

编译原理实验3

LLVM IR与LightIR

计科210X 甘晴void 202108010XXX

【回答三个问题；实验难点与实验反馈在报告最后】

实验要求

详细的实验项目文档为 https://gitee.com/coderwym/cminus_compiler-2023-fall/tree/master/Documentations/lab3

简单陈述如下：

- 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（完成4个案例：根据的.c文件手写.ll文件）
- 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（完成4个案例：根据的.c文件手写.cpp文件）
- 理解Visitor Pattern，回答问题。

★

参考文献

- A橙: <https://blog.csdn.net/Aaron503/article/details/128324964>
- https://blog.csdn.net/weixin_45428457/article/details/123095236

实验过程

0 环境配置

使用 `lli --version` 命令获取当前系统LLVM版本。

要求版本为10.0.1，我这里的版本为10.0.0，后续使用暂时没有问题。

```
root@LAPTOP-S8GDLRKI:~# lli --version
LLVM (http://llvm.org/):
  LLVM version 10.0.0

  Optimized build.
  Default target: x86_64-pc-linux-gnu
  Host CPU: tigerlake
root@LAPTOP-S8GDLRKI:~#
```

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。

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](#)。（很重要，建议先看）

1.2 gcd例子: 利用clang生成的.ll

这里提供了一个范例gcd_array.ll

根据clang -S -emit-llvm gcd_array.c指令，可以得到对应的gcd_array.ll文件.

我们的任务是结合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，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

后续会经常用到这两条指令。

★1.3 我的提交1：手动编写.ll

任务：

对于 `tests/lab3/c_cases/` 目录下的 `assign.c`、`fun.c`、`if.c`、`while.c` 这四个文件，在 `tests/lab3/stu_11/` 目录中，手工完成自己的 `assign_hand.ll`、`fun_hand.ll`、`if_handf.ll`、`while_hand.ll`，以实现与上述四个C程序相同的逻辑功能。

必要时，可以参考 `clang -S -emit-llvm` 的输出，但不能照抄。

①assign_hand.ll

查看 `assign.c` 文件

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

根据 `assign.c` 编写相应 `.ll` 文件。

★使用 `inbounds` 检查指针是否越界，第一个 `i32 0` 必须存在第二个才是偏移量

`dso_local` 是一个 Runtime Preemption, 表明该变量会在同一个链接单元内解析符号

代码

```
define dso_local i32 @main() #0 {
    %1 = alloca [10 x i32]      ;分配a[10]空间并返回指针存入%1

    %2 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0,
    i32 0      ;获取a[0]地址的指针
    store i32 10, i32* %2      ;给a[0]赋值10

    %3 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0,
    i32 1      ;获取a[1]地址的指针
    %4 = load i32, i32* %2      ;%4 = a[0]
```

```

%5 = mul i32 %4,2      ;%5 = a[0]*2
store i32 %5, i32* %3   ;a[1] = %5 = a[0]*2
%6 = load i32, i32* %3  ;%6 = a[1]

ret i32 %6              ;return a[1]
}

```

验证

在c_cases文件夹内，使用gcc -o assign assign.c编译之后执行./assign;echo \$?;

之后在stu_ll文件夹内，使用lli assign_hand.ll; echo \$?，直接执行。

结果如下：

```

root@LAPTOP-S8GDLRKI: /home  X  +  v
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# ls
CMakeLists.txt  c_cases  stu_cpp  stu_ll  ta_gcd
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd c_cases
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ls
assign.c  fun.c  if.c  while.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# gcc -o assign assign.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ./assign;echo $?
20
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# cd ..
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# ls
CMakeLists.txt  c_cases  stu_cpp  stu_ll  ta_gcd
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd stu_ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# ls
assign_hand.ll  fun_hand.ll  if_hand.ll  while_hand.ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# lli assign_hand.ll; echo $?
20
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll#

```

结果一样，都为20。结果正确。

②fun_hand.ll

查看fun.c文件

```

int callee(int a){
    return 2 * a;
}

int main(){
    return callee(110);
}

```

根据fun.c编写相应.ll文件。

代码

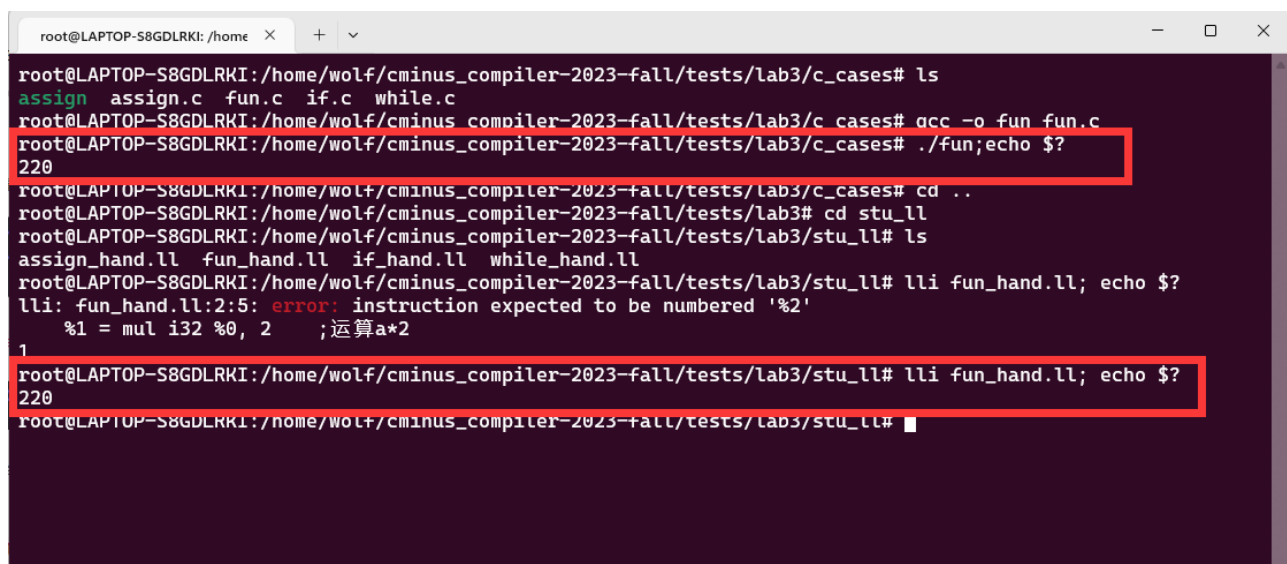
```
define dso_local i32 @callee(i32 %0) #0 {  
    %2 = mul i32 %0, 2    ;运算a*2  
    ret i32 %2            ;返回运算结果  
}  
  
define dso_local i32 @main() #0 {  
    %1 = call i32 @callee(i32 110)    ;以给定参数调用函数  
    ret i32 %1                        ;返回调用结果  
}
```

