LAB4 实验报告

学号 201908030406 姓名 陈叹

一、实验要求：

主要工作
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

二、实验难点

都是难点。
主要在于代码量巨大，并且对于前三个实现必须有一个清晰的认识。
需要数量掌握并理解运用 lab1~3 的相关知识，综合实现 lab4 的代码，对每个产生式进行分析，并得出代码实现。

三、实验设计

1.概述
首先阅读相关资料
cminus-f 文法一共有 31 项，而这里我们只需要实现 15 项。另外的项，比如 addop 等已经包含在更高层次的非终结符号的代码中了。
我们本次实现的核心就是根据每个产生式，来编写其对应的代码，基于 LightIR 结构，输入一个 cminus-f 语言编写的文件，利用 lab1 的 flex 进行词法分析，用 lab2 的 bison 对 flex 词法分析的结果进行语法分析，生成语法树，然后传入到我们当前的代码中，进行树的遍历，在遍历过程中，借助 lab3 的知识，利用 builder 进行辅助，构造出与 cminus-f 对应的 IR 语言，从而能够使 clang 能够基于我们的输出文件直接编译运行。

2.下面，我们来考虑具体的遍历过程。
这也需要一些前置知识，主要是关于环境和一些有用的函数等。

重要 API 如下图：

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();
```

这几个函数在后面会反复用到，在这里就不过多解释了，上面也有注释。
他们都被定义在 Scope 中。

另外，在 cminusf_builder.hpp 中，还有一个主要的类，是用于构建 IR 的主要的类，如下图：

```
54 class CminusfBuilder: public ASTVisitor {
55 public:
56     CminusfBuilder() {
57         module = std::unique_ptr<Module>(new Module("Cminus code"));
58         builder = new IRBuilder(nullptr, module.get());
59         auto TyVoid = Type::get_void_type(module.get());
60         auto TyInt32 = Type::get_int32_type(module.get());
61         auto TyFloat = Type::get_float_type(module.get());
62 
63         auto input_type = FunctionType::get(TyInt32, {});
64         auto input_fun =
65             Function::create(
66                     input_type,
67                     "input",
68                     module.get());
69 
70         std::vector<Type *> output_params;
71         output_params.push_back(TyInt32);
72         auto output_type = FunctionType::get(TyVoid, output_params);
73         auto output_fun =
74             Function::create(
75                     output_type,
76                     "output",
77                     module.get());
78 
```

其构造函数很长，就不截完了，而构造函数的作用就是用来定义另外一些固定函数，如 input，output 等，也不过多阐述。
其内部还有本次实验中我们需要实现的所有函数，这些函数都是在访问到某个特定节点时，需要做一些特定操作，来告诉 builder 生成特定的 IR 指令。
具体如下图：

```
105 private:
106     virtual void visit(ASTProgram &) override final;
107     virtual void visit(ASTNum &) override final;
108     virtual void visit(ASTVarDeclaration &) override final;
109     virtual void visit(ASTFunDeclaration &) override final;
110     virtual void visit(ASTParam &) override final;
111     virtual void visit(ASTCompoundStmt &) override final;
112     virtual void visit(ASTExpressionStmt &) override final;
113     virtual void visit(ASTSelectionStmt &) override final;
114     virtual void visit(ASTIterationStmt &) override final;
115     virtual void visit(ASTReturnStmt &) override final;
116     virtual void visit(ASTAssignExpression &) override final;
117     virtual void visit(ASTSimpleExpression &) override final;
118     virtual void visit(ASTAdditiveExpression &) override final
119     virtual void visit(ASTVar &) override final;
120     virtual void visit(ASTTerm &) override final;
121     virtual void visit(ASTCall &) override final;
122
123     IRBuilder *builder;
124     Scope scope;
125     std::unique_ptr<Module> module;
126 };
```

在实验手册中，还有一类 accept 函数，是用于访问者模式的树遍历的，大部分节点是简单的直接访问需要访问的部分，即调用 visit 函数，让当前的 builder 访问它本身，如下图。

```
void ASTProgram::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTNum::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTVarDeclaration::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTFunDeclaration::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTParam::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTCompoundStmt::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTExpressionStmt::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTSelectionStmt::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTIterationStmt::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTReturnStmt::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTAssignExpression::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTSimpleExpression::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTAdditiveExpression::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTVar::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTTerm::accept(ASTVisitor &visitor) { visitor.visit(*this); }
void ASTCall::accept(ASTVisitor &visitor) { visitor.visit(*this); }

void ASTFactor::accept(ASTVisitor &visitor) {
    auto expr =
        dynamic_cast<ASTExpression *>(this);
    if (expr) {
        expr->accept(visitor);
        return;
```

不过，还有一些类型的 accepted 函数是有操作需要完成的。

后面的代码设计会借鉴这部分内容。

再后面，实例程序中还有 visit 函数的实现实例，也会作参考。
还有 cminus-f 的语法和语义，需要熟练掌握，才可以编写出代码。
还有 LightIR 的基本架构，以及各个类的调用，都需要了解。
总之，这就是一个大型综合项目。

3.
下面开始考虑代码实现这些函数。
(1)Program

