lab3 实验报告

学号：202008010415 姓名：温先武

0. 前言

本次实验作为 Lab4 的前驱实验，独立于 Lab1、Lab2。

本次实验的目的是让大家熟悉 Lab4 所需要的相关知识: LLVM IR、 LightIR（LLVM IR 的轻量级 C++接口）和 Visitor Pattern（访问者模式）。

在开始实验之前，如果你使用的不是助教提供的虚拟机镜像，请根据之前的环境准备确保 LLVM 的版本为 10.0.1，且 PATH 环境变量配置正确。可以通过 `lli --version` 命令是否可以输出 10.0.1 的版本信息来验证。

主要工作

1. 第一部分: 了解 LLVM IR。通过 clang 生成的.ll，了解 LLVM IR 与 c 代码的对应关系。完成 1.3
2. 第二部分: 了解 LightIR。通过助教提供的 c++例子，了解 LightIR 的 c++接口及实现。完成 2.3
3. 第三部分: 理解 Visitor Pattern。
4. 实验报告：在 report.md 中**回答 3 个问题**。

1. LLVM IR 部分

1.1 LLVM IR 介绍

根据维基百科的介绍，LLVM 是一个自由软件项目，它是一种编译器基础设施，以 C++写成，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。IR 的全称是 Intermediate Representation，即中间表示。LLVM IR 是一种类似于汇编的底层语言。

LLVM IR 的具体指令可以参考 Reference Manual。但是你会发现其内容庞杂。虽然助教认为，高效地查阅官方文档及手册是非常必要的一项技能，但是由于其手册过于复杂，因此助教筛选了后续实验中将要用到的子集，总结为了精简的 IR Reference 手册。

作为一开始的参考，你可以先阅读其中 `IR Features` 和 `IR Format` 两节，后续有需要再反复参考。实验的最后，你需要在 report.md 中**回答问题 3**。

1.2 gcd 例子：利用 clang 生成的.ll

阅读 tests/lab3/ta_gcd/gcd_array.c。

根据 `clang -S -emit-llvm gcd_array.c` 指令，你可以得到对应的 `gcd_array.ll` 文件.你需要结合 gcd_array.c 阅读 `gcd_array.ll`，理解其中每条 LLVM IR 指令与 c 代码的对应情况。

通过 `lli gcd_array.ll; echo $?`指令，你可以测试 `gcd_array.ll` 执行结果的正确性。其中，

- `lli` 会运行`*.ll` 文件
- `$?`的内容是上一条命令所返回的结果，而 `echo $?`可以将其输出到终端中

后续你会经常用到这两条指令。

1.3 你的提交 1：手动编写.ll

助教提供了四个简单的 c 程序，分别是 `tests/lab3/c_cases/`目录下的 assign.c、fun.c、if.c 和 while.c.你需要在 `tests/lab3/stu_ll/`目录中，手工完成自己的 assign_hand.ll、fun_hand.ll、if_handf.ll 和 while_hand.ll，以实现与上述四个 C

程序相同的逻辑功能.你需要添加必要的注释.`.ll`文件的注释是以";"开头的。

必要的情况下，你可以参考`clang -S -emit-llvm`的输出，但是你提交的结果必须避免同此输出一字不差。
助教会用`lli`检查你结果的正确性，并用肉眼检查你的注释。

2. LightIR 部分

2.1 LightIR - LLVM IR 的 C++接口

由于 LLVM IR 官方的 C++接口的文档同样过于冗长，助教提供了`LightIR`这一 C++接口库。你需要阅读 LightIR 核心类的介绍。

lab4 部分会要求大家通过`LightIR`根据`AST`构建生成 LLVM IR。所以你需要仔细阅读文档了解其接口的设计。

2.2 gcd 例子: 利用 LightIR + cpp 生成.ll

为了让大家更直观地感受并学会`LightIR`接口的使用，助教提供了 tests/lab3/ta_gcd/gcd_array_generator.cpp。该 cpp 程序会生成与 gcd_array.c 逻辑相同的 LLVM IR 文件。助教提供了非常详尽的注释，一定要好好利用！

该程序的编译与运行请参考 4.2 节。

2.3 你的提交 2: 利用 LightIR + cpp 编写生成.ll 的程序

你需要在`tests/lab3/stu_cpp/`目录中，编写 assign_generator.cpp、fun_generator.cpp、if_generator.cpp 和 while_generator.cpp，以生成与 1.3 节的四个 C 程序相同逻辑功能的`.ll`文件。你需要添加必要的注释。你需要在 report.md 中**回答问题 1**。

3. Lab4 的准备

3.1 了解 Visitor Pattern

Visitor Pattern(访问者模式)是一种在 LLVM 项目源码中被广泛使用的设计模式。在遍历某个数据结构（比如树）时，如果我们需要对每个节点做一些额外的特定操作，Visitor Pattern 就是个不错的思路。

Visitor Pattern 是为了解决**稳定的数据结构**和**易变的操作耦合问题**而产生的一种设计模式。解决方法就是在被访问的类里面加一个对外提供接待访问者的接口，其关键在于在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。这里举一个应用实例来帮助理解访问者模式: 您在朋友家做客，您是访问者；朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