验证

在c_cases文件夹内，使用gcc -o fun fun.c编译之后执行./fun;echo \$?;

之后在stu_ll文件夹内，使用lli fun_hand.ll; echo \$?，直接执行。

结果如下：



```
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ls  
assign assign.c fun.c if.c while.c  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# gcc -o fun fun.c  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ./fun;echo $?  
220  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# cd ..  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3# cd stu_ll  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# ls  
assign_hand.ll fun_hand.ll if_hand.ll while_hand.ll  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# lli fun_hand.ll; echo $?  
lli: fun_hand.ll:2:5: error: instruction expected to be numbered '%2'  
    %1 = mul i32 %0, 2    ;运算a*2  
1  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# lli fun_hand.ll; echo $?  
220  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll#
```

结果一样，都为220。结果正确。

注意，这里之前我有一个错误，是因为我%1和%2的顺序换了一下。LLVM对于寄存器的顺序有很严格的要求，在按程序的逻辑顺序执行过程中必须严格逐个递增。我将%1和%2调换的结果是，在读取%0之后先读取%2再读取%1，这是不正确的。上面现在给出的是正确的代码。

③if_hand.ll

查看if.c文件

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

根据if.c编写相应.ll文件。

fcmp是cmplnst的子类，执行float类型的compare（icmp执行int型的compare）

ugt是无序或大于，ult是无序或小于

br的原型：br i1, label, label ; Conditional branch

代码

```
define dso_local i32 @main() #0{
    %1 = alloca float ;分配float型变量空间并返回指针
    store float 0x40163851E0000000, float* %1 ;使用指针给该变量a赋值
    5.555

    %2 = load float, float* %1 ;取出a中的值临时保存在%2
    %3 = fcmp ugt float %2, 1.0 ;比较a和1.0的大小，存入%3
    br i1 %3, label %4, label %5 ;若%3为真，跳转到%4否则跳转到%5
4:
    ret i32 233 ;返回233
5:
    ret i32 0 ;返回0
}
```

验证

在c_cases文件夹内，使用gcc -o if if.c编译之后执行 ./if;echo \$?;

之后在stu_ll文件夹内，使用lli if_hand.ll; echo \$?，直接执行。

结果如下：

```
root@LAPTOP-S8GDLRKI: /home X + v
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# ls
CMakeLists.txt  c_cases  stu_cpp  stu_ll  ta_gcd
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd c_cases/
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ls
assign  assign.c  fun  fun.c  if.c  while.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# gcc -o if if.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ./if;echo $?
233
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# cd ..
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd stu_ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# ls
assign_hand.ll  fun_hand.ll  if_hand.ll  while_hand.ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# lli if_hand.ll; echo $?
233
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll#
```

结果一样，都为233。结果正确。

④while_hand.ll

查看while.c文件

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

根据while.c编写相应.ll文件。

add后加的参数nsw(无符号包装)标志，一般Rust编译器不会加这个标记，clang会加上这个标记。

代码

```
define dso_local i32 @main() #0{
    %1 = alloca i32 ;int a
    %2 = alloca i32 ;int i
    store i32 10, i32* %1 ;a = 10
    store i32 0, i32* %2 ;i = 0
    br label %3 ;直接跳转到%3

;判断if i < 10
3:
    %4 = load i32, i32* %2 ;取出i
    %5 = icmp slt i32 %4, 10 ;i与10比较
    br i1 %5, label %6, label %10 ;若i<10成立，跳转6，否则跳转10

6:
    %7 = add nsw i32 %4, 1 ;计算i+1，结果放在%7
    store i32 %7, i32* %2 ;将i+1结果存回i地址
    %8 = load i32, i32* %1 ;取出a的值，放在%8
    %9 = add nsw i32 %7, %8 ;计算a+i的值，结果放在%9
    store i32 %9, i32* %1 ;a+i结果存回a地址
    br label %3 ;跳转%3，进行判断

10:
    ret i32 %9 ;返回a的值
}
```

验证

在c_cases文件夹内，使用gcc -o while while.c编译之后执行./while;echo \$?;

之后在stu_ll文件夹内，使用llc while_hand.ll; echo \$?，直接执行。

结果如下：


```
root@LAPTOP-S8GDLRKI: /home X + v
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# ls
CMakeLists.txt  c_cases  stu_cpp  stu_ll  ta_gcd
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd c_cases
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ls
assign  assign.c  fun  fun.c  if  if.c  while.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# gcc -o while while.c
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# ./while;echo $?
65
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/c_cases# cd ..
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# ls
CMakeLists.txt  c_cases  stu_cpp  stu_ll  ta_gcd
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3# cd stu_ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# ls
assign  hand.ll  fun  hand.ll  if  hand.ll  while  hand.ll
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll# lli while_hand.ll; echo $?
65
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall/tests/lab3/stu_ll#
```

结果一样，都为65。结果正确。

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。所以这里需要仔细阅读文档了解其接口的设计，为Lab4做准备。

