lab2 实验报告

学号 姓名

问题1: 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`

  第一种表示对于一个10个i32组成的数组，以指向这个数组的指针 [10 x i32]* %1 为基地址，i32 0 表示索引到这个数组第一个元素对应的地址，然后 i32 %0 表示再向后索引 %0 个地址，表示的是两次索引，得到数组中第 %0 个元素。

  而第二种表示对 i32 这种类型操作，以一个指向i32的指针 i32* %1 为基地址，然后向后索引%0个地址，从而得到 %1 后面第%0个地址，表示的是一次索引。由于一开始就是对于i32*指针操作，所以少了一次索引。

问题2: cpp 与 .ll 的对应

请说明你的 cpp 代码片段和 .ll 的每个 BasicBlock 的对应关系。

str_assign_generator:

cpp生成的 .ll是这样的:

```
@a = global [10 x i32] zeroinitializer
define i32 @main() {
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0
  %op1 = getelementptr [10 x i32], [10 x i32]* @a, i32 0, i32 0
  store i32 10, i32* %op1
  %op2 = load i32, i32* %op1
  %op3 = getelementptr [10 x i32], [10 x i32]* @a, i32 0, i32 1
  %op4 = mul i32 %op2, 2
  store i32 %op4, i32* %op3
  ret i32 %op4
}
```

其中BasicBlock label_entry对应代码

```
auto mainFun = Function::create(FunctionType::get(Int32Type, {}),
                                "main", module);
  auto bb = BasicBlock::create(module, "entry", mainFun);

  builder->set_insert_point(bb);

  auto retAlloca = builder->create_alloca(Int32Type);
  builder->create_store(CONST_INT(0), retAlloca);             // 默认 ret 0

  auto *arrayType = ArrayType::get(Int32Type, 10);
  auto initializer = ConstantZero::get(Int32Type, module);
```

```
  auto a = GlobalVariable::create("a", module, arrayType, false, initializer);

  auto a0GEP = builder->create_gep(a, {CONST_INT(0), CONST_INT(0)});
  builder->create_store(CONST_INT(10), a0GEP);
  auto a0Load = builder->create_load(a0GEP);
  auto a1GEP = builder->create_gep(a, {CONST_INT(0), CONST_INT(1)});
  auto a1Load = builder->create_imul(a0Load, CONST_INT(2));
  builder->create_store(a1Load, a1GEP);


  builder->create_ret(a1Load);
```

stu_fun_generator:

生成的 .ll文件:

```
define i32 @callee(i32 %arg0) {
label_entry:
  %op1 = alloca i32
  %op2 = alloca i32
  store i32 %arg0, i32* %op2
  %op3 = load i32, i32* %op2
  %op4 = mul i32 %op3, 2
  ret i32 %op4
}
define i32 @main() {
label_entry:
  %op0 = call i32 @callee(i32 110)
  ret i32 %op0
}
```

其中callee中的BasicBlock对应代码片段:

```
// BB的名字在生成中无所谓,但是可以方便阅读
  auto bb = BasicBlock::create(module, "entry", calleeFun);

  builder->set_insert_point(bb);                          // 一个BB的开始,将当前插入指
令点的位置设在bb

  auto retAlloca = builder->create_alloca(Int32Type);   // 在内存中分配返回值的位置
  auto aAlloca = builder->create_alloca(Int32Type);     // 在内存中分配参数a的位置

  std::vector<Value *> args;  // 获取callee函数的形参,通过Function中的iterator
  for (auto arg = calleeFun->arg_begin(); arg != calleeFun->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
  }

  builder->create_store(args[0], aAlloca);  // 将参数a store下来

  auto aLoad = builder->create_load(aAlloca);           // 将参数a load上来
  auto mul = builder->create_imul(aLoad, CONST_INT(2));  // MUL - mul
  builder->create_ret(mul);
```