```
28 void CminusfBuilder::visit(ASTProgram &node) {
29     //Program -> declaration-list
30     for (auto decl: node.declarations){
31         decl->accept(*this);
32     }
33 }
```

1. \program → declaration-list\
2. \declaration-list → declaration-list declaration | declaration\

该函数将语法 1 和 2 合并，对当前的每个 declaration 进行简单的顺序遍历即可，并不需要操作。
(2)num

```
35 void CminusfBuilder::visit(ASTNum &node) {
36     if (node.type==CminusType::TYPE_INT){
37         ret=Const_int(node.i_val);
38     }
39     else if (node.type==CminusType::TYPE_FLOAT){
40         ret=Const_fp(node.f_val);
41     }
42 }
```

27. \integer → **INTEGER**\
28. \float → **FLOATPOINT**\

该函数对应 27,28 两句。
该节点是终结符号，不过有两种可能，int 和 float，分别返回即可。
(3)vardeclaration
变量的声明。
对应语句：

4. \var-declaration → type-specifier **ID** ; | type-specifier **ID** [**INTEGER**] ;\

需要分情况讨论。
如果在不在全局变量中，那么只需要声明，不需要分配空间(空间在运行的时候在栈上分配)。
而此时这句话是具体的语句了，需要利用 builder 来添加一句话，这句话具体由当前节点的参数确定。
还需要注意当前节点可能是单个变量，也有可能是数组，这个也需要分情况讨论。

具体代码如下：

```
void CminusfBuilder::visit(ASTVarDeclaration &node) {
    Type* int32type=Type::get_int32_type(module.get());
    Type* floattype=Type::get_float_type(module.get());
    if (!CminusfBuilder::scope.in_global()){
        if (node.num){ //local array declaration
            ArrayType* arraytype;
            if (node.type==CminusType::TYPE_INT)
                arraytype=ArrayType::get(int32type,node.num->i_val);
            else
                arraytype=ArrayType::get(floattype,node.num->f_val);
            auto localarray=builder->create_alloca(arraytype);
            scope.push(node.id,localarray);
        }
        else{
            if (node.type==CminusType::TYPE_INT){
                auto localvar=builder->create_alloca(int32type);
                scope.push(node.id,localvar);
            }
            else{
                auto localvar=builder->create_alloca(floattype);
                scope.push(node.id,localvar);
            }
        }
    }
```

而如果在全局变量中，不仅需要声明，还需要具体的开辟空间。

根据 lab3 的知识，利用 ConstantZero 和 GlobalVariable 合作进行空间的申请，具体代码：

```
    else{
        if (node.num){
            ArrayType* arraytype;
            auto initializer=ConstantZero::get(int32type,module.get());
            if (node.type==CminusType::TYPE_INT){
                arraytype=ArrayType::get(int32type,node.num->i_val);
            }
            else{
                arraytype=ArrayType::get(floattype,node.num->i_val);
                initializer=ConstantZero::get(floattype,module.get());
            }
            auto glarray=GlobalVariable::create(node.id,module.get(),arraytype,false,initializer);
            scope.push(node.id,glarray);
        }
        else{
            if (node.type==CminusType::TYPE_INT){
                auto initializer=ConstantZero::get(int32type,module.get());
                auto globalvar=GlobalVariable::create(node.id,module.get(),int32type,false,initializer)
                scope.push(node.id,globalvar);
            }
            else{
                auto initializer=ConstantZero::get(floattype,module.get());
                auto globalvar=GlobalVariable::create(node.id,module.get(),floattype,false,initializer)
                scope.push(node.id,globalvar);
            }
        }
    }
```

这样，变量声明就写好了。

(4)fundeclaration

函数声明。

对应语句：

6. \fun-declaration → type-specifier **ID** (params) compound-stmt\

这个部分就更麻烦了。由于是函数，变量等的作用域会变化。因此首先，需要调用 scope 中的 enter 函数，进入新的作用域。

然后，考虑参数列表，可能没有参数，也可能有。

若有，则需要确定参数的类型，用于声明函数。
每个参数有 int 和 float，还有是否 array，所以一共 8 种，分类讨论。
代码：

```
97 void CminusfBuilder::visit(ASTFunDeclaration &node) {
98     //函数的作用域从参数声明开始.
99     scope.enter();
00     func_with_array=false;
01     //fundeclaration -> type-specifier id (params) {}
02     Type* int32type=Type::get_int32_type(module.get());
03     Type* floattype=Type::get_float_type(module.get());
04     Type* voidtype =Type::get_void_type (module.get());
05     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
06     Type* floatptrtype=Type::get_float_ptr_type(module.get());
07     std::vector<Type *> argstype;
08     if (node.params.size()>0){
09         for (auto p:node.params){
10             if (p->isarray){
11                 if (p->type==CminusType::TYPE_INT)
12                     argstype.push_back(int32ptrtype);
13                 else
14                     argstype.push_back(floatptrtype);
15             }
16             else{
17                 if (p->type==CminusType::TYPE_INT)
18                     argstype.push_back(int32type);
19                 else
20                     argstype.push_back(floattype);
21             }
22         }
23     }
```

然后，返回值也需要确定类型。
最后，调用 create 函数，创建函数块。