有关 Visitor Pattern 的含义、模式和特点，有梯子的同学可参考维基百科。

下面的例子可以清晰地展示 Visitor Pattern 的运作方式。这是助教编写的计算表达式 `4 * 2 - 2 / 4 + 5` 结果的 C++程序。

其中较为重要的一点原则在于，C++中对函数重载特性的支持。在代码 `treeVisitor.visit(node)`中，根据 `node` 对象具体类型的不同，编译器会在 `visit(AddSubNode& node)`、`visit(NumberNode& node)`、`visit(MulDivNode& node)` 三者中，选择对应的实现进行调用。你需要理解下面这个例子中 tree 是如何被遍历的。请在 report.md 中**回答问题 2**。

例子:简单的表达式计算 - `visitor.cpp`

该文件的执行结果如下:

```
$ g++ visitor.cpp -std=c++14; ./a.out
4 * 2 - 2 / 4 + 5 evaluates: 13
```

4. 实验要求

4.1 目录结构

除了下面指明你所要修改或提交的文件，其他文件请勿修改。

```
.
├── CMakeLists.txt
├── Documentations
│   ├── ...
│   ├── common                          <- LightIR 相关文档
│   └── lab3
│       └── README.md                    <- lab3 实验文档说明（你在这里）
├── include                             <- 实验所需的头文件
│   ├── ...
│   ├── lightir
├── README.md
├── Reports
│   ├── ...
│   └── lab3
│       └── report.md                    <- lab3 所需提交的实验报告，含 3 个问题（你要交）
├── src
│   ├── ...
│   └── lightir
└── tests
    ├── CMakeLists.txt
    ├── ...
    └── lab3                            <- lab3 文件夹
        ├── c_cases                      <- 4 个 c 程序
        │   ├── assign.c
        │   ├── fun.c
        │   ├── if.c
        │   └── while.c
        ├── CMakeLists.txt               <- 你在 2.3 节需要去掉注释（我们不收，你要改）
        ├── stu_cpp                      <- lab3 所需提交的 cpp 目录（你要交）
        │   ├── assign_generator.cpp
        │   ├── fun_generator.cpp
        │   ├── if_generator.cpp
        │   └── while_generator.cpp
        ├── stu_ll                       <- lab3 所需提交的.ll 目录（你要交）
        │   ├── assign_hand.ll
        │   ├── fun_hand.ll
        │   ├── if_hand.ll
        │   └── while_hand.ll
        └── ta_gcd
            ├── gcd_array.c
            └── gcd_array_generator.cpp <- 助教提供的生成 gcd_array.ll 的 cpp
```

4.2 编译、运行和验证

- 编译与运行 在 `${WORKSPACE}/build/` 下执行:

```
# 如果存在 CMakeCache.txt 要先删除
# rm CMakeCache.txt
cmake ..
make
make install
```

  你可以得到对应 `gcd_array_generator.cpp` 的可执行文件。

  在完成 2.3 时，在`${WORKSPACE}/tests/lab3/CMakeLists.txt` 中去掉对应的注释，再在`${WORKSPACE}/build/`下执行 `cmake ..`与 `make` 指令，即可得到对应的可执行文件。

- 验证

  本次试验测试案例只有`${WORKSPACE}/tests/lab3/c_cases` 中的 4 个样例。

  请大家自行验证。

  助教会执行你们的代码，并使用 `diff` 命令进行验证。

4.3 提交要求和评分标准

- 提交要求

  本实验的提交要求分为两部分: 实验部分的文件和报告，git 提交的规范性。

  - 实验部分:

    - 需要完成 `./tests/lab3/stu_ll` 目录下的 4 个文件
    - 需要完成 `./tests/lab3/stu_cpp` 目录下的 4 个文件
    - 需要在 `./Report/lab3/` 目录下撰写实验报告
      - 实验报告内容包括:

        - 实验要求、3 个问题、实验难点、实验反馈(具体参考 report.md)
        - 本次实验报告**参与**评分标准.
    - 本次实验收取 `./tests/lab3/stu_ll` 目录、`./tests/lab3/stu_cpp` 目录和 `./Report/lab3` 目录
- git 提交规范:
    - 不破坏目录结构(`report.md` 如果需要放图片，请放在`./Reports/lab3/figs/`下)
    - 不上传临时文件(凡是自动生成的文件和临时文件请不要上传)
    - git log 言之有物(不强制，请不要 git commit -m 'commit 1'，git commit -m 'sdfsdf'，每次 commit 请提交有用的 comment 信息)

实验难点：

（1） 难点一：实验要求我们根据 assign.c、fun.c、if.c 和 while.c 四个文件，写出对应的 LLVM IR 指令（也就是.ll 文件）。主要的难点就是如何根据 C 代码写出对应的 LLVM IR 指令。如图：

```c
int x[1];
int y[1];

int gcd (int u, int v) {
    if (v == 0) return u;
    else return gcd(v, u - u / v * v);
}

int funArray (int u[], int v[]) {
    int a;
    int b;
    int temp;
    a = u[0];
    b = v[0];
    if (a < b) {
        temp = a;
        a = b;
        b = temp;
    }
    return gcd(a, b);
}

int main(void) {
    x[0] = 90;
    y[0] = 18;
    return funArray(x, y);
}
```

使用 clang 生成对应的 LLVM IR 指令：

```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/ta_gcd$ clang -S -emit-l
lvm gcd_array.c
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/ta_gcd$
```

