



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

编译原理实验报告

了解你的编译器 & LLVM IR 编程

周延霖 2013921

年级：2020 级

专业：信息安全

指导教师：王刚

2022 年 9 月 26 日

摘要

此次实验是对编译器和整个编译过程的了解，在学习了高级语言以及计算机组成原理这些基础课之后，也应该了解一些关于执行文件方式方面的知识，在本次试验中也对自己将要实现的 SysY 语言特性进行 LLVM IR 程序的编写

关键字：编译过程、SysY 语言、LLVM IR

目录

一、 总体编译过程	1
(一) 总体编译过程总述	1
(二) 程序演练	1
二、 预处理器	3
(一) 预处理器概述	3
(二) 预处理指令	3
1. 指令类型	3
2. 指令通用规则	3
(三) 预处理器代码生成	3
三、 编译器	4
(一) 编译器概述	4
(二) 编译器过程	4
1. 词法分析	4
2. 语法分析	4
3. 语义分析	5
4. 中间代码生成与优化	5
5. 目标代码的生成	5
四、 汇编器	6
(一) 汇编器概述	6
(二) 汇编器过程	6
1. 运行截图	6
2. 生成代码内容	6
3. 代码分析	8
五、 链接器	9
(一) 链接器概述	9
1. 识别名称	9
2. 输入与输出	9
3. 外部引用	9
(二) 链接器工作	9
1. 构造符号表	9
2. 文件生成	10

六、 LLVM IR 编程	11
(一) 组员分工	11
(二) 语言特性描述	11
1. 特性一	11
2. 特性二	11
3. 特性三	11
4. 特性四	12
5. 特性五	13
6. 特性六	13
7. 特性七	13
8. 特性八	14
9. 特性九	14
10. 特性十	15
11. 特性十一	15
七、 总结与展望	16
(一) 文件内容展示	16
(二) 实验总结	16
(三) 未来展望	16

一、 总体编译过程

(一) 总体编译过程总述

应用程序从程序员编写的源文件到内存中执行的进程（或者目标文件），大致分为了以下几个阶段：

- 首先通过预处理器将源程序进行预处理，变成可以让编译器编译的程序。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。
- 接下来将经过处理的源程序交给编译器，经过多个步骤后完成编译。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。
- 然后将目标汇编文件交给汇编器。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。
- 最后将可重定位的机器代码交给链接器（或者加载器）。其功能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连接在一起，形成真正能在机器上运行的目标机器代码。

可以总结如图1所示：

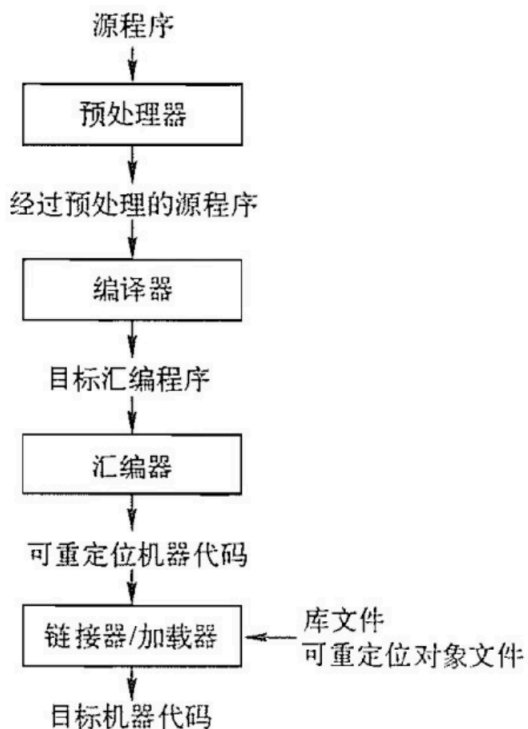


图 1: 总体编译过程

各个部分的具体细节功能将在下面的几节中进行介绍

(二) 程序演练

由于文书的书写毕竟太过枯燥和不变理解，所以本篇文章将以一个程序的逐渐进行来描述整个的编译过程，编译代码如下：

基础阶乘程序

```
1  #include <stdio.h>
2  int main()
3  {
4      int i, n, f;
5      //n = 10;
6      scanf("%d", &n);
7      i = 2;
8      f = 1;
9      while(i <= n)
10     {
11         f = f * i;
12         i = i + 1;
13     }
14     //cout << f << endl;
15     printf("%d\n", f);
16 }
```

