

编译原理实验4

cminus-f语言（由AST生成IR）

计科210X 甘晴void 202108010XXX

实验要求

cminus-f编译器做的事情主要如下：

- 词法分析（Lab1完成）
- 语法分析（Lab2完成）
- 生成语法分析树（Lab2完成）
- 语法分析树->抽象语法树（即AST）【Lab4框架提供】
- 抽象语法树->中间代码（即IR）【★Lab4需要完成★】
- 中间代码->（优化）->目标代码【使用clang实现】

在本实验中，只要完成抽象语法树到中间表示IR的生成这一步。

具体来讲，在理解cminus-f 语法与语义的基础上，参考 `cminusf_builder.hpp` 文件以及 `include` 中的文件，补充完成 `cminusf_builder.cpp` 中的16个visit函数，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。

在自动产生 IR 的过程中，利用访问者模式自顶向下遍历抽象语法树的每一个结点，调用我们补充完成的函数对每一个抽象语法树的结点进行分析，如果程序是合法的则编译应该显示 Success，否则编译不通过显示 Fail。

具体步骤如下：

1. 阅读cminus-f 的语义规则，助教将按照语义实现程度进行评分
2. 阅读LightIR 核心类介绍
3. 阅读实验框架，理解如何使用框架以及注意事项
4. 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
5. 在 `report.md` 中解释该设计，遇到的困难和解决方案

待完成函数如下：

- `void CminusfBuilder::visit(ASTProgram &node) { }`
- `void CminusfBuilder::visit(ASTNum &node) { }`
- `void CminusfBuilder::visit(ASTVarDeclaration &node) { }`
- `void CminusfBuilder::visit(ASTFunDeclaration &node) { }`
- `void CminusfBuilder::visit(ASTParam &node) { }`
- `void CminusfBuilder::visit(ASTCompoundStmt &node) { }`
- `void CminusfBuilder::visit(ASTExpressionStmt &node) { }`
- `void CminusfBuilder::visit(ASTSelectionStmt &node) { }`
- `void CminusfBuilder::visit(ASTIterationStmt &node) { }`
- `void CminusfBuilder::visit(ASTReturnStmt &node) { }`
- `void CminusfBuilder::visit(ASTVar &node) { }`
- `void CminusfBuilder::visit(ASTAssignExpression &node) { }`
- `void CminusfBuilder::visit(ASTSimpleExpression &node) { }`
- `void CminusfBuilder::visit(ASTAdditiveExpression &node) { }`
- `void CminusfBuilder::visit(ASTTerm &node) { }`
- `void CminusfBuilder::visit(ASTCall &node) { }`

实验难点

1 全局变量的设置

调用的函数在 `cminusf_builder.hpp` 中被定义为了 `void` 类型的函数，故返回的结果不能通过函数本身实现，只能通过全局变量来进行传递。

此外用于 `Param` 的处理的参数指针也需要全局进行传递。

设置有效的全局变量也是实验的要点。

2 数组类型的判断与特殊处理

在 `void CminusfBuilder::visit(ASTVar &node)`，`void CminusfBuilder::visit(ASTParam &node)` 等函数中，对于数组类型都做了特殊的判断和处理，而且判断的方式有所不同。

如在前者中，使用 `node.num != nullptr` 进行判断，在后者中使用 `node.expression != nullptr` 进行判断。而相应的处理也是不同的。

3 作用域

一个变量，在放入表的时候，要考虑它的作用域，这显然是符合我们常识的。

如在实现 `void CminusfBuilder::visit(ASTFunDeclaration &node)` 函数中创建函数之后，要进入函数的作用域再创建基本块，并在处理完函数体内的语句之后退出基本块。

同样的还有在实现 `void CminusfBuilder::visit(ASTCompoundStmt &node)` 函数中处理复杂语句时，要先进入本基本块的作用域，并在结束之后退出这个作用域。这也是符合我们常识的，考虑下面这个场景，处理一个复杂（比如像 `if` 语句或者 `while` 语句后面的 `{}` 内部的语句块）语句块时，有可能我们会定义一些临时变量，那么显然这个临时变量的生命周期只在这个语句块内，如果不引入作用域，可能会导致溢出的后果。

对于作用域的使用也是实验的一个难点。

4 阅读资料，理解含义

完成本实验需要阅读的资料实在是太多了。

需要吃透的文件至少有

- `cminusf_builder.cpp`
- `cminusf_builder.hpp`
- `ast.cpp`
- `ast.hpp`

对于一个变量，它的原型或者说定义，往往要跳转好几次才能找到，这个理解代价还是很大的。

实验设计

1.作用域类：Scope

实验文档提到，实验框架中提示如下：

在 `include/cminusf_builder.hpp` 中，助教定义了一个用于存储作用域类 `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：

```

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

```

在实验时，我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，在进行名字查找时不需要顾虑是否需要特殊函数进行特殊操作。

2.全局变量与宏定义

补充了几个宏，所谓宏实际上是为了避免代码重复，所以如果一段代码经常用到，就可以把它封装成宏（或者函数）。如果没有实际上关系也不大，就是把这个代码多写几遍。

1.补充了获取类型的宏，分别获取 `int32` 型的类型和 `float` 的类型。

2.补充了整型，浮点型，指针类型判断的宏，能够判断该指针的类型是否是指定的。

如下：

```

1 #define Int32Type \
2     Type::get_int32_type(module.get()) /* 获取int32类型 */
3 #define FloatType \
4     Type::get_float_type(module.get()) /* 获取float类型 */
5
6 #define checkInt(num) \
7     num->get_type()->is_integer_type() /* 整型判断 */
8 #define checkFloat(num) \
9     num->get_type()->is_float_type() /* 浮点型判断 */
10 #define checkPointer(num) \
11     num->get_type()->is_pointer_type() /* 指针类型判断 */

```

全局变量ret用于节点返回值。arg用于传递参数。ifAssign表示访问Var节点时，应该返回值还是变量地址。

```
1 value *Res; /* 存储返回的结果 */
2 value *arg; /* 存储参数指针，用于Param的处理 */
3 bool need_as_address = false; /* 标志是返回值还是返回地址 */
```

3.使用LOG输出方便调试

参考实验文档提及的有关logging，在cminus_compiler-2023-fall\Documentations\common目录下阅读logging.md。

助教为大家设计了一个C++简单实用的分级日志工具。该工具将日志输出信息从低到高分成四种等级：DEBUG，INFO，WARNING，ERROR。通过设定环境变量LOGV的值，来选择输出哪些等级的日志。LOGV的取值是0~3,分别对应到上述的4种级别(0:DEBUG,1:INFO,2:WARNING,3:ERROR)。此外输出中还会包含打印该日志的代码所在位置。

【注意】如需使用LOGGING，需要在cminusf_builder.cpp最前面加上#include "logging.hpp"

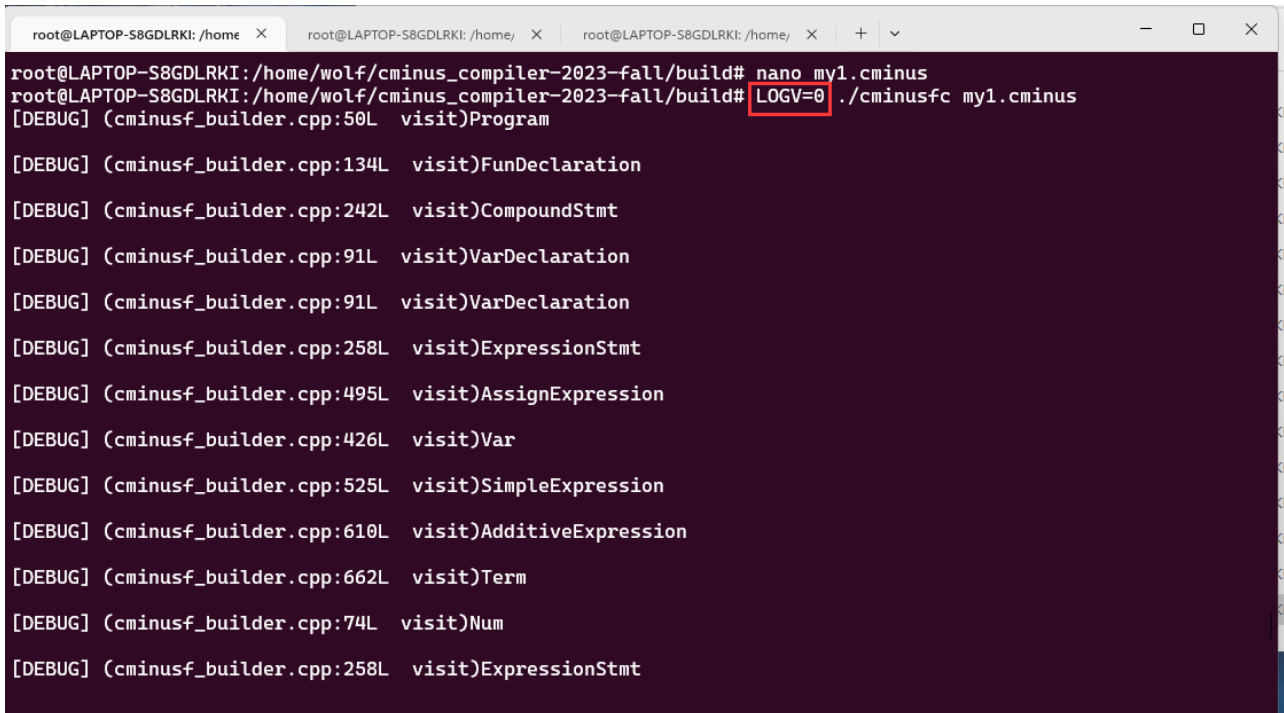
使用方法大致如下：

```
1 #include "logging.hpp"
2 // 引入头文件
3 int main(){
4     LOG(DEBUG) << "This is DEBUG log item.";
5     // 使用关键字LOG，括号中填入要输出的日志等级
6     // 紧接着就是<<以及日志的具体信息，就跟使用std::cout一样
7     LOG(INFO) << "This is INFO log item";
8     LOG(WARNING) << "This is WARNING log item";
9     LOG(ERROR) << "This is ERROR log item";
10    return 0;
11 }
```

具体调试程序时使用环境变量（Linux中采取写在最前面的方式），可以调用LOG输出日志信息

```
1 LOGV=0 ./test_logging
```

类似于这样



```
root@LAPTOP-S8GDLRKi: /home X root@LAPTOP-S8GDLRKi: /home/ X root@LAPTOP-S8GDLRKi: /home/ X + v
root@LAPTOP-S8GDLRKi: /home/wolf/cminus_compiler-2023-fall/build# nano my1.cminus
root@LAPTOP-S8GDLRKi: /home/wolf/cminus_compiler-2023-fall/build# LOGV=0 ./cminusfc my1.cminus
[DEBUG] (cminusf_builder.cpp:50L visit)Program
[DEBUG] (cminusf_builder.cpp:134L visit)FunDeclaration
[DEBUG] (cminusf_builder.cpp:242L visit)CompoundStmt
[DEBUG] (cminusf_builder.cpp:91L visit)VarDeclaration
[DEBUG] (cminusf_builder.cpp:91L visit)VarDeclaration
[DEBUG] (cminusf_builder.cpp:258L visit)ExpressionStmt
[DEBUG] (cminusf_builder.cpp:495L visit)AssignExpression
[DEBUG] (cminusf_builder.cpp:426L visit)Var
[DEBUG] (cminusf_builder.cpp:525L visit)SimpleExpression
[DEBUG] (cminusf_builder.cpp:610L visit)AdditiveExpression
[DEBUG] (cminusf_builder.cpp:662L visit)Term
[DEBUG] (cminusf_builder.cpp:74L visit)Num
[DEBUG] (cminusf_builder.cpp:258L visit)ExpressionStmt
```

如我们觉得程序已经没有问题了，不想看那么多的DEBUG信息，那么我们就可以设定环境变量 `LOGV=1`，选择只看级别大于等于1的日志信息。当然 `LOGV` 值越大，日志的信息将更加简略。如果没有设定 `LOGV` 的环境变量，将默认不输出任何信息。

我在所有函数的开头都添加了DEBUG级别的LOG，照理INFO，WARNING，ERROR等级别的LOG也是要写的，但由于时间仓促，就没有具体去一个一个分别写出来。其中有些用 `std::cout<<` 的方式进行输出了。

