Lab4 实验报告

学号：202008010415　　姓名：温先武

实验介绍

实验框架

本次实验使用了由 C++编写的 LightIR 来生成 LLVM IR。为了便于大家进行实验，该框架自动完成了语法树到 C++ 上的抽象语法树的转换。我们可以使用访问者模式来设计抽象语法树中的算法。大家可以参考打印抽象语法树的算法，以及运行 test_ast 来理解访问者模式下算法的执行流程。

在 include/cminusf_builder.hpp 中，我还定义了一个用于存储作用域的类 Scope。它的作用是辅助我们在遍历语法树时，管理不同作用域中的变量。它提供了以下接口：

```
// 进入一个新的作用域
void enter();
// 退出一个作用域
void exit();
// 往当前作用域插入新的名字->值映射
bool push(std::string name, Value *val);
// 根据名字，寻找到值
Value* find(std::string name);
// 判断当前是否在全局作用域内
bool in_global();
```

你们需要根据语义合理调用 enter 与 exit，并且在变量声明和使用时正确调用 push 与 find。在类 CminusfBuilder 中，有一个 Scope 类型的成员变量 scope，它在初始化时已经将 input、output 等函数加入了作用域中。因此，你们在进行名字查找时不需要顾虑是否需要对特殊函数进行特殊操作。

运行与调试

1.运行 cminusfc

```
mkdir build && cd build
cmake .. -DLLVM_DIR=/path/to/your/llvm/install/lib/cmake/llvm/
make -j
# 安装它以便于链接 libcminus_io.a
make install
```

编译后会产生 cminusfc 程序，它能将 cminus 文件输出为 LLVM IR，也可以利用 clang 将 IR 编译成二进制。程序逻辑写在 src/cminusfc/cminusfc.cpp 中。

当需要对 .cminus 文件测试时，可以这样使用：

```
# 假设 cminusfc 的路径在你的$PATH 中
# 利用构建好的 Module 生成 test.ll
# 注意，如果调用了外部函数，如 input, output 等，则无法使用 lli 运行
cminusfc test.cminus -emit-llvm
# 假设 libcminus_io.a 的路径在$LD_LIBRARY_PATH 中，clang 的路径在$PATH 中
# 1. 利用构建好的 Module 生成 test.ll
# 2. 调用 clang 来编译 IR 并链接上静态链接库 libcminus_io.a，生成二进制文件 test
cminusfc test.cminus
```

2.自动测试

助教贴心地为大家准备了自动测试脚本，它在 tests/lab4 目录下，使用方法如下：

```
# 在 tests/lab4 目录下运行：
./lab4_test.py
```

如果完全正确，它会输出：

```
==========TEST START==========
Case 01: Success
Case 02: Success
Case 03: Success
Case 04: Success
Case 05: Success
Case 06: Success
Case 07: Success
Case 08: Success
Case 09: Success
Case 10: Success
```

```
Case 11: Success
Case 12: Success
============TEST END============
```

实验要求

- 阅读 cminus-f 的语义规则成为语言律师我们将按照语义实现程度进行评分
- 阅读 LightIR 核心类介绍
- 阅读实验框架，理解如何使用框架以及注意事项
- 修改 src/cminusfc/cminusf_builder.cpp 来实现自动 IR 产生的算法，使得它能正确编译任何合法的 cminus-f 程序
- 在 report.md 中解释你们的设计，遇到的困难和解决方案

实验难点

1. 因为 cminusf_builder.cpp 中的 16 个函数都是 void 类型的，没有返回值，一时不知道怎么保存计算的结果，后面思考了一段时间察觉到全局变量好像可以实现，每次全局变量的类型或者值发生变化，就更新全局变量，这样在访问下一个结点时获取全局变量就可以取得最新的值，这样信息可以传入或者传出信息给下层或上层时。

   若子节点需要向父节点传值，则通过全局变量来实现。在visit节点时还要加入语义分析的部分，判断语义是否正确。

2. 在 cminus-f 的语义中，有很多需要注意的小细节，比如五种类型转化，

- 赋值时
- 返回值类型和函数签名中的返回类型不一致时
- 函数调用时实参和函数签名中的形参类型不一致时
- 二元运算的两个参数类型不一致时

- 下标计算时

要区分类型转化时是左边的类型进行转化还是右边的类型进行转化，例如：浮点数和整型一起运算时，谁是整型就需要进行类型提升，转换成浮点数类型。而在函数调用时若实参类型和函数形参不相同时，需要把实参的类型转化为形参的类型。本次实验考虑了很多次类型转化，需要很小心。

3. 对于 void CminusfBuilder::visit(ASTVar &node)函数，虽然语法中说了其 var 可以是一个整型变量、浮点变量，或者一个取了下标的数组变量，但我在实现的过程中感觉有些困难。

4. 访问者模式是什么？

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
```

5. 实验目的：lab4 和 lab3 的区别就是 lab3 是利用 llvm 接口来生成中间代码，而 lab4 直接写一个编译器将 cminus 转成中间代码。

通过查阅资料发现，使用 AST 和 visit 生成 IR 指令的过程：

由于cminus-f的语法还是比lab3中计算器要复杂的多，供参考的抽象语法树的打印代码也还是和生成IR指令由很大差别，在开始编写时存在困难。通过阅读AST的头文件和lab3的visitor.cpp，首先确定了visit函数生成IR指令的过程：

- visit函数调用当前节点的子节点accept函数，即调用子节点的visit函数
- 处理当前节点，生成指令

若子节点需要向父节点传值，则通过全局变量来实现。在visit节点时还要加入语义分析的部分，判断语义是否正确。

6. LightIR 核心类介绍中 API 的使用，例如，创建一个函数，创建一个常量等等。如：

- API：

