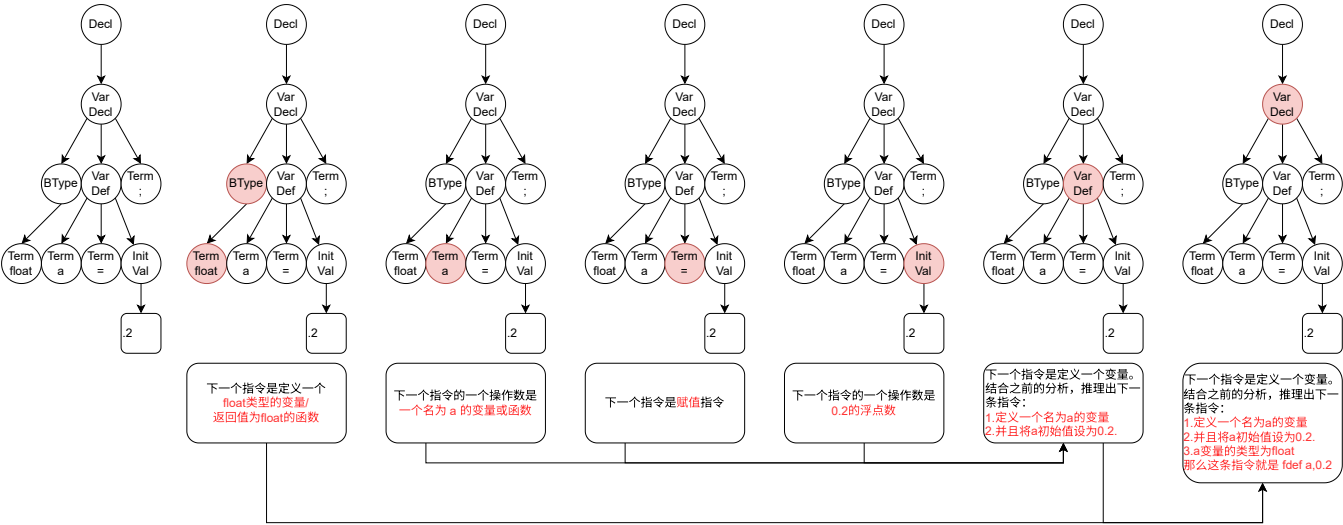


实验二

将实验1的AST抽象语法树转换成IR（中间表示）



数据结构

IR

实验对 SysY 语言设计了统一的 IR 框架，所有 IR 采用四元的形式，即：

`opcode, des, operand1, operand2`

每个IR表示可以在实验指导书中查到。举例来说，你需要将形如

```
1 | int a = 8;
```

转换为

```
1 | def a, 8
```

这种IR表示。

入口函数与基本数据结构

在 `main.cpp` 文件中，有

```
1 | frontend::Analyzer analyzer;
2 | auto program = analyzer.get_ir_program(node);
```

我们需要完成这个 `get_ir_program`

函数入口在这里， `src/front/semantic.cpp`

```
1 | ir::Program frontend::Analyzer::get_ir_program(CompUnit* root)
```

你可以发现这个函数的输入，是实验一分析得到的根节点 `CompUnit` 的指针。让你输出的就是一个 `ir::Program`。

来看一下 `ir::Program` 的数据结构

```
1 namespace ir
2 {
3     // ...
4     struct Program {
5         std::vector<Function> functions;
6         std::vector<GlobalVal> globalVal;
7         Program();
8         void addFunction(const Function& proc);
9         std::string draw();
10    };
11
12 }
```

总的来说，我们需要得知 `Program` 中的 `functions` 和 `globalVal` 两个变量。

当我们把一个程序的 `functions` 和 `globalVal` 都解析完成的时候，实验也就做完了！😊

好耶！一定很简单吧？🤖

Function

这个 `Function` 的数据结构到底是什么呢？请查看 `include/ir/ir_function.h`：

```
1 struct Function {
2     std::string name;
3     ir::Type returnType;
4     std::vector<Operand> ParameterList;
5     std::vector<Instruction*> InstVec;
6     Function();
7     Function(const std::string&, const ir::Type&);
8     Function(const std::string&, const std::vector<Operand>&, const ir::Type&);
9     void addInst(Instruction* inst);
10    std::string draw();
11 };
```

就跟我们的程序里面的函数（`main`函数，`add`函数等等）一样，通过这些命名可以读出来一个 `function` 必须要有

- `name`: 函数名
- `returnType`: 返回类型，数据结构是一个 `Type`。
- `ParameterList`: 参数列表，数据结构是一个 `Operand` 的数组。
- `InstVec`: 指令的列表，数据结构是一个 `Instruction` 指针的数组。