LOG功能还是值得看看的。平时使用c++，如果没有用DEV++等这些IDE而是直接手动配置gcc然后在终端输出信息的话，或者是Linux环境下的c/c++，在终端输出的编译信息，那么应该对终端弹出的ERROR和WARNING这些不陌生。或者使用Linux环境下的rust时，也能在终端看到ERROR这些信息，这些都是用类似于LOG的功能嵌在编译器内实现的。

4.具体实现Visit函数

<0> 总述

Visit函数的实现在 `cminusf_builder.cpp` 中完成。

如果使用VSCODE在 `cminus_compiler-2023-fall` 文件夹下打开 `cminusf_builder.cpp`，配置好相关的设置，单机结构或函数，会跳转到它定义的地方，这个功能十分方便。

在 `ast.hpp` 内定义了节点结构体的类型，包括这个节点结构体有哪些成员变量是我们可以直接调用的，同时需要弄清楚节点结构体成员变量的含义。同时每个该结构体还定义了一个纯虚函数 `virtual void accept(ASTVisitor &) override;` 或者 `virtual void accept(ASTVisitor &) override final;`，在其命名空间下的实现由 `ast.cpp` 对应命名空间的函数完成实现

在 `ast.cpp` 内给出了 `ast.hpp` 中纯虚函数的实现。

比对 `cminus_compiler-2023-fall\Documentations\common\cminusf.md` 中的 `cminus-f` 的语法和语义，包括补充内容，完成 `cminusf_builder.cpp`。

下面是每个具体函数的实现。

<1> void CminusfBuilder::visit(ASTProgram &node)

该节点结构定义如下。

```
1 struct ASTProgram : ASTNode {
2     virtual void accept(ASTVisitor &) override final;
3     std::vector<std::shared_ptr<ASTDeclaration>>
4         declarations;
5 };
```

`cminus-f`语法要求：

一个程序中至少要有一个声明，且最后一个声明必须是 `void main(void)` 形式的函数声明。

（样例好像没有检查这个，不写检查void好像也可以过，但还是写一下）

根据要求，实现如下。

```
1 /* Program, 程序, program->declaration-list */
```

```

2  void CminusfBuilder::visit(ASTProgram &node)
3  {
4      // ASTProgram有一个
      std::vector<std::shared_ptr<ASTDeclaration>>类型的向量
5      // 我们要对这个向量进行遍历
6
7      LOG(DEBUG) << "Program\n";
8      // 语义检查
9      if (node.declarations.size() == 0)
10     {
11         std::cout << "ERROR: 该程序中没有声明。\\n";
12         return;
13     }
14     if (!(node.declarations.back()->id == "main" &&
node.declarations.back()->type == TYPE_VOID))
15     {
16         std::cout << "ERROR: 最后一个声明不是void main(void)\\n";
17         return;
18     }
19     for (auto decl : node.declarations) /* 遍历declaration-list子
      结点 */
20         decl->accept(*this);          /* 处理每一个declaration
      */
21     return;
22 }

```

<2> void CminusfBuilder::visit(ASTNum &node) {}

该节点结构定义如下:

```

1  struct ASTNum: ASTFactor {
2      virtual void accept(ASTVisitor &) override final;
3      CminusType type;
4      union {
5          int i_val;
6          float f_val;
7      };
8  };

```


数值节点没有子节点，直接进行处理，根据`type`确认数值类型，然后将值保存到全局变量中。根据语义规则，只能有整型和浮点数两个类型。分别赋值即可。

根据要求，实现如下：

```
1  /* Num, 对数值进行处理 */
2  void CminusfBuilder::visit(ASTNum &node)
3  {
4      // ASTNum中有两个成员变量
5      // CminusType type; 类型决定这片空间的读取方式
6      // 联合体union{ int i_val; float f_val;}是这片空间保存的值
7
8      LOG(DEBUG) << "Num\n";
9      if (node.type == TYPE_INT) /* 若为整型 */
10         // 调用ConstantInt中的API
11         Res = ConstantInt::get(node.i_val, module.get()); /* 获取
12         结点中存储的整型数值 */
13     else if (node.type == TYPE_FLOAT) /* 若为
14     浮点型 */
15         // 调用ConstantFP中的API
16         Res = ConstantFP::get(node.f_val, module.get()); /* 获取
17         结点中存储的浮点型数值 */
18     return;
19 }
```

<3> void CminusfBuilder::visit(ASTVarDeclaration &node) { }

该节点结构定义如下：

```
1  struct ASTVarDeclaration: ASTDeclaration {
2      virtual void accept(ASTVisitor &) override final;
3      CminusType type;
4      std::shared_ptr<ASTNum> num;
5  };
```

根据节点的定义，节点中包含一个类型和一个指针，还有继承自ASTDeclaration的id。对于变量声明节点的处理，需要产生分配空间的IR，在处理需考虑两个维度：是否全局变量，是否数组定义（根据节点的指针是否为空区分），分4类讨论。并且要把声明的变量放入当前作用域中，保证后续使用可以找到。

根据语义规则，全局变量需要初始化为0，数组变量声明时，大小应该大于0。

根据要求，实现如下：

```
1  /* Var-Declaration, 变量声明, var-declaration -> type-specifier
   ID | type-specifier ID [INTEGER] */
2  void CminusfBuilder::visit(ASTVarDeclaration &node)
3  {
4      // ASTVarDeclaration有两个成员变量:
5      // CminusType type;
6      // std::shared_ptr<ASTNum> num;
7
8      LOG(DEBUG) << "VarDeclaration\n";
9      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 */
10     if (node.type == TYPE_INT)                    /* 若为整型 */
11         tmpType = Int32Type;                      /* 则type为整数类型 */
12     else if (node.type == TYPE_FLOAT)             /* 则为浮点型 */
13         tmpType = FloatType;                     /* 则type为浮点类型 */
14     else
15         std::cout << "ERROR: 在变量声明中, 只有整型和浮点型可以使用\n";
16
17     // 需考虑两个维度: 是否全局变量, 是否数组定义, 分4类讨论
18     if (node.num != nullptr)
19     { /* 若为数组类型 */
20         if (node.num->i_val <= 0)
21             std::cout << "ERROR: 数组长度必须大于0\n";
22         /* 获取需开辟的对应大小的空间的类型指针 */
23         auto *arrayType = ArrayType::get(tmpType, node.num-
   >i_val); /* 获取对应的数组Type */
24         auto initializer = CONST_ZERO(tmpType);
25         /* 全局变量初始化为0 */
26         value *arrayAlloca;
27         /* 存储申请到的数组空间的地址 */
28         if (scope.in_global())
29             /* 若为全局数组, 则开辟全局数组 */
```

```

27         arrayAlloca = GlobalVariable::create(node.id,
module.get(), arrayType, false, initializer);
28         else /* 若不是全局数组，则开辟局部数组 */
29             arrayAlloca = builder->create_alloca(arrayType);
30         scope.push(node.id, arrayAlloca); /* 将获得的数组变量加入域
*/
31     }
32     else
33     { /* 若不是数组类型
*/
34         auto initializer = CONST_ZERO(tmpType); /* 全局变量初始化为
0 */
35         value *varAlloca; /* 存储申请到的变量
空间的地址 */
36         if (scope.in_global()) /* 若为全局变量，则
申请全局空间 */
37             varAlloca = GlobalVariable::create(node.id,
module.get(), tmpType, false, initializer);
38         else /* 若不是全局变量，则申请局部空间 */
39             varAlloca = builder->create_alloca(tmpType);
40         scope.push(node.id, varAlloca); /* 将获得变量加入域 */
41     }
42 }

```

<4> void CminusfBuilder::visit(ASTFunDeclaration &node) { }

该节点结构定义如下：

```

1 struct ASTFunDeclaration: ASTDeclaration {
2     virtual void accept(ASTVisitor &) override final;
3     std::vector<std::shared_ptr<ASTParam>> params;
4     std::shared_ptr<ASTCompoundStmt> compound_stmt;
5 };

```

FunDeclaration节点包含一个形参列表param和复合语句compound-stmt。需要创建的IR是创建函数和创建函数的第一个BasicBlock的指令，然后处理复合语句。在进入函数时要进入函数作用域，创建函数时要处理参数与返回值。对于每个参数，用全局变量取出实参，调用accept函数进行处理，在Param的visit函数中完成存储空间的分配，并加入到函数作用域当

中。

根据要求，实现如下：

```
1 void CminusfBuilder::visit(ASTFunDeclaration &node)
2 {
3     // ASTFunDeclaration有两个成员变量:
4     // std::vector<std::shared_ptr<ASTParam>> params;
5     // std::shared_ptr<ASTCompoundStmt> compound_stmt;
6
7     LOG(DEBUG) << "FunDeclaration\n";
8     // 考虑函数返回类型+函数名+参数列表+复合语句（局部声明与语句列表）
9
10    Type *retType; /* 函数返回类型 */
11    /* 根据不同的返回类型，设置retType */
12    if (node.type == TYPE_INT)
13    {
14        retType = Int32Type;
15    }
16    else if (node.type == TYPE_FLOAT)
17    {
18        retType = FloatType;
19    }
20    else if (node.type == TYPE_VOID)
21    {
22        retType = Type::get_void_type(module.get());
23    }
24
25    /* 根据函数声明，构造形参列表（此处的形参即参数的类型） */
26    std::vector<Type *> paramsType; /* 参数类型列表 */
27    for (auto param : node.params)
28    {
29        if (param->isarray)
30        {
31            /* 若参数为数组形式，则存入首地址指针 */
32
33            if (param->type == TYPE_INT) /* 若为整型 */
34
35            paramsType.push_back(Type::get_int32_ptr_type(module.get()));
36            else if (param->type == TYPE_FLOAT) /* 若为浮点型 */
37
38            paramsType.push_back(Type::get_float_ptr_type(module.get()));
39        }
40    }
```

```

36         else
37         {
对应类型 */
38             if (param->type == TYPE_INT) /* 若为整型 */
39                 paramType.push_back(Int32Type);
40             else if (param->type == TYPE_FLOAT) /* 若为浮点型 */
41                 paramType.push_back(FloatType);
42         }
43     }
44     auto funType = FunctionType::get(retType, paramType);
/* retType返回结构, paramType函数形参结构 */
45     auto function = Function::create(funType, node.id,
module.get()); /* 创建函数 */
46     scope.push(node.id, function);
/* 将函数加入到全局作用域 */
47     scope.enter();
/* 进入此函数作用域 */
48     auto bb = BasicBlock::create(module.get(), node.id +
"_entry", function); /* 创建基本块 */
49     builder->set_insert_point(bb);
/* 将基本块加入到builder中 */
50     /* 函数传参, 要将实参和形参进行匹配 */
51     std::vector<Value*> args; /* 创建vector存储实参 */
52     for (auto arg = function->arg_begin(); arg != function-
>arg_end(); arg++)
53     {
/* 遍历实参列表 */
54         args.push_back(*arg); /* 将实参加入vector */
55     }
56     for (int i = 0; i < node.params.size(); i++)
57     {
/* 遍历形参列表 (=遍历实参列表)
*/
58         auto param = node.params[i]; /* 取出对应形参 */
59         arg = args[i]; /* 取出对应实参 */
60         param->accept(*this); /* 调用param的accept进行处理
*/
61     }
62     node.compound_stmt->accept(*this); /* 处理函数体内语句compound-
stmt */
63     // 判断返回值的类型, 根据对应的返回值类型, 执行ret操作
64     // 这里应该是实现若没有显式返回, 默认返回对应类型的0或void
65     if (builder->get_insert_block()->get_terminator() ==
nullptr)

```

```

66     {
67         if (function->get_return_type()->is_void_type())
68             builder->create_void_ret();
69         else if (function->get_return_type()->is_float_type())
70             builder->create_ret(CONST_FP(0.0));
71         else
72             builder->create_ret(CONST_INT(0));
73     }
74     scope.exit(); /* 退出此函数作用域 */
75 }

```

<5> void CminusfBuilder::visit(ASTParam &node) { }

该节点结构定义如下：

```

1  struct ASTParam: ASTNode {
2      virtual void accept(ASTVisitor &) override final;
3      CminusType type;
4      std::string id;
5      // true if it is array param
6      bool isarray;
7  };

```

在处理参数时，要为参数分配空间，使参数能够保留在函数的作用域内。在lab3中自行编写.ll文件时直接使用参数，不进行存储，直接使用就可以实现相同的逻辑。但在将cminus转换为IR时，cminus的语义规定了每次函数调用都会产生一组独立内存的参数，因此为参数分配空间，并存入作用域。

根据要求，实现如下：

```

1  /* Param, 参数 */
2  void CminusfBuilder::visit(ASTParam &node)
3  {
4      // ASTParam有3个成员变量
5      // CminusType type;      参数类型
6      // std::string id;      参数名
7      // bool isarray;        是否数组标记
8      LOG(DEBUG) << "Param\n";