二、 预处理器

(一) 预处理器概述

预处理器执行预处理指令，并在处理过程中删除这些指令。

预处理阶段会处理预编译指令，包括绝大多数的 # 开头的指令，如 include define if 等等，对 include 指令会替换对应的头文件，对 define 的宏命令会直接替换相应内容，同时会删除注释，添加行号和文件名标识。

对于 gcc，通过添加参数-E 令 gcc 只进行预处理过程，参数-o 改变 gcc 输出文件名，因此通过命令 gcc main.c -E -o main.i，即可得到预处理后文件。

观察预处理文件，可以发现文件长度远大于源文件，这就是将代码中的头文件进行了替代导致的结果。另外，实际上预处理过程是 gcc 调用了另一个程序（C Pre-Processor 调用时简写作 cpp）完成的过程。

(二) 预处理指令

1. 指令类型

- 宏定义——#define 定义一个宏，#undef 删除宏
- 文件包含——#include 将指定文件包含进程序中，如各种头文件 (*.h)
- 条件编译——#if、#ifdef、#ifndef、#elif、#else、#endif 根据测试条件确定包含还是排除一个文本块。

2. 指令通用规则

- 指令以 # 开头。无需位于行首，但要求前面有空白字符
- 指令符号间空格数量任意。如 # define N 50，包括 Tab 符
- 指令在换行后结束
- 指令在程序中的位置任意。通常在文件的开始，也可以放在后面，甚至函数定义的中间
- 指令行可添加注释

(三) 预处理器代码生成

由实验指导书的操作生成的 main.i 文件确实要比源文件大上许多，主要原因是因为包含了许多头文件，而头文件中又包含了许多其他文件，这样一直递归可以想象出其内容的丰富，由于代码量过于庞大，这里就不进行展示。

三、 编译器

(一) 编译器概述

编译器也是一种电脑程序。它会将用某种编程语言写成的源代码(原始语言),转换成另一种编程语言(目标语言)。高级计算机语言便于人编写,阅读,维护。低阶机器语言是计算机能直接解读、运行的。编译器主要的目的是将便于人编写,阅读,维护的高级计算机语言所写作的源代码,翻译为计算机能解读、运行的低阶机器语言的程序。编译器将原始程序(Source program)作为输入,翻译产生使用目标语言(Target language)的等价程序。源代码一般为高阶语言(High-level language),如 Pascal、C、C++、C#、Java 等,而目标语言则是汇编语言或目标机器的目标代码(Object code),有时也称作机器代码(Machine code)。

(二) 编译器过程

编译可以分为五个基本步骤:词法分析、语法分析、语义分析及中间代码的生成、优化、目标代码的生成。这是每个编译器都必须的基本步骤和流程,从源头输入高级语言源程序输出目标语言代码。大致流程可以总结如图2所示:

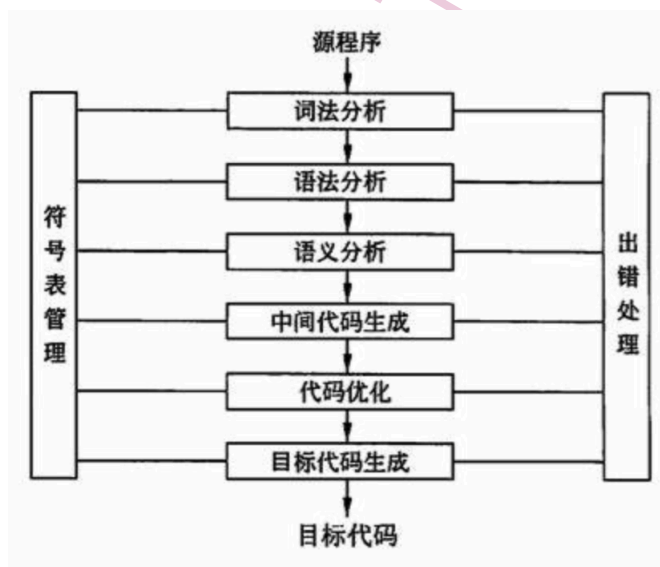


图 2: 编译器编译过程

1. 词法分析

词法分析器是通过词法分析程序对构成源程序的字符串从左到右的扫描,逐个字符地读,识别出每个单词符号,识别出的符号一般以二元式形式输出,即包含符号种类的编码和该符号的值。词法分析器一般以函数的形式存在,供语法分析器调用。当然也可以一个独立的词法分析器程序存在。完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。

