lab2 实验报告

学号：PB20000024

姓名：陈奕衡

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`

对于第一条指令，这里生成了类似结构体的指针，其中结构体内为10个`i32`（`int`）类型变量，后面的第一个参数表示结构体指针（10 * `i32`）大小的偏移，第二个参数表示结构体内部（`i32`）大小的偏移，因此上式代表了a[0]元素

对于第二条指令，这里生成的就是普通的`i32`型指针，从而没有内部偏移量，只会有一个参数，代表的就是指针偏移（`i32`），上式同样代表a[0]元素

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。

assign.c

该片段只有一个主函数的 BasicBlock，cpp代码如下：

```
// main function
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); // assign ret = 0

auto *arrayType = ArrayType::get(Int32Type, 10);
auto a = builder->create_alloca(arrayType); //get a[10]

auto a0GEP = builder->create_gep(a, {CONST_INT(0), CONST_INT(0)});
builder->create_store(CONST_INT(10), a0GEP); // a[0] = 10

auto a0Load = builder->create_load(a0GEP);
auto mul = builder->create_imul(a0Load, CONST_INT(2)); // a[0] * 2
auto a1GEP = builder->create_gep(a, {CONST_INT(0), CONST_INT(1)});
builder->create_store(mul, a1GEP); // a[1] = a[0] * 2

auto a1Load = builder->create_load(a1GEP);
```

```
    builder->create_store(a1Load, retAlloca);
    auto retLoad = builder->create_load(retAlloca);
    builder->create_ret(retLoad); // return a[1]
```

llvm代码如下：

```
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0 ; assign ret = 0

  %op1 = alloca [10 x i32] ; get a[10]

  %op2 = getelementptr [10 x i32], [10 x i32]* %op1, i32 0, i32 0
  store i32 10, i32* %op2 ; a[0] = 10

  %op3 = load i32, i32* %op2
  %op4 = mul i32 %op3, 2 ; a[0] * 2
  %op5 = getelementptr [10 x i32], [10 x i32]* %op1, i32 0, i32 1
  store i32 %op4, i32* %op5 ; a[1] = a[0] * 2

  %op6 = load i32, i32* %op5
  store i32 %op6, i32* %op0
  %op7 = load i32, i32* %op0
  ret i32 %op7 ; return a[1]
```

fun.c

首先是`callee`函数的 BasicBlock，cpp代码如下：

```
    auto bb = BasicBlock::create(module, "entry", calleeFun);
    builder->set_insert_point(bb);

    auto aAlloca = builder->create_alloca(Int32Type);
    std::vector<Value *> args;
    for (auto arg = calleeFun->arg_begin(); arg != calleeFun->arg_end(); arg++) {
      args.push_back(*arg);
    }
    builder->create_store(args[0], aAlloca);  // handle a

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

    auto aLoad = builder->create_load(aAlloca);
    auto mul = builder->create_imul(aLoad, CONST_INT(2)); // a * 2

    builder->create_store(mul, retAlloca);
    auto retLoad = builder->create_load(retAlloca);
    builder->create_ret(retLoad); // return a * 2
```

llvm代码如下：

```
label_entry:
  %op1 = alloca i32
  store i32 %arg0, i32* %op1 ; handle a

  %op2 = alloca i32
  store i32 0, i32* %op2 ; assign ret = 0

  %op3 = load i32, i32* %op1
  %op4 = mul i32 %op3, 2 ; a * 2

  store i32 %op4, i32* %op2
  %op5 = load i32, i32* %op2
  ret i32 %op5 ; return a * 2
```

之后是主函数的 BasicBlock，cpp代码如下：

```
  bb = BasicBlock::create(module, "entry", mainFun);
  builder->set_insert_point(bb);

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

  auto call = builder->create_call(calleeFun, {CONST_INT(110)});
  builder->create_ret(call); // return callee(110)
```

llvm代码如下：

```
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0 ; assign ret = 0

  %op1 = call i32 @callee(i32 110)
  ret i32 %op1 ;  return callee(110)
```

if.c

首先是主函数的 BasicBlock，cpp代码如下：

```
  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); // assign ret = 0
```

```
    auto aAlloca = builder->create_alloca(Float32Type);
    builder->create_store(CONST_FP(5.555), aAlloca); // get a = 5.555

    auto aLoad = builder->create_load(aAlloca);
    auto fcmp = builder->create_fcmp_gt(aLoad, CONST_FP(1)); // cond

    auto trueBB = BasicBlock::create(module, "trueBB", mainFun);    // true br
    auto falseBB = BasicBlock::create(module, "falseBB", mainFun);  // false br
    auto retBB = BasicBlock::create(
        module, "", mainFun);
    builder->create_cond_br(fcmp, trueBB, falseBB); // branch
```

llvm代码如下：

```
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0 ; assign ret = 0

  %op1 = alloca float
  store float 0x40163851e0000000, float* %op1 ;get a = 5.555

  %op2 = load float, float* %op1
  %op3 = fcmp ugt float %op2,0x3ff0000000000000 ; cond

  br i1 %op3, label %label_trueBB, label %label_falseBB ; branch
```