```

```

9
10     value *paramAlloca; // 分配该参数的存储空间
11     // 区分是否为数组，为整型还是浮点型，共分为4类讨论
12     if (node.isarray)
13     {
14         // 若为数组 */
15         if (node.type == TYPE_INT) /* 若为整型数组，则开辟整型数组存储
16         空间 */
17             paramAlloca = builder-
18             >create_alloca(Type::get_int32_ptr_type(module.get()));
19         else if (node.type == TYPE_FLOAT) /* 若为浮点数数组，则开辟浮
20         点数数组存储空间 */
21             paramAlloca = builder-
22             >create_alloca(Type::get_float_ptr_type(module.get()));
23     }
24     else
25     {
26         // 若不是数组 */
27         if (node.type == TYPE_INT) /* 若为整型，则开辟整型存储空间 */
28             paramAlloca = builder->create_alloca(Int32Type);
29         else if (node.type == TYPE_FLOAT) /* 若为浮点数，则开辟浮点数
30         存储空间 */
31             paramAlloca = builder->create_alloca(FloatType);
32     }
33     builder->create_store(arg, paramAlloca); /* 将实参load到开辟的
34     存储空间中 */
35     // 此处arg通过全局变量传递、
36     // 函数原型: StoreInst *create_store(value *val, value *ptr)
37     scope.push(node.id, paramAlloca); /* 将参数push到域中 */
38 }

```

<6> void CminusfBuilder::visit(ASTCompoundStmt &node) { }

该节点结构定义如下：

```

1 struct ASTCompoundStmt: ASTStatement {
2     virtual void accept(ASTVisitor&) override final;
3     std::vector<std::shared_ptr<ASTVarDeclaration>>
    local_declarations;
4     std::vector<std::shared_ptr<ASTStatement>> statement_list;
5 };

```

每个函数内部都有一个复合语句，根据ASTCompoundStmt的定义，复合语句由局部声明和一系列语句构成。只需要逐个调用相应的accept函数，不需要产生IR。

根据要求，实现如下：

```

1 void CminusBuilder::visit(ASTCompoundStmt &node)
2 {
3     // ASTCompoundStmt有两个成员变量
4     // std::vector<std::shared_ptr<ASTVarDeclaration>>
    local_declarations;
5     // std::vector<std::shared_ptr<ASTStatement>>
    statement_list;
6
7     LOG(DEBUG) << "CompoundStmt\n";
8     scope.enter();                                /* 进
入函数体内的作用域 */
9     for (auto local_declaration : node.local_declarations) /* 遍
历 */
10         local_declaration->accept(*this);          /* 处
理每一个局部声明 */
11     for (auto statement : node.statement_list)      /* 遍
历 */
12         statement->accept(*this);                  /* 处
理每一个语句 */
13     scope.exit();                                  /* 退
出作用域 */
14 }

```


<7> void CminusfBuilder::visit(ASTExpressionStmt &node) { }

该节点结构定义如下：

```
1 struct ASTExpressionStmt: ASTStatement {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTExpression> expression;
4 };
```

ExpressionStmt对应一条表达式或空，只要表达式存在，就处理该表达式，否则不需要做处理。

根据要求，实现如下：

```
1  /* ExpressionStmt, 表达式语句, expression-stmt -> expression;
2     *                                     | ; */
3  void CminusfBuilder::visit(ASTExpressionStmt &node)
4  {
5      // ASTExpressionStmt只有一个成员变量:
6      // std::shared_ptr<ASTExpression> expression;
7
8      LOG(DEBUG) << "ExpressionStmt\n";
9
10     if (node.expression != nullptr)    /* 若对应表达式存在 */
11         node.expression->accept(*this); /* 则处理该表达式 */
12 }
```

<8> void CminusfBuilder::visit(ASTSelectionStmt &node) { }

该节点结构定义如下：

```
1 struct ASTSelectionStmt: ASTStatement {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTExpression> expression;
4     std::shared_ptr<ASTStatement> if_statement;
5     // should be nullptr if no else structure exists
6     std::shared_ptr<ASTStatement> else_statement;
7 };
```

SelectionStmt包含一个条件表达式，一个**if**语句块，还有可能存在的**else**语句块。先处理表达式，产生条件跳转语句。

若指向**else**语句块的指针为空，就说明只有**if**语句。考虑只有**if**的情况，在执行到**if**时，应该通过**br**指令条件跳转到**if**语句块或**if**后的部分。

若还有**else**语句，则通过**br**指令条件跳转到**if**语句块或**else**语句块，然后从这两个语句块的结尾返回或者跳转到**ifelse**语句之后的部分。

在**SelectionStmt**的**visit**函数中应该至少生成三个**BasicBlock**，并生成**br**指令。根据**else**指针是否为空判断是否需要生成条件判断为**false**的**BasicBlock**。

根据要求，实现如下：

```
1 void CminusBuilder::visit(ASTSelectionStmt &node)
2 {
3     // ASTSelectionStmt有3个成员变量:
4     // std::shared_ptr<ASTExpression> expression;      判断条件
5     // std::shared_ptr<ASTStatement> if_statement;      if语句执
    行体
6     // std::shared_ptr<ASTStatement> else_statement;    else语句
    执行体
7     // 若无else语句, else_statement == nullptr
8
9     LOG(DEBUG) << "SelectionStmt\n";
10
11     // 构建判断条件代码
12     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 */
13     auto resType = Res->get_type(); /* 获取表达式得到的结果类型 */
14     if (resType->is_pointer_type())
15     {                                     /* 若结果为指针型, 则针对指
    针型进行处理 */
16         Res = builder->create_load(Res); /* 大于0视为true */
17     }
18     else if (resType->is_integer_type())
19     {
20         /* 若结果为整型, 则针对整型进行处理(bool类型视为整型) */
21         Res = builder->create_icmp_gt(Res,
22         CONST_ZERO(Int32Type)); /* 大于0视为true */
23     }
24     else if (resType->is_float_type())
```

```

23     {
24         /* 若结果为浮点型，则针对浮点数进行处理 */
25         Res = builder->create_fcmp_gt(Res,
26         CONST_ZERO(FloatType)); /* 大于0视为true */
27     }
28     auto function = builder->get_insert_block()->get_parent();
29     /* 获得当前所对应的函数 */
30
31     auto trueBB = BasicBlock::create(module.get(), "true",
32     function); /* 创建符合条件块 */
33
34     // 构建语句执行体代码
35     // 根据是否存在else语句分2类讨论
36     if (node.else_statement != nullptr)
37     {
38         /* 若存在else语句 */
39         auto falseBB = BasicBlock::create(module.get(), "false",
40         function); /* 创建else块 */
41         builder->create_cond_br(Res, trueBB, falseBB);
42         /* 设置跳转语句 */
43         // 处理trueBB
44         builder->set_insert_point(trueBB); /* 符合if条
45         件的块 */
46         node.if_statement->accept(*this); /* 处理if语
47         句执行体 */
48         auto curTrueBB = builder->get_insert_block(); /* 将块加入
49         */
50         // 处理falseBB
51         builder->set_insert_point(falseBB); /* else的
52         块 */
53         node.else_statement->accept(*this); /* 处理
54         else语句执行体 */
55         auto curFalseBB = builder->get_insert_block(); /* 将块加
56         入 */
57
58         /* 处理返回，避免跳转到对应块后无return */
59         // 判断true语句中是否存在ret语句
60         auto trueTerm = builder->get_insert_block()-
61         >get_terminator();
62         // 判断else语句中是否存在ret语句
63         auto falseTerm = builder->get_insert_block()-
64         >get_terminator();

```

```

50
51     if (trueTerm == nullptr || falseTerm == nullptr)
52     { /* 若有一方不存在return语句，则需要创建返回块 */
53         BasicBlock *retBB;
54         retBB = BasicBlock::create(module.get(), "ret",
function); /* 创建return块 */
55         builder->set_insert_point(retBB);
56         /* return块（即后续语句） */
57         if (trueTerm == nullptr)
58         {
59             /* 若if语句
执行体不存在return */
60             builder->set_insert_point(curTrueBB); /* 则设置跳
转 */
61         }
62         else if (falseTerm == nullptr)
63         {
64             /* 若else
语句执行体中不存在return */
65             builder->set_insert_point(curFalseBB); /* 则设置
跳转 */
66         }
67         builder->create_br(retBB); /* 跳转到刚刚设置的return块
*/
68     }
69     }
70     else
71     {
72         /* 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 */
73         auto retBB = BasicBlock::create(module.get(), "ret",
function); /* 后续语句块 */
74         builder->create_cond_br(Res, trueBB, retBB);
75         /* 根据条件设置跳转指令 */
76
77         builder->set_insert_point(trueBB);
78         /* true语句块 */
79         node.if_statement->accept(*this);
80         /* 执行条件符合后要执行的语句 */
81         if (builder->get_insert_block()->get_terminator() ==
nullptr) /* 补充return（同上） */
82         {
83             builder->create_br(retBB);
84             /* 跳转到刚刚设置的return块 */
85             builder->set_insert_point(retBB);
86             /* return块（即后续语句） */

```

```

77     }
78 }

```

<9> void CminusfBuilder::visit(ASTIterationStmt &node) {}

该节点结构定义如下：

```

1 struct ASTIterationStmt: ASTStatement {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTExpression> expression;
4     std::shared_ptr<ASTStatement> statement;
5 };

```

while迭代语句有一个条件表达式，进行条件跳转（与if类似）。可以创建一个用于判断的ifBasicBlock，一个循环的loopBasicBlock，一个while语句后的afterWhileBasicBlock，添加相应的指令。当条件表达式为True时，进行ifBB->loopBB->ifBB的循环跳转。

根据要求，实现如下：

```

1  /* IterationStmt, while语句, iteration-stmt -> while (expression)
   statement */
2  void CminusfBuilder::visit(ASTIterationStmt &node)
3  {
4      // ASTIterationStmt有2个成员变量:
5      // std::shared_ptr<ASTExpression> expression;    循环判断条件
6      // std::shared_ptr<ASTStatement> statement;      while循环执行
   体
7
8      LOG(DEBUG) << "IterationStmt\n";
9      auto function = builder->get_insert_block()->get_parent();
   /* 获得当前所对应的函数 */
10     auto conditionBB = BasicBlock::create(module.get(),
"condition", function); /* 创建条件判断块 */
11     auto loopBB = BasicBlock::create(module.get(), "loop",
function); /* 创建循环语句块 */
12     auto retBB = BasicBlock::create(module.get(), "ret",
function); /* 创建后续语句块 */

```

```

13     if (builder->get_insert_block()->get_terminator() ==
nullptr)
14         builder->create_br(conditionBB); // 跳转到条件判断块
15
16     // 构建条件判断代码
17     builder->set_insert_point(conditionBB); /* 条件判断块 */
18     node.expression->accept(*this);          /* 处理条件判断对应的表
19     达式，得到返回值存到expression中 */
20     auto resType = Res->get_type();          /* 获取表达式得到的结果
21     类型 */
22     if (resType->is_pointer_type())
23     {
24         /* 若结果为指针型，则针对指
25         针型进行处理 */
26         Res = builder->create_load(Res); /* 大于0视为true */
27     }
28     else if (resType->is_integer_type())
29     {
30         /* 若结果为整型，则针对整型进行处理(bool类型视为整型) */
31         Res = builder->create_icmp_gt(Res,
CONST_ZERO(Int32Type)); /* 大于0视为true */
32     }
33     else if (resType->is_float_type())
34     {
35         /* 若结果为浮点型，则针对浮点数进行处理 */
36         Res = builder->create_fcmp_gt(Res,
CONST_ZERO(FloatType)); /* 大于0视为true */
37     }
38     builder->create_cond_br(Res, loopBB, retBB); /* 设置条件跳转语
39     句 */
40
41     // 构建循环语句代码
42     builder->set_insert_point(loopBB);
43     /* 循环语句执行块 */
44     node.statement->accept(*this);
45     /* 执行对应的语句 */
46     if (builder->get_insert_block()->get_terminator() ==
nullptr) /* 若无返回，则补充跳转 */
47         builder->create_br(conditionBB);
48     /* 跳转到条件判断语句 */
49
50     builder->set_insert_point(retBB); /* return块（即后续语句） */
51 }

```

<10> void CminusfBuilder::visit(ASTReturnStmt &node) { }

该节点结构定义如下：

```
1 struct ASTReturnStmt: ASTStatement {
2     virtual void accept(ASTVisitor &) override final;
3     // should be nullptr if return void
4     std::shared_ptr<ASTExpression> expression;
5 };
```