2. 语法分析

语法分析是编译过程的第二个阶段。这阶段的任务是在词法分析的基础上将识别出的单词符号序列组合成各类语法短语,如“语句”,“表达式”等。语法分析程序的主要步骤是判断源程序语句是否符合定义的语法规则,在语法结构上是否正确。而一个语法规则又称为文法,乔姆斯

基将文法根据施加不同的限制分为 0 型、1 型、2 型、3 型文法, 0 型文法又称短语文法, 1 型称为上下文有关文法, 2 型称为上下文无关文法, 3 型文法称为正规文法, 限制条件依次递增。

3. 语义分析

词法分析注重的是每个单词是否合法, 以及这个单词属于语言中的哪些部分。语法分析的上下文无关文法注重的是输入语句是否可以依据文法匹配产生式。那么, 语义分析就是要了解各个语法单位之间的关系是否合法。实际应用中就是对结构上正确的源程序进行上下文有关性质的审查, 进行类型审查等。

4. 中间代码生成与优化

在进行了语法分析和语义分析阶段的工作之后, 有的编译程序将源程序变成一种内部表示形式, 这种内部表示形式叫做中间语言或中间表示或中间代码。所谓“中间代码”是一种结构简单、含义明确的记号系统, 这种记号系统复杂性介于源程序语言和机器语言之间, 容易将它翻译成目标代码。另外, 还可以在中间代码一级进行与机器无关的优化。

5. 目标代码的生成

根据优化后的中间代码, 可生成有效的目标代码。而通常编译器将其翻译为汇编代码, 此时还需要将汇编代码经汇编器汇编为目标机器的机器语言。

四、 汇编器

(一) 汇编器概述

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”。汇编器将 `main.s` 翻译成机器语言指令，把这些指令打包成可重定位目标程序。得到 `.o` 文件，是一个二进制文件，它的字节码是机器语言指令，不再是字符。前面两个阶段都还有字符。