之后是真分支的 BasicBlock，cpp代码如下：

```
    builder->set_insert_point(trueBB);  // if true
    builder->create_store(CONST_INT(233), retAlloca);
    builder->create_br(retBB);  // br retBB
```

llvm代码如下：

```
label_trueBB:
  store i32 233, i32* %op0
  br label %label4
```

之后是假分支的 BasicBlock，cpp代码如下：

```
    builder->set_insert_point(falseBB);  // if false
    builder->create_br(retBB);  // br retBB
```

llvm代码如下：

```
label_falseBB:
  br label %label4
```

最后是返回值的 BasicBlock，cpp代码如下：

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

llvm代码如下：

```
label4:
  %op5 = load i32, i32* %op0
  ret i32 %op5
```

while.c

首先是主函数的 BasicBlock，cpp代码如下：

```
// main function
    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); // assign ret = 0

    auto aAlloca = builder->create_alloca(Int32Type);
    auto iAlloca = builder->create_alloca(Int32Type);
    builder->create_store(CONST_INT(10), aAlloca);
    builder->create_store(CONST_INT(0), iAlloca); // i = 0, a = 10

    auto condBB = BasicBlock::create(module, "cond", mainFun);
    auto loopBB = BasicBlock::create(module, "loop", mainFun);
    auto retBB = BasicBlock::create(module, "", mainFun);
    builder->create_br(condBB); // while branch
```

llvm代码如下：

```
label_entry:
  %op0 = alloca i32
  store i32 0, i32* %op0 ; assign ret = 0
```

```
  %op1 = alloca i32
  %op2 = alloca i32
  store i32 10, i32* %op1
  store i32 0, i32* %op2 ; i = 0, a = 10

  br label %label_cond ; while branch
```

之后是每次循环条件判断的 BasicBlock，cpp代码如下：

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

llvm代码如下：

```
label_cond:
  %op3 = load i32, i32* %op2
  %op4 = icmp slt i32 %op3, 10
  br i1 %op4, label %label_loop, label %label10
```

之后是主循环体的 BasicBlock，cpp代码如下：

```
// loopBB
builder->set_insert_point(loopBB);
iLoad = builder->create_load(iAlloca);
auto add = builder->create_iadd(iLoad, CONST_INT(1));
builder->create_store(add, iAlloca); // i = i + 1

auto aLoad = builder->create_load(aAlloca);
iLoad = builder->create_load(iAlloca);
add = builder->create_iadd(aLoad, iLoad);
builder->create_store(add, aAlloca); // a = a + i

builder->create_br(condBB);
```

llvm代码如下：

```
label_loop:
  %op5 = load i32, i32* %op2
  %op6 = add i32 %op5, 1
  store i32 %op6, i32* %op2 ; i = i + 1

  %op7 = load i32, i32* %op1
```

```
  %op8 = load i32, i32* %op2
  %op9 = add i32 %op7, %op8
  store i32 %op9, i32* %op1 ; a = a + i

  br label %label_cond
```

最后是返回的 BasicBlock，cpp代码如下：

```
// retBB
builder->set_insert_point(retBB);
aLoad = builder->create_load(aAlloca);
builder->create_store(aLoad, retAlloca);
auto retLoad = builder->create_load(retAlloca);
builder->create_ret(retLoad);
```