返回语句中有一个表达式计算返回值，如果指向该返回语句的指针为空，说明没有返回值，创建一个void返回IR，否则需要调用该表达式的accept函数，并检查返回类型是否和函数的返回类型相同。

根据要求，实现如下：

```
1  /* ReturnStmt, 返回语句, return-stmt -> return;
2     *                               | return expression; */
3  void CminusfBuilder::visit(ASTReturnStmt &node)
4  {
5      // ASTReturnStmt只有1个成员变量
6      // std::shared_ptr<ASTExpression> expression;    返回语句表达式
7
8      LOG(DEBUG) << "ReturnStmt\n";
9      auto function = builder->get_insert_block()->get_parent();
10     /* 获得当前所对应的函数 */
11     auto retType = function->get_return_type();
12     /* 获取返回类型 */
13
14     // 处理返回语句表达式
15     if (retType->is_void_type())
16     {
17         // 如果本来就是void类型的函数，需要返回void
18         builder->create_void_ret(); /* 则创建void返回，随后return，
19         无需后续操作 */
17         return;
18     }
19     // 如果需要返回非void
```

```

20     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 */
21     auto resType = Res->get_type(); /* 获取表达式得到的结果类型 */
22     /* 处理expression返回的结果和需要return的结果类型不匹配的问题 */
23     // 使用强制类型转换
24     if (retType->is_integer_type() && resType->is_float_type())
25         Res = builder->create_fptosi(Res, Int32Type);
26     if (retType->is_float_type() && resType->is_integer_type())
27         Res = builder->create_sitofp(Res, Int32Type);
28     builder->create_ret(Res); /* 创建return，将expression的结果进行
    返回 */
29 }
30

```

<11> void CminusfBuilder::visit(ASTVar &node) { }

该节点结构定义如下：

```

1 struct ASTVar: ASTFactor {
2     virtual void accept(ASTVisitor &) override final;
3     std::string id;
4     // nullptr if var is of int type
5     std::shared_ptr<ASTExpression> expression;
6 };

```

根据cminus的语义说明，Var可以是整型变量，浮点变量或数组变量。如果是数组变量，需要判断下标是否为负，如果为负则添加neg_idx_except指令退出程序，否则计算对应元素的地址(gep指令)。如果是数组，则下标可能是个表达式，需要确保表达式的返回结果为整型，然后才能进行取元素的操作。

根据要求，实现如下：

```

1 void CminusfBuilder::visit(ASTVar &node)
2 {
3     // ASTVar有2个成员变量：
4     // std::string id;                变量名
5     // std::shared_ptr<ASTExpression> expression;    变量类型
6     // 如果变量var是int型的，这个取nullptr

```



```

7
8     LOG(DEBUG) << "var\n";
9     auto var = scope.find(node.id);          /* 从域中取出对应变量
*/
10    bool if_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) */
11    need_as_address = false;                  /* 重置全局变量
need_as_address */
12    value *index = CONST_INT(0);              /* 初始化index */
13    if (node.expression != nullptr)
14    {                                          /* 若有
expression, 代表不是int类型的引用*/
15        node.expression->accept(*this);      /* 处理
expression, 得到结果Res */
16        auto res = Res;                      /* 存储
结果 */
17        if (checkFloat(res))                 /* 判断
结果是否为浮点数 */
18            res = builder->create_fptosi(res, Int32Type); /* 若
是, 则矫正为整数 */
19        index = res;                          /* 赋值
给index, 表示数组下标 */
20        /* 判断下标是否为负数。若是, 则调用neg_idx_except函数进行处理 */
21        auto function = builder->get_insert_block()-
>get_parent();                               /* 获取当前函数 */
22        auto indexTest = builder->create_icmp_lt(index,
CONST_ZERO(Int32Type));                      /* test是否为负数 */
23        auto failBB = BasicBlock::create(module.get(), node.id +
"_failTest", function); /* fail块 */
24        auto passBB = BasicBlock::create(module.get(), node.id +
"_passTest", function); /* pass块 */
25        builder->create_cond_br(indexTest, failBB, passBB);
/* 设置跳转语句 */
26
27        // 下标为负数, 调用neg_idx_except函数进行处理
28        builder->set_insert_point(failBB);
/* fail块, 即下标为负数 */
29        auto fail = scope.find("neg_idx_except");
/* 取出neg_idx_except函数 */
30        builder->create_call(static_cast<Function *>(fail), {});
/* 调用neg_idx_except函数进行处理 */

```

```

31         builder->create_br(passBB);
    /* 跳转到pass块 */
32
33         // 下标合法,
34         builder->set_insert_point(passBB);
    /* pass块 */
35         if (var->get_type()->get_pointer_element_type()-
>is_array_type()) /* 若为指向数组的指针 */
36             var = builder->create_gep(var, {CONST_INT(0),
index}); /* 则进行两层寻址 */
37         else
38         {
39             if (var->get_type()->get_pointer_element_type()-
>is_pointer_type()) /* 若为指针 */
40                 var = builder->create_load(var);
    /* 则取出指针指向的元素 */
41             var = builder->create_gep(var, {index});
    /* 进行一层寻址（因为此时并非指向数组） */
42         }
43         if (if_return_lvalue)
44         {
    /* 若要返回值 */
45             Res = var; /* 则返回var对应的地址 */
46             need_as_address = false; /* 并重置全局变量
need_as_address */
47         }
48         else
49             Res = builder->create_load(var); /* 否则则进行load */
50         return;
51     }
52     else
53     { // 处理不是数组的情况
54         if (if_return_lvalue)
55         {
    /* 若要返回值 */
56             Res = var; /* 则返回var对应的地址 */
57             need_as_address = false; /* 并重置全局变量
need_as_address */
58         }
59         else
60         { /* 否则 */
61             // 数组的指针即a[]类型就返回数组的起始地址，否则load取值
62             if (var->get_type()->get_pointer_element_type()-
>is_array_type()) /* 若指向数组，需要两个偏移取地址 */

```

```

63         Res = builder->create_gep(var, {CONST_INT(0),
CONST_INT(0)}); /* 则寻址 */
64         else
65             Res = builder->create_load(var); /* 否则则进行load
*/
66     }
67 }
68 }

```

<12> void CminusfBuilder::visit(ASTAssignExpression &node) { }

该节点结构定义如下：

```

1 struct ASTAssignExpression: ASTExpression {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTVar> var;
4     std::shared_ptr<ASTExpression> expression;
5 };

```

对于Assign语句，将全局变need_as_address量置为true，调用子节点var的accept函数得到变量的地址，然后计算表达式的值，创建store指令将值存入地址。需要确认表达式结果是否与变量类型相同，如果不同需要将表达式结果转换为和变量相同的类型。

根据要求，实现如下：

```

1  /* AssignExpression, 赋值语句, var = expression */
2  void CminusfBuilder::visit(ASTAssignExpression &node)
3  {
4      // ASTAssignExpression有2个成员变量
5      // std::shared_ptr<ASTVar> var;                左值（被赋值的
对象）
6      // std::shared_ptr<ASTExpression> expression;    右值（赋的值）
7
8      LOG(DEBUG) << "AssignExpression\n";
9      need_as_address = true; /* 设置need_as_address, 表示需要返回值
*/
10
11     // 获取左值, 右值

```

```

12     node.var->accept(*this);           /* 处理左var */
13     auto left = Res;                   /* 获取地址 */
14     node.expression->accept(*this);    /* 处理右expression */
15     auto right = Res;                  /* 获得结果 */
16
17     // 处理左值，右值类型冲突问题
18     auto leftType = left->get_type()-
>get_pointer_element_type(); /* 获取var的类型 */
19     /* 若赋值语句左右类型不匹配，则进行匹配 */
20     if (leftType == FloatType && checkInt(right))
21         right = builder->create_sitofp(right, FloatType);
22     if (leftType == Int32Type && checkFloat(right))
23         right = builder->create_fptosi(right, Int32Type);
24
25     // 赋值
26     builder->create_store(right, left);
27 }

```

<13> void CminusfBuilder::visit(ASTSimpleExpression &node) {}

该节点结构定义如下：

```

1 struct ASTSimpleExpression: ASTExpression {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTAdditiveExpression> additive_expression_l;
4     std::shared_ptr<ASTAdditiveExpression> additive_expression_r;
5     RelOp op;
6 };

```

简单表达式SimpleExpression是一个加法表达式或两个加法表达式的关系运算。在节点中有两个加法表达式的指针和一个运算符类型为RelOp的运算符op，RelOp是一个枚举类型，包含了所有比较运算符。根据语义，对于该节点的处理，应该先处理加法表达式，将表达式的值保存下来，如果两个表达式指针都不为空，说明为关系运算，再比较两个运算结果，根据结果将表达式的值赋为0或1。进行比较时需要注意两个值的类型，整型和浮点型比较时要将整型转换为浮点型。

具体实现中，应该调用加法表达式的accept函数(如果指针不为空)，暂存结果，对于比较运算，根据op生成icmp或fcmp的指令，最后返回的值就是指令结果。

根据要求，实现如下：

```
1 void CminusBuilder::visit(ASTSimpleExpression &node)
2 {
3     // ASTSimpleExpression有3个成员变量
4     // std::shared_ptr<ASTAdditiveExpression>
5     additive_expression_l;
6     // std::shared_ptr<ASTAdditiveExpression>
7     additive_expression_r;
8     // RelOp op;
9
10    LOG(DEBUG) << "SimpleExpression\n";
11    node.additive_expression_l->accept(*this); /* 处理左边的
12    expression */
13    auto lres = Res; /* 获取结果存到lres
14    中 */
15    if (node.additive_expression_r == nullptr)
16    {
17        return;
18    } /* 若不存在右
19    expression, 则直接返回 */
20    node.additive_expression_r->accept(*this); /* 处理右边的
21    expression */
22    auto rres = Res; /* 结果存到rres中
23    */
24
25    // 确保两个表达式的类型相同, 若存在浮点和整型混存, 全部转换为浮点型
26    if (checkInt(lres) && checkInt(rres))
27    { /* 确保两边都是整数 */
28        switch (node.op)
29        {
30            /* 根据不同的比较操作, 调用icmp进行处理 */
31            // 比较的返回结果
32            // enum RelOp:
33            // <= 对应 OP_LE,
34            // < 对应 OP_LT,
35            // > 对应 OP_GT,
36            // >= 对应 OP_GE,
37            // == 对应 OP_EQ,
38            // != 对应 OP_NEQ
39            case OP_LE:
40                Res = builder->create_icmp_le(lres, rres);
```

```

34         break;
35     case OP_LT:
36         Res = builder->create_icmp_lt(lres, rres);
37         break;
38     case OP_GT:
39         Res = builder->create_icmp_gt(lres, rres);
40         break;
41     case OP_GE:
42         Res = builder->create_icmp_ge(lres, rres);
43         break;
44     case OP_EQ:
45         Res = builder->create_icmp_eq(lres, rres);
46         break;
47     case OP_NEQ:
48         Res = builder->create_icmp_ne(lres, rres);
49         break;
50     }
51 }
52 else
53 {
54     /* 若有一边是浮点类型，则需要先将另一边也转
55     为浮点数，再进行比较 */
56     if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
57         lres = builder->create_sitofp(lres, FloatType);
58     if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
59         rres = builder->create_sitofp(rres, FloatType);
60     switch (node.op)
61     {
62         /* 根据不同的比较操作，调用fcmp进行处理 */
63         case OP_LE:
64             Res = builder->create_fcmp_le(lres, rres);
65             break;
66         case OP_LT:
67             Res = builder->create_fcmp_lt(lres, rres);
68             break;
69         case OP_GT:
70             Res = builder->create_fcmp_gt(lres, rres);
71             break;
72         case OP_GE:
73             Res = builder->create_fcmp_ge(lres, rres);
74             break;
75         case OP_EQ:
76             Res = builder->create_fcmp_eq(lres, rres);

```

```

75         break;
76     case OP_NEQ:
77         Res = builder->create_fcmp_ne(lres, rres);
78         break;
79     }
80 }
81 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） */
82 }

```

<14> void CminusfBuilder::visit(ASTAdditiveExpression &node) { }

该节点结构定义如下：

```

1 struct ASTAdditiveExpression: ASTNode {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTAdditiveExpression> additive_expression;
4     AddOp op;
5     std::shared_ptr<ASTTerm> term;
6 };