```
125     if (node.type==CminusType::TYPE_INT){
126         functype=int32type;
127     }
128     else if (node.type==CminusType::TYPE_FLOAT){
129         functype=floattype;
130     }
131     else{
132         functype=voidtype;
133     }
134     auto func=Function::create(FunctionType::get(functype,argstype),node.id,module.get());
```

下一步，非常坑，需要将该函数块的定义加到 module 下面。
这个定义是在全局变量下面的，而当前由于上面 enter，进入了当前函数的作用域内。
因此，需要先退出，将块加入，然后再进入进行后续操作。
代码：

```
135     scope.exit();
136     scope.push(node.id,func);
137     scope.enter();
```

重新进入函数作用域之后，进行后续操作。
由 lab3 的知识，函数下面是若干个 basicblock。

显然，我们需要新建一个 basicblock，函数体内的语句(非终结符号)需要在 basicblock 内部定义。
代码如下图，插入块 bb。

```
138     auto bb=BasicBlock::create(module.get(),"",func);
139     currentfunc=func;
140     builder->set_insert_point(bb);
```

然后，params，即参数，的具体遍历还没有完成，之前只是将他们加入 module 下方。
现在，遍历每个 param，这样就可以得到每个 param 的具体值，然后将其值和名字，一并加入到当前块 bb 下方(当前已经在块 bb 内部，直接加入即可)。
代码：

```
141     //store parameters' value
142     for (auto param:node.params){
143         param->accept(*this);
144     }
145     std::vector<Value *>argsvalue;
146     for (auto arg=func->arg_begin();arg!=func->arg_end();arg++)
147         argsvalue.push_back(*arg);
148     if (node.params.size() && argsvalue.size()){
149         int i=0;
150         for (auto arg:node.params){
151             auto p=scope.find(arg->id);
152             builder->create_store(argsvalue[i++],p);
153         }
154     }
```

最后，需要遍历后面的函数体，最后退出，
这部分比较简单，代码：

```
155     if (node.compound_stmt){
156         node.compound_stmt->accept(*this);
157     }
158     scope.exit();
159 }
```

(4)param
对应语句：

7. \params → param-list | **void**\
8. \param-list → param-list , param | param\
9. \param → type-specifier **ID** | type-specifier **ID** []\

这是上面刚用到过的，单个函数参数节点的遍历。
主要的问题就在于判断它是什么数据类型，然后合理的设置值即可。
数据类型在上面 vardeclaration 中已经分析了，一共有 4 种类型。
对于某种类型，用 builder 声明，并 push 进当前作用域即可。
分类讨论即可：

```
161 void CminusfBuilder::visit(ASTParam &node) {
162     // param -> type-specifier id ; type-specifier id []
163     Type* int32type=Type::get_int32_type(module.get());
164     Type* floattype=Type::get_float_type(module.get());
165     Type* voidtype =Type::get_void_type (module.get());
166     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
167     Type* floatptrtype=Type::get_float_ptr_type(module.get());
168     if(node.isarray){ //array
169         if (node.type==CminusType::TYPE_INT){
170             auto param_ptr=builder->create_alloca(int32ptrtype);
171             scope.push(node.id,param_ptr);
172         }
173         else{
174             auto param_ptr=builder->create_alloca(floatptrtype);
175             scope.push(node.id,param_ptr);
176         }
177     }
178     else{
179         if(node.type==CminusType::TYPE_INT){
180             auto param=builder->create_alloca(int32type);
181             scope.push(node.id, param);
182         }
183         else if(node.type==CminusType::TYPE_FLOAT){
184             auto param=builder->create_alloca(floattype);
185             scope.push(node.id, param);
186         }
187 
188     }
189 }
```

(5) CompoundStmt

这部分非常简单

对应语句：

10. \compound-stmt → { local-declarations statement-list}\

11. \local-declarations → local-declarations var-declaration | empty\

12. \statement-list → statement-list statement | empty\

几乎什么都不用做，只要遍历子节点就行了。

显然，先遍历 local-declarations,再遍历 statement-list.

由于有'{}'，需要进入新的作用域(即使之前 fundeclaration 可能已经进入新的作用域了，因为{}语句可以单独使用的)

另外，每个 statemenet_list 相当于一个 basicblock，因此，需要插入块之后再运行。

```
191 void CminusfBuilder::visit(ASTCompoundStmt &node) {
192     //compoundstmt->localdeclaration|statementlist
193     scope.enter();
194     for (auto decl:node.local_declarations)
195         decl->accept(*this);
196     for (auto stmt:node.statement_list){
197         if (builder->get_insert_block()->get_terminator()==nullptr)
198             stmt->accept(*this);
199     }
200     scope.exit();
201 }
```

(6)expressionstmt

对应产生式

14. \expression-stmt → expression ; | ;\

其中非终结符号只有 expression，遍历该节点即可：

```
203 void CminusfBuilder::visit(ASTExpressionStmt &node) {
204     if (node.expression) node.expression->accept(*this);
205 }
```

(7)selectionstmt

对应产生式

15. $$\text{selection-stmt} \rightarrow \textbf{if}\ (\ \text{expression}\ )\ \text{statement} \mid \textbf{if}\ (\ \text{expression}\ )\ \text{statement}\ \textbf{else}\ \text{statement}$$

该语句也十分麻烦。

首先，一定需要遍历 expression，这个在另外的函数中，我们直接调用即可。

并且 expression 会设置返回值，这个在后面会对应到，返回值设置在 ret 中。

首先，为 ret 返回值开辟语句，利用 builder 存储 ret 的值。

然后，需要将 ret 的值和 0 进行比较，为后面的 if 语句做准备。

然后将比较结果也开辟语句存储起来。

代码：