而main中的BasicBlock对应代码片段:

```
bb = BasicBlock::create(module, "entry", mainFun);
  // BasicBlock的名字在生成中无所谓,但是可以方便阅读
  builder->set_insert_point(bb);
  auto call = builder->create_call(calleeFun, {CONST_INT(110)});
  builder->create_ret(call);
```

stu_if_generator:

cpp生成的.ll文件：

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0
  %op1 = alloca float
  store float 0x40163851e0000000, float* %op1
  %op2 = load float, float* %op1
  %op3 = fcmp ugt float %op2,0x3ff0000000000000
  br i1 %op3, label %label_trueBB, label %label_retBB
label_trueBB:                                             ; preds =
%label_entry
  store i32 233, i32* %op0
  br label %label_retBB
label_retBB:                                              ; preds =
%label_entry, %label_trueBB
  %op4 = load i32, i32* %op0
  ret i32 %op4
}
```

其中label_entry对应的BasicBlock对应的代码片段为：

```
auto bb = BasicBlock::create(module, "entry", mainFun);
  // BasicBlock的名字在生成中无所谓,但是可以方便阅读
  builder->set_insert_point(bb);

  auto retAlloca = builder->create_alloca(Int32Type);   // 在内存中分配返回值的位置
  builder->create_store(CONST_INT(0), retAlloca);
  auto aAlloca = builder->create_alloca(FloatType);
  builder->create_store(CONST_FP(5.555), aAlloca);
  auto aLoad = builder->create_load(aAlloca);
  auto fpcmp = builder->create_fcmp_gt(aLoad, CONST_FP(1.0));
```

label_trueBB对应的BasicBlock对应的代码片段为：

```
  auto br = builder->create_cond_br(fpcmp, trueBB, retBB);  // 条件BR

  builder->set_insert_point(trueBB);  // if true; 分支的开始需要SetInsertPoint设置
  builder->create_store(CONST_INT(233), retAlloca);
  builder->create_br(retBB);  // br retBB
```

label_retBB对应的BasicBlock对应的代码片段为：

```
  builder->set_insert_point(retBB);
  auto retLoad = builder->create_load(retAlloca);
  builder->create_ret(retLoad);
```

stu_while_generator:

cpp生成的.ll文件：

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = alloca i32
  %op2 = alloca i32
  store i32 10, i32* %op1
  store i32 0, i32* %op2
  br label %label_condBB
label_condBB:                                     ; preds =
%label_entry, %label_loopBB
  %op3 = load i32, i32* %op2
  %op4 = icmp slt i32 %op3, 10
  br i1 %op4, label %label_loopBB, label %label10
label_loopBB:                                     ; preds =
%label_condBB
  %op5 = load i32, i32* %op2
  %op6 = add i32 %op5, 1
  store i32 %op6, i32* %op2
  %op7 = load i32, i32* %op2
  %op8 = load i32, i32* %op1
  %op9 = add i32 %op8, %op7
  store i32 %op9, i32* %op1
  br label %label_condBB
label10:                                          ; preds = %label_condBB
  %op11 = load i32, i32* %op1
  ret i32 %op11
}
```

label_entry对应的代码：

```
  auto bb = BasicBlock::create(module, "entry", mainFun);
  // BasicBlock的名字在生成中无所谓,但是可以方便阅读
  builder->set_insert_point(bb);

  auto retAlloca = builder->create_alloca(Int32Type);   // 在内存中分配返回值的位置
  auto aAlloca = builder->create_alloca(Int32Type);     // 在内存中分配参数u的位置
  auto iAlloca = builder->create_alloca(Int32Type);     // 在内存中分配参数v的位置

  builder->create_store(CONST_INT(10), aAlloca);
  builder->create_store(CONST_INT(0), iAlloca);
```

label_condBB对应的代码：

```
builder->set_insert_point(condBB);
auto iLoad = builder->create_load(iAlloca);
auto icmp = builder->create_icmp_lt(iLoad, CONST_INT(10));
auto br = builder->create_cond_br(icmp, loopBB, retBB);  // 条件BR
```

label_loopBB对应的代码：

```
builder->set_insert_point(loopBB);
iLoad = builder->create_load(iAlloca);
auto iLoad1 = builder->create_iadd(iLoad, CONST_INT(1));
builder->create_store(iLoad1, iAlloca);
iLoad = builder->create_load(iAlloca);
auto aLoad = builder->create_load(aAlloca);
auto aLoad1 = builder->create_iadd(aLoad, iLoad);
builder->create_store(aLoad1, aAlloca);
builder->create_br(condBB);  // br retBB
```

label_retBB对应的代码：

```
builder->set_insert_point(retBB);
auto retLoad = builder->create_load(aAlloca);
builder->create_ret(retLoad);
```

问题3: Visitor Pattern

分析 `calc` 程序在输入为 `4 * (8 + 4 - 1) / 2` 时的行为：

1. 请画出该表达式对应的抽象语法树（使用 `calc_ast.hpp` 中的 `CalcAST*` 类型和在该类型中存储的值来表示），并给节点使用数字编号。
2. 请指出示例代码在用访问者模式遍历该语法树时的遍历顺序。

序列请按如下格式指明（序号为问题 3.1 中的编号）：
3->2->5->1

1.