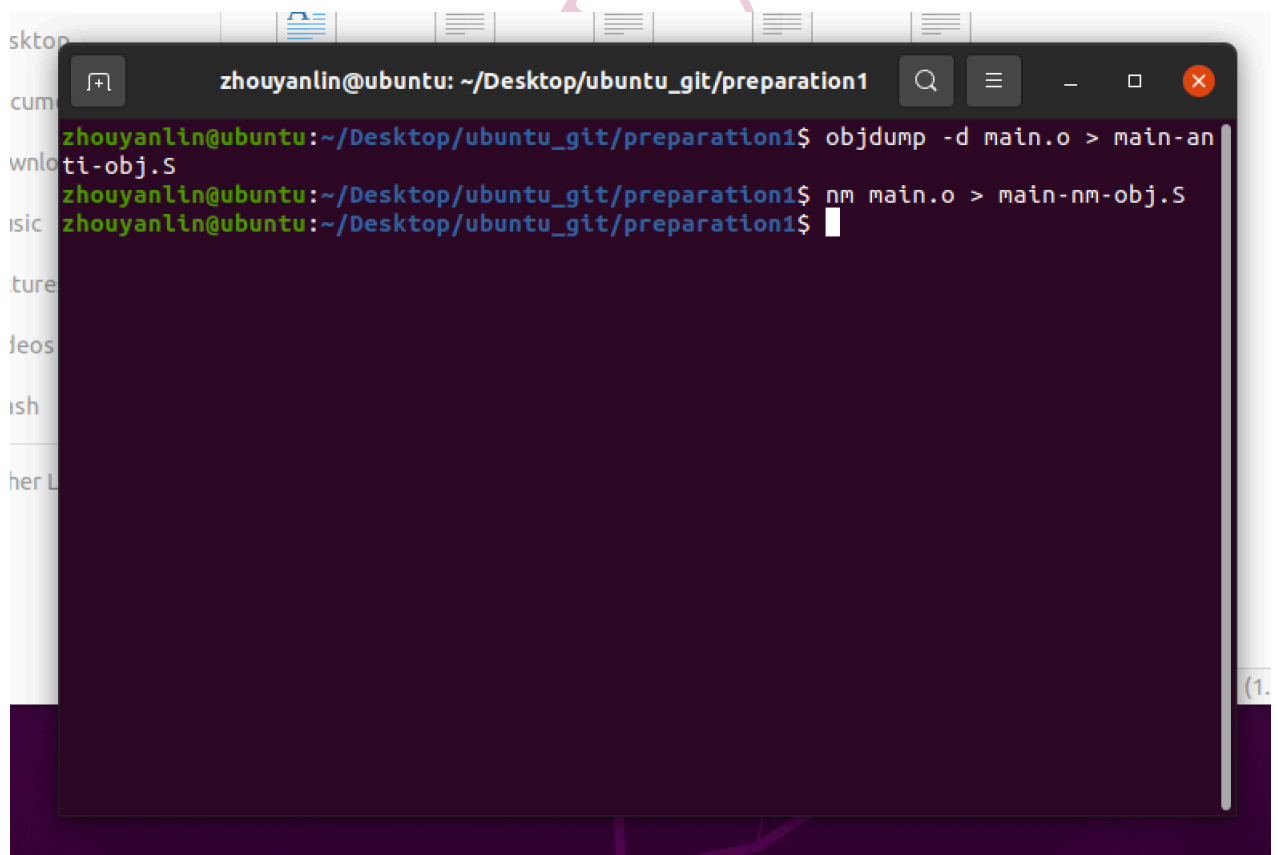
并且汇编器需要加载程序到硬件，主要分为以下两个步骤：

1. 指定要加载的目标位置，是一个内存地址，可以由程序计数器指定
2. 手动设置一系列开关，实现加载的功能，将二进制模式（存储在某个非易失性存储设备（纸带等））输入到 1 中的目标内存地址中。

(二) 汇编器过程

1. 运行截图

运行时的代码如图3所示：



```
zhouyanlin@ubuntu: ~/Desktop/ubuntu_git/preparation1
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation1$ objdump -d main.o > main-anti-obj.s
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation1$ nm main.o > main-nm-obj.s
zhouyanlin@ubuntu:~/Desktop/ubuntu_git/preparation1$
```

图 3: 运行截图

2. 生成代码内容

运行完如上代码后会生成 `main-anti-obj.S` 和 `main-nm-obj.txt` 两个文件，文件的内容如下所示：

main-anti-obj.S

```

1 main.o:      file format elf64-x86-64
2
3
4 Disassembly of section .text:
5
6 0000000000000000 <main>:
7   0:   f3 0f 1e fa      endbr64
8   4:   55                push   %rbp
9   5:   48 89 e5          mov    %rsp,%rbp
10  8:   48 83 ec 20       sub    $0x20,%rsp
11  c:   64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
12 13:  00 00
13 15:  48 89 45 f8       mov    %rax,-0x8(%rbp)
14 19:  31 c0             xor    %eax,%eax
15 1b:  48 8d 45 ec       lea    -0x14(%rbp),%rax
16 1f:  48 89 c6          mov    %rax,%rsi
17 22:  48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi      # 29 <main+0x29>
18 29:  b8 00 00 00 00    mov    $0x0,%eax
19 2e:  e8 00 00 00 00    callq 33 <main+0x33>
20 33:  c7 45 f0 02 00 00 00 movl    $0x2,-0x10(%rbp)
21 3a:  c7 45 f4 01 00 00 00 movl    $0x1,-0xc(%rbp)
22 41:  eb 0e            jmp    51 <main+0x51>
23 43:  8b 45 f4          mov    -0xc(%rbp),%eax
24 46:  0f af 45 f0       imul   -0x10(%rbp),%eax
25 4a:  89 45 f4          mov    %eax,-0xc(%rbp)
26 4d:  83 45 f0 01       addl   $0x1,-0x10(%rbp)
27 51:  8b 45 ec          mov    -0x14(%rbp),%eax
28 54:  39 45 f0          cmp    %eax,-0x10(%rbp)
29 57:  7e ea            jle    43 <main+0x43>
30 59:  8b 45 f4          mov    -0xc(%rbp),%eax
31 5c:  89 c6            mov    %eax,%esi
32 5e:  48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi      # 65 <main+0x65>
33 65:  b8 00 00 00 00    mov    $0x0,%eax
34 6a:  e8 00 00 00 00    callq 6f <main+0x6f>
35 6f:  b8 00 00 00 00    mov    $0x0,%eax
36 74:  48 8b 55 f8       mov    -0x8(%rbp),%rdx
37 78:  64 48 33 14 25 28 00 xor    %fs:0x28,%rdx
38 7f:  00 00
39 81:  74 05            je     88 <main+0x88>
40 83:  e8 00 00 00 00    callq 88 <main+0x88>
41 88:  c9              leaveq
42 89:  c3              retq