```
207 void CminusfBuilder::visit(ASTSelectionStmt &node) {
208     Type* int32type=Type::get_int32_type(module.get());
209     Type* floattype=Type::get_float_type(module.get());
210     node.expression->accept(*this);
211     if (ret->get_type()->is_pointer_type()) ret=builder->create_load(ret);
212     else if (ret->get_type()->is_array_type()) ret=builder->create_gep(ret,{Const_int(0),Const_int(0)});
213     auto icmp=ret;
214     if (icmp->get_type()==int32type) icmp=builder->create_icmp_gt(icmp,Const_int(0));
215     else if (icmp->get_type()==floattype) icmp=builder->create_fcmp_gt(icmp,Const_fp(0));
```

上图中，返回值存储在 ret 中，比较结果存储在 icmp 中。

若 icmp=0，则不成立；若 icmp=1，则成立。不过我们不是具体的在执行 IR 语句，而是在构建 IR 语句。

下一步，回忆 lab3 中的 if 语句，我们需要利用跳转指令，以及很多的 basicblock，来实现一句 if 语句，在这里也是如此。

另外，还需要分这句 if 语句中有无 else 块。

先考虑有 else 块的情形。

新建 trueBB，falseBB 和 exitBB。

然后，trueBB 和 falseBB 利用 builder 的调用 sreate_cond_br()函数，和 icmp 联系在一起，形成一个跳转指令。

然后，将后面的两个块，false 和 exit，从当前的 BB 链表中删去。

因为当前函数之后直接的块就是 trueBB，其他两个块是基于块末尾的跳转指令链接在一起的，此时只是统一创建，链接在后面会考虑。

代码：

```
216     if (node.else_statement!=nullptr){
217         auto trueBB=BasicBlock::create(module.get(),"",currentfunc);
218         auto falseBB=BasicBlock::create(module.get(),"",currentfunc);
219         auto exitBB=BasicBlock::create(module.get(),"",currentfunc);
220         builder->create_cond_br(icmp,trueBB,falseBB);
221         falseBB->erase_from_parent();
222         exitBB->erase from parent();
```

这样，BB 和原函数的关系就建立好了。

考虑 trueBB，首先设置 builder 当前插入指令的位置为 trueBB，然后，需要遍历表达式中跟在 if 后面的 statement，即 true 条件下的 statement。

代码：

```
223         builder->set_insert_point(trueBB);
224         node.if_statement->accept(*this);
```

然后考虑终止，如果块的最后不是终止指令(一般不是)，那么需要加一条跳转到 exit 的语句，当然，也有可能 statement 中有跳转语句，直接跳出了 if。

代码：

```
225         bool isreturned=true;
226         if (builder->get_insert_block()->get_terminator()==nullptr){
227             builder->create_br(exitBB);
228             isreturned=false;
229         }
```

这样，trueBB 的遍历和根其他 BB 的链接关系就搞好了。其内部的表达式由孩子取完成。

然后是 flaseBB，跟 trueBB 类似即可：

```
230         currentfunc->add_basic_block(falseBB);
231         builder->set_insert_point(falseBB);
232         node.else_statement->accept(*this);
233         if (builder->get_insert_block()->get_terminator()==nullptr){
234             builder->create_br(exitBB);
235             isreturned=false;
236         }
```

最后，如果有 exit，则需要新建 exitBB，然后后面的在嘛在 exitBB 中实现，这里只需要创建即可。

```
237         if (!isreturned){
238             currentfunc->add_basic_block(exitBB);
239             builder->set_insert_point(exitBB);
240         }
```

另外一边，如果没有 else 语句，那么就非常简单了，不多描述，代码：

```
242     else{
243         auto trueBB=BasicBlock::create(module.get(),"",currentfunc);
244         auto falseBB=BasicBlock::create(module.get(),"",currentfunc);
245         builder->create_cond_br(icmp,trueBB,falseBB);
246         falseBB->erase_from_parent();
247         builder->set_insert_point(trueBB);
248         node.if_statement->accept(*this);
249         currentfunc->add_basic_block(falseBB);
250         if (builder->get_insert_block()->get_terminator()==nullptr)
251             builder->create_br(falseBB);
252         builder->set_insert_point(falseBB);
253     }
254 }
```

(8) IterationStmt

循环语句。

对应产生式：

16. \iteration-stmt → **while** (expression) statement\

有了上面的经验，还是比较好写的。

首先，创建相关的块，然后，设置跳转指令，然后考虑循环条件为真的 trueBB，遍历孩子，

然后再创建 false 的块，就是退出循环语句的块，后面的代码在该块中实现。

代码：