不知道你们还记不记得，在实验一中有节点类型专门就是用来解析名字、返回类型的，是什么呢，好难猜啊？

```
FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block
```

Operand

顾名思义，**操作数**。

还记得刚才我们IR的具体表示为一个四元组吗？`opcode, des, operand1, operand2`

`opcode`很好理解，无非就是一个操作的枚举类型，比如有`add, store, ...`

这里面其余三个，`op1`、`op2`和`des`都是操作数。

```
1 namespace ir {
2
3 enum class Type {
4     Int,
5     Float,
6     IntLiteral,
7     FloatLiteral,
8     IntPtr,
9     FloatPtr,
10    null
11 };
12
13 std::string toString(Type t);
14
15 struct Operand {
16     std::string name;
17     Type type;
18     Operand(std::string = "null", Type = Type::null);
19 };
20
21 }
```

一个 Operand 有它的类型和它的名字

类型`type`:

- `Int`: 整型。比如 `int x` 这里面 `x` 就是一个整型。
- `Float`: 浮点型。
- `IntLiteral`: 整型字面量。比如 `1, 2, 1024` 这种。
- `FloatLiteral`: 浮点型字面量。
- `IntPtr`: 整型指针，数组需要用。
- `FloatPtr`: 浮点型指针。

比如 `int x = 2;` 这条指令，`x` 是一个 `Int` 的操作数，`1` 是一个 `IntLiteral` 的操作数。

Instruction

怎么定义一条指令？在 `ir_instruction.h` 中有：

```

1 struct Instruction {
2     Operand op1; // 操作数1
3     Operand op2; // 操作数2
4     Operand des; // 结果
5     Operator op; // 操作
6     Instruction();
7     Instruction(const Operand& op1, const Operand& op2, const Operand& des, const
      Operand& op);
8     virtual std::string draw() const;
9 };

```

就跟刚才的四元组一样，我们需要照着这个来生成指令。

比如：

```

1 int x = a + b; // 假设a, b 都是int类型

```

这条指令，你就需要构造类似这样的Instruction

```

1 auto inst = ir::Instruction{Operand{"a", Type::Int}, Operand{"b", Type::Int},
      Operand{"x", Type::Int}, Operator::add}

```

回顾一下，一个Function有它的name，returnType，参数列表和一堆指令。而这些数据我们现在都可以在某些节点获得，通过这些节点来完成这个function，我们的任务就完成了。

globalVal

全局变量，定义为这样：

```

1 struct GlobalVal
2 {
3     ir::Operand va;
4     int maxlen = 0; //为数组长度设计
5     GlobalVal(ir::Operand va);
6     GlobalVal(ir::Operand va, int len);
7 };

```

有两个数据，一个操作数，和一个maxlen，记录数组长度，当不是数组时值为0.

全局变量，顾名思义，我们可以在全局中访问到这些操作数。与之一个相关的概念是作用域。

scope（作用域）

回顾一下程序设计基础，假设有这么一段程序

```

1 int main() {
2     int a = 1;
3     if (true) {
4         int a = 3;
5         while (true) {
6             if (a == 5) {

```

```

7         break;
8     }
9     a++;
10 }
11 }
12 cout << a << endl;
13 return 0;
14 }

```

这里打印的值会是多少？运行程序跑出来是 `1`，这是因为作用域的缘故。

在进入一个block（大括号括起来的区域），变量的作用域会发生变更，也就是我们讨论变量的上下文发生了切换。

执行 `int a = 3` 时，已经是在一个新的作用域发生的事情了，直到遇到大括号的末尾退出block，才退回到了原来的作用域。

仔细想想，这跟我们学过的一个数据类型很相似，当遇到一个 `{` 进入新的作用域，访问变量时去获取这个变量存在的最新作用域。

没错，这是一个**栈**。

在程序中，有一个数据结构叫**符号表**，我们可以在这里看到相关的数据和函数：

```

1 // definition of symbol table
2 struct SymbolTable {
3     vector<ScopeInfo> scope_stack;
4     map<std::string, ir::Function*> functions;
5     int scope_cnt;
6
7     void add_scope();
8     void exit_scope();
9
10    string get_scoped_name(string id) const;
11    ir::Operand get_operand(string id) const;
12    STE get_ste(string id) const;
13    void add_ste(const string& id, STE ste);
14 };

```

这里的scope_stack就是一个作用域的栈。

符号表是一个在实验二中重要的数据结构，请根据以上内容自行研究。

算法

该从哪里入手

我们最终要写的函数：

```

1  ir::Program frontend::Analyzer::get_ir_program(CompUnit* root){
2      ir::Program program;
3      // do something
4      // ...
5      reutrn program;
6  }

```

不妨我们先来试一下这个程序

```

1  int main(){
2      return 3;
3  }

```

它的AST应该是

```

1  {
2      "name" : "CompUnit",
3      "subtree" : [
4          {
5              "name" : "FuncDef",
6              "subtree" : [
7                  {
8                      "name" : "FuncType",
9                      "subtree" : [
10                     //...