```

main-nm-obj.txt

```

1      U _GLOBAL_OFFSET_TABLE_
2      U __isoc99_scanf

```

```
3 0000000000000000 T main
4             U printf
5             U __stack_chk_fail
```

3. 代码分析

对于 **main-nm-obj.txt** 文件来看，可以分析出其主要是包含了一些需要导入的外部函数的信息，例如输入输出流函数等等。

最核心的应该是 **main-anti-obj.S** 文件，对其可做如下分析：

- 第 8-10 行很明显是对于调用 `main` 函数时给函数分配栈帧的操作
- 第 13-18 行是为三个 `int` 标识符来分配空间
- 第 19 行是调用 `scanf`
- 第 20、21 行是对 `i`、`f` 的立即数赋值操作
- 第 22 行代表要进入 `while` 循环体中
- 第 23-27 行是对 `while` 循环体中运算的进行
- 第 28、29 行是对 `while` 循环条件的判断
- 第 30-42 行是对 `printf` 函数的调用以及返回 `main` 函数的值并退出程序

汇编器的分析到这里也到此结束。

五、 链接器

(一) 链接器概述

典型的链接器把由编译器或汇编器生成的若干个目标模块，整合成一个被称为载入模块或可执行文件的实体—该实体能被操作系统直接执行。其中，某些目标模块是直接作为输入提供给链接器的；而另外一些目标模块则是根据链接过程的需要，从包括有类似 `printf` 函数的库文件中取得的。

1. 识别名称

链接器通常把目标模块看成是由一组外部对象（external object）组成的。每个外部对象代表机器内存中的某个部分，并通过一个外部名称来识别。因此，程序中的每个函数和每个外部变量，如果没有声明为 `static`，就都是一个外部对象。某些 C 编译器会对静态函数和静态变量的名称做一定改变，将它们也作为外部对象。由于经过了“名称修饰”，因此它们不会与其他源程序文件中的同名函数或同名变量发生命令冲突。

2. 输入与输出

链接器的输入是一组目标模块和库文件。链接器的输出是一个载入模块。链接器读入目标模块和库文件，同时生成载入模块。对每个目标模块中的每个外部对象，链接器都要检查载入模块，看是否有同名的外部对象。如果没有，链接器就将该外部对象添加到载入模块中；如果有，链接器就要开始处理命名冲突。

3. 外部引用

除了外部对象，目标模块还可能包括了对其他模块中的外部对象的引用，例如，一个调用了函数 `printf` 的 C 程序所生成的目标模块，就包括了一个对函数 `printf` 的引用。可以推测得出，该引用指向的是一个位于某个库文件中的外部对象。在链接器读入一个目标模块时，它必须解析出这个目标模块中定义的所有外部对象的引用，并做出标记说明这些外部对象不再是未定义的。

(二) 链接器工作

链接器之所以存在或者产生，基本上是由于程序开发的模块化。这里讲的模块，主要是编译概念上的模块，通常他们按照功能划分，比如一个 `.c` 或者 `.cpp` 文件就是一个编译单元，就是一个模块，编译后就产生一个 `.o` 目标文件。为了最终生成一个可执行文件、静态库或者动态库，就需要把各个编译单元按照特定的约定组合到一起。这里特定的约定指的就是“目标文件格式”，它定义了目标文件、库文件和可执行文件的格式，这里组合这一过程就叫做链接。

1. 构造符号表

一个编译模块中，通常是函数的定义和全局数据的定义，数据类型的定义通常在头文件中，编译时会被包含在编译模块中。函数和数据由符号来标识，一般符号有全局和静态之分，全局符号可以被其他模块引用，而静态符号只能在本模块中引用。编译各个模块时，编译器会解析该模块。重要的一项工作就是建立符号表，符号表中包含了本模块有哪些符号可以被其他模块引用（导出符号），还包括本模块引用（导入符号，即未定义符号）、但在其他模块中定义的符号。每一个符号都关联一个地址，这个地址指明了该符号在本模块中的偏移地址（通常是一个从 0 开始的地址）。