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。

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
```

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。★否则无法生成可执行文件

★2.3 我的提交2: 利用LightIR + cpp编写生成.ll的程序

任务:

在 `tests/lab3/stu_cpp/` 目录中，编写 `assign_generator.cpp`、`fun_generator.cpp`、`if_generator.cpp`、`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。需要添加必要的注释。

① `assign_generator.cpp`

代码

```
#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
```

```

int main() {
    auto module = new Module("Assign");
    auto builder = new IRBuilder(nullptr, module);
    //使用IRBuilder创建指令

    //创建函数与BasicBlock
    Type *Int32Type = Type::get_int32_type(module);
    auto mainTy = FunctionType::get(Int32Type, {});
    //返回值为i32, 参数为空
    auto main = Function::create(mainTy, "main", module);

    auto BB = BasicBlock::create(module, "BasicBlock1", main);
    builder->set_insert_point(BB);
    //设置插入指令的BasicBlock

    //与.ll中的指令对应
    auto *arrayType = ArrayType::get(Int32Type, 10);
    //数组的类型为[10 x i32]
    auto a = builder->create_alloca(arrayType);
    //分配数组
    auto a0Ptr = builder->create_gep(a, {CONST_INT(0),
CONST_INT(0)}); //计算a[0]地址
    builder->create_store({CONST_INT(10)}, a0Ptr);
    //a[0] = 10
    auto a1Ptr = builder->create_gep(a, {CONST_INT(0),
CONST_INT(0)}); //计算a[1]地址
    auto a0 = builder->create_load(a0Ptr);
    //取出a[0]
    auto temp = builder->create_imul(a0, {CONST_INT(2)});
    //a[0]*2
    builder->create_store(temp, a1Ptr);
    //a1 = a[0]*2
    auto a1 = builder->create_load(a1Ptr);
    builder->create_ret(a1);
    //return a1 = a[0]*2
    //指令创建结束
    std::cout << module->print();
    delete module;
    return 0;
}

```

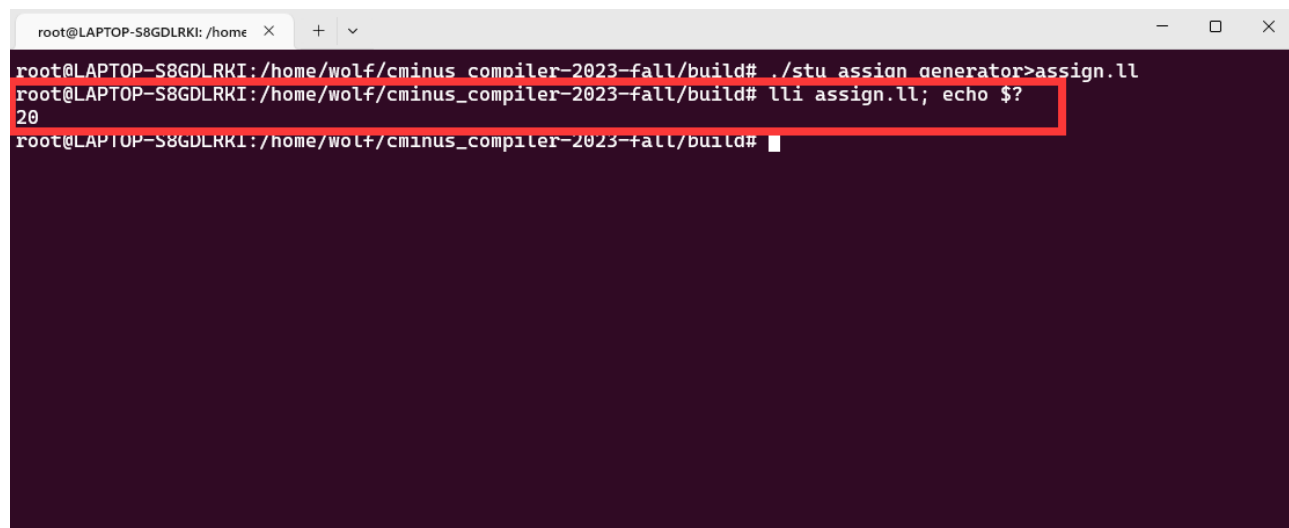
验证

在build文件夹内，make成功后，`./stu_assign_generator>assign.ll`

查看assign.ll结果如下：

```
define i32 @main() {  
  label_BasicBlock1:  
    %op0 = alloca [10 x i32]  
    %op1 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 0  
    store i32 10, i32* %op1  
    %op2 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 0  
    %op3 = load i32, i32* %op1  
    %op4 = mul i32 %op3, 2  
    store i32 %op4, i32* %op2  
    %op5 = load i32, i32* %op2  
    ret i32 %op5  
}
```

使用`lli assign.ll; echo $?`测试输出结果如下



```
root@LAPTOP-S8GDLRKI: /home  X  +  v  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./stu_assign_generator>assign.ll  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# lli assign.ll; echo $?  
20  
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#
```

结果正确

basicblock对应关系

仅有一个BasicBlock,

```
auto BB = BasicBlock::create(module, "BasicBlock1", main);
```

创建名为BasicBlock1的基本块，并将其保存到变量BB中，对应.ll文件中的label_BasicBlock1块。

②fun_generator.cpp

代码

```
#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) // 得到常数值表示,方便后面多次用到

int main() {
    auto module = new Module("fun");
    auto builder = new IRBuilder(nullptr, module); // 使用IRBuilder创建指令
    Type *Int32Type = Type::get_int32_type(module);

    // 创建callee函数与callee_BasicBlock
    auto calleeTy = FunctionType::get(Int32Type, {Int32Type}); // 通过返回值类型与参数类型列表得到函数类型
    auto callee = Function::create(calleeTy, "callee", module); // 由函数类型创建函数
```

```

    auto BB0 = BasicBlock::create(module, "callee_BasicBlock" ,
callee);
    builder->set_insert_point(BB0);

    //插入callee的BasicBlock中的指令
    std::vector<Value *> args;//获取函数的形参,通过Function中的iterator
    for (auto arg = callee->arg_begin(); arg != callee->arg_end();
arg++) {
        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
    }
    auto ans = builder->create_imul(args[0], CONST_INT(2));
    //mul = a*2
    builder->create_ret(ans);