```
static Function *create(FunctionType *ty, const std::string &name, Module *parent);
// 创建并返回Function，参数依次是待创建函数类型ty，函数名字name(不可为空)，函数所属的Module
FunctionType *get_function_type() const;
// 返回此函数类的函数类型
Type *get_return_type() const;
// 返回此函数类型的返回值类型
void add_basic_block(BasicBlock *bb);
// 将bb添加至Function的bb链表上（调用bb里的创建函数时会自动调用此函数挂在function的bb链表上）
unsigned get_num_of_args() const;
// 得到函数形参数数量
unsigned get_num_basic_blocks() const;
// 得到函数基本块数量
Module *get_parent() const;
// 得到函数所属的Module
std::list<Argument *>::iterator arg_begin()
// 得到函数形参的list的起始迭代器
std::list<Argument *>::iterator arg_end()
// 得到函数形参的list的终止迭代器
void remove(BasicBlock* bb)
// 从函数的bb链表中删除一个bb
std::list<BasicBlock *> &get_basic_blocks()
// 返回函数bb链表
std::list<Argument *> &get_args()
// 返回函数的形参链表
void set_instr_name();
// 给函数中未命名的基本块和指令命名
```

实验设计

从实验说明中知道，要编写的代码是 cminusf_builder.cpp，其内含各个 visit 函数，我们需要补全这些 visit 的函数。在写代码之前，要好好看看打印抽象语法树的算法的代码（虽然参考价值不多，但其中的 visit 的代码还是很有启发性的，需要仔细阅读），掌握 logging 工具（虽然掌握了之后还是特别难 debug），仔细看 Light IR 文档（主要看 C++ APIs 部分），仔细看 cminus-f 的语法与语义（这个特别重要）。

- 宏定义和一些全局变量

```
// use these macros to get constant value
#define CONST_INT(num) \
  ConstantInt::get(num, module.get())        /* 增加一个有关整型的宏 */
#define CONST_FP(num) \
  ConstantFP::get((float)num, module.get())
#define CONST_ZERO(type) \
  ConstantZero::get(type, module.get())     /* 此处要修改为 type */
```

```
#define Int32Type \
  Type::get_int32_type(module.get())          /* 获取 int32 类型 */
#define FloatType \
  Type::get_float_type(module.get())          /* 获取 float 类型 */

#define checkInt(num) \
  num->get_type()->is_integer_type()          /* 整型判断 */
#define checkFloat(num) \
  num->get_type()->is_float_type()            /* 浮点型判断 */
#define checkPointer(num) \
  num->get_type()->is_pointer_type()          /* 指针类型判断 */

// You can define global variables here
// to store state
Value* Res;                                   /* 存储返回的结果 */
Value* arg;                                   /* 存储参数指针，用于 Param 的处理 */
bool need_as_address = false;     /* 标志是返回值还是返回地址 */
```

- Program，程序

program → declaration-list

```
/* Program，程序, program->declaration-list */
void CminusfBuilder::visit(ASTProgram &node) {
  for (auto decl : node.declarations)             /* 遍历 declaration-list */
      decl->accept(*this);                        /* 处理每一个 declaration */
}
```

- Num，数值

```
/* Num，数值 */
void CminusfBuilder::visit(ASTNum &node) {
  if (node.type == TYPE_INT)                      /* 若为整型 */
      Res = ConstantInt::get(node.i_val, module.get());    /* 获取结点中存储的整型数值 */
  else if (node.type == TYPE_FLOAT)               /* 若为浮点型 */
      Res = ConstantFP::get(node.f_val, module.get());     /* 获取结点中存储的浮点型数值 */
}
```

- Var-Declaration，变量声明

var-declaration → type-specifier ID ; | type-specifier ID [INTEGER]

注意：根据 cminus 语法规则，全局变量需要初始化为全 0 。cminus-f 的基

础类型只有整型(int)、浮点型(float)和 void。而在变量声明中，只有整型和浮点型可以使用，void 仅用于函数声明。一个变量声明定义一个整型或者浮点型的变量，或者一个整型或浮点型的数组变量（这里整型指的是 32 位有符号整型，浮点数是指 32 位浮点数）。数组变量在声明时，INTEGER \textbf{INTEGER}INTEGER 应当大于 0。一次只能声明一个变量。

```
/* Var-Declaration, 变量声明, var-declaration -> type-specifier ID
 *                                                | type-specifier ID [INTEGER] */
void CminusfBuilder::visit(ASTVarDeclaration &node) {
    Type* tmpType;                          /* 类型暂存变量，用于存储变量的类型，用于后续申请空间 */
    if (node.type == TYPE_INT)          /* 若为整型 */
        tmpType = Int32Type;              /* 则 type 为整数类型 */
    else if (node.type == TYPE_FLOAT)     /* 则为浮点型 */
        tmpType = FloatType;                  /* 则 type 为浮点类型 */
    if (node.num != nullptr) {              /* 若为数组类型 */
        /* 获取需开辟的对应大小的空间的类型指针 */
        auto* arrayType = ArrayType::get(tmpType, node.num->i_val); /* 获取对应的数组 Type */
        auto initializer = CONST_ZERO(tmpType);                                 /* 全局变量初始化为 0 */
        Value* arrayAlloca;                   /* 存储申请到的数组空间的地址 */
        if (scope.in_global())              /* 若为全局数组，则开辟全局数组 */
            arrayAlloca = GlobalVariable::create(node.id, module.get(), arrayType, false, initializer);
        else                                    /* 若不是全局数组，则开辟局部数组 */
            arrayAlloca = builder->create_alloca(arrayType);
        scope.push(node.id, arrayAlloca);/* 将获得的数组变量加入域 */
    }
    else {                                      /* 若不是数组类型 */
        auto initializer = CONST_ZERO(tmpType); /* 全局变量初始化为 0 */
        Value* varAlloca;                     /* 存储申请到的变量空间的地址 */
        if (scope.in_global())              /* 若为全局变量，则申请全局空间 */
            varAlloca = GlobalVariable::create(node.id, module.get(), tmpType, false, initializer);
        else                                    /* 若不是全局变量，则申请局部空间 */
            varAlloca = builder->create_alloca(tmpType);
        scope.push(node.id, varAlloca); /* 将获得变量加入域 */
```

