预备工作 1——了解编译器及 LLVM IR 编程

杨俣哲 李煦阳
孙一丁 李世阳 杨科迪
周辰霏 尧泽斌

日期：2020 年 9 月至 2022 年 9 月

目录

1 实验描述

以你熟悉的编译器，如 GCC、LLVM 等为研究对象，深入地探究语言处理系统的完整工作过程：

1. 完整的编译过程都有什么？
2. 预处理器做了什么？
3. 编译器做了什么？
4. 汇编器做了什么？
5. 链接器做了什么？
6. 通过编写 LLVM IR 程序，熟悉 LLVM IR 中间语言。

并尽可能地对其实现方式有所了解。

1.1 方法

以一个简单的 C（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，以此撰写调研报告。二进制文件或许需要利用某些系统工具理解，如 `objdump`、`nm`。

进一步地，可以调整你认为**关键**的编译参数（如优化参数、链接选项参数），比较目标程序的大小、运行性能等。

你的源程序可以包含尽可能丰富的语言特性（如函数、全局变量、常量、各类宏、头文件...），以更全面探索每一个阶段编译器进行的工作。

1.2 实验要求

要求：

撰写调研报告（符合科技论文写作规范，包含完整结构：题目、摘要、关键字、引言、你的工作和结果的具体介绍、结论、参考文献，文字、图、表符合格式规范，建议使用 latex 撰写）[1]

（可基于此模板，该模板所在网站是一个很流行的 latex 文档协同编辑网站，copy 此 project 即可成为自己的项目，在其上编辑即可，更多 latex 参考资料[2]）。

期望： 不要当作“命题作文”，更多地发挥主观能动性，做更多探索。如：

1. 细微修改程序，观察各阶段输出的变化，从而更清楚地了解编译器的工作；
2. 调整编译器的程序选项，例如加入调试选项、优化选项等，观察输出变化、了解编译器；
3. 尝试更深入的内容，例如令编译器做自动并行化，观察输出变化、了解编译器；
4. 与预习作业 1 中的优化问题相结合等等。

[1] 你可以搜索“vscode+latex workshop”以配置 `latex` 环境，再进一步了解“如何用 latex 书写中文”。

[2] LaTeX 入门、LaTeX 命令与符号汇总、LaTeX 数学公式等符号书写

基础样例程序：

```
int main()
{
    int i, n, f;
    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
}
```

2) 斐波那契数列

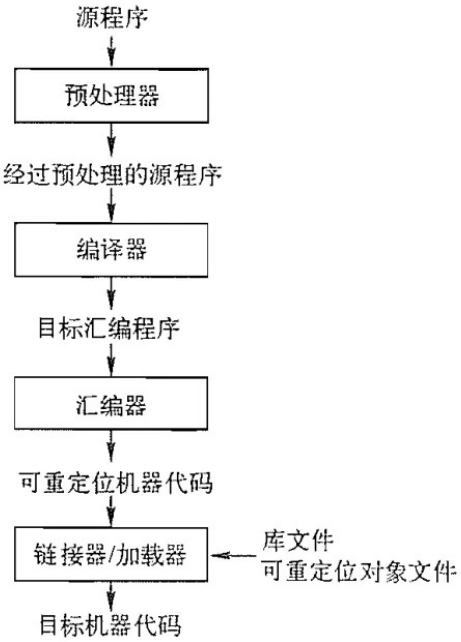
```
int main()
{
    int a, b, i, t, n;

    a = 0;
    b = 1;
    i = 1;
    cin >> n;
    cout << a << endl;
    cout << b << endl;
    while (i < n)
    {
        t = b;
        b = a + b;
        cout << b << endl;
        a = t;
        i = i + 1;
    }
}
```

2 参考流程

以下内容仅供参考，更多的细节希望同学们亲自动手体验，详细了解各阶段的作用。

以一个 C 程序为例，整体的流程如图所示：



简单来说，不同阶段的作用如下：

预处理器 处理源代码中以 # 开始的预编译指令，例如展开所有宏定义、插入 `include` 指向的文件等，以获得经过预处理的源程序。

编译器 将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。

汇编器 将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。

链接器 将可重定位的机器代码和相应的一些目标文件以及库文件连接在一起，形成真正能在机器上运行的目标机器代码。

我们将以一段简单的 C 代码为例：

```c
#include<stdio.h>
int main(){
    int a,b;
    // 输入变量
    scanf("%d%d",&a,&b);
    // 输出结果
    printf("Hello World %d\n",a+b);
    return 0;
}
```