    //创建main函数与main_BasicBlock
    auto mainTy = FunctionType::get(Int32Type, {});//通过返回值类型与
参数类型列表得到函数类型
    auto main = Function::create(mainTy, "main", module);//由函数类型
创建函数
    auto BB1 = BasicBlock::create(module, "main_BasicBlock" ,
main);
    builder->set_insert_point(BB1);

    //插入main的BasicBlock中的指令
    auto callret = builder->create_call(callee, {CONST_INT(110)});
    //callret = callee(110)
    builder->create_ret(callret);

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

```

思考

这里可能会这样写

```
auto aAlloca = builder->create_alloca(Int32Type); // 在内存中分配参数
a的位置
builder->create_store(args[0], aAlloca); // 将参数a store下来
```

即先把a存下来，再进行操作，但这样与我之前写的.ll文件的含义是不一样的，最终生成的.ll文件也会不一样。故不这样写。

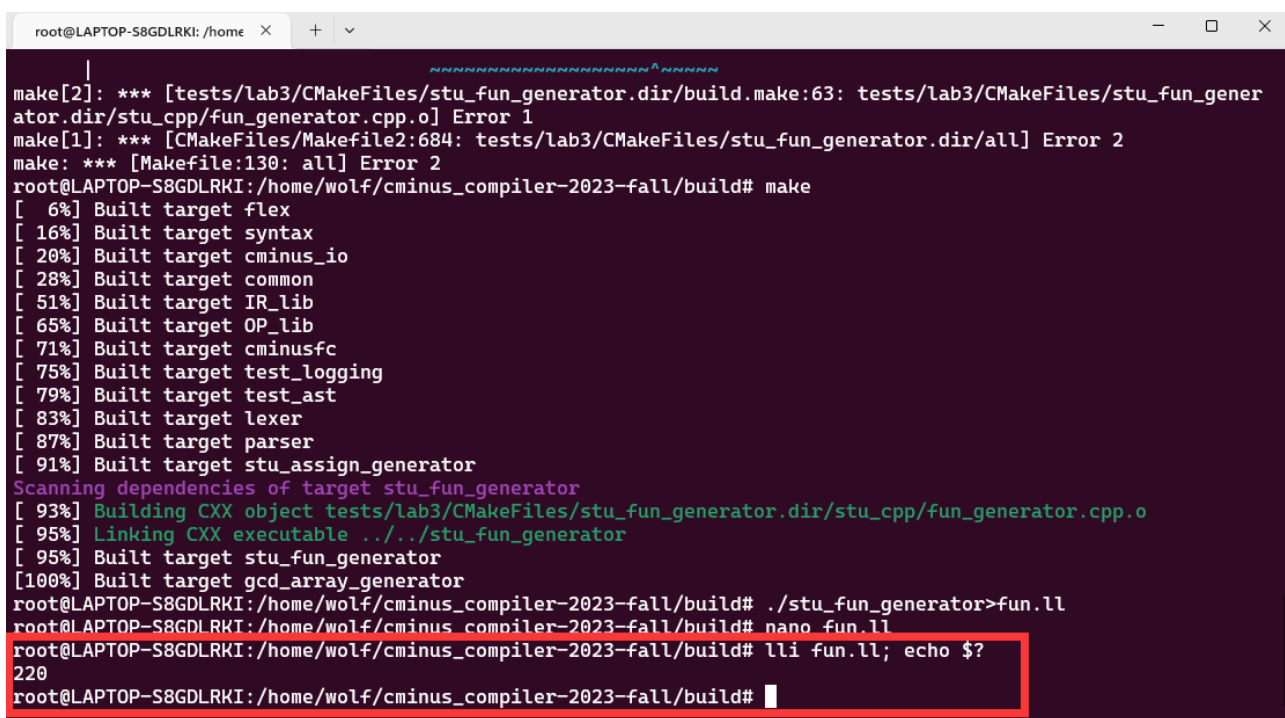
验证

在build文件夹内，make成功后，`./stu_fun_generator>fun.ll`

查看assign.ll结果如下：

```
define i32 @callee(i32 %arg0) {
label_callee_BasicBlock:
    %op1 = mul i32 %arg0, 2
    ret i32 %op1
}
define i32 @main() {
label_main_BasicBlock:
    %op0 = call i32 @callee(i32 110)
    ret i32 %op0
}
```

使用`lli fun.ll; echo $?`测试输出结果如下



```
root@LAPTOP-S8GDLRKI: /home  X  +  v
make[2]: *** [tests/lab3/CMakeFiles/stu_fun_generator.dir/build.make:63: tests/lab3/CMakeFiles/stu_fun_generator.dir/stu_cpp/fun_generator.cpp.o] Error 1
make[1]: *** [CMakeFiles/Makefile2:684: tests/lab3/CMakeFiles/stu_fun_generator.dir/all] Error 2
make: *** [Makefile:130: all] Error 2
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# make
[ 6%] Built target flex
[ 16%] Built target syntax
[ 20%] Built target cminus_io
[ 28%] Built target common
[ 51%] Built target IR_lib
[ 65%] Built target OP_lib
[ 71%] Built target cminusfc
[ 75%] Built target test_logging
[ 79%] Built target test_ast
[ 83%] Built target lexer
[ 87%] Built target parser
[ 91%] Built target stu_assign_generator
Scanning dependencies of target stu_fun_generator
[ 93%] Building CXX object tests/lab3/CMakeFiles/stu_fun_generator.dir/stu_cpp/fun_generator.cpp.o
[ 95%] Linking CXX executable ../../stu_fun_generator
[ 95%] Built target stu_fun_generator
[100%] Built target gcd_array_generator
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./stu_fun_generator>fun.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# nano fun.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# lli fun.ll; echo $?
220
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#
```

结果正确。

basicblock对应关系

创建了两个块，分别为callee_BasicBlock和main_BasicBlock。

其分别对应.ll文件中 callee 函数中的标签label_callee_BasicBlock和main函数中的标签label_main_BasicBlock。

③if_generator.cpp

代码