```
    }
}
```

- Fun-Declaration，函数声明

fun-declaration → type-specifier ID (params) compound-stmt

注意：函数声明包含了返回类型，标识符，由逗号分隔的形参列表，还有一个复合语句。当函数的返回类型是 void 时，函数不返回任何值。函数的参数可以是 void ，也可以是一个列表。当函数的形参是 void 时，调用该函数时不用传入任何参数。形参中跟着中括号代表数组参数，它们可以有不同长度。整型参数通过值来传入函数（pass by value），而数组参数通过引用来传入函数（pass by reference，即指针）。函数的形参拥有和函数声明的复合语句相同的作用域，并且每次函数调用都会产生一组独立内存的参数。（和 C 语言一致）函数可以递归调用。

- 首先需要确定返回值的类型，然后确定参数列表中各个参数的类型，最后才能确定函数的类型。
- 函数定义后，需要把全局变量中的 `cur_fun` 指向当前这个函数，并在创建函数块且执行 `builder->set_insert_point(BB);` 后进入新的作用域。
- 在这个函数块中，我们需要获取函数的形参，给每个参数在内存中分配位置，并把值给对应的内存地址，然后加入当前作用域。
- 下一步就是访问函数中的复合语句，当访问结束后，函数将根据当前函数返回值的类型执行 `ret` 操作。
- 最后退出当前作用域

```
/* Fun-Declaration，函数声明，fun-declaration -> type-specifier ID ( params ) compound-stmt */
void CminusfBuilder::visit(ASTFunDeclaration &node) {
    Type* retType;                  /* 返回类型 */
    /* 根据不同的返回类型，设置 retType */
    if (node.type == TYPE_INT) { retType = Int32Type; }
    if (node.type == TYPE_FLOAT) { retType = FloatType; }
    if (node.type == TYPE_VOID) { retType = Type::get_void_type(module.get()); }
    /* 根据函数声明，构造形参列表（此处的形参即参数的类型） */
    std::vector<Type*> paramsType;  /* 参数类型列表 */
    for (auto param : node.params) {
        if (param->isarray) {       /* 若参数为数组形式，则存入首地址指针 */
            if (param->type == TYPE_INT)        /* 若为整型 */
                paramsType.push_back(Type::get_int32_ptr_type(module.get()));
```

```
            else if(param->type == TYPE_FLOAT)  /* 若为浮点型 */
                paramsType.push_back(Type::get_float_ptr_type(module.get()));
        }
        else {                          /* 若为单个变量形式，则存入对应类型 */
            if (param->type == TYPE_INT)          /* 若为整型 */
                paramsType.push_back(Int32Type);
            else if (param->type == TYPE_FLOAT) /* 若为浮点型 */
                paramsType.push_back(FloatType);
        }
    }
    auto funType = FunctionType::get(retType, paramsType);                  /* 函数
结构 */
    auto function = Function::create(funType, node.id, module.get());   /* 创建函数
*/
    scope.push(node.id, function);  /* 将函数加入到域 */
    scope.enter();                          /* 进入此函数作用域 */
    auto bb = BasicBlock::create(module.get(), node.id + "_entry", function);/* 创建基
本块 */
    builder->set_insert_point(bb);  /* 将基本块加入到 builder 中 */
    /* 将实参和形参进行匹配 */
    std::vector<Value*> args;          /* 创建 vector 存储实参 */
    for (auto arg = function->arg_begin();arg != function->arg_end();arg++) {/* 遍
历实参列表 */
        args.push_back(*arg);          /* 将实参加入 vector */
    }
    for (int i = 0;i < node.params.size();i++) {      /* 遍历形参列表（=遍历实参列表）
*/
        auto param = node.params[i];           /* 取出对应形参 */
        arg = args[i];                         /* 取出对应实参 */
        param->accept(*this);               /* 调用 param 的 accept 进行处理 */
    }
    node.compound_stmt->accept(*this);         /* 处理函数体内语句 compound-
stmt */
    if (builder->get_insert_block()->get_terminator() == nullptr) {
        if (function->get_return_type()->is_void_type())
            builder->create_void_ret();
        else if (function->get_return_type()->is_float_type())
            builder->create_ret(CONST_FP(0.));
        else
            builder->create_ret(CONST_INT(0));
    }
    scope.exit();                           /* 退出此函数作用域 */
}
```

- Param，参数

param → type-specifier ID | type-specifier ID []