2.1 预处理器

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，有兴趣的同学可以自行尝试。

2.2 编译器

编译过程是我们整门课程着重讲述的过程，具体来说分为六步，详细解释可以查看课程的预习 PPT，简单来说分别为

词法分析 将源程序转换为单词序列。对于 LLVM，你可以通过以下命令获得 token 序列：

```
clang -E -Xclang -dump-tokens main.c
```

语法分析 将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree，即 AST）。对于 gcc，你可以通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出。LLVM 可以通过如下命令获得相应的 AST：

```
clang -E -Xclang -ast-dump main.c
```

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。

中间代码生成 完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。

你可以通过`-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，vscode 中直接有相应插件。你可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。你可以额外使用`-Ox`、`-fno-*` 等 flag 控制编译行为，使输出文件更可读、了解其优化行为。

LLVM 可以通过下面的命令生成 LLVM IR：

```
clang -S -emit-llvm main.c
```

代码优化 进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。

在第一周的预习作业中，很多同学对编译器如何进行代码优化感到疑问，在这个步骤中你可以通过 LLVM 现有的优化 pass 进行代码优化探索。

在 LLVM 官网对所有 pass 的分类[3]中，共分为三种：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

LLVM 可以通过下面的命令生成每个 pass 后生成的 LLVM IR，以观察差别：

```
llc -print-before-all -print-after-all a.ll > a.log 2>&1
# 因为输出的内容过长，在命令行中无法完整显示，这时必须要对输出进行重定向
# 0、1、2 是三个文件描述符，分别表示标准输入 (stdin)、标准输出 (stdout)、标准错误 (stderr)
# 因此 2>&1 的具体含义就不难理解，你也可以试试去掉重定向描述，看看实际效果
```

同样，你也可以通过下面的命令指定使用某个 pass 以生成 LLVM IR，以特别观察某个 pass 的差别：

```
opt -<module name> <test.bc> /dev/null
```

所有的 module name 对应的命令行参数也可以在[4]查到。

上面的指令需要用到 bc 格式，即 LLVM IR 的二进制代码形式，而我们之前生成的是 LLVM IR 的文本形式。当然我们也可以通过添加额外命令行参数的方式直接使用 ll 格式的 LLVM IR，这里留给同学们自行探究。

我们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：

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

如果你认为本部分的探索内容过于“黑盒”，你也可以尝试去阅读各大编译器，如 LLVM 各个 pass 的源码[5]。尽管你现在可能并不了解大多数 pass 实际上是怎么工作的，但可能对你大作业的代码优化部分程序的编写有帮助。

代码生成 以中间表示形式作为输入，将其映射到目标语言

[3] LLVM 对所有 pass 的简述
[4] LLVM 的 pass 参数
[5] LLVM 源码所在仓库

```
gcc main.i -S -o main.S # 生成 x86 格式目标代码
arm-linux-gnueabihf-gcc main.i -S -o main.S # 生成 arm 格式目标代码
llc main.ll -o main.S # LLVM 生成目标代码
```

2.3 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”，你可以在一些网上资料，比如这里，进行宏观的了解。

希望同学们在报告中详细分析并阐述汇编器处理的结果以及汇编器的具体功能分析。你可能会用到反编译工具（你可以在文后的 Makefile 中找到简单的使用示例）。

x86 格式汇编可以直接用 gcc 完成汇编器的工作，如使用下面的命令：

```
gcc main.S -c -o main.o
```

arm 格式汇编需要用到交叉编译，如使用下面的命令：

```
arm-linux-gnueabihf-gcc main.S -o main.o
```

LLVM 可以直接使用 llc 命令同时汇编和链接 LLVM bitcode：

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

2.4 链接器加载器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而连接器对该机器代码进行执行生成可执行文件。可以尝试对可执行文件反汇编，看一看与上一阶段反汇编结果的不同。