链接器在链接过程中，会扫描各个模块的符号表，得到一个“全局符号表”，链接器由此决定一个符号在哪里被定义，在哪里被引用。并且，将符号引用处替换为定义处的地址，这一过程就叫做符号解析。

2. 文件生成

链接器的一项终极目标就是生成可执行文件。通常，可执行文件和普通目标文件的重要区别就是地址空间的使用。主流操作系统中，可执行文件都是基于虚拟地址空间的，即每个可执行文件都有相同且独立的地址空间，并且文件中各个段（代码段，数据段，以及进程空间中的堆栈段）都有相似的布局。而普通目标文件却使用从零开始的地址空间，这样一来，模块 M 中的符号 m 就可能和模块 N 中的符号 n 拥有“相同”的地址。在链接器链接各个模块时，会从各个模块中“提取”类型相同的段进行合并，并将合并后的段写入可执行文件中。这一过程被称为存储空间的分配。值得一提的是，栈、堆以及未初始化的数据这些“运行时”需要的空间不会在可执行文件中占据磁盘空间，但它们占用相应的地址空间。

由于存在上述“合并”过程，前面提到的符号解析就涉及到另外一个过程：重定位。由于各个模块中的函数/数据地址会被重新排放，那么对这些符号的引用也必须被相应地调整。这一调整过程被称作重定位。

符号解析，存储空间分配，还有重定位，这三个过程是一个有机的整体，是“同时”进行的，且这三个过程也是模块化所带来的必须要解决的问题。

六、 LLVM IR 编程

(一) 组员分工

本次实验我们两个组员针对语言特性进行分工，并最后对自己分到的语言特性进行 LLVM 中间语言的编写代码并撰写文档，最后将两个人的整合后放到 ubuntu20.04 下用虚拟机可以跑通，所以也算是通过了验证。

具体分工任务如下（共有两个任务，每个组员选一个）：

- 主函数中各种运算、条件、循环语句及对应的终止符等语言特性，对应特性 1、4、5、7、9、11
- 全局变量、子两数、空函数、函数的声明与定义、各种变量类型等语言特性，对应特性 2、3、6、8、10

本次代码的链接如下：

[LLVM IR 编写代码](#)

(二) 语言特性描述

接下来对手写的 LLVM IR 语言特性进行描述，共分为 11 个方面进行描述：

1. 特性一

```
1 ; ModuleID = 'test.c'
2 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
3 target triple = "x86_64-pc-linux-gnu"
4
```

图 4: 语言特性 1-1

如上图所示，LLVM IR 的基本单位成为 module（只要是单文件编译就只涉及单 module），对应 SysY 中的 CompUnit—— $\text{CompUnit} ::= [\text{CompUnit}] (\text{Decl} \mid \text{FuncDef})$ ，一个 CompUnit 中有且仅有一个 main 函数定义，是程序的入口。

2. 特性二

一个 module 中可以包含多个顶层实体，如 function 和 global variable，CompUnit 的顶层变量/常量声明语句（对应 Decl），函数定义（对应 FuncDef）都不可以重复定义同名标识符（IDENT），即便标识符的类型不同也不允许。如图5所示，在同一个 module 中包含了多个全局变量；如图6和图7所示，在同一个 module 中包含了多个函数。如图5所示，在定义整形的 globalX 后，若再定义一个浮点型的 globalX 则会报错，故将后者注释掉；如图7和图8所示，在定义了一个无参的 emptyFunction 之后，若再定义一个有参的 emptyFunction 则会报错，故将后者注释掉。综上可知，CompUnit 的顶层变量/常量声明语句，函数定义都不可以重复定义同名标识符

3. 特性三

一个 function define 中至少有一个 basicblock。basicblock 对应 SysY 中的 Block 语句块，语句块内声明的变量的生存期在该语句块内。如图9所示，即使是一个空函数，也至少需要一个

```
@str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@globalX = common global i32 @, align 4
; @globalX = common global float 0.000000e+00, align 4
@globalY = common global float 0.000000e+00, align 4
; @globalZ = common global @, align 4
@globalArr = common global [3 x i32] zeroinitializer, align 4
@globalPoint = common global i32* null, align 8
```

图 5: 特性二-1

```
; Function Attrs: nounwind uwtable
define i32 @Distance(i32 %a, i32 %b) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %res = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
```

图 6: 特性二-2

```
; Function Attrs: nounwind uwtable
define void @emptyFunction() #0 {
    ret void
}
```