以下仅截取 gcd 函数：

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @gcd(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %4, align 4
  store i32 %1, i32* %5, align 4
  %6 = load i32, i32* %5, align 4
  %7 = icmp eq i32 %6, 0
  br i1 %7, label %8, label %10

8:                                                ; preds = %2
  %9 = load i32, i32* %4, align 4
  store i32 %9, i32* %3, align 4
  br label %20

10:                                               ; preds = %2
  %11 = load i32, i32* %5, align 4
  %12 = load i32, i32* %4, align 4
  %13 = load i32, i32* %4, align 4
  %14 = load i32, i32* %5, align 4
  %15 = sdiv i32 %13, %14
  %16 = load i32, i32* %5, align 4
  %17 = mul nsw i32 %15, %16
  %18 = sub nsw i32 %12, %17
  %19 = call i32 @gcd(i32 %11, i32 %18)
  store i32 %19, i32* %3, align 4
  br label %20

20:                                               ; preds = %10, %8
  %21 = load i32, i32* %3, align 4
  ret i32 %21
}
```

（2）难点二：实验要求我们阅读 tests/lab3/ta_gcd/gcd_array_generator.cpp 这个 cpp 文件，理解这个 cpp

文件时如何生成与 gcd_array.c 逻辑相同的 LLVM IR 指令的，同时要求我们编写相应的 cpp 文件然后生成与 1.3 节四个 C 程序逻辑相同的.ll 文件。主要难点是阅读 LightIR 这一 C++接口库，理解对应接口的使用，编写对应的文件。实验给出的例子是 gcd_array_generator.cpp 这一文件，在/build 目录下执行

cmake ../；make 即可得到对应的可执行文件，，执行即可得到对应的 LLVM IR 指令，使用 lli 运行指令即可得到对应的输出。

实验设计：

assign.c：

```c
int main(){
  int a[10];
  a[0] = 10;
  a[1] = a[0] * 2;
  return a[1];
}
```

.ll：

```
;main函数中，dso_local i32 @main()表示在main函数中
define dso_local i32 @main() #0 {

;对应语句int a[10]
%1 = alloca [10 x i32]  ;创建数组，可容纳10个int类型的数据

;a[0] = 10;
%2 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 0    ;%2为a[0]，这里是获取a[0]的指针
store i32 10, i32* %2   ;%2赋值为10，a[0]赋值为10

;a[1] = a[0] * 2;
%3 = load i32, i32* %2  ;%3加载%2的数据
%4 = mul nsw i32 %3, 2  ;%4赋值为%3的二倍
;%4就是a[0] * 2
%5 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i64 0, i64 1    ;%5存储a[1]
store i32 %4, i32* %5   ;把%4的值赋值给%5，完成a[1]赋值

;return a[1];
%6 = load i32, i32* %5  ;%6存储%5
ret i32 %6              ;返回%6

;main函数的右括号
}

```

fun.c：

```c
int callee(int a){
  return 2 * a;
}
int main(){
  return callee(110);
}
```

.ll：

```
;这里的意思是一个函数int callee(int a){
define dso_local i32 @callee(i32 %0) #0{
        ;return 2 * a
        %2 = mul i32 %0, 2; 将参数%1乘以2存入%2中，mul代表乘法
        ret i32 %2; 返回%2
}

;main函数{
define dso_local i32 @main() #0 {

;return callee(110);
%1 = call i32 @callee(i32 110)  ;%用1存储callee(110)的返回值
ret i32 %1                      ;返回%1

;右括号
}
```

这里在 callee 函数里面，我本来是打算用%1 开辟第一个空间的，但是如果使用%1 的话会报出下面的错误：

```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ lli fun_hand.ll
lli: fun_hand.ll:4:2: error: instruction expected to be numbered '%2'
        %1 = mul i32 %0, 2; 将参数%1乘以2存入%2中，mul代表乘法
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$
```

表示应该给这个命名为%2，我的理解是：程序已经有一个变量占用了%1 的位置了，所以只能用%2。

if.c：

```
int main(){
  float a = 5.555;
  if(a > 1)
    return 233;
  return 0;
}
```

.ll：

```
; main 函数
define dso_local i32 @main() #0 {
        ; float a = 5.555;
        %1 = alloca float;创建一个float空间，并返回
        store float 0x40163851E0000000, float* %1; 将浮点数5.555存入%1所指向的空间中

        ; if(a > 1)
        %2 = load float, float* %1; 取出%1的空间中的值，也就是a
        %3 = fcmp ugt float %2, 1.000;将其和1进行比较，返回结果到%3中
        br i1 %3, label %4, label %5;如果大于，则跳转到%4，否则跳转到%5

4:      ; return 233;
        ret i32 233; 返回233

5:      ; return 0;
        ret i32 0; 返回0
}

```

while.c：

```
int main(){
  int a;
  int i;
  a = 10;
  i = 0;
  while(i < 10){
    i = i + 1;
    a = a + i;
  }
  return a;
}
```

.ll 文件：