```
#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

int main(){
    auto module = new Module("if");
    auto builder = new IRBuilder(nullptr, module);
    //使用IRBuilder创建指令
    Type *Int32Type = Type::get_int32_type(module);
```



```

//创建main函数和entry_BasicBlock
auto mainTy = FunctionType::get(Int32Type, {});
//返回值为i32, 参数为空
auto main = Function::create(mainTy, "main", module);
auto BBEntry = BasicBlock::create(module, "entry_BasicBlock" ,
main);
builder->set_insert_point(BBEntry);

//entryBasicBlock插入指令, 对应.ll的BasicBlock0
Type *FloatType = Type::get_float_type(module);
//浮点类型
auto aPtr = builder->create_alloca(FloatType);
//为float a分配空间并返回指针
builder->create_store(CONST_FP(5.555), aPtr);
//a = 5.555
auto a = builder->create_load(aPtr);
//取出a
auto fcmp = builder->create_fcmp_gt(a, CONST_FP(1.0));
//fcmp = if a>1
//创建true, false对应的BasicBlock
auto BBTrue = BasicBlock::create(module, "true_BasicBlock" ,
main);
auto BBFalse = BasicBlock::create(module, "false_BasicBlock",
main);
builder->create_cond_br(fcmp, BBTrue, BBFalse);
//br跳转指令

//true对应的BasicBlock插入指令
builder->set_insert_point(BBTrue);
builder->create_ret(CONST_INT(233));

//false对应的BasicBlock插入指令
builder->set_insert_point(BBFalse);
builder->create_ret(CONST_INT(0));

//指令创建结束
std::cout << module->print();
delete module;
return 0;
}

```

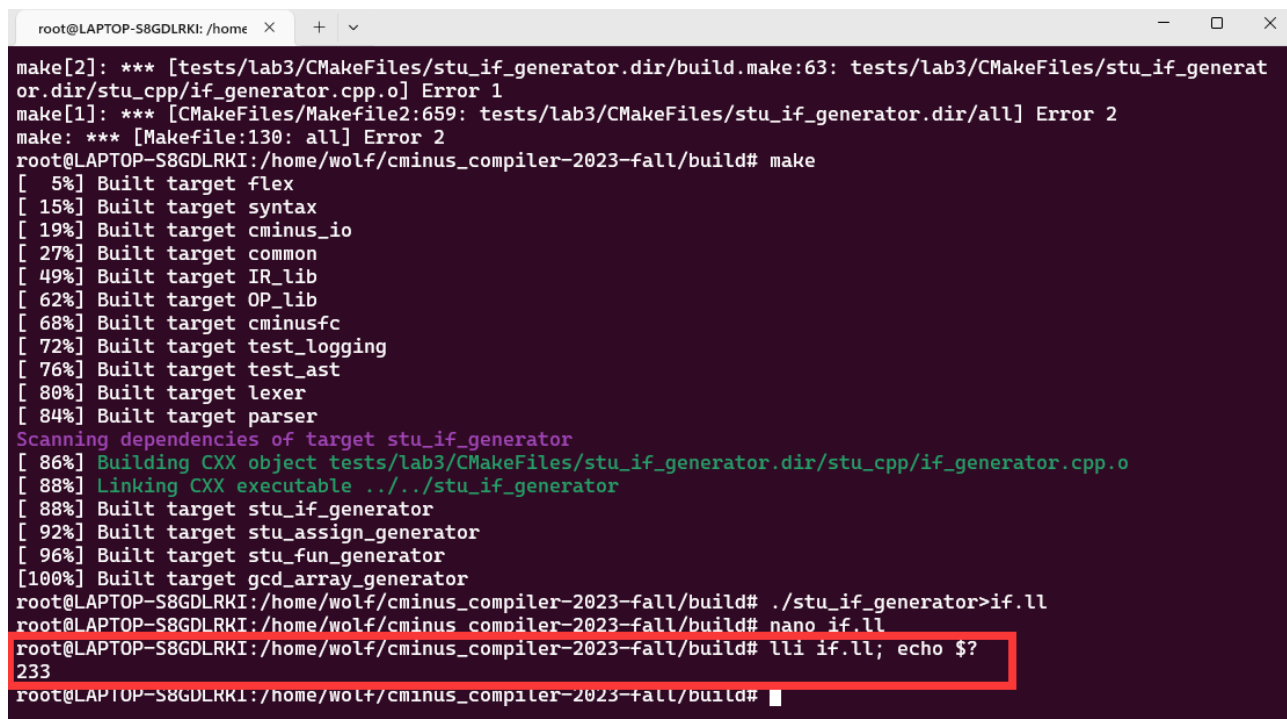
验证

在build文件夹内，make成功后，`./stu_if_generator>if.ll`

查看assign.ll结果如下：

```
define i32 @main() {
label_entry_BasicBlock:
    %op0 = alloca float
    store float 0x40163851e0000000, float* %op0
    %op1 = load float, float* %op0
    %op2 = fcmp ugt float %op1, 0x3ff0000000000000
    br i1 %op2, label %label_true_BasicBlock, label
%label_false_BasicBlock
label_true_BasicBlock:
    ; preds = %label_entry_BasicBlock
    ret i32 233
label_false_BasicBlock:
    ; preds = %label_entry_BasicBlock
    ret i32 0
}
```

使用`lli if.ll; echo $?`测试输出结果如下