```
/* Param, 参数 */
void CminusfBuilder::visit(ASTParam &node) {
  Value* paramAlloca; /* 该参数的存储空间 */
  if (node.isarray) { /* 若为数组 */
        if (node.type == TYPE_INT)            /* 若为整型数组，则开辟整型数组存储空间 */
              paramAlloca                                                                    = builder->create_alloca(Type::get_int32_ptr_type(module.get()));
        else if (node.type == TYPE_FLOAT) /* 若为浮点数数组，则开辟浮点数数组存储空间 */
              paramAlloca                                                                    = builder->create_alloca(Type::get_float_ptr_type(module.get()));
  }
  else {                        /* 若不是数组 */
        if (node.type == TYPE_INT)            /* 若为整型，则开辟整型存储空间 */
              paramAlloca = builder->create_alloca(Int32Type);
        else if (node.type == TYPE_FLOAT) /* 若为浮点数，则开辟浮点数存储空间 */
              paramAlloca = builder->create_alloca(FloatType);
  }
  builder->create_store(arg, paramAlloca);      /* 将实参 load 到开辟的存储空间中 */
  scope.push(node.id, paramAlloca);               /* 将参数 push 到域中 */
}
```

- CompoundStmt, 函数体语句

compound-stmt → { local-declarations statement-list }

```
/* CompoundStmt, 函数体语句, compound-stmt -> {local-declarations statement-list} */
void CminusfBuilder::visit(ASTCompoundStmt &node) {
  scope.enter();          /* 进入函数体内的作用域 */
  for (auto local_declaration : node.local_declarations)    /* 遍历 */
        local_declaration->accept(*this);    /* 处理每一个局部声明 */
  for (auto statement : node.statement_list)                    /* 遍历 */
        statement->accept(*this);               /* 处理每一个语句 */
  scope.exit();           /* 退出作用域 */
}
```

- ExpressionStmt, 表达式语句

expression-stmt → expression ; | ;

```
/* ExpressionStmt, 表达式语句, expression-stmt -> expression;
 *                                                  | ; */
```

```
void CminusfBuilder::visit(ASTExpressionStmt &node) {          /*  */
    if (node.expression != nullptr)       /* 若对应表达式存在 */
        node.expression->accept(*this); /* 则处理该表达式 */
}
```

- SelectionStmt, if 语句

selection-stmt → if (expression) statement | if (expression) statement else statement

注意：if 语句中的表达式将被求值，若结果的值等于 0，则第二个语句执行（如果存在的话），否则第一个语句会执行。为了避免歧义，else 将会匹配最近的 if 。

```
/* SelectionStmt, if 语句, selection-stmt -> if (expression) statement
 *                                          | if (expression) statement else statement */
void CminusfBuilder::visit(ASTSelectionStmt &node) {
    auto function = builder->get_insert_block()->get_parent();  /* 获得当前所对应的函数 */
    node.expression->accept(*this);             /* 处理条件判断对应的表达式，得到返回值存到 expression 中 */
    auto resType = Res->get_type();             /* 获取表达式得到的结果类型 */
    Value* TrueFalse;                           /* 存储 if 判断的结果 */
    if (resType->is_integer_type()) {           /* 若结果为整型，则针对整型进行处理(bool 类型视为整型) */
        auto intType = Int32Type;
        TrueFalse = builder->create_icmp_gt(Res, CONST_ZERO(intType));  /* 大于 0 视为 true */
    }
    else if (resType->is_float_type()) {        /* 若结果为浮点型，则针对浮点数进行处理 */
        auto floatType = FloatType;
        TrueFalse = builder->create_fcmp_gt(Res, CONST_ZERO(floatType));/* 大于 0 视为 true */
    }
    if (node.else_statement != nullptr) {       /* 若存在 else 语句 */
        auto trueBB = BasicBlock::create(module.get(), "true", function);    /* 创建符合条件块 */
        auto falseBB = BasicBlock::create(module.get(), "false", function); /* 创建 else 块 */
        builder->create_cond_br(TrueFalse, trueBB, falseBB);       /* 设置跳转语句
```

```
*/

        builder->set_insert_point(trueBB);   /* 符合 if 条件的块 */
        node.if_statement->accept(*this);    /* 处理符合条件后要执行的语句 */
        auto curTrueBB = builder->get_insert_block();    /* 将块加入 */

        builder->set_insert_point(falseBB); /* else 的块 */
        node.else_statement->accept(*this); /* 处理 else 语句 */
        auto curFalseBB = builder->get_insert_block();   /* 将块加入 */

        /* 处理返回，避免跳转到对应块后无 return */
        auto trueTerm = builder->get_insert_block()->get_terminator();   /* 判断 true 语句中是否存在 ret 语句 */
        auto falseTerm = builder->get_insert_block()->get_terminator(); /* else 语句中是否存在 ret 语句 */
        BasicBlock* retBB;
        if (trueTerm == nullptr || falseTerm == nullptr) {   /* 若有一方不存在 return 语句，则需要创建返回块 */
            retBB = BasicBlock::create(module.get(), "ret", function);   /* 创建 return 块 */
            builder->set_insert_point(retBB);                    /* return 块（即后续语句）  */
        }
        if (trueTerm == nullptr) {              /* 若符号条件后要执行的语句中不存在 return */
            builder->set_insert_point(curTrueBB);    /* 则设置跳转 */
            builder->create_br(retBB);                  /* 跳转到刚刚设置的 return 块 */
        }
        if (falseTerm == nullptr) {             /* 若 else 语句中不存在 return */
            builder->set_insert_point(curFalseBB);   /* 则设置跳转 */
            builder->create_br(retBB);                  /* 跳转到刚刚设置的 return 块 */
        }
    }
    else {   /* 若不存在 else 语句，则只需要设置 true 语句块和后续语句块即可 */
        auto trueBB = BasicBlock::create(module.get(), "true", function);    /* true 语句块 */
        auto retBB = BasicBlock::create(module.get(), "ret", function);       /* 后续语句块 */
        builder->create_cond_br(TrueFalse, trueBB, retBB);   /* 根据条件设置跳转指令 */

        builder->set_insert_point(trueBB);   /* true 语句块 */
```

```
        node.if_statement->accept(*this);    /* 执行条件符合后要执行的语句 */
        if (builder->get_insert_block()->get_terminator() == nullptr)    /* 补充 return（同上） */
            builder->create_br(retBB);        /* 跳转到刚刚设置的 return 块 */
        builder->set_insert_point(retBB);    /* return 块（即后续语句） */
    }
}
```

- IterationStmt, while 语句

iteration-stmt → while (expression) statement

注意：while 语句是 cminus-f 中唯一的迭代语句。它执行时，会不断对表达式进行求值，并且在对表达式的求值结果等于 0 前，循环执行执下面的语句

