mov    -0x4(%rbp),%eax
32  6e: 89 45 ec            mov    %eax,-0x14(%rbp)
33  71: 8b 45 f4            mov    -0xc(%rbp),%eax
34  74: 03 45 fc            add    -0x4(%rbp),%eax
35  77: 89 45 fc            mov    %eax,-0x4(%rbp)
36  7a: 48 bf 00 00 00 00    movabs $0x0,%rdi
37  81: 00 00 00
38  84: 48 8d 75 fc          lea    -0x4(%rbp),%rsi
39  88: b0 00              mov    $0x0,%al
40  8a: e8 00 00 00 00      call  8f <main+0x8f>
41  8f: 8b 45 ec            mov    -0x14(%rbp),%eax
42  92: 89 45 f4            mov    %eax,-0xc(%rbp)
43  95: 8b 45 f8            mov    -0x8(%rbp),%eax
44  98: 83 c0 01            add    $0x1,%eax
45  9b: 89 45 f8            mov    %eax,-0x8(%rbp)
46  9e: eb c3              jmp    63 <main+0x63>
47  a0: 8b 45 e8            mov    -0x18(%rbp),%eax
48  a3: 48 83 c4 20          add    $0x20,%rsp
49  a7: 5d                  pop    %rbp
50  a8: c3                  ret

```

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，得到了更为纯净的汇编代码。

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:      file format elf64-x86-64
3
4
5  Disassembly of section .init:
6
7  00000000000001000 <_init>:
8  1000: 48 83 ec 08          sub    $0x8,%rsp
9  1004: 48 8b 05 c5 2f 00 00    mov    0x2fc5(%rip),%rax      # 3fd0 <__gmon_start__@Base>

```

```

10    100b: 48 85 c0          test    %rax,%rax
11    100e: 74 02          je      1012 <_init+0x12>
12    1010: ff d0          call   *%rax
13    1012: 48 83 c4 08     add     $0x8,%rsp
14    1016: c3             ret
15
16    Disassembly of section .plt:
17
18    0000000000001020 <printf@plt-0x10>:
19    1020: ff 35 ca 2f 00 00    push   0x2fca(%rip)        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
20    1026: ff 25 cc 2f 00 00    jmp     *0x2fcc(%rip)      # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
21    102c: 0f 1f 40 00        nopl    0x0(%rax)
22
23    0000000000001030 <printf@plt>:
24    1030: ff 25 ca 2f 00 00    jmp     *0x2fca(%rip)      # 4000 <printf@GLIBC_2.2.5>
25    1036: 68 00 00 00 00      push   $0x0
26    103b: e9 e0 ff ff ff      jmp     1020 <_init+0x20>
27
28    0000000000001040 <__isoc99_scanf@plt>:
29    1040: ff 25 c2 2f 00 00    jmp     *0x2fc2(%rip)      # 4008 <__isoc99_scanf@GLIBC_2.7>
30    1046: 68 01 00 00 00      push   $0x1
31    104b: e9 d0 ff ff ff      jmp     1020 <_init+0x20>
32
33    Disassembly of section .plt.got:
34
35    0000000000001050 <__cxa_finalize@plt>:
36    1050: ff 25 8a 2f 00 00    jmp     *0x2f8a(%rip)      # 3fe0 <__cxa_finalize@GLIBC_2.2.5>
37    1056: 66 90            xchg    %ax,%ax
38
39    Disassembly of section .text:
40
41    0000000000001060 <_start>:
42    1060: 31 ed            xor     %ebp,%ebp
43    1062: 49 89 d1          mov     %rdx,%r9
44    1065: 5e              pop     %rsi
45    1066: 48 89 e2          mov     %rsp,%rdx
46    1069: 48 83 e4 f0       and     $0xfffffffffffffff0,%rsp
47    106d: 50              push    %rax
48    106e: 54              push    %rsp
49    106f: 45 31 c0          xor     %r8d,%r8d
50    1072: 31 c9            xor     %ecx,%ecx
51    1074: 48 8d 3d d5 00 00 00 lea     0xd5(%rip),%rdi      # 1150 <main>
52    107b: ff 15 3f 2f 00 00    call   *0x2f3f(%rip)      # 3fc0 <__libc_start_main@GLIBC_2.34>
53    1081: f4              hlt