```
256 void CminusfBuilder::visit(ASTIterationStmt &node) {
257     //while(expression)statement
258     Type* int32type=Type::get_int32_type(module.get());
259     Type* floattype=Type::get_float_type(module.get());
260     auto whileBB=BasicBlock::create(module.get(),"",currentfunc);
261     auto trueBB=BasicBlock::create(module.get(),"",currentfunc);
262     auto falseBB=BasicBlock::create(module.get(),"",currentfunc);
263     builder->create_br(whileBB);
264     //whileBB
265     builder->set_insert_point(whileBB);
266     node.expression->accept(*this);
267     if (ret->get_type()->is_pointer_type()) ret=builder->create_load(ret);
268     else if (ret->get_type()->is_array_type()) ret=builder->create_gep(ret,{Const_int(0),Const_int(0)});
269     auto icmp=ret;
270     if (icmp->get_type()==int32type) icmp=builder->create_icmp_gt(icmp,Const_int(0));
271     else if (icmp->get_type()==floattype) icmp=builder->create_fcmp_gt(icmp,Const_fp(0));
272     builder->create_cond_br(icmp,trueBB,falseBB);
273     //trueBB
274     builder->set_insert_point(trueBB);
275     falseBB->erase_from_parent();
276     node.statement->accept(*this);
277     builder->create_br(whileBB);
278     //falseBB
279     currentfunc->add_basic_block(falseBB);
280     builder->set_insert_point(falseBB);
281 }
```

实现思路和 if 语句很类似，注意 trueBB 的结束跳转是到 whileBB，构成循环即可。

(9) ReturnStmt

对应产生式：

17. \return-stmt → **return ;** | **return** expression ;\

显然，需要分有无返回值来考虑。

若没有返回值，直接创建一个 void_ret 语句即可，利用 builder 的函数，代码：

```
310     else{
311         builder->create_void_ret();
312     }
```

接下来考虑有返回值的情况，返回值可以是一个 expression。

这里涉及两个数据类型，一个是返回的数据类型，还有一个是 expression 的数据类型，如果不一致，需要考虑强制类型转换，这是最麻烦的点。

首先，将两个数据类型存储下了，分别在 returntype 和 ret 中。

代码：

```
288     auto returntype=currentfunc->get_return_type();  //currentfunc 的返回值类型
289     if(node.expression!=nullptr){   //返回类型不为空
290         node.expression->accept(*this);
291         if (ret->get_type()->is_pointer_type()){
292             if (ret->get_type()->get_pointer_element_type()==int32type)
293                 ret=builder->create_load(int32type,ret);
294             else
295                 ret=builder->create_load(floattype,ret);
296         }
```

然后，考虑数据类型不一致的情形，需要进行强制类型转换。

builder 中有提供转换的函数，当然，也就是有相应的 IR 代码。

转换之后，再创建返回语句即可。

代码：

```
297         if(returntype==int32type&&ret->get_type()==floattype){ // 函数返回值类型为int但ret类型为float
298             auto tmp=builder->create_fptosi(ret, int32type);
299             builder->create_ret(tmp);
300         }
301         // 函数返回类型为float但ret类型为int
302         else if(returntype==floattype&&ret->get_type()==int32type){
303             auto tmp=builder->create_sitofp(ret, floattype);
304             builder->create_ret(tmp);
305         }
306         else{
307             builder->create_ret(ret);
308         }
```

(10) Var

对应产生式：

19. \var → ID | ID [expression]\

该函数并不产生语句，只是设置返回值。

一出现 expression，就又非常麻烦，可能需要类型转换了。

首先是没有 expression 的情形，非常简单：

```
316 void CminusfBuilder::visit(ASTVar &node) {
317     Type* int32type=Type::get_int32_type(module.get());
318     Type* floattype=Type::get_float_type(module.get());
319     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
320     Type* floatptrtype=Type::get_float_ptr_type(module.get());
321     auto id=scope.find(node.id);
322     if (node.expression==nullptr){
323         ret=id;
324     }
```

如果有 expression，显然首先遍历 expression。

原表达式中 expression 的结果是作为数组下标的，显然应当为 int 类型。

如果是浮点型，则转换。

代码：

```
325     else{
326         node.expression->accept(*this);
327         auto num=ret;
328         if (num->get_type()==int32type) num=num;
329         else if (num->get_type()==floattype) num=builder->create_fptosi(num,int32type);
```

然后，需要设置返回值为数组对应的元素。

```
358         auto* idpload=builder->create_load(id);
359         if (idpload->get_type()->is_array_type()){
360             auto idgep=builder->create_gep(id,{Const_int(0),num});
361             ret=idgep;
362         }
363         else{
364             auto idgep=builder->create_gep(idpload,{num});
365             ret=idgep;
366         }
```

这样即可。

(11) AssignExpression

赋值表达式，对应产生式：

18. \expression → var = expression | simple-expression\ 的前半部分。

和上面类似，主要是数据类型转换。

代码：