```
/* IterationStmt, while 语句, iteration-stmt -> while (expression) statement */
void CminusfBuilder::visit(ASTIterationStmt &node) {
    auto function = builder->get_insert_block()->get_parent();  /* 获得当前所对应的函数 */
    auto conditionBB = BasicBlock::create(module.get(), "condition", function); /* 创建条件判断块 */
    auto loopBB = BasicBlock::create(module.get(), "loop", function);              /* 创建循环语句块 */
    auto retBB = BasicBlock::create(module.get(), "ret", function);                /* 创建后续语句块 */
    builder->create_br(conditionBB);            /* 跳转到条件判断块 */

    builder->set_insert_point(conditionBB); /* 条件判断块 */
    node.expression->accept(*this);             /* 处理条件判断对应的表达式，得到返回值存到 expression 中 */
    auto resType = Res->get_type();             /* 获取表达式得到的结果类型 */
    Value* TrueFalse;                           /* 存储 if 判断的结果 */
    if (resType->is_integer_type()) {           /* 若结果为整型，则针对整型进行处理(bool 类型视为整型) */
        auto intType = Int32Type;
        TrueFalse = builder->create_icmp_gt(Res, CONST_ZERO(intType));  /* 大于 0 视为 true */
    }
    else if (resType->is_float_type()) {        /* 若结果为浮点型，则针对浮点数进行处理 */
        auto floatType = FloatType;
```

```
        TrueFalse = builder->create_fcmp_gt(Res, CONST_ZERO(floatType));/* 大于
0 视为 true */
    }
    builder->create_cond_br(TrueFalse, loopBB, retBB);  /* 设置条件跳转语句 */

    builder->set_insert_point(loopBB);          /* 循环语句执行块 */
    node.statement->accept(*this);                /* 执行对应的语句 */
    if (builder->get_insert_block()->get_terminator() == nullptr)    /* 若无返回，则
补充跳转 */
        builder->create_br(conditionBB);      /* 跳转到条件判断语句 */

    builder->set_insert_point(retBB);           /* return 块（即后续语句） */
}
```

- ReturnStmt，返回语句

return-stmt → return ; | return expression ;

注意：return 语句可以返回值，也可以不返回值。未声明为 void 类型的函数必须返回和函数返回类型相同的值

```
/* ReturnStmt，返回语句, return-stmt -> return;
 *                                          | return expression; */
void CminusfBuilder::visit(ASTReturnStmt &node) {
    auto function = builder->get_insert_block()->get_parent();  /* 获得当前所对应
的函数 */
    auto retType = function->get_return_type();       /* 获取返回类型 */
    if (retType->is_void_type()) {          /* 如果是 void */
        builder->create_void_ret();       /* 则创建 void 返回，随后 return，无需后
续操作 */
        return;
    }
    /* 处理非 void 的情况 */
    node.expression->accept(*this);       /* 处理条件判断对应的表达式，得到返回
值存到 expression 中 */
    auto resType = Res->get_type();       /* 获取表达式得到的结果类型 */
    /* 处理 expression 返回的结果和需要 return 的结果类型不匹配的问题 */
    if (retType->is_integer_type() && resType->is_float_type())
        Res = builder->create_fptosi(Res, Int32Type);
    if (retType->is_float_type() && resType->is_integer_type())
        Res = builder->create_sitofp(Res, Int32Type);
    builder->create_ret(Res);                 /* 创建 return，将 expression 的结果进行
返回 */
}
```

- Var，变量引用

var → ID | ID [expression]

注意：var 可以是一个整型变量、浮点变量，或者一个取了下标的数组变量。数组的下标值是整型，它的值是表达式计算结果或结果进行类型转换后的整型值一个负的下标会导致程序终止，需要调用框架中的内置函数 neg_idx_except（该内部函数会主动退出程序，只需要调用该函数即可），但是对于上界并不做检查。

```
/* Var，变量引用，var -> ID
 *                    | ID [expression] */
void CminusfBuilder::visit(ASTVar &node) {
    auto var = scope.find(node.id);                    /* 从域中取出对应变量 */
    bool should_return_lvalue = need_as_address;    /* 判断是否需要返回地址（即是否由赋值语句调用） */
    need_as_address = false;                    /* 重置全局变量 need_as_address */
    Value* index = CONST_INT(0);                /* 初始化 index */
    if (node.expression != nullptr) {    /* 若有 expression */
        node.expression->accept(*this); /* 处理 expression，得到结果 Res */
        auto res = Res;                         /* 存储结果 */
        if (checkFloat(res))                    /* 判断结果是否为浮点数 */
            res = builder->create_fptosi(res, Int32Type);    /* 若是，则矫正为整数 */
        index = res;                            /* 赋值给 index，表示数组下标 */
        /* 判断下标是否为负数。若是，则调用 neg_idx_except 函数进行处理 */
        auto function = builder->get_insert_block()->get_parent();  /* 获取当前函数 */
        auto indexTest = builder->create_icmp_lt(index, CONST_ZERO(Int32Type)); /* test 是否为负数 */
        auto failBB = BasicBlock::create(module.get(), node.id + "_failTest", function);/* fail 块 */
        auto passBB = BasicBlock::create(module.get(), node.id + "_passTest", function);/* pass 块 */
        builder->create_cond_br(indexTest, failBB, passBB); /* 设置跳转语句 */

        builder->set_insert_point(failBB);  /* fail 块，即下标为负数 */
        auto fail = scope.find("neg_idx_except");                    /* 取出 neg_idx_except 函数 */
        builder->create_call(static_cast<Function*>(fail), {}); /* 调用 neg_idx_except 函数进行处理 */
        builder->create_br(passBB);             /* 跳转到 pass 块 */
```