```

```

54 1082: 66 2e 0f 1f 84 00 00 cs nopw 0x0(%rax,%rax,1)
55 1089: 00 00 00
56 108c: 0f 1f 40 00          nopl  0x0(%rax)
57
58 00000000000001090 <deregister_tm_clones>:
59 1090: 48 8d 3d 89 2f 00 00 lea    0x2f89(%rip),%rdi      # 4020 <__TMC_END__>
60 1097: 48 8d 05 82 2f 00 00 lea    0x2f82(%rip),%rax      # 4020 <__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_deregisterTMCloneTab
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 000000000000010c0 <register_tm_clones>:
72 10c0: 48 8d 3d 59 2f 00 00 lea    0x2f59(%rip),%rdi      # 4020 <__TMC_END__>
73 10c7: 48 8d 35 52 2f 00 00 lea    0x2f52(%rip),%rsi      # 4020 <__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_registerTMCloneTable
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 15 2f 00 00 00 cmpb    $0x0,0x2f15(%rip)      # 4020 <__TMC_END__>
92 110b: 75 2b               jne    1138 <__do_global_dtors_aux+0x38>
93 110d: 55                  push   %rbp
94 110e: 48 83 3d ca 2e 00 00 cmpq    $0x0,0x2eca(%rip)      # 3fe0 <__cxa_finalize@GLIBC_2.2.5>
95 1115: 00
96 1116: 48 89 e5             mov    %rsp,%rbp
97 1119: 74 0c               je     1127 <__do_global_dtors_aux+0x27>

```



```

98    111b: 48 8b 3d f6 2e 00 00    mov     0x2ef6(%rip),%rdi        # 4018 <__dso_handle>
99    1122: e8 29 ff ff ff          call    1050 <__cxa_finalize@plt>
100   1127: e8 64 ff ff ff          call    1090 <deregister_tm_clones>
101   112c: c6 05 ed 2e 00 00 01    movb    $0x1,0x2eed(%rip)        # 4020 <__TMC_END__>
102   1133: 5d                      pop     %rbp
103   1134: c3                      ret
104   1135: 0f 1f 00                nopl    (%rax)
105   1138: c3                      ret
106   1139: 0f 1f 80 00 00 00 00    nopl    0x0(%rax)
107
108   00000000000001140 <frame_dummy>:
109   1140: f3 0f 1e fa            endbr64
110   1144: e9 77 ff ff ff          jmp     10c0 <register_tm_clones>
111   1149: 0f 1f 80 00 00 00 00    nopl    0x0(%rax)
112
113   00000000000001150 <main>:
114   1150: 55                      push    %rbp
115   1151: 48 89 e5                mov     %rsp,%rbp
116   1154: 48 83 ec 20             sub     $0x20,%rsp
117   1158: c7 45 e8 00 00 00 00    movl    $0x0,-0x18(%rbp)
118   115f: c7 45 f4 00 00 00 00    movl    $0x0,-0xc(%rbp)
119   1166: c7 45 fc 01 00 00 00    movl    $0x1,-0x4(%rbp)
120   116d: c7 45 f8 01 00 00 00    movl    $0x1,-0x8(%rbp)
121   1174: 48 bf 04 20 00 00 00    movabs $0x2004,%rdi
122   117b: 00 00 00
123   117e: 48 8d 75 f0             lea     -0x10(%rbp),%rsi
124   1182: b0 00                  mov     $0x0,%al
125   1184: e8 b7 fe ff ff          call    1040 <__isoc99_scanf@plt>
126   1189: 48 bf 04 20 00 00 00    movabs $0x2004,%rdi
127   1190: 00 00 00
128   1193: 48 8d 75 f4             lea     -0xc(%rbp),%rsi
129   1197: b0 00                  mov     $0x0,%al
130   1199: e8 a2 fe ff ff          call    1040 <__isoc99_scanf@plt>
131   119e: 48 bf 04 20 00 00 00    movabs $0x2004,%rdi
132   11a5: 00 00 00
133   11a8: 48 8d 75 fc             lea     -0x4(%rbp),%rsi
134   11ac: b0 00                  mov     $0x0,%al
135   11ae: e8 8d fe ff ff          call    1040 <__isoc99_scanf@plt>
136   11b3: 8b 45 f8                mov     -0x8(%rbp),%eax
137   11b6: 3b 45 f0                cmp     -0x10(%rbp),%eax
138   11b9: 7d 35                  jge     11f0 <main+0xa0>
139   11bb: 8b 45 fc                mov     -0x4(%rbp),%eax
140   11be: 89 45 ec                mov     %eax,-0x14(%rbp)
141   11c1: 8b 45 f4                mov     -0xc(%rbp),%eax