```
;main函数和左括号
define dso_local i32 @main() #0 {

;int a
%1 = alloca i32;开辟int类型空间，存储a

;int i
%2 = alloca i32;开辟int类型空间，存储i

;a = 10
store i32 10, i32* %1;%1赋值为10，也就是a = 10

;i = 0
store i32 0, i32* %2;%2赋值为0，也就是i = 0
br label %3;跳转到3

;下面是while语句while(i < 10){
3:
%4 = load i32, i32* %2;把%2也就是i的值加载到%4
;比较i和10的大小，如果小于则5为真，否则为否
%5 = icmp slt i32 %4, 10
;5若真则跳转到6，5若否则跳转到11
br i1 %5, label %6, label %11

;i = i + 1
6:
%7 = add nsw i32 %4, 1;%7保存加一后的值
```

```
%7 = add nsw i32 %4, 1;%7保存加一后的值
store i32 %7, i32* %2;存储回%2

;a = a + i
%8 = load i32, i32* %1;a的数据加载到%8
%9 = load i32, i32* %2;i的数据加载到%9
%10 = add nsw i32 %8, %9;%10保存相加的数据
store i32 %10, i32* %1;存储回%1
br label %3;跳转到3，也就是继续判断while循环是否成立

;}

;return a

11:
%12 = load i32, i32* %1;a的数据加载到%12
ret i32 %12;返回%12

;}，右括号
}
```

然后是四个.cpp 文件的编写：

助教给我们提供了 LightIR 核心类的介绍与 C++相关的接口，下面是一些要用到的知识点：

Type

- 含义：IR的类型，该类是所有类型的超类
- 成员：
  - tid_：枚举类型，表示type的类型（包含VoidType、LabelType、FloatType、Int1、Int32、ArrayType、PointerType）

这是 IR 的类型，里面声明了我们编写 cpp 文件所需要用到的类型，例如 32 位宽的整数类型 Int32Type 和数组类型 arrayType 等。

- ArrayType
  - 含义：数组类型
  - 成员：
    - contained_：数组成员的类型
    - num_elements_：数组维数
  - API：

```
static ArrayType *get(Type *contained, unsigned num_elements)
// 返回数组类型，参数依次是 数组元素的类型contained，数组元素个数num_elements
Type *get_element_type() const
// 返回数组元素类型
unsigned get_num_of_elements() const
// 返回数组元素个数
```

这是定义数组类型的代码所需要用到的 API，我们传进去的参数是数组元素的类型 Int32Type 和 10，表示定义一个可包含 10 个类型为 32 位宽的整数类型的数组。

BasicBlock

- 含义：基本块，是一个是单入单出的代码块，该类维护了一个指令链表，基本块本身属于 Value, 类型是 <label>，会被分支指令调用
- 成员：
  - instr_list_：指令链表
  - pre_basic_blocks_： bb前驱集合
  - succ_basic_blocks_：bb后继集合

这个 API 的含义是基本块，可以简单理解为我们程序中的代码块，例如:

while(i < 100) {

sum += i;

}

while(i < 100)是一个基本块，循环体中又是一个基本块。我们需要创建基本块来存储代码块里面所对应的可生成 LLVM IR 指令的 C++代码。

Function

- 含义：函数，该类描述 LLVM 的一个简单过程，维护基本块表，格式化参数表
- 成员：
  - basic_blocks_：基本块列表
  - arguments_：形参列表
  - parent_：函数属于的module

含义是函数，里面的 API Function::create 可以用来创建函数，实现函数调用。

GlobalVariable

- 含义：全局变量，该类用于表示全局变量，是 GlobalValue 的子类，根据地址来访问
- 成员：
  - is_const：是否为常量
  - init_val_：初始值
- API：由于cminusf语义要求所有的全局变量都默认初始化为0，故 `GlobalVariable` 中成员和API再构造CminusFBuilder用不到

定义全局变量的时候需要用到。

assign.cpp：

```cpp
#include "BasicBlock.h"
#include "Constant.h"
#include "Function.h"
#include "IRBuilder.h"
#include "Module.h"
#include "Type.h"

#include <iostream>
#include <memory>

#ifdef DEBUG    // 用于调试信息,大家可以在编译过程中通过" -DDEBUG"来开启这一选项
#define DEBUG_OUTPUT std::cout << __LINE__ << std::endl;        // 输出行号的简单示例
#else
#define DEBUG_OUTPUT
#endif

#define CONST_INT(num) \
    ConstantInt::get(num, module)

#define CONST_FP(num) \
    ConstantFP::get(num, module)        // 得到常数值的表示,方便后面多次用到

```

这一部分开头的声明与实验给出的例子的开头相同，主要是声明相关的头文件等，不再赘述。

```cpp
        //int a[10];
        auto *arrayType = ArrayType::get(Int32Type, 10);// 在内存中为数组分配空间,参数表示数组的元素类型和元素个数
        auto aAlloca = builder->create_alloca(arrayType);
        //a[0]=10;
        auto a0GEP = builder->create_gep(aAlloca, {CONST_INT(0), CONST_INT(0)});// 获取a[0]地址
        builder->create_store(CONST_INT(10), a0GEP);                            // a[0]=10
        a0GEP = builder->create_gep(aAlloca, {CONST_INT(0), CONST_INT(0)});     // 更新a[0]
        //a[1] = a[0] * 2;
        auto a0Load = builder->create_load(a0GEP);                              // 加载a[0]
        auto a0mul2 = builder->create_imul(a0Load, CONST_INT(2));               // a[0]*2
        auto a1GEP = builder->create_gep(aAlloca, {CONST_INT(0), CONST_INT(1)});// 获取a[1]地址
        builder->create_store(a0mul2, a1GEP);                                   // 将a[0]*2存入a[1]
        a1GEP = builder->create_gep(aAlloca, {CONST_INT(0), CONST_INT(1)});     // 更新a[1]
        //return a[1];
        auto a1Load = builder->create_load(a1GEP);                              // 加载a[1]
        builder->create_ret(a1Load);                                            // 返回a[1]

        std::cout << module->print();
        delete module;
        return 0;
}
```

这一部分就调用上述提到的模块中的 API 即可。

fun.cpp：