图 7: 特性二-3

```
; Function Attrs: nounwind uwtable
; define void @emptyFunction(i32 %a) #0 {
;     ret void
; }
```

图 8: 特性二-4

```
; Function Attrs: nounwind uwtable
define void @emptyFunction() #0 {
    ret void
}
```

图 9: 特性三-1

```
; <label>:14                                ; preds = %10, %6
    %15 = load i32, i32* %res, align 4
    store i32 %c, i32* %16, align 4
    ret i32 %15
}
```

图 10: 特性三-2

基本块。如图10所示，虽然主函数中定义了变量 C，但是在子函数中不能使用 C，这说明声明变量的生存期在对应语句块内。

4. 特性四

以下四张图片展示了 LLVM 中间语言的不同 instruction 的不同操作，基本上覆盖了各种的指令类型

```
73 store i32 0, i32* %1, align 4
74 store i32 1, i32* %a, align 4
75 store i32 5, i32* %b, align 4
76 store float 9.000000e+00, float* %c, align 4
77 store float 1.000000e+01, float* %d, align 4
78 store i32 0, i32* %i, align 4
```

图 11: 赋值指令

```
81 ; <label>:2                                ; preds = %22, %0
82 %3 = load i32, i32* %i, align 4
83 %4 = icmp slt i32 %3, 5
84 br i1 %4, label %5, label %24
```

图 12: 跳转指令

```

89  %8 = add nsw i32 %6, %7
90  store i32 %8, i32* %a, align 4
91  %9 = load float, float* %c, align 4
92  %10 = load float, float* %d, align 4
93  %11 = fmul float %9, %10

```

图 13: 运算指令

```

135  %35 = load i32, i32* %a, align 4
136  %36 = load i32, i32* %b, align 4
137  %37 = call i32 @Distance(i32 %35, i32 %36)
138  store i32 %37, i32* %res, align 4

```

图 14: 函数调用指令

5. 特性五

如下图所示, llvm IR 中注释以; 开头, SysY 中与 C 语言一致:

```

58  ; <label>:14                                ; preds = %10, %6
59  %15 = load i32, i32* %res, align 4
60  ; store i32 %c, i32* %16, align 4
61  ret i32 %15

```

图 15: 注释

6. 特性六

```

; @globalX = common global float 0.000000e+00, align 4
@globalY = common global float 0.000000e+00, align 4
; @globalZ = common global 0, align 4
@globalArr = common global [3 x i32] zeroinitializer, align 4

```

图 16: 特性六-1

llvm IR 是静态类型的, 即每个值的类型在编写时是确定的。如图16所示, 在定义变量 globalZ 时并没有声明变量类型, 导致结果报错, 这说明每个值的类型在编写时是确定的。

7. 特性七

如以下两张图片所示, llvm IR 中全局变量和函数都以 @ 开头, 且会在类型 (如 i32) 之前用 global 标明, 局部变量以 % 开头, 其作用域是单个函数, 临时寄存器 (上文中的 %1 等) 以升序阿拉伯数字命名


```

5  @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
6  @globalX = common global i32 0, align 4
7  ; @globalX = common global float 0.000000e+00, align 4
8  @globalY = common global float 0.000000e+00, align 4
9  ; @globalZ = common global 0, align 4
10 @globalArr = common global [3 x i32] zeroinitializer, align 4
11 @globalPoint = common global i32* null, align 8
12

```

图 17: 全局变量

```

65 define i32 @main() #0 {
66   %1 = alloca i32, align 4
67   %a = alloca i32, align 4
68   %b = alloca i32, align 4
69   %c = alloca float, align 4
70   %d = alloca float, align 4
71   %i = alloca i32, align 4
72   %res = alloca i32, align 4

```

图 18: 局部变量

8. 特性八

```

; Function Attrs: nounwind uwtable
define i32 @Distance(i32 %a, i32 %b) #0 {

```

图 19: 特性八-1

```

declare i32 @printf(i8*, ...) #1
declare i32 @sum(i32 %a, i32 %b) #1

```

图 20: 特性八-2

函数定义的语法可以总结为：define + 返回值 (i32) + 函数名 (@main) + 参数列表 ((i32 %a,i32 %b)) + 函数体 (ret i32 0)，函数声明你可以在 main.ll 的最后看到，即用 declare 替换 define。如图19所示，展示了函数定义的例子；如图20所示，展示了函数声明的例子。