```
        builder->set_insert_point(passBB);   /* pass 块 */
        if (var->get_type()->get_pointer_element_type()->is_array_type())     /* 若为指向数组的指针 */
            var = builder->create_gep(var, { CONST_INT(0), index });      /* 则进行两层寻址（原因在上一实验中已说明） */
        else {
            if (var->get_type()->get_pointer_element_type()->is_pointer_type()) /* 若为指针 */
                var = builder->create_load(var);            /* 则取出指针指向的元素 */
            var = builder->create_gep(var, { index });   /* 进行一层寻址（因为此时并非指向数组） */
        }
        if (should_return_lvalue) {              /* 若要返回值 */
            Res = var;                                  /* 则返回 var 对应的地址 */
            need_as_address = false;            /* 并重置全局变量 need_as_address */
        }
        else
            Res = builder->create_load(var);/* 否则则进行 load */
        return;
    }
    /* 处理无 expression 的情况 */
    if (should_return_lvalue) { /* 若要返回值 */
        Res = var;                      /* 则返回 var 对应的地址 */
        need_as_address = false;/* 并重置全局变量 need_as_address */
    }
    else {                              /* 否则 */
        if (var->get_type()->get_pointer_element_type()->is_array_type())     /* 若指向数组 */
            Res = builder->create_gep(var, { CONST_INT(0), CONST_INT(0) }); /* 则寻址 */
        else
            Res = builder->create_load(var);/* 否则则进行 load */
    }
}
```

- AssignExpression，赋值语句

注意：赋值语义为：先找到 var 代表的变量地址（如果是数组，需要先对下标表达式求值），然后对右侧的表达式进行求值，求值结果将在转换成变量类型后存储在先前找到的地址中。同时，存储在 var 中的值将作为赋值表达式的求

值结果。在 C 中，赋值对象（即 var ）必须是左值，而左值可以通过多种方式获得。cminus-f 中，唯一的左值就是通过 var 的语法得到的，因此 cminus-f 通过语法限制了 var 为左值，而不是像 C 中一样通过类型检查，这也是为什么 cminus-f 中不允许进行指针算数。

```
/* AssignExpression，赋值语句, var = expression */
void CminusfBuilder::visit(ASTAssignExpression &node) {
    need_as_address = true;              /* 设置 need_as_address，表示需要返回值 */
    node.var->accept(*this);             /* 处理左 var */
    auto var = Res;                      /* 获取地址 */
    node.expression->accept(*this); /* 处理右 expression */
    auto res = Res;                      /* 获得结果 */
    auto varType = var->get_type()->get_pointer_element_type(); /* 获取 var 的类型
*/
    /* 若赋值语句左右类型不匹配，则进行匹配 */
    if (varType == FloatType && checkInt(res))
        res = builder->create_sitofp(res, FloatType);
    if (varType == Int32Type && checkFloat(res))
        res = builder->create_fptosi(res, Int32Type);
    builder->create_store(res, var);/* 进行赋值 */
}
```

- SimpleExpression，比较表达式

simple-expression → additive-expression relop additive-expression | additive-expression

注意：一个简单表达式是一个加法表达式或者两个加法表达式的关系运算。当它是加法表达式时，它的值就是加法表达式的值。而当它是关系运算时，如果关系运算结果为真则值为整型值 1，反之则值为整型值 0。

```
/* SimpleExpression，比较表达式，simple-expression -> additive-expression relop
additive-expression
 *                                                    | additive-expression */
void CminusfBuilder::visit(ASTSimpleExpression &node) {
    node.additive_expression_l->accept(*this);   /* 处理左边的 expression */
    auto lres = Res;                                  /* 获取结果存到 lres 中 */
```

```
    if (node.additive_expression_r == nullptr) { return; }  //* 若不存在右 expression，则直接返回 */
    node.additive_expression_r->accept(*this);  /* 处理右边的 expression */
    auto rres = Res;                                    /* 结果存到 rres 中 */
    if (checkInt(lres) && checkInt(rres)) {       /* 确保两边都是整数 */
        switch (node.op) {  /* 根据不同的比较操作，调用 icmp 进行处理 */
        case OP_LE:
            Res = builder->create_icmp_le(lres, rres);break;
        case OP_LT:
            Res = builder->create_icmp_lt(lres, rres);break;
        case OP_GT:
            Res = builder->create_icmp_gt(lres, rres);break;
        case OP_GE:
            Res = builder->create_icmp_ge(lres, rres);break;
        case OP_EQ:
            Res = builder->create_icmp_eq(lres, rres);break;
        case OP_NEQ:
            Res = builder->create_icmp_ne(lres, rres);break;
        }
    }
    else {  /* 若有一边是浮点类型，则需要先将另一边也转为浮点数，再进行比较 */
        if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
            lres = builder->create_sitofp(lres, FloatType);
        if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
            rres = builder->create_sitofp(rres, FloatType);
        switch (node.op) {  /* 根据不同的比较操作，调用 fcmp 进行处理 */
        case OP_LE:
            Res = builder->create_fcmp_le(lres, rres);break;
        case OP_LT:
            Res = builder->create_fcmp_lt(lres, rres);break;
        case OP_GT:
            Res = builder->create_fcmp_gt(lres, rres);break;
        case OP_GE:
            Res = builder->create_fcmp_ge(lres, rres);break;
        case OP_EQ:
            Res = builder->create_fcmp_eq(lres, rres);break;
        case OP_NEQ:
            Res = builder->create_fcmp_ne(lres, rres);break;
        }
    }
    Res = builder->create_zext(Res, Int32Type); /* 将结果作为整数保存（可作为判断语句中的判断条件） */
}
```

- AdditiveExpression，加法表达式

additive-expression → additive-expression addop term | term

注意：浮点数和整型一起运算时，整型值需要进行类型提升，转换成浮点数类型，且运算结果也是浮点数类型