```
27      // callee函数，创建函数
28      std::vector<Type*> Ints(1, Int32Type);                  /* 函数参数类型的vector，内含1个int类型 */
29      auto calleeFunTy = FunctionType::get(Int32Type, Ints);  /* 通过返回值类型与参数类型列表得到函数类型 */
30      auto calleeFun = Function::create(calleeFunTy,          /* 由函数类型得到函数 */
31      "callee", module);
32      auto bb = BasicBlock::create(module, "fun", calleeFun); /* 创建基本块，命名为fun */
33      builder->set_insert_point(bb);                          /* 将基本块插入builder中 */
34      // 传参
35      auto aAlloca = builder->create_alloca(Int32Type);       /* 在内存中分配参数a的位置 */
36      std::vector<Value*> args;                       /* 获取callee函数的形参,通过Function中的iterator */
37      for (auto arg = calleeFun->arg_begin(); arg != calleeFun->arg_end(); arg++) {
38              args.push_back(*arg);   // * 号运算符是从迭代器中取出迭代器当前指向的元素
39      }
40      builder->create_store(args[0], aAlloca);                /* 存储参数a */
41      // 具体执行
42      auto aLoad = builder->create_load(aAlloca);             /* 将参数a存到变量aLoad中 */
43      auto res = builder->create_imul(aLoad, CONST_INT(2));   /* 将值乘以2存入变量res中 */
44      builder->create_ret(res);                               /* 创建返回，将res返回 */
45
```

上述是 callee 函数的实现方式。

```
46      // main函数
47      auto mainFun = Function::create(FunctionType::get(Int32Type, {}),   /* 创建 main 函数 */
48      "main", module);
49      bb = BasicBlock::create(module, "main", mainFun);           /* 创建基本块，命名为main */
50      builder->set_insert_point(bb);                              /* 将基本块加入到builder中 */
51      // 设置默认返回
52      auto retAlloca = builder->create_alloca(Int32Type);         /* 创建返回默认量 */
53      builder->create_store(CONST_INT(0), retAlloca);             /* 给默认量赋0，表示默认ret 0 */
54      // 具体执行
55      auto call = builder->create_call(calleeFun, {CONST_INT(110)});/* 调用函数calleeFun，将结果存到变量call中 */
56      builder->create_ret(call);      /* 返回结果值 */
57
58      std::cout << module->print();
59      delete module;
60
61      return 0;
62  }
```

main 函数的实现方式。

if.cpp：

```
30      //main函数
31      auto mainFunTy = FunctionType::get(Int32Type, {});      // 通过返回值类型与参数类型列表得到函数类型
32      auto mainFun = Function::create(mainFunTy, "main", module);     // 通过函数类型得到函数
33      auto bb = BasicBlock::create(module, "entry", mainFun);
34      builder->set_insert_point(bb);  // 将当前插入指令点的位置设在bb
35
36      auto retAlloca = builder->create_alloca(Int32Type);   // 在内存中分配返回值的位置
37
38      //float a =5.555;
39      auto aAlloca = builder->create_alloca(FloatType);     // 在内存中分配浮点数a的位置
40      builder->create_store(CONST_FP(5.555), aAlloca);
41      auto aLoad = builder->create_load(aAlloca);     //load上来
42
```

创建 main 函数以及声明以及初始化浮点数 a。

```
43        //if(a>1)
44        auto fcmp = builder->create_fcmp_gt(aLoad, CONST_FP(1));                //比较大小
45        auto trueBB = BasicBlock::create(module, "trueBB", mainFun);     // true分支
46        auto falseBB = BasicBlock::create(module, "falseBB", mainFun);   // false分支
47        auto retBB = BasicBlock::create(module, "", mainFun);    // return分支,提前create,以便true分支可以br
48
49        auto br = builder->create_cond_br(fcmp, trueBB, falseBB);  // 条件BR
50        DEBUG_OUTPUT // 我调试的时候故意留下来的,以醒目地提醒你这个调试用的宏定义方法
51
52        //return 233;
53        builder->set_insert_point(trueBB);  // if true; 分支的开始需要SetInsertPoint设置
54        builder->create_store(CONST_INT(233), retAlloca);
55        builder->create_br(retBB);  // br retBB
56
57        //return 0;
58        builder->set_insert_point(falseBB);  // if false
59        builder->create_store(CONST_INT(0), retAlloca);
60        builder->create_br(retBB);  // br retBB
61
62        builder->set_insert_point(retBB);  // ret分支
63        auto retLoad = builder->create_load(retAlloca);
64        builder->create_ret(retLoad);
65
66        std::cout << module->print();
67        delete module;
68        return 0;
```

剩下的逻辑判断。

while.cpp：