llvm代码如下：

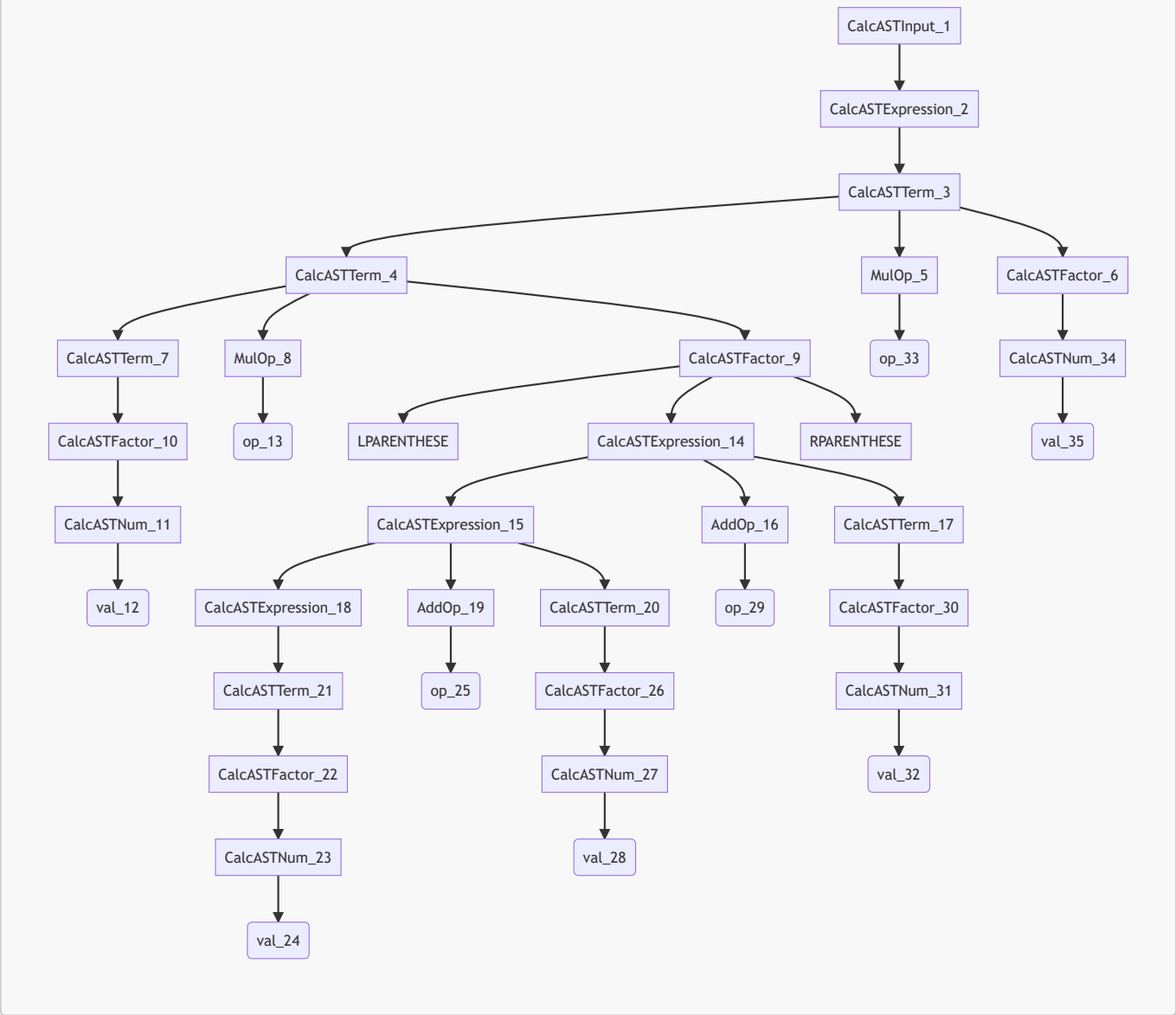
```
label10:                                          ; preds = %label_cond
  %op11 = load i32, i32* %op1
  store i32 %op11, i32* %op0
  %op12 = load i32, i32* %op0
  ret i32 %op12
```

以上便是两者的对应关系

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 2.1 中的编号）：

1->2->3->4->7->10->11->12->8->13->9->14->15->18->21->22->23->24->19->25->20->26->27->28->16->29->17->31->32->34->5->35->6->36->37

实验难点

本次实验相对来说难点较少，大部分时间都是在阅读文档和代码，并且给出的实验案例也比较简单。比较难的地方就是回答实验报告中的问题了，这两个问题所需阅读的材料还是挺多的。

下面是gcd_array_generator.cpp中一个问题的回答

```
  auto x0GEP = builder->create_gep(x, {CONST_INT(0), CONST_INT(0)});  // GEP: 这里为什么是{0, 0}呢？（实验报告相关）
  builder->create_store(CONST_INT(90), x0GEP);
  auto y0GEP = builder->create_gep(y, {CONST_INT(0), CONST_INT(0)});  // GEP: 这里为什么是{0, 0}呢？（实验报告相关）
  builder->create_store(CONST_INT(18), y0GEP);

  x0GEP = builder->create_gep(x, {CONST_INT(0), CONST_INT(0)});
  y0GEP = builder->create_gep(y, {CONST_INT(0), CONST_INT(0)});
```

```
    call = builder->create_call(funArrayFun, {x0GEP, y0GEP});           // 为什么这里
传的是{x0GEP, y0GEP}呢?
```

这里的{0, 0}指的就是问题一中第一种分配方式的两个参数，因此只有在结构体指针不偏移，内部偏移也为0的时候，才能够正确表示出x[0]，y[0]。后面传入{x0GEP, y0GEP}是因为函数的参数列表是一个指针型的`vector`，因此需要传入指针而非值，并且需要以数组的形式传入

实验反馈

本次实验设计挺合理的，没有什么更好的意见或建议