

Compilers Principals - Lab2

Zhixin Zhang, 3210106357

1 实验内容

本次实验，我们基于 lab1 的语法分析，实现了代码的语法树的构建，并基于语法树，构建出了符号表，实现了更加复杂的语义分析功能，包括类型检查，数组初始化检查。

通过

```
make compiler  
./compiler <input 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。同时在错误流中输出程序的语法树，并且显示：

```
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 lab1 的基础上，增加了语法树的输出和语义分析。

```
Root->print(0);  
Checker checker;  
if(!checker.check(Root))  
{  
    std::cerr << "Failed in semantic analysis : " << argv[1] << std::endl;  
    return 1;  
}  
std::cerr << "\nParse success !" << std::endl;
```

2.2 类型检查的依据：class Type

为了更方便的对比函数，表达式，变量的类型，我们用一个“类型类”来封装一个对象类型的所有信息。比如对于一个函数，它应当包括的信息有：返回类型，参数类型。对于一个数组变量，应当包括：数组类型，每维的宽度等等。

class Type 对类型信息进行了很好的封装，并且添加了基本的比较算子，使其可以用常用的 STL 容器进行存储，使之后的处理更加方便。

```
class Type // to recognize variables and functions
{
public:
    bool isfunc; // whether the object is a function or not.
    string type;
    vector<Type> args; // if it is a function, it will have params
    deque<int> wid; // width for array
    ...
};
```

2.3 类型检查

类型检查基于语法树实现，在语法树上通过深度优先搜索的方式，对所有节点进行检查。

2.3.1 符号表的实现

我们按照 dfs 的顺序对所有被定义的 ident（函数，单变量，数组）标号，对于每一个 ident 字符串，维护一个栈作为其符号表，栈中维护的 ident 的标号。为了方便查找，我们用 map 存储符号表。

```
map<string, stack<int>> get_var; // get the position of the variable in the stack
```

同时，为了方便查询变量所绑定的类型，我们按照 ident 的标号，存储其类型。

```
map<int, Type> get_type;
```

2.3.2 check 函数

```
int check(Node* o, int L = 0);
```

用于检查语法树上的一个节点。基本的思路为，首先递归扫描所有子节点，判断其是否合法。然后根据当前节点的类型分别进行特殊的判断。

```
for(auto &x : o->child)
    if(!check(x, L)) { DEBUG("type error"); return 0;}
```

对于函数、变量的定义，要同时维护符号表。在 dfs 的同时还需要维护当前的作用域（可以直接维护当前作用域内变量的标号的最小值），离开当前作用域的时候，需要对所有符号表的栈，弹出当前作用域的所有变量。

```
for(auto &[_ , t] : get_var)
    if(t.size() && t.top() > L && !get_type[t.top()].isfunc) t.pop();
```

对于作用域还有一个特殊的需要考虑的问题，就是函数的参数也应当属于当前函数的作用域，因此，函数中节点中的 block 不应当更新作用域的范围。因此对于 block，需要额外传入一个变量来表示是否使用最新的作用域。

```
int check_Block(Node* o, int L, bool modify = 1)
{
    if(modify) L = num_var;
    // ...
}
```

对于类型检查，我们以函数调用为例：需要判断传入的参数类型是否与函数参数本身相同。

```
for(int i = 0; i < args->child.size(); ++i)
{
    if(!check(args->child[i], L)) { DEBUG("type error"); return 0;}
    Type arg_type = args->child[i]->exp_type;
    if (arg_type != type.args[i]) { DEBUG("type error"); return 0; }
}
```

2.4 数组范围检查

这里数组范围检查主要指的是在初始化阶段的检查，对于程序执行过程中的数组越界，属于段错误（不在编译阶段处理）。数组初始化相关样例见：/test/lab2/arr_defn2.sy、arr_defn3.sy、array_init_error2.sy 等。

比较复杂的情况是初始化数组的格式正确，但是超过了原定数组的大小，我们需要处理初始化阶段的数组元素补齐操作（将部分位置设置为 0）。

具体的，考虑大括号的层数可以用语法树上 initVal 的层数来表示，所以，我们在对 initVal 类型的节点的 check 时维护一下当前的大括号的层数，对于层数为 1，也就是最外层的值，它有可能是一个单值，也有可能是一个内部的大括号，我们根据它的最大层数，作为这个元素的贡献。

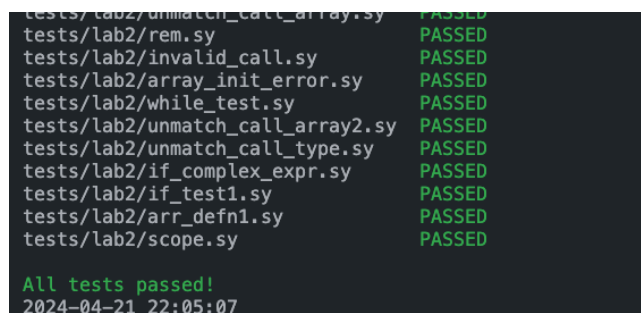
该处理方法并不完全匹配 sy 语法，但是可以处理绝大多数的数组范围检查。

```
if(layer >= 0)
{
    num[layer] = 1;
    if(layer == 0)
    {
        int mx = 0;
        for(int i = 1; i < MAXLAYER; ++i) if(num[i]) mx = i, num[i] = 0;
        calc_sum += widths[mx];
    }
}
```

3 测试结果

```
python3 test.py ./compiler lab2
```

tests 下的测试样例全部通过：



```
tests/lab2/unmatch_call_array.sy PASSED
tests/lab2/rem.sy PASSED
tests/lab2/invalid_call.sy PASSED
tests/lab2/array_init_error.sy PASSED
tests/lab2/while_test.sy PASSED
tests/lab2/unmatch_call_array2.sy PASSED
tests/lab2/unmatch_call_type.sy PASSED
tests/lab2/if_complex_expr.sy PASSED
tests/lab2/if_test1.sy PASSED
tests/lab2/arr_defn1.sy PASSED
tests/lab2/scope.sy PASSED

All tests passed!
2024-04-21 22:05:07
```

图 1 All tests passed!