1. CalcASTInput: expression 指向节点 2
2. CalcASTExpression: expression = nullptr, op 是任意值(因为只看term项)，term 指向节点 3
3. CalcASTTerm: term 指向节点 4, op = OP_DIV, factor 指向节点 5
4. CalcASTTerm: term 指向节点 6, op = OP_MUL, factor 指向节点 7
5. CalcASTNum: val = 2
6. CalcASTTerm: term = nullptr, op任意，factor 指向节点 8
7. CalcASTExpression: expression 指向节点9，op = OP_SUB，term 指向节点 10
8. CalcASTNum: val = 4
9. CalcASTExpression: expression 指向节点11, op = OP_ADD，term 指向节点 12
10. CalcASTTerm: term = nullptr, op任意，factor 指向节点 13
11. CalcASTExpression: expression = nullptr, op任意，term指向节点 14
12. CalcASTTerm: term = nullptr, op任意，factor 指向节点 15
13. CalcASTNum: val = 1
14. CalcASTTerm: term = nullptr, op任意，factor 指向节点 16
15. CalcASTNum: val = 4
16. CalcASTNum: val = 8

```
1. CalASTInput
|--2. CalcASTExpression
   |--3. CalcASTTerm term->4 op=OP_DIV factor->5
      |--4. CalcASTTerm term->6 op=OP_MUL factor->7
      |  |--6. CalcASTTerm
      |  |  |--8. CalcASTNum val=4
      |  |--7. CalcASTExpression expression->9 op=OP_SUB term->10
      |     |--9. CalcASTExpression expression->11 op=OP_ADD term->12
      |     |  |--11. CalcASTExpression
      |     |  |  |--14. CalcASTTerm
      |     |  |     |--16. CalcASTNum val=8
      |     |  |--12. CalcASTTerm
      |     |     |--15. CalcASTNum=8
      |     |--10. CalcASTTerm
      |        |13. CalcASTNum val=1
      |--5. CalcASTNum val=2
```

2.

1->2->3->4->6->8->7->9->11->14->16->12->15->10->13->5

实验难点

本次实验中，其实实验内容并不难，只要认真阅读理解就可以写出来。

不过主要还是由于我对于实验中的这些工具使用不太熟练，导致卡了比较长的时间。

比如我在 docker 里面运行 clang 的时候突然出现 clang not found 的问题，我重装了 docker 多次都没有解决，后来还是在朋友的提醒下通过添加 clang 的路径到环境变量里解决了问题，后来根据群友的解释，似乎是 docker 出现了一些问题，有一个配置文件没有正确加载，`exec bash` 重新加载一下就好了。其实归根结底还是我基础太差，碰到问题难以解决，比如不清楚Linux如何添加PATH，导致没有及时尝试这种方法。现在我知道了 `export PATH=路径` 就可以了(T_T).

还有对于git的使用不够熟练，在pull了之后有一个 Makefile.txt 文件有冲突，需要手动去掉冲突，结果我没有理解，仅仅把 “>>>>>” 去掉，同时保留了之前的版本和修改后的版本。结果 cmake 报错，我意识到了这个问题，又去掉了那个 Makefile.txt 之前的那部分，但是不小心多删了一行，结果 make 又无数次报错，非常令人崩溃，debug了好久，还拉上助教们一起迷惑，最后和上流仓库的文件对比了一下才发现了这个错误。

总之本次实验一方面增加了我对于以上问题的一些理解，另一方面让我感受到了自己能力的不足(我真是bug制造小能手)，让我感受到自己提升空间的巨大。感觉自己怎么努力都是菜鸡，但是还是希望每天都有进步吧。

实验反馈

没有什么，感觉助教都很热心，感谢助教们给予一个小菜鸡的帮助 :-)