Lab2 语义分析实验报告

史子凡　151180115　vivianszf9@gmail.com

一、 实验进度。

完成了实验二必做和选做部分要求的全部内容，包括 17 个基本错误类型和 3 个选做的要求。

二、 编译方式。

根目录下写有 Makefile 文件，提供以下编译方式：

1. make lab2: 即编译所有相关文件并生成所需要的 parser 可执行文件；
2. make parser: 对 main.c, syntax.tab.c, treeop.c, symbol_table.c 还有 semantic.c 进行联合编译，生成可执行文件 parser；
3. make clean: 删除 parser 文件；
4. make testcN：N 用 1 到 17 的数字替代，是对 test/compulsory/下的第 N 个必做样例进行测试。比如，make testc2 表示对 test/compulsory/2.cmm 进行测试。
5. make testoN：N 用 1 到 3 的数字替代，是对 test/others/下的第 N 个选做样例进行测试。比如，make testo2 表示对 test/others/2.cmm 进行测试。
6. make testm1: 测试自己写的 test/my/1.cmm 这个测试样例。
7. make testall: 对所有测试样例（17 个必做+3 个选做）进行测试。错误信息会依次显示在命令行上。

三、 实现细节。

1. 总体思路

我实现的方法是在实验一生成的语法树上从根节点 Program 开始按照产生式对每个产生式中的元素一个一个往下进行递归分析，在分析的过程中如果遇到 ExtDef 就知道产生了定义，将 DecList，FunDec，StructSpecifier，中定义的东西放入栈中（也确定在哈希表中的位置）。遇到其余的产生式则进行进一步往下分析，并维护属性等信息。在遍历过程中进行相应的错误分析。在遍历完整棵树后，在栈里检查是不是有没有定义的函数，如果没有定义，那就报错。

2. 符号表的实现

采用了讲义中给的基于十字链表和 open hashing 散列表的符号表实现方式，

定义的数据结构如下：

符号表中每个符号对应的结构：

```
struct Symbolele
{
    char *name;
    int funcornot;
    int lineno;
    union{
        Type *type;
        Func *func;
    };
};
```

符号再用链表连接起来构成符号表，连接起来的结构为：

```
struct Symbolt
{
    Symbolele *s;
    Stack *stack;
    Symbolt *hash_pre;
    Symbolt *hash_next;
    Symbolt *stack_pre;
    Symbolt *stack_next;
};
```

其中，符号类型分为两种，一种是函数的 Func 类型，还有一种是非函数的 Type 类型，借鉴了讲义上给的结构：

```
struct Type_
{
    enum{VTINT,VTFLOAT,VTARRAY,VTSTRUCT}kind;
    union
    {
        struct Array
        {
            Type *elem;
            int size;
        }*array;
        FieldList *structure;
    };
};
```

```
struct Func
{
    Type *type;
    int defordec;
    struct Args
    {
        Symbolele *a;
        Args *next;
    }*args;
};
```

3. 函数声明

在实验一写好的 syntax.y 中对文法进行修改，添加产生式 ExtDef->Specifier FunDec SEMI.由此就可以将函数声明正确识别出来。在添加函数到符号表的时候需记录其只是被声明还是已经被定义了，这个由 Func 结构体中的 defordec 来记录。

4. 支持多层作用域

根据讲义中的符号表可以很好很方便的实现多层作用域的支持。

5. 类型等价机制为结构等价。

写了一个函数叫 type_equiv_detect（在 symbol_table.c 中），其可以递归的判断每一个域是否等价，只有全部等价且域的个数相同才会返回两者等价的信息。

6. 运算类型

规定对于逻辑运算规定了必须是 int 类型，对于四则运算规定必须是 int 或 float，否则都将进行报错。

7. 错误检测

未经定义、重复定义或名字重复这类的错误只需遍历一遍符号表看看之前有

没有定义或重名的即可；

类型不匹配的错误通过对 type 进行比较即可以查找出；

左边出现右值的错误通过在对 Exp 进行分析的时候除了 type 再多加上一个属性 lorr，用于记录是不是右值。这样在递归分析 Exp 的过程中也不会丢失是不是右值的分析结果。例如 INT，FLOAT，四则运算等等肯定都是右值。记录下来最后在 ASSIGNOP 的地方进行判断即可；

函数的实参和形参数目或类型不匹配是通过对他们的 arguments 进行一一比较得到的。对 Args 进行分析的时候会返回所有 Args 构成的一个链表。

对于涉及“()”、“[]”、“.”的错误，在 Exp 分析到这些括号的时候，对括号前的符号的 type 进行检测，看看是否符合要求，如果不符合，那就报错。

对于结构体在定义时对域进行初始化的错误（给的测试样例里没有，自己写了一个测试样例在 test/my/1.cmm 里，运行 make testm1 就可以看到结果了），设置一个全局变量 struct_varassign，在 StructSpecifier 中往下分析 DefList 之前将其设置为 1，从 DefList 返回时设为 0，这样在 DefList 向下分析到 Exp 为 ASSIGNOP 的时候就知道这个是在结构体定义时出现的赋值，进行报错。

因为我的报错机制并不能分析出哪个是最根本的错误，所以会报出所有连锁导致的错误，这就会造成可能同一个位置报错很多次。因此我在每次报错前都进行报错扫描，看看之前有没有在这个地方报过同款错误，以此来避免重复。

四、 实验总结。

很久没接触过链表了，所以一开始遇到各种 segmentation fault 的错误，原因都是因为访问了空指针：要么是忘了判断 NULL 的情况，要么就是忘了用 malloc 分配空间。定位 segmentation fault 的错误非常麻烦，就我而言，都是靠肉眼分析产生式分析到哪里然后判断出错误在哪里这种低级方法…

本来觉得递归分析太慢了，想用先序遍历的方式处理，即只有看到了定义或者产生式才进行处理，但事实证明，那样写的话重复的代码比较多，而且逻辑并没有直接一层层递归下去清晰，所以还是采用了产生式递归的方法进行分析，这样也可以更好的清晰的包含每一种情况。

另外，因为支持一次分析处理多个文件，所以还要记得每次开始分析前进行初始化，否则要么错误报不出来，要么报一些奇奇怪怪的错误。