```
343 void CminusfBuilder::visit(ASTAssignExpression &node) {
344     //expression 的返回值都放在ret中
345     //变量已经在声明的时候分配了内存空间
346     //var = expression
347     Type* int32type=Type::get_int32_type(module.get());
348     Type* floattype=Type::get_float_type(module.get());
349     node.var->accept(*this);
350     auto assign_var=ret;
351     node.expression->accept(*this);
352     //如果赋值语句两边类型不同，进行类型转换
353     // var is of int type
354     if (ret->get_type()->is_pointer_type()){
355         ret=builder->create_load(ret);
356     }
357     if(assign_var->get_type()->get_pointer_element_type()==ret->get_type())  // expression's type == var's type
358         builder->create_store(ret, assign_var);
359     else if (ret->get_type()==floattype){ //expression is of float type  强制转换类型
360         ret=builder->create_fptosi(ret, int32type);
361         builder->create_store(ret, assign_var);
362     }
363     else{
364         ret=builder->create_sitofp(ret,floattype);
365         builder->create_store(ret, assign_var);
366     }
367 }
```

(12) SimpleExpression

对应产生式

18. \expression → var = expression | simple-expression\ 的后半部分

以及

20. \simple-expression → additive-expression relop additive-expression | additive-expression\

21. \relop → <= | < | > | >= | == | !=\

这里我们需要返回表达式的值。加法表达式不用我们实现，但是不等关系还是需要实现的。首先，分为有无不等关系，如果没有，直接遍历孩子表达式即可。

```
369 void CminusfBuilder::visit(ASTSimpleExpression &node) {
370     //addexpr-l relop addexpr-r
371     Type* int32type=Type::get_int32_type(module.get());
372     Type* floattype=Type::get_float_type(module.get());
373     Type* int1type=Type::get_int1_type(module.get());
374     if (node.additive_expression_r==nullptr){
375         node.additive_expression_l->accept(*this);
376     }
```

然后，考虑有不等关系，则先遍历两个孩子，将其值存储下来：

```
377     else{
378         node.additive_expression_l->accept(*this);
379         Value* lval;
380         if (ret->get_type()==int32type || ret->get_type()==floattype || ret->get_type()==int1type){
381             lval=ret;
382         }
383         node.additive_expression_r->accept(*this);
384         Value* rval;
385         if (ret->get_type()==int32type || ret->get_type()==floattype || ret->get_type()==int1type){
386             rval=ret;
387         }
```

然后，进行各种类型转换，使得两边类型一致。

```
388     //二元运算两边类型不一样，强制类型转换  int 转 float
389         if (lval->get_type()==int1type) lval=builder->create_zext(lval,int32type);
390         if (rval->get_type()==int1type) rval=builder->create_zext(rval,int32type);
391         if(lval->get_type()==floattype&&rval->get_type()==int32type)
392             builder->create_sitofp(rval, floattype);
393         else if(rval->get_type()==floattype&&lval->get_type()==int32type)
394         builder->create_sitofp(lval, floattype);
```

然后，基于不等符号，以及两边的数据类型，进行判断：

```
396     //判断ret的值
397     /* <= OP_LE //< OP_LT,// > OP_GT,// >= OP_GE, // == OP_EQ, // !=  OP_NEQ*/
398         if(lval->get_type()==int32type){// lval rval  int32type  icmp
399             switch (node.op) {
400                 case OP_LE:
401                     ret=builder->create_icmp_le(lval, rval);
402                     break;
403                 case OP_LT:
404                     ret=builder->create_icmp_lt(lval, rval);
405                     break;
406                 case OP_GT:
407                     ret=builder->create_icmp_gt(lval, rval);
408                     break;
409                 case OP_GE:
410                     ret=builder->create_icmp_ge(lval, rval);
411                     break;
412                 case OP_EQ:
413                     ret=builder->create_icmp_eq(lval, rval);
414                     break;
415                 default:
416                     ret=builder->create_icmp_ne(lval, rval);
417             }
418         }
419         else { //lval rval  floattype  fcmp
420             switch (node.op) {
421                 case OP_LE:
422                     ret=builder->create_fcmp_le(lval, rval);
423                     break;
424                 case OP_LT:
425                     ret=builder->create_fcmp_lt(lval, rval);
426                     break;
427                 case OP_GT:
428                     ret=builder->create_fcmp_gt(lval, rval);
429                     break;
430                 case OP_GE:
431                     ret=builder->create_fcmp_ge(lval, rval);
432                     break;
433                 case OP_EQ:
434                     ret=builder->create_fcmp_eq(lval, rval);
435                     break;
436                 default:
437                     ret=builder->create_fcmp_ne(lval, rval);
438                     break;
439             }
440         }
```

这样就行了。

(13) AdditiveExpression

加法表达式，因为加法优先级低，因此，先按+-来分表达式，每个子表达式内部还可能有

*/

对应产生式：

22. \additive-expression → additive-expression addop term | term\

23. \addop → + | -\

该函数非常类似于前面的 SimpleExpression 函数，先判断是否有+-号，然后如果有，则先遍历左右，得到值，然后数据类型转换，最后根据+-号来进行合理的计算即可。

代码：

下图这部分在进行判断有无+-和遍历子树

```
445 void CminusfBuilder::visit(ASTAdditiveExpression &node) {
446     Type* int32type=Type::get_int32_type(module.get());
447     Type* floattype=Type::get_float_type(module.get());
448     Type* int1type=Type::get_int1_type(module.get());
449     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
450     Type* floatptrtype=Type::get_float_ptr_type(module.get());
451     if (node.additive_expression==nullptr){
452         //additive_expression -> term
453         node.term->accept(*this);
454     }
455     else{
456         //additive_expression -> additive_expression addop term
457         node.additive_expression->accept(*this);
458         Value* leftvalue;
459         leftvalue=ret;
460         node.term->accept(*this);
461         Value* rightvalue;
462         rightvalue=ret;
```

下面这部分在进行类型转换和根据+-进行运算。