```
43        // 具体执行
44        auto aAlloca = builder->create_alloca(Int32Type);       /* 申请存a的空间，将地址赋值给指针aAlloca */
45        auto iAlloca = builder->create_alloca(Int32Type);       /* 申请存i的空间，将地址赋值给指针iAlloca */
46        builder->create_store(CONST_INT(10), aAlloca);          /* 将值10存入a的空间 */
47        builder->create_store(CONST_INT(0), iAlloca);           /* 将值0存入i的空间 */
48        builder->create_br(whileBB);                            /* 跳转到while循环条件判断，判断是否进入循环 */
49
```

声明以及初始化 a 和 i，然后进入 while 循环。

```
38        // 创建基本块
39        auto whileBB = BasicBlock::create(module, "whileBB", mainFun);  /* 进行while判断的基本块 */
40        auto trueBB = BasicBlock::create(module, "trueBB", mainFun);    /* 符合判断条件的基本块分支 */
41        auto falseBB = BasicBlock::create(module, "falseBB", mainFun);  /* 不符合判断条件的基本块分支 */
42
```

创建基本块。

```
50        //while判断的基本块
51        builder->set_insert_point(whileBB);                     /* while条件判断，设置SetInsertPoint */
52        auto i = builder->create_load(iAlloca);                 /* 取出i */
53        auto icmp = builder->create_icmp_lt(i, CONST_INT(10));  /* 判断i是否小于10，并将判断结果存到icmp中 */
54        builder->create_cond_br(icmp, trueBB, falseBB);         /* 根据icmp创建跳转语句 */
```

while 判断的基本块，对应 while(i<10)

```
56        builder->set_insert_point(trueBB);                  // if true; 分支的开始需要SetInsertPoint设置
57        i = builder->create_load(iAlloca);                  /* 取出i */
58        auto tmp = builder->create_iadd(i, CONST_INT(1));   /* 将i加1，存到暂存变量tmp中，tmp=i+1 */
59        builder->create_store(tmp, iAlloca);                /* 将tmp的值存到i中，i=tmp*/
60        auto a = builder->create_load(aAlloca);             /* 取出a */
61        i = builder->create_load(iAlloca);                  /* 取出i */
62        tmp = builder->create_iadd(a, i);                   /* 将a加i的值存到tmp中，tmp=i+a */
63        builder->create_store(tmp, aAlloca);                /* 将tmp存到a中，a=tmp */
64        builder->create_br(whileBB);                        /* 跳转到while循环条件判断，判断是否继续循环 */
```

如果 i<10 成立所对应的基本块。

