语义分析 实验报告

151220151 殷瀚 411629507@qq.com

1.实现的功能

完成了所有必做及选做任务：

1. 必做内容中的 17 种错误类型；
2. 支持函数声明；
3. 作用域可嵌套；
4. 结构体类型结构等价。

2.实现方法及数据结构

2.1 符号表

符号表实现为一个名为 SymbolTable 的 C++类，在 SymbolTable.cpp 和 SymbolTable.h 中。采用 hash 表实现。所有的变量、结构体、函数都在一个表中，这个类对外提供的主要方法有：增加一个表项、查找一个表项、检查某个符号是否已经在表中。

表结构的定义大体上参照了实验手册，不再赘述，下面讲一下自己添加的部分：

2.1.1 ErrorType 与内存泄漏

在 Type 的 kind 域中，除了 BASIC, ARRAY, STRUCTURE, FUNCTION 之外，增加了一个 ERROR，增加这个类型对功能的影响不大，主要是为了写语义分析的时候方便一些。

在语义分析器中，预先定义一个名为 errorType 的 Type 型变量，整个语义分析过程中只有它的 kind 是 ERROR。这样，在检测出语义错误时，那些需要返回一个 Type 的 function 就只需要返回这个 errorType，而不用返回一个 NULL（这样就需要在许多地方进行空指针检查）或是 new 一个临时的 Type（由于这个错误的符号根本不会进入符号表，就带来了一处内存泄漏）。

P.S. 关于防止内存泄漏的问题，主要做的就是确保所有 new 出来的东西要么在符号表里面有一个指针，以便在语义分析结束后全释放掉，要么就及时 delete 掉。我的代码中还有其他一些与内存泄漏有关的处理细节，应该还有漏掉的，讲起来比较繁琐，跟实验要求也无关，报告里就不多说了。

2.1.2 AssignType

在 Type 中的一个域，指明该符号是左值还是右值，用来检测 Error type 6。

2.2 语义分析

为所有的产生式写相应的语义动作即可，实现在 SemanticAnalyzer.cpp 和 SemanticAnalyzer.h 中。

虽然这个部分占了实验二的绝大部分时间，但是在实验报告里似乎也没什么值得讲的，因为总的来说没什么难度，根据产生式的右部以及所有错误类型一个一个写过去就行，就是工作量比较大（七百多行 Orz）。

2.2.1 类型检查

类型检查也比较简单，就是根据 kind，分类型把相应的域一个个比过去就行了。对于检查的结果，除了 MATCHED（匹配）和 NOT_MATCHED（不匹配）以外，还有一种 NOT_SET，当参与比较的两者中有某一个包含语义错误的话，就会返回 NOT_SET。

比如赋值语句“a=x”，如果发现 x 是未定义的变量，那么如 2.1.1 节所述，x 的 type 就是 errorType，此时这条赋值语句的类型检查结果就是 NOT_SET，这样我们只会报告“未定义变量 x”错误，不会再多报一个“类型不匹配”的错误。

2.2.2 产生式序号

为了方便，把之前语法分析的部分小修改了一下。由于写语义动作时需要知道当前结点的子节点的类型，这些信息完全可以在语法分析时记下来，所以现在语法树上的每个结点会记录下产生式的序号，这样我们在语义分析时针对这个 productionNO 的值进行 switch-case 即可。

```
ExtDecList  : VarDec{
                $$ = new Node(NODE_TYPE_NON_TERMINAL, "ExtDecList", @$.first_line);
                $$->addChild($1);
                $$->setProductionNo(0);
            }
            | VarDec COMMA ExtDecList{
                $$ = new Node(NODE_TYPE_NON_TERMINAL, "ExtDecList", @$.first_line);
                $$->addChild($1);
                $$->addChild($2);
                $$->addChild($3);
                $$->setProductionNo(1);
            }
            ;
```