```

一开始，传进来的时候一个CompUnit*，它的文法是这样的

```

1  // CompUnit -> (Decl | FuncDef) [CompUnit]

```

我们要去分析这个节点，就和实验一一样，整个过程是递归的。

我们定义一个 `analysisCompUnit` 函数：

```

1  // CompUnit -> (Decl | FuncDef) [CompUnit]
2  void frontend::Analyzer::analysisCompUnit(CompUnit* root,
3      ir::Program& program) {
4      if (Decl* node = dynamic_cast<Decl*>(root->children[0])) { // Decl
5          vector<Instruction*> decl_insts;
6          analysisDecl(node, decl_insts);
7          for (auto& inst : decl_insts) {
8              symbol_table.functions["global"]->addInst(inst);
9          }
10     } else if (FuncDef* node = dynamic_cast<FuncDef*>(root->children[0])) {
11         Function* new_func = new Function();
12         analysisFuncDef(node, new_func);
13         program.addFunction(*new_func);
14     } else {
15         assert(0 && "Unknown node type");
16     }
17     if (root->children.size() == 2) {

```

```

18     GET_CHILD_PTR(child_comp, CompUnit, 1);
19     analysisCompUnit(child_comp, program);
20 }
21 }

```

`root->children[0]` 只有两种情况，一个是 `Decl` 节点，一个是 `FuncDef` 节点。我们该怎么做判断，`dynamic_cast<Decl*>(root->children[0])` 这种方式，其含义是，将 `root->children[0]` 动态转换为 `Decl*` 的类型，如果不可以转换，`node` 则是一个 `nullptr`，`nullptr` 的 `bool` 值为 `false`。所以等价于做了一个类型判断。

回到程序中，我们的程序在这一个节点会得到其第一个子节点的类型是 `FuncDef`，于是在这里进入第二个逻辑分支。此时我们要主动 `new` 一个新的 `function`（因为已经要函数定义了），我们继续写 `analysisFuncDef` 函数。

```

1 // FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block
2 void frontend::Analyzer::analysisFuncDef(frontend::FuncDef* root,
3     Function* func) {
4     symbol_table.add_scope();
5     GET_CHILD_PTR(func_type, FuncType, 0);
6     GET_CHILD_PTR(ident, Term, 1);
7     GET_IDENFR_NAME(id, ident);
8     // ...
9     symbol_table.exit_scope();
10 }

```

在 `semantic.cpp` 中，写好了几个宏可以调用，比如：

```

1 #define GET_CHILD_PTR(node, type, index) \
2     auto node = static_cast<type*>(root->children[index])

```

`GET_CHILD_PTR(func_type, FuncType, 0)`；其含义是，我需要把第一个子节点赋值到 `func_type` 这个变量里，其类型是 `FuncType`

此时 `func_type` 就是一个节点指针了。

还有一个宏是：

```

1 #define GET_IDENFR_NAME(_id, _term) \
2     std::string _id; \
3     _id = _term->token.value

```

`GET_IDENFR_NAME(id, ident)`；其含义是，我需要把 `ident`（是一个 `Term*`）的值放在 `id` 这里，`id` 是一个 `string` 类型。

也就是说，通过分析 `GET_IDENFR_NAME`，我们获得了 `id` 其值为 `main`

你要写每一个节点的 `analyze` 函数，形如：

```

1 // analysis functions
2 void analysisCompUnit(CompUnit*, ir::Program&);
3 void analysisDecl(Decl*, vector<ir::Instruction*>&);
4 void analysisConstDecl(ConstDecl*, vector<ir::Instruction*>&);
5 void analysisVarDecl(VarDecl*, vector<ir::Instruction*>&);
6 void analysisConstDef(ConstDef*, ir::Type, vector<ir::Instruction*>&);
7 void analysisVarDef(VarDef*, ir::Type, vector<ir::Instruction*>&);