```
66        builder->set_insert_point(falseBB);                 // if false; 分支的开始需要SetInsertPoint设置
67        auto res = builder->create_load(aAlloca);           /* 取出a的值，存到res中，res=a */
68        builder->create_ret(res);                           /* 将res返回，即return res */
69
```

如果不成立所对应的基本块。

问题 1：cpp 与.ll 的对应

请描述你的 cpp 代码片段和.ll 的每个 BasicBlock 的对应关系。描述中请附上两者代码。

assign_generator.cpp 与 1.ll 只有一个 basicblock,对应关系为：

auto bb = BasicBlock::create(module, "entry", mainFun);

对应标签 main 函数中的 label_entry。

fun_generator.cpp 与 2.ll 有两个 basicblock,对应关系为：

auto bb = BasicBlock::create(module, "entry", calleeFun);对应 callee 函数中的标签 label_entry。

bb = BasicBlock::create(module, "entry", mainFun);对应 main 函数中的标签 label_entry。

if_generator.cpp 与 3.ll 有有 4 个 basicblock,对应关系为：

auto bb = BasicBlock::create(module, "entry", mainFun);对应 main 函数中的标签 label_entry

auto trueBB = BasicBlock::create(module, "trueBB", mainFun);对应标签 label_trueBB

auto falseBB = BasicBlock::create(module, "falseBB", mainFun);对应标签 label_falseBB

auto retBB = BasicBlock::create(module, "", mainFun);对应标签 label4

while_generator.cpp 与 4.ll 有 4 个 basicblock,对应关系为：

auto bb = BasicBlock::create(module, "entry", mainFun);对应 main 函数中的标签 label_entry

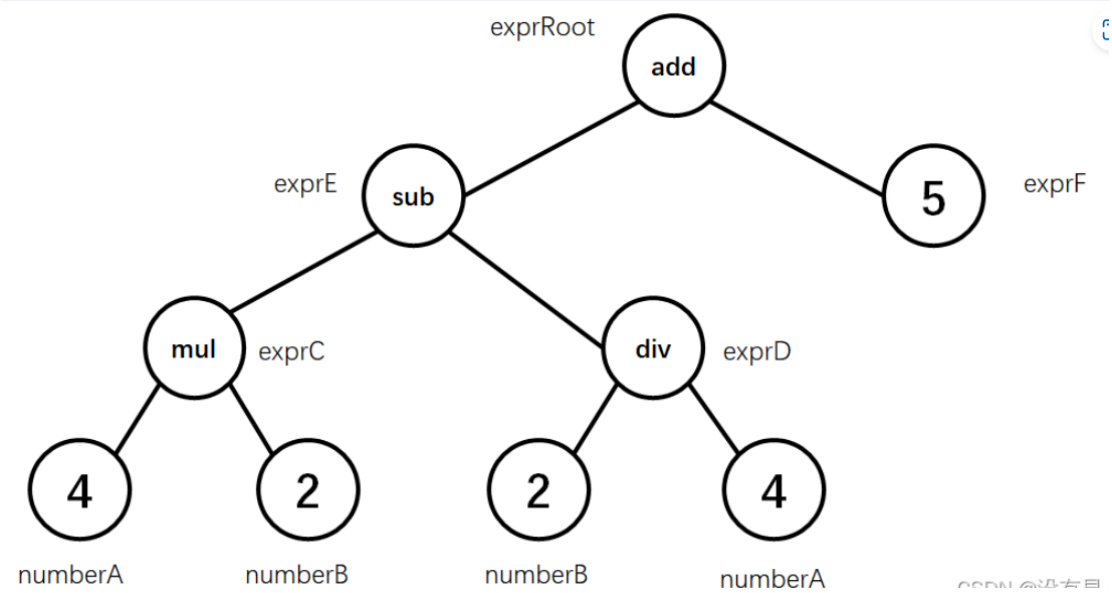
auto condBB = BasicBlock::create(module, "condBB", mainFun);对应标签 label_condBB

auto trueBB = BasicBlock::create(module, "trueBB", mainFun);对应标签 label_trueBB

auto retBB = BasicBlock::create(module, "retBB", mainFun);对应标签 label_retBB

问题 2: Visitor Pattern

请指出 visitor.cpp 中，`treeVisitor.visit(exprRoot)`执行时，以下几个 Node 的遍历序列:numberA、numberB、exprC、exprD、exprE、numberF、exprRoot。
序列请按如下格式指明：
exprRoot->numberF->exprE->numberA->exprD



顺序为：
exprRoot->numberF->exprE->exprD->numberB->numberA->exprC->numberA->numberB

问题 3: getelementptr

请给出 IR.md 中提到的两种 getelementptr 用法的区别,并稍加解释:

- %2 = getelementptr [10 x i32], [10 x i32]* %1, i32 0, i32 %0

- `%2 = getelementptr i32, i32* %1 i32 %0`

第一种方法：数组在 C 中会分割指针，但在 LLVM IR 中,它只能确定数组类型的大小然后强制转换为指针，但不会分割它们。%1 是我们的基址，有两个索引 0 和%0。因为它是数组，但是我们是通过指针访问它，所以我们需要两个索引：第一个用于分割指针(因为用的指针是[10 x i32]，但是我们返回的是一个 i32 的，所以需要分割)，第二个用于索引数组本身（即偏移量）。该方法适用的数组为 int nums[] = {1, 2, 3};或者 int a[10]

第二种方法：%1 做为我们的基址地址，%0 做为我们的索引（偏移量），并把计算出来的地址给%2。该方法适用的数组为 int *nums = {1, 2, 3};

实验结果验证：

首先是验证四个.ll 文件：

assign.c 文件：

```c
int main(){
  int a[10];
  a[0] = 10;
  a[1] = a[0] * 2;
  return a[1];
}
```

验证：

```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ lli assign_hand.
ll
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ echo $?
20
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$
```

输出为 20，验证正确。

fun.c 文件：

```c
int callee(int a){
  return 2 * a;
}
int main(){
  return callee(110);
}
```

验证：

```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ lli fun_hand.ll
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ echo $?
220
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$
```

输出为 220，验证正确。

if.c 文件：

```c
int main(){
  float a = 5.555;
  if(a > 1)
    return 233;
  return 0;
}
```

验证：

```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ lli if_hand.ll
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ echo $?
233
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$
```

输出为 233，验证正确。

while.c 文件：

```c
int main(){
  int a;
  int i;
  a = 10;
  i = 0;
  while(i < 10){
    i = i + 1;
    a = a + i;
  }
  return a;
}
```

验证：

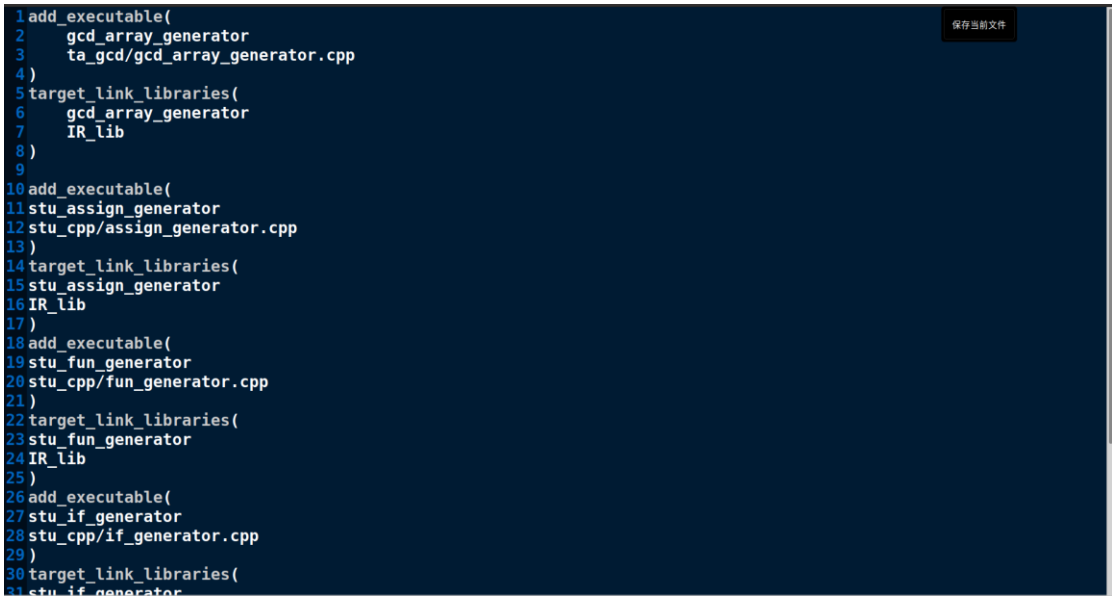
```
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ lli while_hand.l
l
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$ echo $?
65
click@click:~/桌面/cminus_compiler-2022-fall/tests/lab3/stu_ll$
```

输出为 65，验证正确。

接下来验证四个.cpp 文件：

在/tests/lab3/CMakeLists.txt 中去掉对应的注释，再在/build/下执行 cmake …与 make 指令，即可得到对应的可执行文件。

删除注释后如下：（仅截取部分）



执行 make 编译：