2.3 函数声明

函数声明实现起来还是比较麻烦的，不仅要添加对应的产生式和动作，往符号表里添加一个函数时的过程也不太一样，最后还需要检查有没有未定义的函数。

2.3.1 修改文法

增加一条产生式：

ExtDef -> Specifier FunDec SEMI

语义动作就是分别解析返回类型和函数名，与函数定义类似，但是向符号表添加表项的过程有所修改。

2.3.2 检查符号表

当我们尝试向符号表中添加一个函数时，可能有如下几种情况：

AddFunctionResult	说明	处理
NEW_ITEM_ADDED	符号表中没有同名项，直接添加一个新的表项。	添加到表中
DIFFERENT_KIND	符号表中有同名项，但该项不是一个函数。	Error Type 19
REDEFINED	符号表中有同名的已定义函数，新增项是一次重定义。	Error Type 4
CONSISTENT_DECLARE	新增项是函数声明，且与表中函数的类型一致。	不需处理
INCONSISTENT_DECLARE	新增项是函数声明，但是与表中函数的类型不一致	Error Type 19
NEWLY_DEFINED	新增项是函数定义，表中函数未定义，且函数类型一致。	修改表项
INCONSISTENT_DEFINE	新增项是函数定义，表中函数未定义，但函数类型不一致。	Error Type 19

因此，我们在符号表中的 Function_中增加一个域：isDefined，记录一个函数是否已经被定义。每次遇到函数定义或声明，就参照上表进行处理和报错即可。

2.3.3 检查未定义函数

在我们执行完语义分析后，还需要遍历一遍符号表，看有没有表项的类型是函数且 isDefined 属性值为 false，若有，则需要报错“Error Type 18”。

按照实验手册的要求，“Error Type 18”报错中需要有函数声明所在的行数和函数名，但是一个未定义函数可能有多条声明，可我们的符号表中不会为多次声明保存多个重复表项，所以我在符号表中增加了一个链表 funDecRecords，专门用来保存所有的函数声明和定义的记录，包括这条声明所在的行数，函数类型等信息，检查未定义函数时，遍历这个链表即可。

2.4 嵌套作用域

手册上讲的很详细，不过我这里为了方便，没有采用手册上的十字链表来实现。

2.4.1 作用域层数和作用域栈

为了实现嵌套作用域，我们只需要记录下每个符号所在的作用域的层数，以及分析程序当前所在的层数即可。

这样，我们只需要在符号表类中增加一个整型变量 scopeDepth，表示当前所在作用域的层数。每次进入一个嵌套作用域，层数加 1；退出作用域时，先把符号表中所有属于这一层的项删掉，再把层数减 1。相应的，添加表项时，把当前的层数也写入表中；在检查表中是否有同名表项时，对函数的操作和之前一样（不支持函数的嵌套定义），对变量和结构体，只检查属于同一层的同名项。

另外，我们在检查结构体的定义时，虽然也可以当作是一层作用域来处理，但是重定义的域和重定义的变量对应于不一样的错误类型，为了输出正确的错误信息，我们需要记录当前作用域类型，以区分是在结构体中还是在函数或全局中，因此，用一个栈来记录：

```
enum ScopeType          { COMMON, STRUCT };
stack<ScopeType> scopeStack;
```

2.4.2 函数参数的作用域

支持嵌套作用域后，函数的形参就不应该还是全局作用域了，对于函数定义，把形参放到函数体所在的作用域中，就能正确实现参数的作用域了。

2.5 结构体类型结构等价

这个附加要求其实几乎没有工作量，因为结构体的结构等价就是要把 FieldList 中的内容逐个相比就行，而我之前在比较函数类型时，为了比较参数列表的类型，已经实现了 compareFieldList 函数，所以比较结构体时把字符串比较换成调用这个函数就行了。

3. 编译运行方法

直接在 Code 目录下 make 即可，将依次调用 flex 和 bison 生成.c 文件，再寻找 Code 目录下所有的.c 和.cpp 文件进行编译链接得到可执行文件 parser（生成的 parser 在**根目录**下）。

在 Code 目录下执行 make test 将调用项目**根目录**下的 parser，对 Test 目录下所有.cmm 文件进行分析。

在 Code 目录下执行 make clean 将 Code 目录下的临时文件清理，不会删除根目录下的可执行文件。

4. 试验总结

语义分析写起来还是挺累人的，为每个产生式写语义动作是个非常繁琐的工作，符号表的设计也是牵一发而动全身，需要十分小心。