```
root@LAPTOP-S8GDLRKI: /home X + v
make[2]: *** [tests/lab3/CMakeFiles/stu_if_generator.dir/build.make:63: tests/lab3/CMakeFiles/stu_if_generat
or.dir/stu_cpp/if_generator.cpp.o] Error 1
make[1]: *** [CMakeFiles/Makefile2:659: tests/lab3/CMakeFiles/stu_if_generator.dir/all] Error 2
make: *** [Makefile:130: all] Error 2
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# make
[ 5%] Built target flex
[ 15%] Built target syntax
[ 19%] Built target cminus_io
[ 27%] Built target common
[ 49%] Built target IR_lib
[ 62%] Built target OP_lib
[ 68%] Built target cminusfc
[ 72%] Built target test_logging
[ 76%] Built target test_ast
[ 80%] Built target lexer
[ 84%] Built target parser
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
[ 92%] Built target stu_assign_generator
[ 96%] Built target stu_fun_generator
[100%] Built target gcd_array_generator
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./stu_if_generator>if.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# nano if.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# lli if.ll; echo $?
233
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#
```

basicblock对应关系

创建了3个块，分别为entry_BasicBlock, true_BasicBlock, false_BasicBlock,

其分别对应.ll文件中label_entry_BasicBlock，判断正确跳转地址label_true_BasicBlock和判断错误跳转地址label_false_BasicBlock。

④while_generator.cpp

代码

```
#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

int main(){
    auto module = new Module("while");
    auto builder = new IRBuilder(nullptr, module); // 使用IRBuilder创建指令
    // 创建main函数
    Type *Int32Type = Type::get_int32_type(module);
    auto mainTy = FunctionType::get(Int32Type, {});
```

```

auto main = Function::create(mainTy, "main", module);

//创建4个BasicBlock
auto BBEntry = BasicBlock::create(module, "entry", main);
auto BBWhile = BasicBlock::create(module, "while", main);
auto BBTrue = BasicBlock::create(module, "true", main);
auto BBFalse = BasicBlock::create(module, "false", main);

//entryBasicBlock插入指令
builder->set_insert_point(BBEntry);
auto aPtr = builder->create_alloca(Int32Type);
    //分配a的空间, 返回指针aPtr
auto iPtr = builder->create_alloca(Int32Type);
    //分配i的空间, 返回指针iPtr
builder->create_store(CONST_INT(0), iPtr);
    //i = 0
builder->create_store(CONST_INT(10), aPtr);
    //a = 10
builder->create_br(BBWhile);
    //br跳转到while循环的判断BasicBlock

//whileBasicBlock插入指令
builder->set_insert_point(BBWhile);
auto i_now = builder->create_load(iPtr);
auto icmp = builder->create_icmp_lt(i_now, CONST_INT(10));
builder->create_cond_br(icmp, BBTrue, BBFalse);
//br跳转到True或False的BasicBlock

//TrueBasicBlock插入指令
builder->set_insert_point(BBTrue);
auto newi = builder->create_iadd(i_now, CONST_INT(1)); //i + 1
builder->create_store(newi, iPtr); //i = i + 1
auto a = builder->create_load(aPtr);
auto newa = builder->create_iadd(newi, a); //a + i
builder->create_store(newa, aPtr); //a = a + i
builder->create_br(BBWhile);
    //br跳转到while循环的判断BasicBlock

//FalseBasicBlock插入指令
builder->set_insert_point(BBFalse);
builder->create_ret(newa); //return a

```

```

//指令创建结束
std::cout << module->print();
delete module;
return 0;
}

```

验证

在build文件夹内，make成功后，`./stu_assign_generator>assign.ll`

查看assign.ll结果如下：

```

define i32 @main() {
label_entry:
    %op0 = alloca i32
    %op1 = alloca i32
    store i32 0, i32* %op1
    store i32 10, i32* %op0
    br label %label_while
label_while:                                     ; preds
= %label_entry, %label_true
    %op2 = load i32, i32* %op1
    %op3 = icmp slt i32 %op2, 10
    br i1 %op3, label %label_true, label %label_false
label_true:                                     ; preds
= %label_while
    %op4 = add i32 %op2, 1
    store i32 %op4, i32* %op1
    %op5 = load i32, i32* %op0
    %op6 = add i32 %op4, %op5
    store i32 %op6, i32* %op0
    br label %label_while
label_false:                                     ; preds
= %label_while
    ret i32 %op6
}

```

使用 `lli assign.ll; echo $?` 测试输出结果如下

```
root@LAPTOP-S8GDLRKI: /home X + v
-- Found BISON: /usr/bin/bison (found version "3.5.1")
-- Found LLVM 10.0.0
-- Using LLVMConfig.cmake in: /usr/lib/llvm-10/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /home/wolf/cminus_compiler-2023-fall/build
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-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
[ 66%] Built target cminusfc
[ 69%] Built target test_logging
[ 73%] Built target test_ast
[ 77%] Built target lexer
[ 81%] Built target parser
[ 84%] Built target stu_while_generator
[ 88%] Built target stu_if_generator
[ 92%] Built target stu_assign_generator
[ 96%] Built target stu_fun_generator
[100%] Built target gcd_array_generator
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./stu_while_generator>while.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# nano while.ll
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# lli while.ll; echo $?
65
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#
```

basicblock对应关系

创建了4个块，分别为entry, while, true, false,

其分别对应.ll文件中label_entry, while循环控制块（通过i检验是否继续循环），判断正确跳转地址label_true和判断错误跳转地址label_false。

3 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，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`

```
#include <iostream>
#include <vector>

class TreeVisitor; // Forward declare TreeVisitor

class Node { // Parent class for the elements (AddSubNode,
            // NumberNode and
            // MulDivNode)
public:
    // This function accepts an object of any class derived from
    // TreeVisitor and must be implemented in all derived classes
    virtual int accept(TreeVisitor& treevisitor) = 0;
};

// Forward declare specific elements (nodes) to be dispatched
class AddSubNode;
class NumberNode;
class MulDivNode;

class TreeVisitor { // Declares the interface for the treevisitor
public:
    // Declare overloads for each kind of a node to dispatch
    virtual int visit(AddSubNode& node) = 0;
    virtual int visit(NumberNode& node) = 0;
    virtual int visit(MulDivNode& node) = 0;
};

class AddSubNode : public Node { // Specific element class #1
public:
    // Resolved at runtime, it calls the treevisitor's overloaded
    function,
    // corresponding to AddSubNode.
```