```

```

142    11c4: 03 45 fc          add    -0x4(%rbp),%eax
143    11c7: 89 45 fc          mov     %eax,-0x4(%rbp)
144    11ca: 48 bf 07 20 00 00  movabs $0x2007,%rdi
145    11d1: 00 00 00
146    11d4: 48 8d 75 fc          lea     -0x4(%rbp),%rsi
147    11d8: b0 00              mov     $0x0,%al
148    11da: e8 51 fe ff ff      call    1030 <printf@plt>
149    11df: 8b 45 ec          mov     -0x14(%rbp),%eax
150    11e2: 89 45 f4          mov     %eax,-0xc(%rbp)
151    11e5: 8b 45 f8          mov     -0x8(%rbp),%eax
152    11e8: 83 c0 01          add     $0x1,%eax
153    11eb: 89 45 f8          mov     %eax,-0x8(%rbp)
154    11ee: eb c3              jmp     11b3 <main+0x63>
155    11f0: 8b 45 e8          mov     -0x18(%rbp),%eax
156    11f3: 48 83 c4 20        add     $0x20,%rsp
157    11f7: 5d                pop     %rbp
158    11f8: c3                ret
159
160    Disassembly of section .fini:
161
162    000000000000011fc <_fini>:
163    11fc: 48 83 ec 08        sub     $0x8,%rsp
164    1200: 48 83 c4 08        add     $0x8,%rsp
165    1204: c3                ret

```

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7：

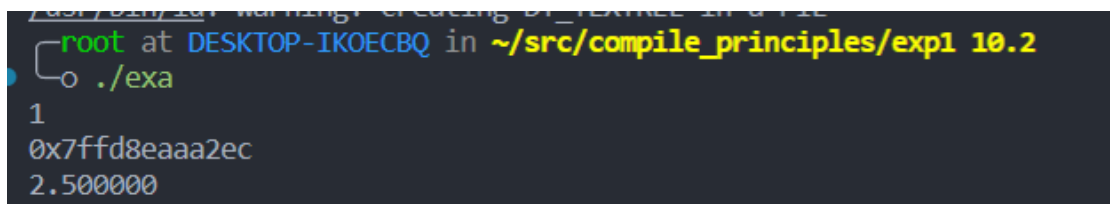


图 7: 变量定义输出结果

(二) 语句块

此处我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 (i8*, ...) @_isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str,
14    store i32 0, i32* %5, align 4
15    %8 = load i32, i32* %5, align 4
16    %9 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str,
17    %10 = load i32, i32* %3, align 4
18    %11 = call i32 (i8*, ...) @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8:

```

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 8: 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

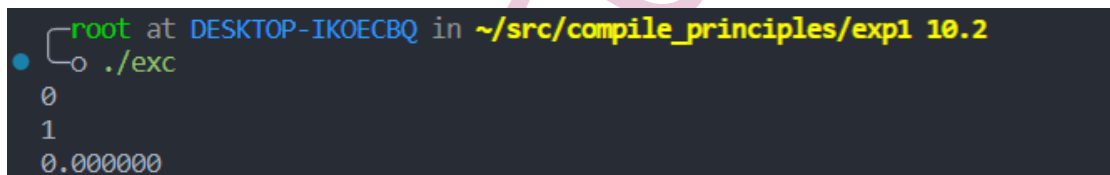
```

```

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9：



```

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

```

图 9: 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。

NIJU