```

加法表达式中包含了一个乘法表达式，一个加法表达式和一个运算符。如果加法表达式指针为空，则表达式的值就是乘法表达式的值，否则分别计算两个表达式，调用相应的accept函数，然后进行根据运算符生成加或减指令。

根据要求，实现如下：

```

1 void CminusfBuilder::visit(ASTAdditiveExpression &node)
2 {
3     // ASTAdditiveExpression有3个成员变量：
4     // std::shared_ptr<ASTAdditiveExpression>
5     additive_expression;
6     // AddOp op;
7     // std::shared_ptr<ASTTerm> term;
8
9     LOG(DEBUG) << "AdditiveExpression\n";
10    if (node.additive_expression == nullptr)
11    { /* 若无加减法运算 */

```

```

11         node.term->accept(*this);
12         return; /* 则直接去做乘除法 */
13     }
14     node.additive_expression->accept(*this); /* 处理左expression
15     */
16     auto lres = Res; /* 结果保存在lres中
17     */
18     node.term->accept(*this); /* 处理右term */
19     auto rres = Res; /* 结果保存在rres中
20     */
21     // 分为整型-整型, 和存在浮点数类型, 这两种情况进行讨论
22     // 若存在浮点数, 则全部强制转换为浮点数实现
23     if (checkInt(lres) && checkInt(rres))
24     { /* 确保两边都是整数 */
25         switch (node.op)
26         { /* 根据对应加法或是减法, 调用iadd或是isub进行处理 */
27             case OP_PLUS:
28                 Res = builder->create_iadd(lres, rres);
29                 break;
30             case OP_MINUS:
31                 Res = builder->create_isub(lres, rres);
32                 break;
33         }
34     }
35     else
36     { /* 若有一边是浮点类型, 则需要先将另一边也转
37       为浮点数, 再进行处理 */
38         if (checkInt(lres)) /* 若左边是整数, 则将其转为浮点数 */
39             lres = builder->create_sitofp(lres, FloatType);
40         if (checkInt(rres)) /* 若右边是整数, 则将其转为浮点数 */
41             rres = builder->create_sitofp(rres, FloatType);
42         switch (node.op)
43         { /* 根据对应加法或是减法, 调用fadd或是fsub进行处理 */
44             case OP_PLUS:
45                 Res = builder->create_fadd(lres, rres);
46                 break;
47             case OP_MINUS:
48                 Res = builder->create_fsub(lres, rres);
49                 break;
50         }
51     }

```


<15> void CminusfBuilder::visit(ASTTerm &node) { }

该节点结构定义如下：

```

1 struct ASTTerm : ASTNode {
2     virtual void accept(ASTVisitor &) override final;
3     std::shared_ptr<ASTTerm> term;
4     MulOp op;
5     std::shared_ptr<ASTFactor> factor;
6 };

```

乘法表达式由乘法表达式和因子或单独一个因子构成。与加法表达式的处理相同。

根据要求，实现如下：

```

1 void CminusfBuilder::visit(ASTTerm &node)
2 {
3     // ASTTerm有3个成员变量：
4     // std::shared_ptr<ASTTerm> term;
5     // MulOp op;
6     // std::shared_ptr<ASTFactor> factor;
7
8     LOG(DEBUG) << "Term\n";
9     if (node.term == nullptr)
10    { /* 若无乘法运算 */
11        node.factor->accept(*this);
12        return; /* 则直接去处理元素 */
13    }
14    node.term->accept(*this); /* 处理左term */
15    auto lres = Res; /* 结果保存在lres中 */
16    node.factor->accept(*this); /* 处理右factor */
17    auto rres = Res; /* 结果保存在rres中 */
18    if (checkInt(lres) && checkInt(rres))
19    { /* 确保两边都是整数 */
20        switch (node.op)
21        { /* 根据对应乘法或是除法，调用imul或是idiv进行处理 */

```

```

22         case OP_MUL:
23             Res = builder->create_imul(lres, rres);
24             break;
25         case OP_DIV:
26             Res = builder->create_isdiv(lres, rres);
27             break;
28     }
29 }
30 else
31 {
32     /* 若有一边是浮点类型，则需要先将另一边也转
33     为浮点数，再进行处理 */
34     if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
35         lres = builder->create_sitofp(lres, FloatType);
36     if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
37         rres = builder->create_sitofp(rres, FloatType);
38     switch (node.op)
39     { /* 根据对应乘法或是除法，调用fmul或是fddiv进行处理 */
40     case OP_MUL:
41         Res = builder->create_fmul(lres, rres);
42         break;
43     case OP_DIV:
44         Res = builder->create_fdiv(lres, rres);
45         break;
46     }
47 }
48 }

```

<16> void CminusfBuilder::visit(ASTCall &node) {}

该节点结构定义如下：

```

1 struct ASTCall: ASTFactor {
2     virtual void accept(ASTVisitor &) override final;
3     std::string id;
4     std::vector<std::shared_ptr<ASTExpression>> args;
5 };

```

call节点需要创建一条函数调用call指令，从作用域中取出函数，然后根据函数的参数将节点的实参传入，并检查类型是否与函数参数的类型一致，不一致则需要转换为函数的形参类型。创建一个参数列表，将转换好的参数存入列表来调用函数。

根据要求，实现如下：

```
1  /* Call, 函数调用, call -> ID (args) */
2  void CminusfBuilder::visit(ASTCall &node)
3  {
4      // ASTCall有3个成员变量:
5      // std::string id;
6      // std::vector<std::shared_ptr<ASTExpression>> args;
7
8      LOG(DEBUG) << "Call\n";
9      auto function = static_cast<Function *>
10 (scope.find(node.id)); /* 获取需要调用的函数 */
11      auto paramType = function->get_function_type()-
12 >param_begin(); /* 获取其参数类型 */
13
14      if (function == nullptr)
15          std::cout << "ERROR: 函数" << node.id << "未定义\n";
16
17      // 处理参数列表
18      std::vector<Value *> args; /* 创建args用于存储函数参数的值，构建调
19 用函数的参数列表 */
20      for (auto arg : node.args)
21      {
22          /* 遍历形参列表 */
23          arg->accept(*this); /* 对每一个参数进行处理，获取参数对应的值
24  */
25
26          if (Res->get_type()->is_pointer_type())
27          {
28              /* 若参数是指针 */
29              args.push_back(Res); /* 则直接将值加入到参数列表 */
30          }
31          else
32          { /* 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
33 类型转换 */
34              if (*paramType == FloatType && checkInt(Res))
35                  Res = builder->create_sitofp(Res, FloatType);
36              if (*paramType == Int32Type && checkFloat(Res))
37                  Res = builder->create_fptosi(Res, Int32Type);
38              args.push_back(Res);
39          }
40      }
41      paramType++; /* 查看下一个形参 */
```

```

32     }
33     Res = builder->create_call(static_cast<Function *>
    (function), args); /* 创建函数调用 */
34 }

```

实验测试

构建

在build文件夹下输入如下指令，使我们写的cminusf_builder.cpp文件纳入编译。

```

1  make clean
2  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。

单个测试

需要对某个（提供的或自己写的）`.cminus` 文件测试时，可以这样使用：

```

1  # 假设 cminusfc 的路径在你的$PATH中
2  # 利用构建好的 Module 生成 test.ll
3  # 注意，如果调用了外部函数，如 input, output 等，则无法使用lli运行
4  cminusfc test.cminus -emit-llvm
5
6  # 假设libcminus_io.a的路径在$LD_LIBRARY_PATH中，clang的路径在$PATH中
7  # 1. 利用构建好的 Module 生成 test.ll
8  # 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
9  cminusfc test.cminus

```

步骤如下：

如对本段代码，为保存在build文件夹内的`my.cminus`

```

1 void main(void){
2     int i;
3     i = 5;
4     while(i >= 7)
5     {
6         i = i - 2;
7     }
8     output(i);
9 }
10

```

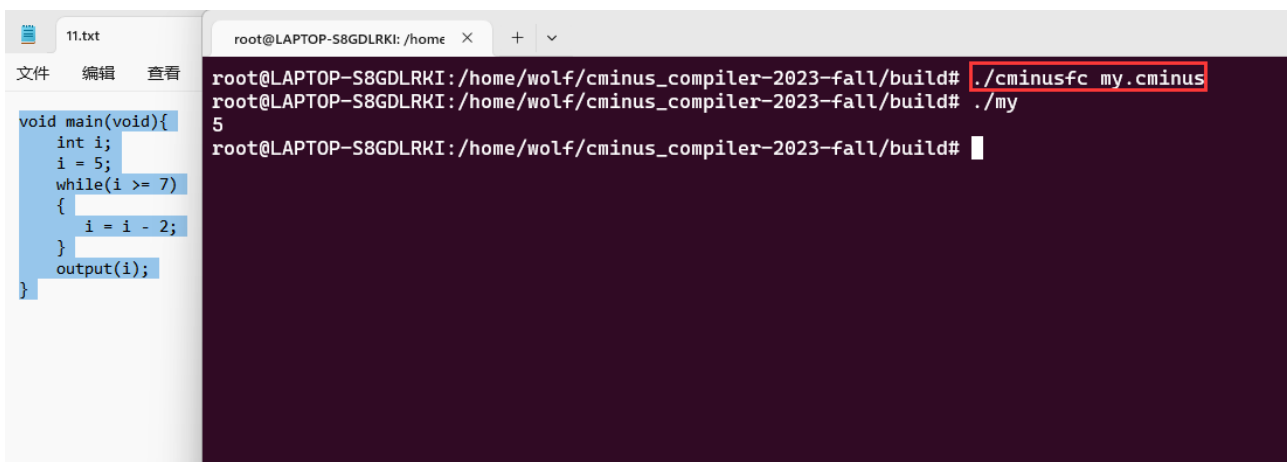
在build文件夹内使用

```
1 ./cminusfc my.cminus
```

编译my.cminus，生成my可执行文件，然后使用

```
1 ./my
```

运行该可执行文件，得到结果。



The screenshot shows a code editor on the left with a file named '11.txt' containing the following Cminus code:

```

void main(void){
    int i;
    i = 5;
    while(i >= 7)
    {
        i = i - 2;
    }
    output(i);
}

```

On the right, a terminal window shows the execution of the program in the directory `/home/wolf/cminus_compiler-2023-fall/build#`:

```

root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./cminusfc my.cminus
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# ./my
5
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#

```

自动测试

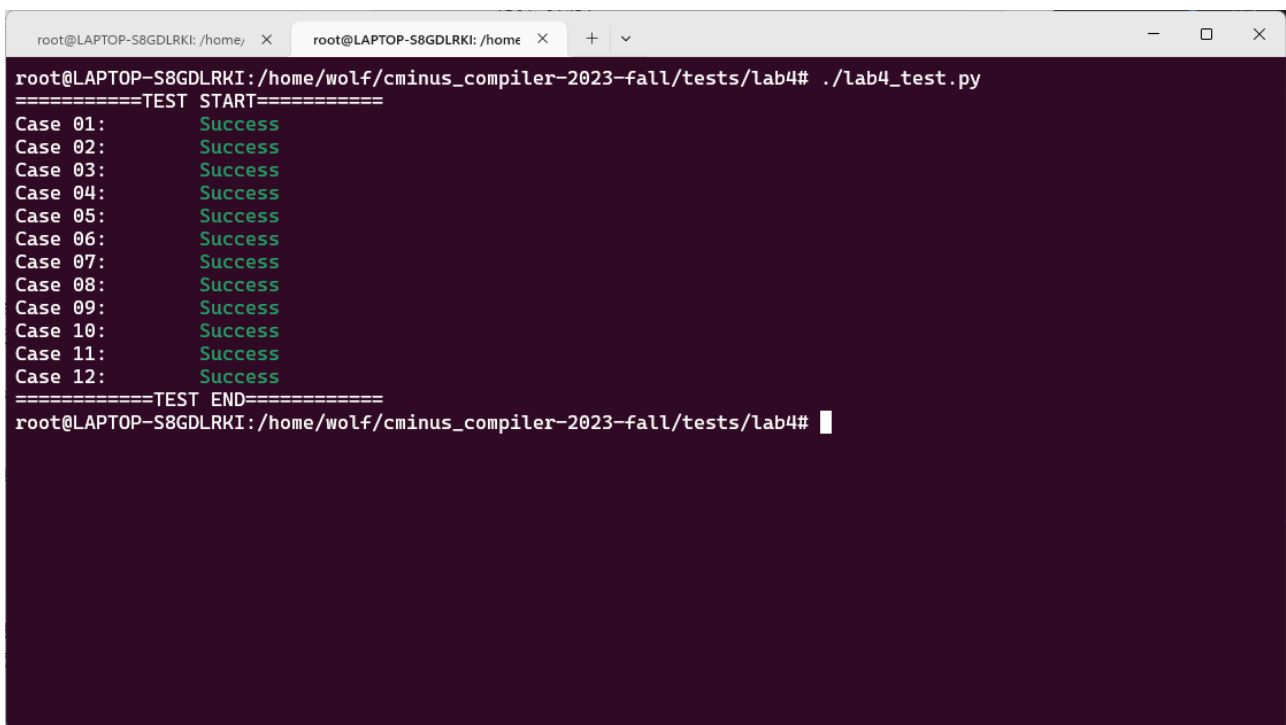
在 tests/lab4 目录下，输入如下指令：