```

    int accept(TreeVisitor& treeVisitor) override {
        return treeVisitor.visit(*this);
    }
    Node& leftNode;
    Node& rightNode;
    std::string op;
    AddSubNode(Node& left, Node& right, std::string op):
    leftNode(left), rightNode(right), op(op){}
};

class NumberNode : public Node { // Specific element class #2
public:
    // Resolved at runtime, it calls the treevisitor's overloaded
    function,
    // corresponding to NumberNode.
    int accept(TreeVisitor& treeVisitor) override {
        return treeVisitor.visit(*this);
    }
    int number;
    NumberNode(int number){
        this->number = number;
    }
};

class MulDivNode : public Node { // Specific element class #3
public:
    // Resolved at runtime, it calls the treevisitor's overloaded
    function,
    // corresponding to MulDivNode.
    int accept(TreeVisitor& treeVisitor) override {
        return treeVisitor.visit(*this);
    }
    Node& leftNode;
    Node& rightNode;
    std::string op;
    MulDivNode(Node& left, Node& right, std::string op):
    leftNode(left), rightNode(right), op(op){}
};

class TreevisitorCalculator : public TreeVisitor { // Implements
    triggering of all

```


// kind of elements

```
(nodes)
public:
    int visit(AddSubNode& node) override {
        auto right = node.rightNode.accept(*this);
        auto left = node.leftNode.accept(*this);
        if (node.op == "add") {
            return left + right;
        }
        else {
            return left - right;
        }
    }

    int visit(NumberNode& node) override {
        return node.number;
    }

    int visit(MulDivNode& node) override {
        auto left = node.leftNode.accept(*this);
        auto right = node.rightNode.accept(*this);
        if (node.op == "mul") {
            return left * right;
        }
        else {
            return left / right;
        }
    }
};

int main() {
    // construct the expression nodes and the tree
    // the expression: 4 * 2 - 2 / 4 + 5
    auto numberA = NumberNode(4);
    auto numberB = NumberNode(2);
    auto exprC = MulDivNode(numberA, numberB, "mul");
    auto exprD = MulDivNode(numberB, numberA, "div");
    auto exprE = AddSubNode(exprC, exprD, "sub");
    auto numberF = NumberNode(5);
    auto exprRoot = AddSubNode(exprE, numberF, "add");

    TreeVisitorCalculator treeVisitor;
```

```
// traverse the tree and calculate
int result = treeVisitor.visit(exprRoot);
std::cout << "4 * 2 - 2 / 4 + 5 evaluates: " << result <<
std::endl;
return 0;
}
```

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
```

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。

【详细在2.3中逐个对比，这里只陈列结果】

①assign

仅有一个BasicBlock，基本块BasicBlock1，对应.ll文件中的label_BasicBlock1。

②fun

创建了两个块，分别为callee_BasicBlock和main_BasicBlock。

其分别对应.ll文件中 callee 函数中的标签label_callee_BasicBlock和main函数中的标签label_main_BasicBlock。

③if

创建了3个块，分别为entry_BasicBlock, true_BasicBlock, false_BasicBlock,

其分别对应.ll文件中label_entry_BasicBlock, 判断正确跳转地址label_true_BasicBlock和判断错误跳转地址label_false_BasicBlock。

④while

创建了4个块，分别为entry, while, true, false,

其分别对应.ll文件中label_entry, while循环控制块（通过i检验是否继续循环），判断正确跳转地址label_true和判断错误跳转地址label_false。

问题2: Visitor Pattern

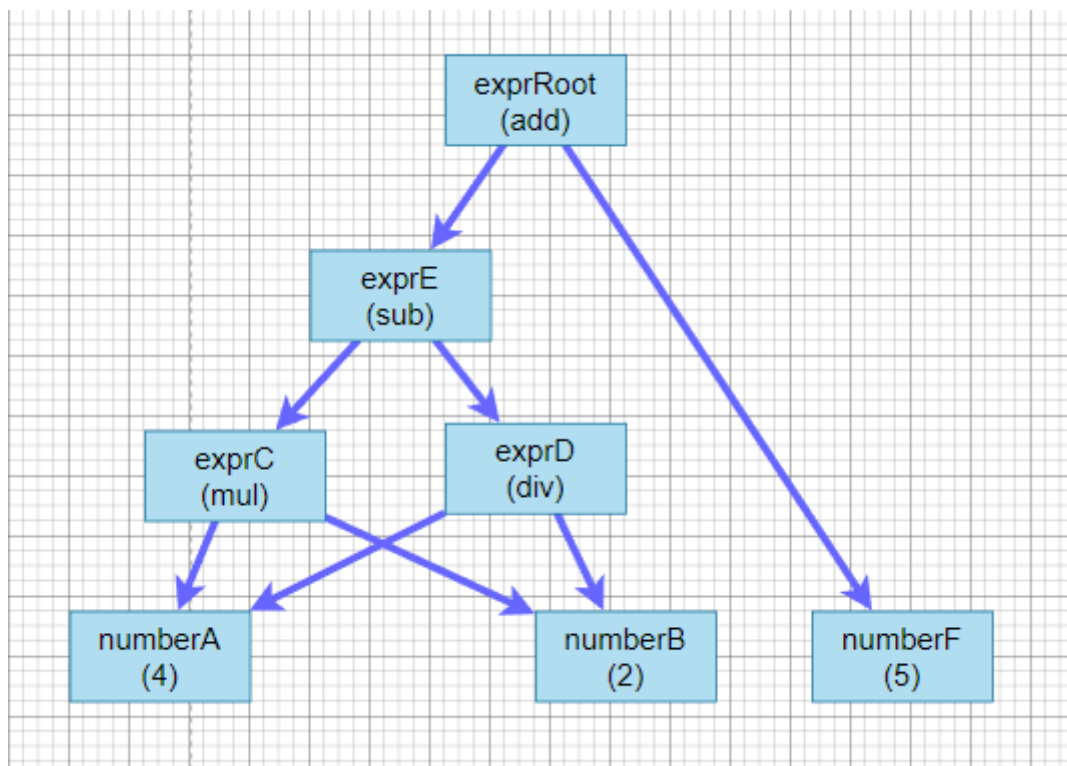
请指出visitor.cpp中，`treeVisitor.visit(exprRoot)`执行时，以下几个Node的遍历序列:numberA、numberB、exprC、exprD、exprE、numberF、exprRoot。