9. 特性九

如以下两张图片所示，终结指令一定位于一个基本块的末尾，如 ret 指令会令程序控制流返回到函数调用者，br 指令会根据后续标识符的结果进行下一个基本块的跳转，br 指令包含无条件 (br+label) 和有条件 (br+ 标志符 +truelabel+falselabel) 两种

```

81 ; <label>:2 ; preds = %22, %0
82 %3 = load i32, i32* %i, align 4
83 %4 = icmp slt i32 %3, 5
84 br i1 %4, label %5, label %24
85

```

图 21: br 指令

```

133 ; <label>:33                                ; preds = %30, %27
134 %34 = call i32 @i8*, ... @printf(i8* getelementptr inbounds
135 %35 = load i32, i32* %a, align 4
136 %36 = load i32, i32* %b, align 4
137 %37 = call i32 @Distance(i32 %35, i32 %36)
138 store i32 %37, i32* %res, align 4
139 ret i32 0

```

图 22: ret 指令

10. 特性十

```

@globalX = common global i32 0, align 4
; @globalX = common global float 0.000000e+00, align 4
@globalY = common global float 0.000000e+00, align 4
; @globalZ = common global 0, align 4
@globalArr = common global [3 x i32] zeroinitializer, align 4
@globalPoint = common global i32* null, align 8

```

图 23: 特性十-1

```

store i32 0, i32* @globalX, align 4
store float 0.000000e+00, float* @globalY, align 4
store i32 0, i32* @globalArr, align 4
store i32* null, i32** @globalPoint, align 8
ret void

```

图 24: 特性十-2

i32 这个变量类型实际上就指 32 bit 长的 integer，类似的还有 void、label、array、pointer 等。如图23所示，分别定义了整形的 globalX、浮点数 globalY、数组 globalArr、指针 globalPoint。如图24所示，分别展示了对 globalX、globalY、globalArr、globalPoint 的赋值操作。

11. 特性十一

如下图所示，绝大多数指令的含义就是其字面意思，load 从内存读值，store 向内存写值，add 相加参数，alloca 分配内存并返回地址等

```

86 ; <label>:5                                ; preds = %2
87 %6 = load i32, i32* %a, align 4
88 %7 = load i32, i32* %b, align 4
89 %8 = add nsw i32 %6, %7
90 store i32 %8, i32* %a, align 4
91 %9 = load float, float* %c, align 4
92 %10 = load float, float* %d, align 4
93 %11 = fmul float %9, %10

```

图 25: ret 指令

七、 总结与展望

(一) 文件内容展示

最后在整个过程中生成的所有的文件如图26所示：

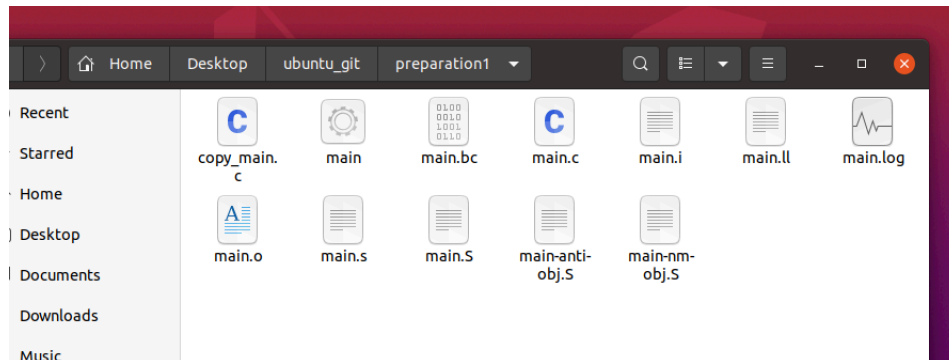


图 26: 运行截图

(二) 实验总结

本次实验是编译原理的第一次实验，首先对总体编译过程进行分析，接下来对预处理器、编译器、汇编器、链接器的具体处理细节进行分析，最后针对自己未来所要实现的 SysY 语言特性进行中间过程语言（LLVM IR）进行编写并分析，也在 clang 的编译下调试通过。

(三) 未来展望

本次是第一次实验，对编译方面的相关知识有了更深入、更全面的理解，对于各个部分所要实现的功能以及未来自己所要实现的语言特性有了一定的掌握，期望自己在这个学期可以取得一个满意的结果。