在这一阶段，你可以尝试调整链接相关参数，如`-static`。

```
gcc main.o -o main
```

当你执行可执行文件时，便会使用到加载器，以将二进制文件载入内存。这不是我们要研究的事了。

2.5 LLVM IR 编程

LLVM IR（Intermediate Representation）是由代码生成器自顶向下遍历逐步翻译语法树形成的，你可以将任意语言的源代码编译成 LLVM IR，然后由 LLVM 后端对 LLVM IR 进行优化并编译为相应平台的二进制程序。LLVM IR 具有类型化、可扩展性和强表现力的特点。LLVM IR 是相对于 CPU

指令集高级、但作为低级的代码中间表示的一种语言。从上述介绍中可以看出 LLVM 后端支持相当多的平台，我们无须担心操作系统等平台的问题，而且我们只需将代码编译成 LLVM IR，就可以由优化水平较高的 LLVM 后端来进行优化。此外，LLVM IR 本身更贴近汇编语言，指令集相对底层，能灵活地进行低级操作。

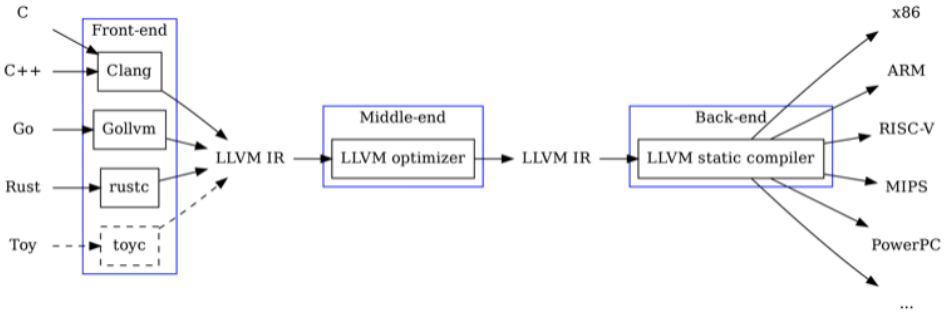


图 2.1: LLVM IR 设计架构,参考

LLVM IR 代码存在三种表示形式：在内存中的表示（BasicBlock、Instruction 等 cpp 类）、二进制代码形式（用于编译器加载）、可读的汇编语言表示形式。除了上面提到的`clang -S -emit-llvm main.c`，你也可以通过`clang -c -emit-llvm main.c -o mian.bc`生成 bitcode 形式的 LLVM IR 文件。

下面以1.2中基本样例程序为例对 LLVM IR 特性进行简单介绍。更多有关 LLVM IR 的结构问题，可以参考LLVM Programmer Manual。

首先你需要在命令行中输入`clang -emit-llvm -S main.c -o main.ll`,打开同目录下的 `main.ll` 文件，你可以得到以下内容（本指导书已删除无用语句，加入 llvm IR 相关注释及其与 SysY 语言特性的对应关系）：

```
; 所有的全局变量都以 @ 为前缀，后面的 global 关键字表明了它是一个全局变量
; SysY 语言中注释的规范与 C 语言一致
; 函数定义以 `define` 开头，i32 标明了函数的返回类型，其中 `main`是函数的名字，`@` 是其前缀
; FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block; FuncDef 表示函数定义，FuncType 指明了函数的返回类型，FuncParam 是函数定义的形参列表
define i32 @main() #0 {
  ; 以 % 开头的符号表示虚拟寄存器，你可以把它当作一个临时变量（与全局变量相区分），或称之为临时寄存器
  %1 = alloca i32, align 4
  ; 为 %1 分配空间，其大小与一个 i32 类型的大小相同。%1 类型即为 i32*，align 4 可以理解为对齐方式为 4 个字节
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  ; 将 0（i32）存入 %1（i32*）
  store i32 0, i32* %1, align 4
  ; 调用函数 @scanf ，i32 表示函数的返回值类型
```

```
15    %5 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8],
      ↪  [3 x i8]* @.str, i64 0, i64 0), i32* %3)
16    store i32 2, i32* %2, align 4
17    store i32 1, i32* %4, align 4
18    ; 这里的 br 是无条件分支，label 可以理解为一个代码标签，指代下面那个代码块
19    br label %6
20
21  6:                                                ; preds = %10, %0
22    %7 = load i32, i32* %2, align 4
23    %8 = load i32, i32* %3, align 4
24    ; icmp 会根据不同的比较规则（这里是 sle，小于等于）比较两个操作数%7 和%8，i32 是操作
  ↪  数类型
25    %9 = icmp sle i32 %7, %8
26    ; 这里的 br 是有条件分支，它根据 i1 和两个 label 的值，用于将控制流传输到当前函数中
  ↪  的不同基本块。
27    ; i1 类型的变量%cmp 的值如果为真，那么执行 label%10，否则执行 label%16
28    br i1 %9, label %10, label %16
29
30  10:                                               ; preds = %6
31    %11 = load i32, i32* %4, align 4
32    %12 = load i32, i32* %2, align 4
33    %13 = mul nsw i32 %11, %12
34    store i32 %13, i32* %4, align 4
35    %14 = load i32, i32* %2, align 4
36    %15 = add nsw i32 %14, 1
37    store i32 %15, i32* %2, align 4
38    br label %6, !llvm.loop !10
39
40  16:                                               ; preds = %6
41    %17 = load i32, i32* %4, align 4
42    %18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
      ↪  i8]* @.str.1, i64 0, i64 0), i32 %17)
43    ret i32 0
44  }
45  ; 函数声明
46  declare dso_local i32 @__isoc99_scanf(i8*, ...)
47
48  declare dso_local i32 @printf(i8*, ...)
49
```