```

```

8 void analysisFuncDef(FuncDef*, ir::Function*);
9 void analysisFuncFParams(FuncFParams*, ir::Function*);
10 void analysisFuncFParam(FuncFParam*, ir::Function*);
11 void analysisBlock(Block*, vector<ir::Instruction*>&, bool);
12 void analysisBlockItem(BlockItem*, vector<ir::Instruction*>&);
13 void analysisStmt(Stmt*, vector<ir::Instruction*>&);
14
15 void analysisCond(Cond*, vector<ir::Instruction*>&);
16
17 Type analysisBType(BType*);
18 Type analysisFuncType(FuncType*);
19
20 void
21     analysisConstInitVal(ConstInitVal*, ir::Type, vector<ir::Instruction*>&);
22 void analysisInitVal(InitVal*, ir::Type, vector<ir::Instruction*>&);
23 void analysisConstExp(ConstExp*, vector<ir::Instruction*>&);
24 void analysisAddExp(AddExp*, vector<ir::Instruction*>&);
25 void analysisMulExp(MulExp*, vector<ir::Instruction*>&);
26 void analysisUnaryExp(UnaryExp*, vector<ir::Instruction*>&);
27 void
28     analysisFuncRParams(FuncRParams*, vector<ir::Instruction*>&, ir::Function*);
29 void analysisPrimaryExp(PrimaryExp*, vector<ir::Instruction*>&);
30 void analysisExp(Exp*, vector<ir::Instruction*>&);
31 void analysisLVal(LVal*, vector<ir::Instruction*>&, bool);
32 void analysisNumber(Number*);
33
34 void analysisLOrExp(LOrExp*, vector<ir::Instruction*>&);
35 void analysisLAndExp(LAndExp*, vector<ir::Instruction*>&);
36 void analysisEqExp(EqExp*, vector<ir::Instruction*>&);
37 void analysisRelExp(RelExp*, vector<ir::Instruction*>&);

```

其中有些接口参数是我的程序需要用的，可以根据实际情况调整。从 `analysisCompUnit` 函数自顶向下调用，过程中会依次分析到function, inst等等，把这些加入到program中，最后返回program即可。

一些处理

这些处理非必要，也可以有自己的实现方式。

函数作用域

正如刚才所说，当进入分析FuncDef节点时，函数就进入了一个新的作用域，这是因为函数参数也在这个作用域当中

```

1 int f(int a, int b){
2     int c = 1;
3     return 100;
4 }

```

当我进入一个函数定义的大括号时，便不再进入一个新的作用域，此时 a, b, c 在一个作用域里。


```

1 {
2     {"a", <传过来的值>},
3     {"b", <传过来的值>},
4     {"c", 100},
5 }

```

所以这里的

```

1 void analysisBlock(Block*, vector<ir::Instruction*>&, bool);

```

多了一个bool类型，表明是否是函数定义的block，如果是就不增加新的scope。除此以外，遇到一个 block 便加一个scope

外部库函数

因为测评需要根据输入来输出结果，所以要支持输入输出的外部库函数，你需要将这个map里面的function添加到的你的符号表中。

```

1 map<std::string, ir::Function*> frontend::get_lib_funcs() {
2     static map<std::string, ir::Function*> lib_funcs = {
3         {"getint", new Function("getint", Type::Int)},
4         {"getch", new Function("getch", Type::Int)},
5         {"getfloat", new Function("getfloat", Type::Float)},
6         {"getarray",
7             new Function("getarray", {Operand("arr", Type::IntPtr)}, Type::Int)},
8         {"getfarray",
9             new Function(
10                "getfarray", {Operand("arr", Type::FloatPtr)}, Type::Int)},
11         {"putint",
12             new Function("putint", {Operand("i", Type::Int)}, Type::null)},
13         {"putch", new Function("putch", {Operand("i", Type::Int)}, Type::null)},
14         {"putfloat",
15             new Function("putfloat", {Operand("f", Type::Float)}, Type::null)},
16         {"putarray",
17             new Function("putarray",
18                 {Operand("n", Type::Int), Operand("arr", Type::IntPtr)},
19                 Type::null)},
20         {"putfarray",
21             new Function("putfarray",
22                 {Operand("n", Type::Int), Operand("arr", Type::FloatPtr)},
23                 Type::null)},
24     };
25     return &lib_funcs;
26 }
27
28 frontend::Analyzer::Analyzer() : tmp_cnt{0}, symbol_table() {
29     symbol_table.add_scope();
30     ir::Function* global_func = new ir::Function("global", ir::Type::null);
31     symbol_table.functions.insert({"global", global_func});
32     auto lib_funcs = *get_lib_funcs();
33     for (const auto& pair : lib_funcs) {

```

```
34 |         symbol_table.functions.insert(pair);
35 |     }
36 | }
```

Global函数

参考指导书。请注意源程序中并没有一个叫做 `global` 的函数，是因为需要对全局变量进行初始化，所以采用了这样一个特殊的做法。

调用处理

IR生成需涉及对全局变量、全局常量的处理。一种可行的方法是将`global`作为一个 `function` 进行处理，除去其中变量、常量定义声明的 IR 外，仍需生成一条 `return null` 的 IR。并在 `main` 函数中首先生成对 `global` 的调用IR。