```
1 ./lab4_test.py
```

输出：

```
1  =====TEST START=====
2  Case 01:      Success
3  Case 02:      Success
4  Case 03:      Success
5  Case 04:      Success
6  Case 05:      Success
7  Case 06:      Success
8  Case 07:      Success
9  Case 08:      Success
10 Case 09:      Success
11 Case 10:      Success
12 Case 11:      Success
13 Case 12:      Success
14 =====TEST END=====
```

截图如下：



```
root@LAPTOP-S8GDLRKi: /home/ X root@LAPTOP-S8GDLRKi: /home X + v
root@LAPTOP-S8GDLRKi: /home/wolf/cminus_compiler-2023-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@LAPTOP-S8GDLRKi: /home/wolf/cminus_compiler-2023-fall/tests/lab4#
```

实验总结

本次实验具有一定的难度，很考验对于实验3知识的融汇贯通。

通过本次实验，理解了从抽象语法树产生中间IR的方法，并进行了实现。在实现过程中，对于IR的指令有了进一步的熟悉与理解，掌握了使用C++接口创建不同IR指令的方法，以及在访问者模式下遍历抽象语法树，完成IR生成的过程。在完成实现时需要多次跳转，阅读相关的头文件，语义规则，原型函数定义，原型结构体定义等。

经过四次实验，结合课程所学的原理，理解了编译器的词法分析，语法分析，中间代码生成的过程，也学习了相关工具的使用并进行了实践，了解了编译器工作的每一个部分的原理和相互之间的配合。

但感觉还是没有融汇贯通，还是有很多需要学的，如果投入的时间能再多一些，感觉能学到更多。

实验反馈

实验难度还是有点大的，时间给的还是太少了。

而且没有给参考的代码，花了很多时间才理解实验需要做什么。前几个函数的实现参考了一些标准代码并进行研究才能稍微领悟实验书写的方式。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。

参考资料

https://blog.csdn.net/qq_51684393/article/details/128474127

<https://blog.csdn.net/Aaron503/article/details/128325057>

<https://github.com/XiaoYaoYouUSTC/Cminusf-Compiler>

https://github.com/kyin2905/cminus_compiler-2022-fall

【附录】实验完整代码

```
1  #include "cminusf_builder.hpp"
2  #include "logging.hpp"
3
4  // use these macros to get constant value
5  #define CONST_INT(num) \
6      ConstantInt::get(num, module.get()) /* 增加一个有关整型的宏 */
7  // 以num值来创建常数类
8  #define CONST_FP(num) \
9      ConstantFP::get((float)num, module.get())
10 // 以num值来创建浮点数类
11 #define CONST_ZERO(type) \
```

```

12     ConstantZero::get(type, module.get()) /* 此处要修改为type */
13 // 用于全局变量初始化的常量0值
14
15 #define Int32Type \
16     Type::get_int32_type(module.get()) /* 获取int32类型 */
17 #define FloatType \
18     Type::get_float_type(module.get()) /* 获取float类型 */
19
20 #define checkInt(num) \
21     num->get_type()->is_integer_type() /* 整型判断 */
22 #define checkFloat(num) \
23     num->get_type()->is_float_type() /* 浮点型判断 */
24 #define checkPointer(num) \
25     num->get_type()->is_pointer_type() /* 指针类型判断 */
26
27 // You can define global variables here
28 // to store state
29 value *Res; /* 存储返回的结果 */
30 value *arg; /* 存储参数指针，用于Param的处理 */
31 bool need_as_address = false; /* 标志是返回值还是返回地址 */
32
33 /*
34  * use CminusfBuilder::Scope to construct scopes
35  * scope.enter: enter a new scope
36  * scope.exit: exit current scope
37  * scope.push: add a new binding to current scope
38  * scope.find: find and return the value bound to the name
39  */
40
41 // 下面的类型在include/ast.hpp有定义
42 // 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
43
44 /* Program, 程序, program->declaration-list */
45 void CminusfBuilder::visit(ASTProgram &node)
46 {
47     // ASTProgram有一个
std::vector<std::shared_ptr<ASTDeclaration>>类型的向量
48     // 我们要对这个向量进行遍历
49
50     LOG(DEBUG) << "Program\n";
51     // 语义检查

```



```

52     if (node.declarations.size() == 0)
53     {
54         std::cout << "ERROR: 该程序中没有声明。\\n";
55         return;
56     }
57     if (!(node.declarations.back()->id == "main" &&
node.declarations.back()->type == TYPE_VOID))
58     {
59         std::cout << "ERROR: 最后一个声明不是void main(void)\\n";
60         return;
61     }
62     for (auto decl : node.declarations) /* 遍历declaration-list
子结点 */
63         decl->accept(*this);          /* 处理每一个declaration
*/
64     return;
65 }
66
67 /* Num, 对数值进行处理 */
68 void CminusfBuilder::visit(ASTNum &node)
69 {
70     // ASTNum中有两个成员变量
71     // CminusType type; 类型决定这片空间的读取方式
72     // 联合体union{ int i_val; float f_val;}是这片空间保存的值
73
74     LOG(DEBUG) << "Num\\n";
75     if (node.type == TYPE_INT) /* 若为整型 */
76         // 调用ConstantInt中的API
77         Res = ConstantInt::get(node.i_val, module.get()); /* 获
取结点中存储的整型数值 */
78     else if (node.type == TYPE_FLOAT) /* 若
为浮点型 */
79         // 调用ConstantFP中的API
80         Res = ConstantFP::get(node.f_val, module.get()); /* 获取
结点中存储的浮点型数值 */
81     return;
82 }
83
84 /* var-Declaration, 变量声明, var-declaration -> type-specifier
ID | type-specifier ID [INTEGER] */
85 void CminusfBuilder::visit(ASTVarDeclaration &node)
86 {

```

```

87 // ASTVarDeclaration有两个成员变量:
88 // CminusType type;
89 // std::shared_ptr<ASTNum> num;
90
91 LOG(DEBUG) << "varDeclaration\n";
92 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 */
93 if (node.type == TYPE_INT) /* 若为整型 */
94     tmpType = Int32Type; /* 则type为整数类型 */
95 else if (node.type == TYPE_FLOAT) /* 则为浮点型 */
96     tmpType = FloatType; /* 则type为浮点类型 */
97 else
98     std::cout << "ERROR: 在变量声明中, 只有整型和浮点型可以使用
\n";
99
100 // 需考虑两个维度: 是否全局变量, 是否数组定义, 分4类讨论
101 if (node.num != nullptr)
102 { /* 若为数组类型 */
103     if (node.num->i_val <= 0)
104         std::cout << "ERROR: 数组长度必须大于0\n";
105     /* 获取需开辟的对应大小的空间的类型指针 */
106     auto *arrayType = ArrayType::get(tmpType, node.num-
>i_val); /* 获取对应的数组Type */
107     auto initializer = CONST_ZERO(tmpType);
108     /* 全局变量初始化为0 */
109     value *arrayAlloca;
110     /* 存储申请到的数组空间的地址 */
111     if (scope.in_global())
112         /* 若为全局数组, 则开辟全局数组 */
113         arrayAlloca = GlobalVariable::create(node.id,
module.get(), arrayType, false, initializer);
114     else /* 若不是全局数组, 则开辟局部数组 */
115         arrayAlloca = builder->create_alloca(arrayType);
116     scope.push(node.id, arrayAlloca); /* 将获得的数组变量加入域
*/
117 }
118 else
119 { /* 若不是数组类型
*/
120     auto initializer = CONST_ZERO(tmpType); /* 全局变量初始化为0 */

```

```

118         value *varAlloca;                                /* 存储申请到的变
量空间的地址 */
119         if (scope.in_global())                            /* 若为全局变量，
则申请全局空间 */
120             varAlloca = GlobalVariable::create(node.id,
module.get(), tmpType, false, initializer);
121         else /* 若不是全局变量，则申请局部空间 */
122             varAlloca = builder->create_alloca(tmpType);
123         scope.push(node.id, varAlloca); /* 将获得变量加入域 */
124     }
125 }
126
127 /* Fun-Declaration, 函数声明, fun-declaration -> type-specifier
ID ( params ) compound-stmt */
128 void CminusfBuilder::visit(ASTFunDeclaration &node)
129 {
130     // ASTFunDeclaration有两个成员变量:
131     // std::vector<std::shared_ptr<ASTParam>> params;
132     // std::shared_ptr<ASTCompoundStmt> compound_stmt;
133
134     LOG(DEBUG) << "FunDeclaration\n";
135     // 考虑函数返回类型+函数名+参数列表+复合语句（局部声明与语句列表）
136
137     Type *retType; /* 函数返回类型 */
138     /* 根据不同的返回类型，设置retType */
139     if (node.type == TYPE_INT)
140     {
141         retType = Int32Type;
142     }
143     else if (node.type == TYPE_FLOAT)
144     {
145         retType = FloatType;
146     }
147     else if (node.type == TYPE_VOID)
148     {
149         retType = Type::get_void_type(module.get());
150     }
151
152     /* 根据函数声明，构造形参列表（此处的形参即参数的类型） */
153     std::vector<Type *> paramsType; /* 参数类型列表 */
154     for (auto param : node.params)
155     {

```

```

156         if (param->isarray)
157         {
158             /* 若参数为数组形式，则存
159             入首地址指针 */
160             if (param->type == TYPE_INT) /* 若为整型 */
161             {
162                 paramsType.push_back(Type::get_int32_ptr_type(module.get()));
163             }
164             else if (param->type == TYPE_FLOAT) /* 若为浮点型 */
165             {
166                 paramsType.push_back(Type::get_float_ptr_type(module.get()));
167             }
168             else
169             {
170                 /* 若为单个变量形式，则存
171                 入对应类型 */
172                 if (param->type == TYPE_INT) /* 若为整型 */
173                 {
174                     paramsType.push_back(Int32Type);
175                 }
176                 else if (param->type == TYPE_FLOAT) /* 若为浮点型 */
177                 {
178                     paramsType.push_back(FloatType);
179                 }
180             }
181         }
182         auto funType = FunctionType::get(retType, paramsType);
183         /* retType返回结构，paramsType函数形参结构 */
184         auto function = Function::create(funType, node.id,
185         module.get()); /* 创建函数 */
186         scope.push(node.id, function);
187         /* 将函数加入到全局作用域 */
188         scope.enter();
189         /* 进入此函数作用域 */
190         auto bb = BasicBlock::create(module.get(), node.id +
191         "_entry", function); /* 创建基本块 */
192         builder->set_insert_point(bb);
193         /* 将基本块加入到builder中 */
194         /* 函数传参，要将实参和形参进行匹配 */
195         std::vector<Value*> args; /* 创建vector存储实参 */
196         for (auto arg = function->arg_begin(); arg != function-
197         >arg_end(); arg++)
198         {
199             /* 遍历实参列表 */
200             args.push_back(*arg); /* 将实参加入vector */
201         }
202         for (int i = 0; i < node.params.size(); i++)
203         {
204             /* 遍历形参列表 (=遍历实参列
205             表) */
206             auto param = node.params[i]; /* 取出对应形参 */

```