根据上述.ll 文件，我们对 LLVM IR 及 SysY 特性做以下总结：

1. 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`——`CompUnit ::= [CompUnit] (Decl | FuncDef)`，一个 `CompUnit` 中有且仅有一个 `main` 函数定义，是程序的入口。

2. 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

3. 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。`Block` 表示为
   Block ::= "" BlockItem "";
   BlockItem ::= Decl | Stmt;

4. 每个 `basicblock` 中有若干 `instruction`，且都以 `terminator instruction` 结尾。SysY 中语句表示为
   Stmt ::= LVal "=" Exp ";"
   | [Exp] ";"
   | Block
   | "if" "(" Exp ")" Stmt ["else" Stmt]
   | "while" "(" Exp ")" Stmt
   | "break" ";"
   | "continue" ";"
   | "return" [Exp] ";";

5. llvm IR 中注释以; 开头，SysY 中与 C 语言一致

6. llvm IR 是静态类型的，即每个值的类型在编写时是确定的

7. 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

8. 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`。SysY 中函数定义表示为`FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block`

9. 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

10. `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

11. 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

关于 LLVM IR 可以从官方文档进行更多了解，一些常用的指令[6]。有关 SysY 语言更多内容可以参考官方文档[7][8]。

[6] LLVM IR 常用指令
[7] SysY 语言定义
[8] SysY 运行时库

2.6 样例 Makefile 文件

```
.PHONY: pre, lexer, ast-gcc, ast-llvm, cfg, ir-gcc, ir-llvm, asm, obj, exe, antiobj,
antiexe

pre:
    gcc main.c -E -o main.i

lexer:
    clang -E -Xclang -dump-tokens main.c

# 生成`main.c.003t.original`
ast-gcc:
    gcc -fdump-tree-original-raw main.c

# 生成`main.ll`
ast-llvm:
    clang -E -Xclang -ast-dump main.c

# 会生成多个阶段的文件（.dot），可以被 graphviz 可视化，可以直接使用 vscode 插件
# (Graphviz (dot) language support for Visual Studio Code)。
# 此时的可读性还很强。`main.c.011t.cfg.dot`
cfg:
    gcc -O0 -fdump-tree-all-graph main.c

# 此时可读性不好，简要了解各阶段更迭过程即可。
ir-gcc:
    gcc -O0 -fdump-rtl-all-graph main.c

ir-llvm:
    clang -S -emit-llvm main.c

asm:
    gcc -O0 -o main.S -S -masm=att main.i

obj:
    gcc -O0 -c -o main.o main.S

antiobj:
    objdump -d main.o > main-anti-obj.S
    nm main.o > main-nm-obj.txt

```

```
41  exe:
42      gcc -O0 -o main main.o
43
44  antiexe:
45      objdump -d main > main-anti-exe.S
46      nm main > main-nm-exe.txt
47
48  clean:
49      rm -rf *.c.*
50
51  clean-all:
52      rm -rf *.c.* *.o *.S *.dot *.out *.txt *.ll *.i main
```