```
463         if (leftvalue->get_type()==int1type) leftvalue=builder->create_zext(leftvalue,int32type);
464         if (rightvalue->get_type()==int1type) rightvalue=builder->create_zext(rightvalue,int32type);
465         if (node.op==OP_PLUS){
466             if (leftvalue->get_type()==int32type && rightvalue->get_type()==int32type)
467                 ret=builder->create_iadd(leftvalue,rightvalue);
468             else{ //整型和浮点数运算，整型变为浮点类型
469                 if(leftvalue->get_type()==int32type)   leftvalue=builder->create_sitofp(leftvalue, floattype);
470                 else rightvalue=builder->create_sitofp(rightvalue, floattype);
471                 ret=builder->create_fadd(leftvalue,rightvalue);
472             }
473         }
474         else{
475             if (leftvalue->get_type()==int32type && rightvalue->get_type()==int32type)
476                 ret=builder->create_isub(leftvalue,rightvalue);
477             else{//整型和浮点数运算，整型变为浮点类型
478                 if(leftvalue->get_type()==int32type)
479                     leftvalue=builder->create_sitofp(leftvalue, floattype);
480                 else rightvalue=builder->create_sitofp(rightvalue, floattype);
481                 ret=builder->create_fsub(leftvalue,rightvalue);
482             }
483         }
484     }
```

(14)term

跟上面也是类似的操作，描述略。

对应产生式

24. \term → term mulop factor | factor\

25. \mulop → * | /\

代码：

```
487 void CminusfBuilder::visit(ASTTerm &node) {
488     Type* int32type=Type::get_int32_type(module.get());
489     Type* floattype=Type::get_float_type(module.get());
490     Type* int1type=Type::get_int1_type(module.get());
491     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
492     Type* floatptrtype=Type::get_float_ptr_type(module.get());
493     if (node.term==nullptr){
494         //additive_expression -> term
495         node.factor->accept(*this);
496     }
497     else{
498         //additive_expression -> additive_expression addop term
499         node.term->accept(*this);
500         Value* leftvalue;
501         leftvalue=ret;
502         node.factor->accept(*this);
503         Value* rightvalue;
504         rightvalue=ret;
```

```
506         if (leftvalue->get_type()==int1type) leftvalue=builder->create_zext(leftvalue,int32type);
507         if (rightvalue->get_type()==int1type) leftvalue=builder->create_zext(leftvalue,int32type);
508         if (node.op==OP_MUL){
509             if (leftvalue->get_type()==int32type && rightvalue->get_type()==int32type)
510                 ret=builder->create_imul(leftvalue,rightvalue);
511             else{  //将其中一个整型变为浮点类型
512                 if(leftvalue->get_type()==int32type)   leftvalue=builder->create_sitofp(leftvalue, floattype);
513                 else rightvalue=builder->create_sitofp(rightvalue, floattype);
514                 ret=builder->create_fmul(leftvalue,rightvalue);
515             }
516         }
517         else{
518             if (leftvalue->get_type()==int32type && rightvalue->get_type()==int32type)
519                 ret=builder->create_isdiv(leftvalue,rightvalue);
520             else{  //将其中一个整型变为浮点类型
521                 if(leftvalue->get_type()==int32type)   leftvalue=builder->create_sitofp(leftvalue, floattype);
522                 else rightvalue=builder->create_sitofp(rightvalue, floattype);
523                 ret=builder->create_fdiv(leftvalue,rightvalue);
524             }
525         }
526     }
527 }
```

(15)call

对应产生式：

29. \call → **ID** (args)\

30. \args → arg-list | empty\

31. \arg-list → arg-list , expression | expression\

该函数的主要难点就在于 args，参数列表。

它是一个 vector，其中的类型为 Value*。

而其值应当由 expression 传递过来。

因此，遍历 args，对每个 expression 进行访问：

```
529 void CminusfBuilder::visit(ASTCall &node) {
530     Value* callfunc=scope.find(node.id);
531     Type* int32type=Type::get_int32_type(module.get());
532     Type* floattype=Type::get_float_type(module.get());
533     Type* int32ptrtype=Type::get_int32_ptr_type(module.get());
534     Type* floatptrtype=Type::get_float_ptr_type(module.get());
535     Type* calltype=callfunc->get_type();
536     FunctionType* argstype=(FunctionType*)calltype;
537     std::vector<Value *> callargs;
538     if (node.args.size()){
539         int i=0;
540         for (auto args:node.args){
541             args->accept(*this);
```

考虑每个 expression 的返回值，可能是 int，float。

不过需要考虑其参数列表中本身的类型，如果是 float，而返回值是 int，则显然需要转换。

代码：

```
542             if (ret->get_type()==int32type){
543                 //return an integer type
544                 if (argstype->get_param_type(i)==floattype)
545                     ret=builder->create_sitofp(ret,floattype);
546                 callargs.push_back(ret);
547             }
548             else if (ret->get_type()==floattype){
549                 if (argstype->get_param_type(i)==int32type)
550                     ret=builder->create_sitofp(ret,int32type);
551                 //return an floattype
552                 callargs.push_back(ret);
553             }
```

这样参数列表就已经搞好了。

最后，调用 builder 的构造 call 语句，参数为 callfunc 和 callargs。