```
/* AdditiveExpression，加法表达式, additive-expression -> additive-expression addop term
 *                                                                  | term */
void CminusfBuilder::visit(ASTAdditiveExpression &node) {
    if (node.additive_expression == nullptr) {   /* 若无加减法运算 */
        node.term->accept(*this);return;            /* 则直接去做乘除法 */
    }
    node.additive_expression->accept(*this);      /* 处理左 expression */
    auto lres = Res;                                  /* 结果保存在 lres 中 */
    node.term->accept(*this);                         /* 处理右 term */
    auto rres = Res;                                  /* 结果保存在 rres 中 */
    if (checkInt(lres) && checkInt(rres)) {       /* 确保两边都是整数 */
        switch (node.op) {   /* 根据对应加法或是减法，调用 iadd 或是 isub 进行处理 */
        case OP_PLUS:
            Res = builder->create_iadd(lres, rres);break;
        case OP_MINUS:
            Res = builder->create_isub(lres, rres);break;
        }
    }
    else {   /* 若有一边是浮点类型，则需要先将另一边也转为浮点数，再进行处理 */
        if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
            lres = builder->create_sitofp(lres, FloatType);
        if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
            rres = builder->create_sitofp(rres, FloatType);
        switch (node.op) {   /* 根据对应加法或是减法，调用 fadd 或是 fsub 进行处理 */
        case OP_PLUS:
            Res = builder->create_fadd(lres, rres);break;
        case OP_MINUS:
            Res = builder->create_fsub(lres, rres);break;
        }
    }
}
```

- Term，乘除法语句

term → term mulop factor | factor

注意：浮点数和整型一起运算时，整型值需要进行类型提升，转换成浮点数类型，且运算结果也是浮点数类型

```
/* Term, 乘除法语句, Term -> term mulop factor
 *                        | factor */
void CminusfBuilder::visit(ASTTerm &node) {
    if (node.term == nullptr) {                  /* 若无乘法运算 */
        node.factor->accept(*this);return;   /* 则直接去处理元素 */
    }
    node.term->accept(*this);     /* 处理左 term */
    auto lres = Res;                  /* 结果保存在 lres 中 */
    node.factor->accept(*this); /* 处理右 factor */
    auto rres = Res;                  /* 结果保存在 rres 中 */
    if (checkInt(lres) && checkInt(rres)) { /* 确保两边都是整数 */
        switch (node.op) {   /* 根据对应乘法或是除法，调用 imul 或是 idiv 进行处理 */
        case OP_MUL:
            Res = builder->create_imul(lres, rres);break;
        case OP_DIV:
            Res = builder->create_isdiv(lres, rres);break;
        }
    }
    else {   /* 若有一边是浮点类型，则需要先将另一边也转为浮点数，再进行处理 */
        if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
            lres = builder->create_sitofp(lres, FloatType);
        if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
            rres = builder->create_sitofp(rres, FloatType);
        switch (node.op) {   /* 根据对应乘法或是除法，调用 fmul 或是 fdiv 进行处理 */
        case OP_MUL:
            Res = builder->create_fmul(lres, rres);break;
        case OP_DIV:
            Res = builder->create_fdiv(lres, rres);break;
        }
    }
}
```

- Call，函数调用

call → ID (args)

注意：函数调用由一个函数的标识符与一组括号包围的实参组成。实参可以为空，也可以是由逗号分隔的的表达式组成的列表，这些表达式代表着函数调用时，传给形参的值。函数调用时的实参数量和类型必须与函数声明中的形参一致，必要时需要进行类型转换。

