Compilers Principals - Lab3

Zhixin Zhang, 3210106357

1 实验内容

本次实验，我们基于 lab2 的语义分析， 实现了 SysY 语言向中间代码的转化． 我们基于先前构建出的语法树和符号表，对语法树上的每一个节点进行递归构建其对应的中间代码，我们使用变量名称和其在符号表上的位置来唯一标识一个变量，解决了变量重名的问题．通过：

```
make compiler
./compiler <input file> [output file]
```

可以对输入的 sy 文件进行语法和语义的检查， 如果可以正确解析出语法树并且通过类型检查和数组检查，程序将正常退出并返回 0，并且将生成中间代码到 output file（如果没有定义，则默认为 ir.out）， 同时在错误流中显示：

```
Parse success!
```

否则，程序将汇报错误，一个错误的代码的解析输出如下：

```
DEBUG: type error at src/semantic.hpp:219
DEBUG: type error at src/semantic.hpp:105
DEBUG: type error at src/semantic.hpp:39
```

报错信息表示语义分析错误在源程序中的位置，在这里，我们并未实现面向用户的报错信息，仅用于个人调试．

2 代码实现

2.1 主接口

main.cc 在 lab2 的基础上，增加了中间代码生成的部分：

```
string IR_OUT = "ir.out";
if(argc >= 3) IR_OUT = string(argv[2]);
IR ir(&checker, Root);
ofstream ir_out(IR_OUT);
ir.print(ir_out);
std::cerr << "\nParse success !" << std::endl;
```

2.2 中间代码存储格式

2.2.1 Class IR

我们以一行作为中间代码的最小存储单元，class IR 用于存储中间代码段，其中可能包括一行或多行中间代码，其成员定义如下（不含成员函数）：

```
  string type;
  unique_ptr<IR_info> info;
```

```
    vector<IR> child;
    IR() : info(nullptr) { type = "NULL"; }
    IR(IR_info *o) : type(o->type), info(unique_ptr<IR_info>(o)) { }
    IR(const IR& o) : type(o.type), info(o.info ? o.info->clone() : nullptr), child(o.child) {}
```

其中 type 表示该段中间代码的类型，如果它包括多行，则类型为 “NULL”. child 为每一行的中间代码. info 表示中间代码的具体信息（如果多行的话为 nullptr）. 这里我们使用指针来存储中间代码的信息，并且维护其深拷贝的操作.

2.2.2 Class IR_info

IR_info 用于描述一行代码输出时的具体信息. 其包括一个虚函数 print()，

```
virtual void print(ostream &OUT) const;
```

我们为不同类型的中间代码定义了不同的子类和相关的输出函数. 如二元运算赋值语句：

```
class info_assign_binary : public IR_info
{
public:
    IR_info* clone() const { return new info_assign_binary(*this); }
    string lv, v1, op, v2;
   info_assign_binary(string lv, string v1, string op, string v2) : lv(lv), v1(v1), op(op), v2(v2)
{ IR_info::type = "ASSIGN"; }
    void print(ostream &OUT) const { OUT << lv << " = " << (v1+" "+upd_op(op)+" "+v2) << "\n"; }
};
```

因此输出中间代码时，只需要从 IR 根节点开始，依次执行每一个 child 的输出函数即可.

2.3 中间代码生成

2.3.1 命名格式

临时变量的命名，代码中全局变量，局部变量的命名，中间代码中 Label 的命名格式参照以下函数生成：

```
    string get_label() { static int tot = 0; ++tot; return "L"+to_string(tot); }
    string get_tmp() { static int tot = 0; ++tot; return "irt3mP"+to_string(tot); }
    string get_name(Node* o) { return "irVar_xxx"+to_string(abs(o->val))+"_"+o->text; }
```

2.3.2 部分实现细节

- 变量定义

全局变量和局部变量对应的中间代码不同，所以需要额外传入一个参数 _inline，表示是否为局部变量.

```
IR from_decl(Node *o, bool _inline = 0);
```

- IR 节点声明

使用如下宏来表示一个新的 IR 节点：

```
#define INFO(type, ...) (new info_##type(__VA_ARGS__))
#define _IR(type, ...) IR(new info_##type(__VA_ARGS__))
```

这样可以更加方便地定义一个 IR 代码段，比如对于 IfElse 节点：

```
string l1 = get_label(), l2 = get_label(), l3 = get_label();
rt._with(from_condition(o->child[0], l1, l2), _IR(label, l1),
    from_stmt(o->child[1]), _IR(_goto, l3),
    _IR(label, l2), from_stmt(o->child[2]), _IR(label, l3));
```

- exp 节点

表达式需要用一个临时变量暂存，因此在转化 exp 节点的时候，还需要传入一个参数表示转换后的代码存在哪个变量里：

```
IR from_exp(Node* o, string tmp);
```

- 变量调用

对于数组变量/全局变量，调用的时候先取其地址，然后再 Load 其中的值，而对于局部变量，可以直接使用普通 Assign 语句来进行赋值操作.

```
if(/*It is an array or global variable*/)
    rt.merge(_IR(assign, tmp, get_name(id)));
else
{
    string t = get_tmp();
    rt._with(_IR(assign_addr, t, get_name(id)), _IR(assign_load, tmp, t));
}
```

3 测试结果

```
python3 test.py ./compiler lab3 -l
```

tests 下的测试样例全部通过：

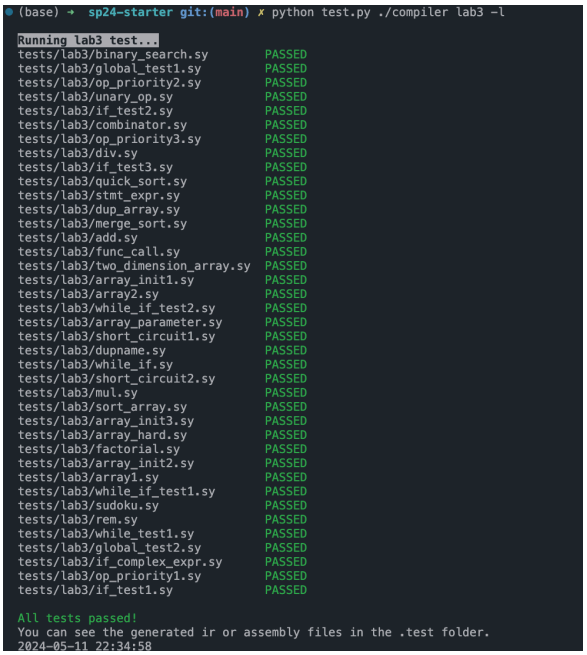

图 1 All tests passed!