```
click@click:~/桌面/cminus_compiler-2022-fall/build$ make
[  5%] Built target flex
[ 15%] Built target syntax
[ 18%] Built target cminus_io
[ 26%] Built target common
[ 47%] Built target IR_lib
[ 60%] Built target OP_lib
[ 62%] Linking CXX executable ../../cminusfc
[ 66%] Built target cminusfc
[ 69%] Built target test_logging
[ 71%] Linking CXX executable ../test_ast
[ 73%] Built target test_ast
[ 77%] Built target lexer
[ 81%] Built target parser
Scanning dependencies of target stu_while_generator
[ 83%] Building CXX object tests/lab3/CMakeFiles/stu_while_generator.dir/stu_cpp/while_generator.cpp.o
[ 84%] Linking CXX executable ../../stu_while_generator
[ 84%] Built target stu_while_generator
Scanning dependencies of target stu_if_generator
[ 86%] Building CXX object tests/lab3/CMakeFiles/stu_if_generator.dir/stu_cpp/if_generator.cpp.o
[ 88%] Linking CXX executable ../../stu_if_generator
[ 88%] Built target stu_if_generator
Scanning dependencies of target stu_assign_generator
[ 90%] Building CXX object tests/lab3/CMakeFiles/stu_assign_generator.dir/stu_cpp/assign_generator.cpp.o
[ 92%] Linking CXX executable ../../stu_assign_generator
[ 92%] Built target stu_assign_generator
Scanning dependencies of target stu_fun_generator
[ 94%] Building CXX object tests/lab3/CMakeFiles/stu_fun_generator.dir/stu_cpp/fun_generator.cpp.o
[ 96%] Linking CXX executable ../../stu_fun_generator
[ 96%] Built target stu_fun_generator
[100%] Built target gcd_array_generator
```

此时当前目录下会出现四个新的可执行文件：

执行./命令，把输出重定向到特定文件下：

```
click@click:~/桌面/cminus_compiler-2022-fall$ cd build/
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_assign_generator > ~/桌面/cminus_compiler-2022-fall/1.tx
t
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_fun_generator > ~/桌面/cminus_compiler-2022-fall/2.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_if_generator > ~/桌面/cminus_compiler-2022-fall/3.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_while_generator > ~/桌面/cminus_compiler-2022-fall/4.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$
```

对四个生成的文件进行验证：

```
click@click:~/桌面/cminus_compiler-2022-fall$ cd build/
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_assign_generator > ~/桌面/cminus_compiler-2022-fall/1.tx
t
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_fun_generator > ~/桌面/cminus_compiler-2022-fall/2.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_if_generator > ~/桌面/cminus_compiler-2022-fall/3.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$ ./stu_while_generator > ~/桌面/cminus_compiler-2022-fall/4.txt
click@click:~/桌面/cminus_compiler-2022-fall/build$ cd ..
click@click:~/桌面/cminus_compiler-2022-fall$ lli 1.txt;echo $?
20
click@click:~/桌面/cminus_compiler-2022-fall$ lli 2.txt;echo $?
220
click@click:~/桌面/cminus_compiler-2022-fall$ lli 3.txt;echo $?
233
click@click:~/桌面/cminus_compiler-2022-fall$ lli 4.txt;echo $?
65
click@click:~/桌面/cminus_compiler-2022-fall$
```

可以看到，与上面直接编写的.ll 文件输出一致，验证正确。

实验反馈：

这次实验是给 lab4 打基础的实验，主要让我们了解了 LLVM IR、 LightIR（LLVM IR 的轻量级 C++接口）和 Visitor Pattern（访问者模式）。要想根据 C 代码写出对应的 LLVM IR 指令，需要阅读助教给出的 IR Reference 手册。还

有，了解了 LightIR 这一接口库的一些相关使用，知道如何编 cpp 文件生成特点逻辑功能的.ll 文件。