```
/* Call，函数调用, call -> ID (args) */
void CminusfBuilder::visit(ASTCall &node) {
    auto function = static_cast<Function*>(scope.find(node.id));    /* 获取需要调用的函数 */
    auto paramType = function->get_function_type()->param_begin();  /* 获取其参数类型 */
    std::vector<Value*> args;          /* 创建 args 用于存储函数参数的值，构建调用函数的参数列表 */
    for (auto arg : node.args) {      /* 遍历形参列表 */
        arg->accept(*this);             /* 对每一个参数进行处理，获取参数对应的值 */
        if (Res->get_type()->is_pointer_type()) {    /* 若参数是指针 */
            args.push_back(Res);      /* 则直接将值加入到参数列表 */
        }
        else {  /* 若不是指针，则需要判断形参和实参的类型是否符合。若不符合则需要类型转换 */
            if (*paramType==FloatType && checkInt(Res))
                Res = builder->create_sitofp(Res, FloatType);
            if (*paramType==Int32Type && checkFloat(Res))
                Res = builder->create_fptosi(Res, Int32Type);
            args.push_back(Res);
        }
        paramType++;                    /* 查看下一个形参 */
    }
    Res = builder->create_call(static_cast<Function*>(function), args); /* 创建函数调用 */
}
```

实验结果

1. 编译

先输入指令 sudo su 并输入密码进入 root 模式，这是下面的 make install 的指令所需要的。

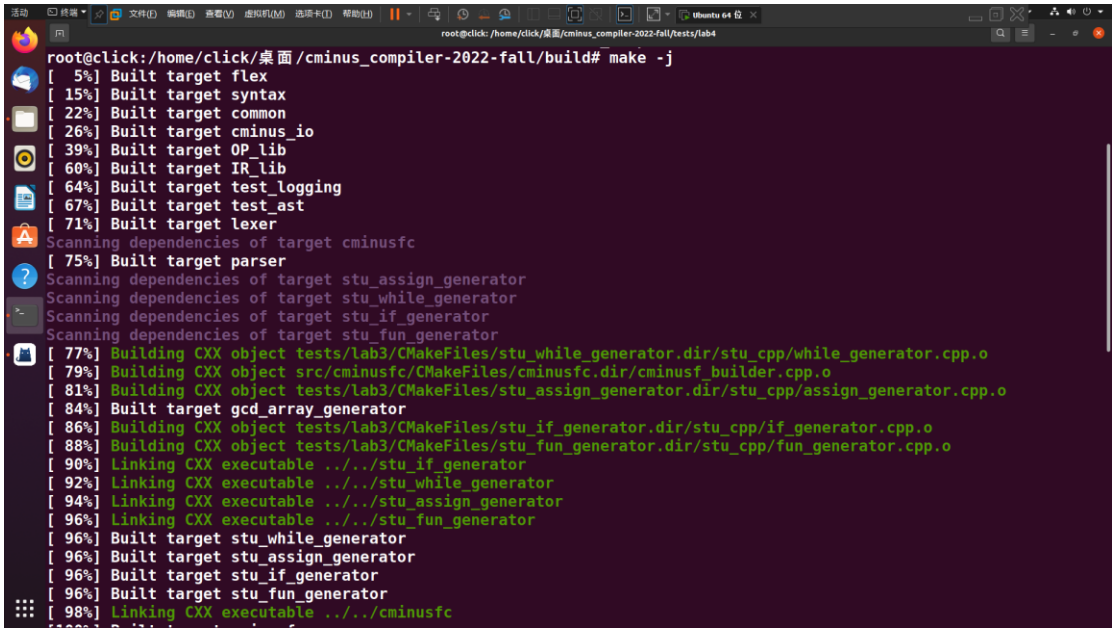
```
click@click:~/桌面/cminus_compiler-2022-fall$ sudo su
[sudo] click 的密码：
root@click:/home/click/桌面/cminus_compiler-2022-fall# cmake .. -DLLVM_DIR=/path/to/your/llvm/install/lib/cmake/llvm/
CMake Error: The source directory "/home/click/桌面" does not appear to contain CMakeLists.txt.
Specify --help for usage, or press the help button on the CMake GUI.
```

进入 build 目录下，执行命令

cmake .. -DLLVM_DIR=/path/to/your/llvm/install/lib/cmake/llvm/:

```
root@click:/home/click/桌面/cminus_compiler-2022-fall# cd build/
root@click:/home/click/桌面/cminus_compiler-2022-fall/build# cmake .. -DLLVM_DIR=/path/to/your/llvm/install/lib/cmake/llvm/
-- 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/click/桌面/cminus_compiler-2022-fall/build
```

然后执行 make -j:



make install:

```
root@click:/home/click/桌面/cminus_compiler-2022-fall/build# make install
[  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
Install the project...
-- Install configuration: "Debug"
-- Installing: /usr/local/lib/libcminus_io.a
-- Installing: /usr/local/bin/cminusfc
root@click:/home/click/桌面/cminus_compiler-2022-fall/build# cd ..
```

2.运行验证

```
root@click:/home/click/桌面/cminus_compiler-2022-fall/build# cd ..
root@click:/home/click/桌面/cminus_compiler-2022-fall# cd tests/lab4
root@click:/home/click/桌面/cminus_compiler-2022-fall/tests/lab4# ./lab4_test.py
===========TEST START===========
Case 01:        Success
Case 02:        Success
Case 03:        Success
Case 04:        Success
Case 05:        Success
Case 06:        Success
Case 07:        Success
Case 08:        Success
Case 09:        Success
Case 10:        Success
Case 11:        Success
Case 12:        Success
============TEST END============
root@click:/home/click/桌面/cminus_compiler-2022-fall/tests/lab4#
```

通过观察脚本运行结果可以知道，全是 success 表示实验结果正确。

实验总结

这次实验整体来说还是有点点小复杂的，因为其涉及到的头文件很多，很难找到正确的切入点，且过程中还经常出现一些小问题不好找原因。需要认真仔细的看 cminus-f 的语法与语义以及各个函数之间的联系才能比较顺利的完成这个实验。通过这个实验，我对 cminusf 语法语义和访问者模式有了更深入的理解，根据 node 对象具体类型的不同，编译器在对语法树自顶向下分析时会选择对应的 visit 函数进行调用，从而较为高效的完成任务。以前写 C++代码时，只知道不符合语法会报错，但并不知道计算机内部是如何实现这个过程的，完成这个实验的过程中我知道了答案。此外，本次实现需要阅读大量的.h 头文件，所以我阅读理解 C 代码的能力有所提升，且养成了仔细严谨阅读文档、多思考的好习惯。