```
580     ret=builder->create_call(callfunc,callargs);
```

这样，所有代码都实现了。

4.debug 之后，运行：

```
a@ac:~/byyl/cminus_compiler-2021-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:        Failed
Case 11:        Failed
Case 12:        Failed
============TEST END============
```

发现 expression 的返回值还有可能是指针。。。

就比如 Case 10：

```
1 void main(void) {
2     int a;
3     a = 1234;
4     output(a);
5     return;
6 }
```

第 4 行的参数为 a，那么由于其 load 存的都是指针，因此调用函数时的参数是指针，而不是简单的变量。

那么，所有 expression 的返回值，都有可能是指针，甚至是数组指针(二阶指针)

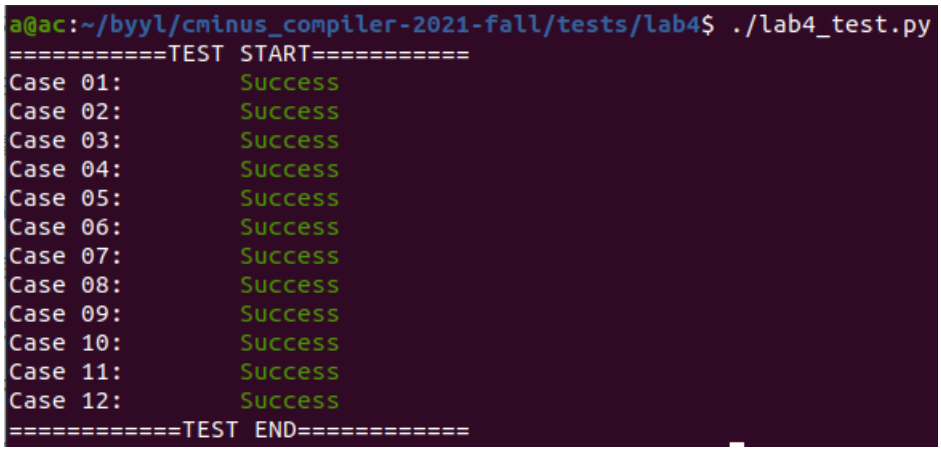
特别的，修改 call 函数如下图：

```
542             if (ret->get_type()==int32type){
543                 //return an integer type
544                 if (argstype->get_param_type(i)==floattype)
545                     ret=builder->create_sitofp(ret,floattype);
546                 callargs.push_back(ret);
547             }
548             else if (ret->get_type()==floattype){
549                 if (argstype->get_param_type(i)==int32type)
550                     ret=builder->create_sitofp(ret,int32type);
551                 //return an floattype
552                 callargs.push_back(ret);
553             }
554             else if (ret->get_type()->get_pointer_element_type()==int32type){
555                 //return a pointer to int
556                 ret=builder->create_load(ret);
557                 if (argstype->get_param_type(i)==floattype)
558                     ret=builder->create_sitofp(ret,floattype);
559                 callargs.push_back(ret);
560             }
561             else if (ret->get_type()->get_pointer_element_type()==floattype){
562                 //return a pointer to float
563                 ret=builder->create load(ret);
564               if (argstype->get_param_type(i)==int32type)
565                   ret=builder->create_fptosi(ret,int32type);
566               callargs.push_back(ret);
567           }
568           else if (ret->get_type()->get_pointer_element_type()->is_array_type()){
569               //return a array
570               callargs.push_back(builder->create_gep(ret,{Const_int(0),Const_int(0)}));
571           }
572           else if (!ret->get_type()->get_pointer_element_type()->is_array_type()){
573               //return a array's pointer
574               callargs.push_back(builder->create_load(ret));
575           }
576
577           i++;
578       }
579   }
580   ret=builder->create_call(callfunc,callargs);
581 }
```

除了本来的 int32，float，之外，还多了各种指针。

一般的，上面每个引用 expression 的位置都要进行配套修改。

改完之后，运行结果：



基础测试样例全部正确了。
而自己写的话，其实 case12 已经比较全面了。所以不另外写 case 测试。

四、实验结果验证

见上方

五、实验反馈

该实验还是设置的非常不错的，综合了前面三个 lab 的知识，来自己实现了一个编译器。从 cminus-f 生成了 IR 语言，而 IR 语言已经是一个非常底层的语言了，很类似汇编。
这个实验也解答了我 lab3 的异或——为什么我们需要去编写生成特定代码的 IR 语言文档，以及利用 LightIR 来生成生成特定文档 IR 语言的代码。
这样，这整个 4 个实验就串联了起来，形成了从 cminus-f 到 IR 语言的一个整体。
当然，我也意识到了 cminus-f 只是 C，乃至 C++的一个很小的子集，并且我们的 LightIR 框架是基于 C++编写的，而这无疑大大方便了我们的工作。而最初始的编译器，在没有其他编译器的时候，需要完成我们现在的工作，一步步的生成中间代码，再根据机器最终生成二进制代码，给机器执行。