```

186         arg = args[i];                /* 取出对应实参 */
187         param->accept(*this);          /* 调用param的accept进行处理
    */
188     }
189     node.compound_stmt->accept(*this); /* 处理函数体内语句
compound-stmt */
190     // 判断返回值的类型，根据对应的返回值类型，执行ret操作
191     // 这里应该是实现若没有显式返回，默认返回对应类型的0或void
192     if (builder->get_insert_block()->get_terminator() ==
nullptr)
193     {
194         if (function->get_return_type()->is_void_type())
195             builder->create_void_ret();
196         else if (function->get_return_type()->is_float_type())
197             builder->create_ret(CONST_FP(0.0));
198         else
199             builder->create_ret(CONST_INT(0));
200     }
201     scope.exit(); /* 退出此函数作用域 */
202 }
203
204 /* Param, 参数 */
205 void CminusBuilder::visit(ASParam &node)
206 {
207     // ASParam有3个成员变量
208     // CminusType type;      参数类型
209     // std::string id;        参数名
210     // bool isarray;          是否数组标记
211     LOG(DEBUG) << "Param\n";
212
213     value *paramAlloca; // 分配该参数的存储空间
214     // 区分是否为数组，为整型还是浮点型，共分为4类讨论
215     if (node.isarray)
216     {
217         /* 若为数组 */
218         if (node.type == TYPE_INT) /* 若为整型数组，则开辟整型数组存
存储空间 */
219             paramAlloca = builder-
>create_alloca(Type::get_int32_ptr_type(module.get()));
220         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 */
221             paramAlloca = builder-
>create_alloca(Type::get_float_ptr_type(module.get()));

```

```

221     }
222     else
223     {
224         if (node.type == TYPE_INT) /* 若为整型，则开辟整型存储空间
225         */
226             paramAlloca = builder->create_alloca(Int32Type);
227         else if (node.type == TYPE_FLOAT) /* 若为浮点数，则开辟浮点
228         数存储空间 */
229             paramAlloca = builder->create_alloca(FloatType);
230     }
231     builder->create_store(arg, paramAlloca); /* 将实参load到开辟的
232     存储空间中 */
233     // 此处arg通过全局变量传递、
234     // 函数原型: StoreInst *create_store(Value *val, Value *ptr)
235     scope.push(node.id, paramAlloca); /* 将参数push到域中 */
236 }
237
238 /* CompoundStmt, 函数体语句, compound-stmt -> {local-declarations
239 statement-list} */
240 void CminusFBuilder::visit(ASTCompoundStmt &node)
241 {
242     // ASTCompoundStmt有两个成员变量
243     // std::vector<std::shared_ptr<ASTVarDeclaration>>
244     local_declarations;
245     // std::vector<std::shared_ptr<ASTStatement>>
246     statement_list;
247
248     LOG(DEBUG) << "CompoundStmt\n";
249     scope.enter(); /*
250     进入函数体内的作用域 */
251     for (auto local_declaration : node.local_declarations) /*
252     遍历 */
253         local_declaration->accept(*this); /*
254     处理每一个局部声明 */
255     for (auto statement : node.statement_list) /*
256     遍历 */
257         statement->accept(*this); /*
258     处理每一个语句 */
259     scope.exit(); /*
260     退出作用域 */
261 }

```

```

251  /* ExpressionStmt, 表达式语句, expression-stmt -> expression;
252      *
253  void CminusBuilder::visit(ASTExpressionStmt &node)
254  {
255      // ASTExpressionStmt只有一个成员变量:
256      // std::shared_ptr<ASTExpression> expression;
257
258      LOG(DEBUG) << "ExpressionStmt\n";
259
260      if (node.expression != nullptr)      /* 若对应表达式存在 */
261          node.expression->accept(*this); /* 则处理该表达式 */
262  }
263
264  /* SelectionStmt, if语句, selection-stmt -> if (expression)
265      *
266      statement
267      | if (expression)
268      statement else statement */
269  void CminusBuilder::visit(ASTSelectionStmt &node)
270  {
271      // ASTSelectionStmt有3个成员变量:
272      // std::shared_ptr<ASTExpression> expression;      判断条件
273      // std::shared_ptr<ASTStatement> if_statement;      if语句执
274      // 行体
275      // std::shared_ptr<ASTStatement> else_statement;      else语句
276      // 执行体
277      // 若无else语句, else_statement == nullptr
278
279      LOG(DEBUG) << "SelectionStmt\n";
280
281      // 构建判断条件代码
282      node.expression->accept(*this); /* 处理条件判断对应的表达式, 得到
283      返回值存到expression中 */
284
285      auto resType = Res->get_type(); /* 获取表达式得到的结果类型 */
286      if (resType->is_pointer_type())
287      {
288          /* 若结果为指针型, 则针对
289          指针型进行处理 */
290          Res = builder->create_load(Res); /* 大于0视为true */
291      }
292      else if (resType->is_integer_type())
293      {
294          /* 若结果为整型, 则针对整型进行处理(bool类型视为整型) */

```

```

285         Res = builder->create_icmp_gt(Res,
CONST_ZERO(Int32Type)); /* 大于0视为true */
286     }
287     else if (resType->is_float_type())
288     {
        /* 若结果为浮点型，则针对浮点数进行处理 */
289         Res = builder->create_fcmp_gt(Res,
CONST_ZERO(FloatType)); /* 大于0视为true */
290     }
291     auto function = builder->get_insert_block()->get_parent();
        /* 获得当前所对应的函数 */
292
293     auto trueBB = BasicBlock::create(module.get(), "true",
function); /* 创建符合条件块 */
294
295     // 构建语句执行体代码
296     // 根据是否存在else语句分2类讨论
297     if (node.else_statement != nullptr)
298     {
        /* 若存在else语句 */
299         auto falseBB = BasicBlock::create(module.get(),
"false", function); /* 创建else块 */
300         builder->create_cond_br(Res, trueBB, falseBB);
        /* 设置跳转语句 */
301         // 处理trueBB
302         builder->set_insert_point(trueBB); /* 符合if
条件的块 */
303         node.if_statement->accept(*this); /* 处理if
语句执行体 */
304         auto curTrueBB = builder->get_insert_block(); /* 将块加
入 */
305         // 处理falseBB
306         builder->set_insert_point(falseBB); /* else
的块 */
307         node.else_statement->accept(*this); /* 处理
else语句执行体 */
308         auto curFalseBB = builder->get_insert_block(); /* 将块加
入 */
309
310         /* 处理返回，避免跳转到对应块后无return */
311         // 判断true语句中是否存在ret语句

```



```

312         auto trueTerm = builder->get_insert_block()-
>get_terminator();
313         // 判断else语句中是否存在ret语句
314         auto falseTerm = builder->get_insert_block()-
>get_terminator();
315
316         if (trueTerm == nullptr || falseTerm == nullptr)
317         { /* 若有一方不存在return语句，则需要创建返回块 */
318             BasicBlock *retBB;
319             retBB = BasicBlock::create(module.get(), "ret",
function); /* 创建return块 */
320             builder->set_insert_point(retBB);
321             /* return块（即后续语句） */
322             if (trueTerm == nullptr)                                /* 若if语
句执行体不存在return */
323                 builder->set_insert_point(curTrueBB); /* 则设置
跳转 */
324             }
325             else if (falseTerm == nullptr)
326             {                                                        /* 若
else语句执行体中不存在return */
327                 builder->set_insert_point(curFalseBB); /* 则设置
跳转 */
328             }
329             builder->create_br(retBB); /* 跳转到刚刚设置的return块
*/
330         }
331     }
332     else
333     {
334         /* 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 */
335         auto retBB = BasicBlock::create(module.get(), "ret",
function); /* 后续语句块 */
336         builder->create_cond_br(Res, trueBB, retBB);
337         /* 根据条件设置跳转指令 */
338
339         builder->set_insert_point(trueBB);
340         /* true语句块 */
341         node.if_statement->accept(*this);
342         /* 执行条件符合后要执行的语句 */

```

```

339         if (builder->get_insert_block()->get_terminator() ==
nullptr) /* 补充return (同上) */
340             builder->create_br(retBB);
            /* 跳转到刚刚设置的return块 */
341         builder->set_insert_point(retBB);
            /* return块 (即后续语句) */
342     }
343 }
344
345 /* IterationStmt, while语句, iteration-stmt -> while
(expression) statement */
346 void CminusBuilder::visit(ASIterationStmt &node)
347 {
348     // ASIterationStmt有2个成员变量:
349     // std::shared_ptr<ASTExpression> expression;    循环判断条件
350     // std::shared_ptr<ASTStatement> statement;      while循环执
行体
351
352     LOG(DEBUG) << "IterationStmt\n";
353     auto function = builder->get_insert_block()->get_parent();
            /* 获得当前所对应的函数 */
354     auto conditionBB = BasicBlock::create(module.get(),
"condition", function); /* 创建条件判断块 */
355     auto loopBB = BasicBlock::create(module.get(), "loop",
function); /* 创建循环语句块 */
356     auto retBB = BasicBlock::create(module.get(), "ret",
function); /* 创建后续语句块 */
357     if (builder->get_insert_block()->get_terminator() ==
nullptr)
358         builder->create_br(conditionBB); // 跳转到条件判断块
359
360     // 构建条件判断代码
361     builder->set_insert_point(conditionBB); /* 条件判断块 */
362     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 */
363     auto resType = Res->get_type(); /* 获取表达式得到的结果
类型 */
364     if (resType->is_pointer_type())
365     { /* 若结果为指针型, 则针对
指针型进行处理 */
366         Res = builder->create_load(Res); /* 大于0视为true */
367     }

```

```

368     else if (resType->is_integer_type())
369     {
        /* 若结果为整型，则针对整型进行处理(bool类型视为整型) */
370         Res = builder->create_icmp_gt(Res,
CONST_ZERO(Int32Type)); /* 大于0视为true */
371     }
372     else if (resType->is_float_type())
373     {
        /* 若结果为浮点型，则针对浮点数进行处理 */
374         Res = builder->create_fcmp_gt(Res,
CONST_ZERO(FloatType)); /* 大于0视为true */
375     }
376     builder->create_cond_br(Res, loopBB, retBB); /* 设置条件跳转语
句 */
377
        // 构建循环语句代码
378     builder->set_insert_point(loopBB);
        /* 循环语句执行块 */
379     node.statement->accept(*this);
        /* 执行对应的语句 */
380     if (builder->get_insert_block()->get_terminator() ==
nullptr) /* 若无返回，则补充跳转 */
381         builder->create_br(conditionBB);
        /* 跳转到条件判断语句 */
382
383
384     builder->set_insert_point(retBB); /* return块（即后续语句） */
385 }
386
387 /* ReturnStmt, 返回语句, return-stmt -> return;
388 *          | return expression; */
389 void CminusfBuilder::visit(ASTReturnStmt &node)
390 {
391     // ASTReturnStmt只有1个成员变量
392     // std::shared_ptr<ASTExpression> expression;    返回语句表达式
393
394     LOG(DEBUG) << "ReturnStmt\n";
395     auto function = builder->get_insert_block()->get_parent();
        /* 获得当前所对应的函数 */
396     auto retType = function->get_return_type();
        /* 获取返回类型 */
397
398     // 处理返回语句表达式

```

```

399     if (retType->is_void_type())
400     {
401         // 如果本来就是void类型的函数，需要返回void
402         builder->create_void_ret(); /* 则创建void返回，随后
return，无需后续操作 */
403         return;
404     }
405     // 如果需要返回非void
406     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 */
407     auto resType = Res->get_type(); /* 获取表达式得到的结果类型 */
408     /* 处理expression返回的结果和需要return的结果类型不匹配的问题 */
409     // 使用强制类型转换
410     if (retType->is_integer_type() && resType->is_float_type())
411         Res = builder->create_fptosi(Res, Int32Type);
412     if (retType->is_float_type() && resType->is_integer_type())
413         Res = builder->create_sitofp(Res, Int32Type);
414     builder->create_ret(Res); /* 创建return，将expression的结果进
行返回 */
415 }
416
417 /* Var，变量引用，var -> ID
*
* | ID [expression] */
418 void CminusFBuilder::visit(ASTVar &node)
419 {
420     // ASTVar有2个成员变量：
421     // std::string id; 变量名
422     // std::shared_ptr<ASTExpression> expression; 变量类型
423     // 如果变量var是int型的，这个取nullptr
424
425     LOG(DEBUG) << "var\n";
426     auto var = scope.find(node.id); /* 从域中取出对应变量
*/
427     bool if_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） */
428     need_as_address = false; /* 重置全局变量
need_as_address */
429     value *index = CONST_INT(0); /* 初始化index */
430     if (node.expression != nullptr)
431     { /* 若
有expression，代表不是int类型的引用*/

```