序列请按如下格式指明：

exprRoot->numberF->exprE->numberA->exprD

根据visit()函数中的实现，对于AddSubNode的访问，是先访问右子节点，然后访问左子节点，对于MulDivNode的访问，先访问左子节点，后访问右子节点。因此可以得出visitor.cpp中访问该树的序列为：

exprRoot->numberF->exprE->exprD->numberB->numberA->exprC->numberA->numberB



具体过程如下：

- 根节点 `exprRoot` 是 `AddSubNode`，按照访问顺序先访问右子树，即 `numberF`
- `numberF` 是 `NumberNode`，直接返回其节点的值5，故开始访问 `exprRoot` 的左子树
由于 `exprRoot` 的第一个左子节点 `exprE` 是 `AddSubNode`，所以要遍历完 `AddSubNode` 才能返回结果，下面访问 `exprE` 的右子树
- `exprE` 的右子节点是 `exprD`，是 `MulDivNode`，所以先访问左子树，即 `numberB`，是 `NumberNode`，返回其值2
- 接着遍历 `exprD` 的右子树，即 `numberA`，是 `NumberNode`，返回其值4，得到两个子树的值后将计算结果 $2/4$ 存在 `exprD` 中
- 接着访问 `exprE` 的左子树，即 `exprC`，是 `MulDivNode`，所以先遍历左子树，是 `numberA`，是 `NumberNode`，返回其值4
- 接着访问 `exprC` 的右子树，即 `numberB`，是 `NumberNode`，返回其值2，接着将运算结果 $2*4$ 的值存在 `exprC` 中
- 得到了 `exprC` 的值（值为8）和 `exprD` 的值（值为0.5），回到上一层将 `exprC-exprD` 的值存入 `exprE` 中
- 得到了 `exprE` 的值和 `numberF` 的值，将 `exprE+numberF` 的值存入根节点，遍历结束，最终结果为13（这里有个小细节，计算结果是12.5，但是结果使用 `int` 型保存，所以四舍五入为13）

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`

第一种用法:

`[10 x i32]` 是数据类型, `[10 x i32]*` 是指针类型, 因此首先用 `i32 0` 表示偏移为0, 这表示直接取第一个 `[10 x i32]` 数组, 在这里不进行偏移。然后的 `i32 %0` 表示在第一个 `[10 x i32]` 数组内, 偏移 `%0` 的元素地址。查资料可见, 这种表示方法可以解决类似于结构体数组这样有内部结构的数组类型的问题。

简单来说, 对于

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

最终取的是 $(\%2) + \text{size}([10 \times i32]) \times (\%1) + \%0$ 的地址。

第二种用法：

指针类型为 $i32^*$ ， $\%1$ 表示的是数组的起始地址，偏移量为 $\%0$ ，取出了数组偏移 $\%0$ 位置的元素地址。

区别：

在第一种用法中，指针类似于指针数组，首先确定在这个指针数组上的偏移，才能得到一个数组的指针，然后通过偏移找到元素的地址。而在第二种用法中，直接对数组的指针进行偏移，找到元素的地址。

当定义全局数组或结构体时，定义的是指针，例如在给出的 `gcd_array.c` 中声明的全局数组 x, y ，就是 $[1 \times i32]^*$ 类型，因此取出元素时使用的是第一种用法。

```
//全局数组x[1]
@x = common dso_local global [1 x i32] zeroinitializer, align 4
//取出元素地址
getelementptr inbounds ([1 x i32], [1 x i32]^* @x, i64 0, i64 0)
```

实验难点

1. 编写.ll文件

① 浮点数的处理

若浮点数不能被精确表示，不能直接赋值，要转化为十六进制的机器表示再赋值。如题目中的浮点数 5.555 ，要转化为 $0x40163851E0000000$ 。

② 寄存器顺序

LLVM 对于寄存器的顺序有很严格的要求，在按程序的逻辑顺序执行过程中必须严格逐个递增。这个我前面有出现过错误。

③编写思维

根据c代码编写.ll文件，颇有点类似上学期计算机系统中学习的汇编代码（实际上发llvm中间表示与之也是类似的），所以如果有一定的汇编功底，会更好写一些，另外LLVM不需要我们人为去分配寄存器，但是寄存器有严格的顺序要求（这一点参照上一条）

④一些标记

如assign中的inbounds标记用来检查是否越界，while中的nsw(无符号包装)标志等。精简文档中没有提到，这个需要自己去探究。

2.编写.cpp文件

①编写思维

我觉得这个更像是用汇编思维写的，甚至感觉上就是对着.ll代码去反推出来的。就是对.ll中的每个语句，要用一定的逻辑去生成它，此外对于函数，块，需要进行特殊的处理。乍看还是很复杂很复杂的，特别是它的范例给的很长很长，根本让人看了就不想读下去。但是这个如果能看懂一个，还是比较好写其它几个的。

实验反馈

经过这个实验，我得以一窥LLVM的基础知识。初步了解了LLVM的.ll文件的编写思路以及使用cpp快速生成.ll文件的代码生成器接口（LightIR）的用法。也初步了解了Visitor Pattern的原理。

实验文档感觉还是给的不够，尤其是代码直接给的很长一段，让人不忍读下去。有些关键的地方还要靠自己去琢磨。在完成实验的过程中还是依靠研究 [A橙学长](#) 和 [芜湖韩金轮学长](#) 在CSDN上发的实验思路才得以完成。

总体而言，在这个实验上还是学到了很多知识，也花了很多时间，有很多收获。