```

443         node.expression->accept(*this);                /* 处
理expression, 得到结果Res */
444         auto res = Res;                                /* 存
储结果 */
445         if (checkFloat(res))                            /* 判
断结果是否为浮点数 */
446             res = builder->create_fptosi(res, Int32Type); /* 若
是, 则矫正为整数 */
447         index = res;                                    /* 赋
值给index, 表示数组下标 */
448         /* 判断下标是否为负数。若是, 则调用neg_idx_except函数进行处理
*/
449         auto function = builder->get_insert_block()-
>get_parent();                /* 获取当前函数 */
450         auto indexTest = builder->create_icmp_lt(index,
CONST_ZERO(Int32Type));        /* test是否为负数 */
451         auto failBB = BasicBlock::create(module.get(), node.id
+ "_failTest", function); /* fail块 */
452         auto passBB = BasicBlock::create(module.get(), node.id
+ "_passTest", function); /* pass块 */
453         builder->create_cond_br(indexTest, failBB, passBB);
/* 设置跳转语句 */
454
455         // 下标为负数, 调用neg_idx_except函数进行处理
456         builder->set_insert_point(failBB);
/* fail块, 即下标为负数 */
457         auto fail = scope.find("neg_idx_except");
/* 取出neg_idx_except函数 */
458         builder->create_call(static_cast<Function *>(fail),
{}); /* 调用neg_idx_except函数进行处理 */
459         builder->create_br(passBB);
/* 跳转到pass块 */
460
461         // 下标合法,
462         builder->set_insert_point(passBB);
/* pass块 */
463         if (var->get_type()->get_pointer_element_type()-
>is_array_type()) /* 若为指向数组的指针 */
464             var = builder->create_gep(var, {CONST_INT(0),
index}); /* 则进行两层寻址 */
465         else
466         {

```

```

457         if (var->get_type()->get_pointer_element_type()-
>is_pointer_type()) /* 若为指针 */
458             var = builder->create_load(var);
                     /* 则取出指针指向的元素 */
459             var = builder->create_gep(var, {index});
                     /* 进行一层寻址（因为此时并非指向数组） */
460         }
461         if (if_return_lvalue)
462         {
                     /* 若要返回值 */
463             Res = var;
                     /* 则返回var对应的地址 */
464             need_as_address = false; /* 并重置全局变量
need_as_address */
465         }
466         else
467             Res = builder->create_load(var); /* 否则则进行load */
468         return;
469     }
470     else
471     { // 处理不是数组的情况
472         if (if_return_lvalue)
473         {
                     /* 若要返回值 */
474             Res = var;
                     /* 则返回var对应的地址 */
475             need_as_address = false; /* 并重置全局变量
need_as_address */
476         }
477         else
478         { /* 否则 */
479             // 数组的指针即a[]类型就返回数组的起始地址，否则load取值
480             if (var->get_type()->get_pointer_element_type()-
>is_array_type()) /* 若指向数组，需要两个偏移取地址 */
481                 Res = builder->create_gep(var, {CONST_INT(0),
CONST_INT(0)}); /* 则寻址 */
482             else
483                 Res = builder->create_load(var); /* 否则则进行
load */
484         }
485     }
486 }
487
488 /* AssignExpression, 赋值语句, var = expression */
489 void CminusBuilder::visit(ASTAssignExpression &node)
490 {

```

```

491 // ASTAssignExpression有2个成员变量
492 // std::shared_ptr<ASTVar> var;           左值（被赋值的
对象）
493 // std::shared_ptr<ASTExpression> expression;  右值（赋的值）
494
495 LOG(DEBUG) << "AssignExpression\n";
496 need_as_address = true; /* 设置need_as_address, 表示需要返回值
*/
497
498 // 获取左值, 右值
499 node.var->accept(*this);           /* 处理左var */
500 auto left = Res;                   /* 获取地址 */
501 node.expression->accept(*this); /* 处理右expression */
502 auto right = Res;                  /* 获得结果 */
503
504 // 处理左值, 右值类型冲突问题
505 auto leftType = left->get_type()-
>get_pointer_element_type(); /* 获取var的类型 */
506 /* 若赋值语句左右类型不匹配, 则进行匹配 */
507 if (leftType == FloatType && checkInt(right))
508     right = builder->create_sitofp(right, FloatType);
509 if (leftType == Int32Type && checkFloat(right))
510     right = builder->create_fptosi(right, Int32Type);
511
512 // 赋值
513 builder->create_store(right, left);
514 }
515
516 /* SimpleExpression, 比较表达式, simple-expression -> additive-
expression relop additive-expression
517 *                                     | additive-
expression */
518 void CminusBuilder::visit(ASTSimpleExpression &node)
519 {
520     // ASTSimpleExpression有3个成员变量
521     // std::shared_ptr<ASTAdditiveExpression>
additive_expression_l;
522     // std::shared_ptr<ASTAdditiveExpression>
additive_expression_r;
523     // Relop op;
524
525     LOG(DEBUG) << "SimpleExpression\n";

```

```

526     node.additive_expression_l->accept(*this); /* 处理左边的
expression */
527     auto lres = Res; /* 获取结果存到
lres中 */
528     if (node.additive_expression_r == nullptr)
529     {
530         return;
531     } /* 若不存在右
expression, 则直接返回 */
532     node.additive_expression_r->accept(*this); /* 处理右边的
expression */
533     auto rres = Res; /* 结果存到rres中
*/
534
535     // 确保两个表达式的类型相同, 若存在浮点和整型混存, 全部转换为浮点型
536     if (checkInt(lres) && checkInt(rres))
537     { /* 确保两边都是整数 */
538         switch (node.op)
539         {
540             /* 根据不同的比较操作, 调用icmp进行处理 */
541             // 比较的返回结果
542             // enum RelOp:
543             // <= 对应 OP_LE,
544             // < 对应 OP_LT,
545             // > 对应 OP_GT,
546             // >= 对应 OP_GE,
547             // == 对应 OP_EQ,
548             // != 对应 OP_NEQ
549             case OP_LE:
550                 Res = builder->create_icmp_le(lres, rres);
551                 break;
552             case OP_LT:
553                 Res = builder->create_icmp_lt(lres, rres);
554                 break;
555             case OP_GT:
556                 Res = builder->create_icmp_gt(lres, rres);
557                 break;
558             case OP_GE:
559                 Res = builder->create_icmp_ge(lres, rres);
560                 break;
561             case OP_EQ:
562                 Res = builder->create_icmp_eq(lres, rres);

```



```

563         break;
564     case OP_NEQ:
565         Res = builder->create_icmp_ne(lres, rres);
566         break;
567     }
568 }
569 else
570 {
571     /* 若有一边是浮点类型，则需要先将另一边也转
572     为浮点数，再进行比较 */
573     if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
574         lres = builder->create_sitofp(lres, FloatType);
575     if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
576         rres = builder->create_sitofp(rres, FloatType);
577     switch (node.op)
578     {
579     /* 根据不同的比较操作，调用fcmp进行处理 */
580     case OP_LE:
581         Res = builder->create_fcmp_le(lres, rres);
582         break;
583     case OP_LT:
584         Res = builder->create_fcmp_lt(lres, rres);
585         break;
586     case OP_GT:
587         Res = builder->create_fcmp_gt(lres, rres);
588         break;
589     case OP_GE:
590         Res = builder->create_fcmp_ge(lres, rres);
591         break;
592     case OP_EQ:
593         Res = builder->create_fcmp_eq(lres, rres);
594         break;
595     case OP_NEQ:
596         Res = builder->create_fcmp_ne(lres, rres);
597         break;
598     }
599     Res = builder->create_zext(Res, Int32Type); /* 将结果作为整数
600     保存（可作为判断语句中的判断条件） */
601 }
602 /* AdditiveExpression, 加法表达式, additive-expression ->
603 additive-expression addop term

```

```

602      *                                     | term
        */
603 void CminusfBuilder::visit(ASTAdditiveExpression &node)
604 {
605     // ASTAdditiveExpression有3个成员变量:
606     // std::shared_ptr<ASTAdditiveExpression>
    additive_expression;
607     // AddOp op;
608     // std::shared_ptr<ASTTerm> term;
609
610     LOG(DEBUG) << "AdditiveExpression\n";
611     if (node.additive_expression == nullptr)
612     { /* 若无加减法运算 */
613         node.term->accept(*this);
614         return; /* 则直接去做乘除法 */
615     }
616     node.additive_expression->accept(*this); /* 处理左expression
    */
617     auto lres = Res; /* 结果保存在lres中
    */
618     node.term->accept(*this); /* 处理右term */
619     auto rres = Res; /* 结果保存在rres中
    */
620
621     // 分为整型-整型, 和存在浮点数类型, 这两种情况进行讨论
622     // 若存在浮点数, 则全部强制转换为浮点数实现
623     if (checkInt(lres) && checkInt(rres))
624     { /* 确保两边都是整数 */
625         switch (node.op)
626         { /* 根据对应加法或是减法, 调用iadd或是isub进行处理 */
627             case OP_PLUS:
628                 Res = builder->create_iadd(lres, rres);
629                 break;
630             case OP_MINUS:
631                 Res = builder->create_isub(lres, rres);
632                 break;
633         }
634     }
635     else
636     { /* 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 */
637         if (checkInt(lres)) /* 若左边是整数, 则将其转为浮点数 */

```

```

638         lres = builder->create_sitofp(lres, FloatType);
639         if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
640             rres = builder->create_sitofp(rres, FloatType);
641         switch (node.op)
642         { /* 根据对应加法或是减法，调用fadd或是fsub进行处理 */
643         case OP_PLUS:
644             Res = builder->create_fadd(lres, rres);
645             break;
646         case OP_MINUS:
647             Res = builder->create_fsub(lres, rres);
648             break;
649         }
650     }
651 }
652
653 /* Term, 乘除法语句, Term -> term mulop factor
654    *                               | factor */
655 void CminusfBuilder::visit(ASTTerm &node)
656 {
657     // ASTTerm有3个成员变量:
658     // std::shared_ptr<ASTTerm> term;
659     // MulOp op;
660     // std::shared_ptr<ASTFactor> factor;
661
662     LOG(DEBUG) << "Term\n";
663     if (node.term == nullptr)
664     { /* 若无乘法运算 */
665         node.factor->accept(*this);
666         return; /* 则直接去处理元素 */
667     }
668     node.term->accept(*this); /* 处理左term */
669     auto lres = Res; /* 结果保存在lres中 */
670     node.factor->accept(*this); /* 处理右factor */
671     auto rres = Res; /* 结果保存在rres中 */
672     if (checkInt(lres) && checkInt(rres))
673     { /* 确保两边都是整数 */
674         switch (node.op)
675         { /* 根据对应乘法或是除法，调用imul或是idiv进行处理 */
676         case OP_MUL:
677             Res = builder->create_imul(lres, rres);
678             break;
679         case OP_DIV:

```

```

680         Res = builder->create_isdiv(lres, rres);
681         break;
682     }
683 }
684 else
685 {
686     /* 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 */
687     if (checkInt(lres)) /* 若左边是整数，则将其转为浮点数 */
688         lres = builder->create_sitofp(lres, FloatType);
689     if (checkInt(rres)) /* 若右边是整数，则将其转为浮点数 */
690         rres = builder->create_sitofp(rres, FloatType);
691     switch (node.op)
692     { /* 根据对应乘法或是除法，调用fmul或是fddiv进行处理 */
693     case OP_MUL:
694         Res = builder->create_fmul(lres, rres);
695         break;
696     case OP_DIV:
697         Res = builder->create_fdiv(lres, rres);
698         break;
699     }
700 }
701
702 /* Call, 函数调用, call -> ID (args) */
703 void CminusBuilder::visit(ASTCall &node)
704 {
705     // ASTCall有3个成员变量:
706     // std::string id;
707     // std::vector<std::shared_ptr<ASTExpression>> args;
708
709     LOG(DEBUG) << "Call\n";
710     auto function = static_cast<Function *>
(scope.find(node.id)); /* 获取需要调用的函数 */
711     auto paramType = function->get_function_type()-
>param_begin(); /* 获取其参数类型 */
712     if (function == nullptr)
713         std::cout << "ERROR: 函数" << node.id << "未定义\n";
714
715     // 处理参数列表
716     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 */
717     for (auto arg : node.args)

```

```

718         {                                     /* 遍历形参列表 */
719             arg->accept(*this); /* 对每一个参数进行处理，获取参数对应的值
*/
720             if (Res->get_type()->is_pointer_type())
721             {                                     /* 若参数是指针 */
722                 args.push_back(Res); /* 则直接将值加入到参数列表 */
723             }
724             else
725             { /* 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 */
726                 if (*paramType == FloatType && checkInt(Res))
727                     Res = builder->create_sitofp(Res, FloatType);
728                 if (*paramType == Int32Type && checkFloat(Res))
729                     Res = builder->create_fptosi(Res, Int32Type);
730                 args.push_back(Res);
731             }
732             paramType++; /* 查看下一个形参 */
733         }
734         Res = builder->create_call(static_cast<Function *>
(function), args); /* 创建函数调用 */
735     }
736

```