编译原理lab2实验报告

姓名：张玄逸	学号：201220194
日期：2022.10.30	邮箱：1822771416@qq.com

实验内容

- 实现了错误1-17的检查
- 完成了要求2.1（包括错误18，19），2.2，2.3

测试方法

代码可使用Makefile进行编译，得到可执行文件parser。

在文件根目录下输入 ./parser [文件名] 即可执行，如 ./parser Test/test1.cmm

实验思路

分析过程

首先，在实验一里我们已经实现了语法分析并得到了语法树，接下来就开始处理这棵树.

对于每个新符号，我们都需要用按照手册提供的ListNode类型记录它的信息。符号表采用AVL树保存，只要记录根节点就能遍历搜索该范围内的所有符号。

```
typedef struct avl_node
{
    ListNode *node;//记录了这个节点代表的符号
    struct avl_node *lc;
    struct avl_node *rc;
    int bf;  //平衡因子
    int sum; //该树总节点个数（不包括自己）
} AVL_node;
```

接下来，观察之前的语法分析规则可以得知，一个Program是由零或多个ExtDef组成的，每个ExtDef都代表一个变量，结构体，函数的定义或声明（函数声明在要求2.1完成），所以把ExtDef视为分析的基础划分。在递归读取语法树时，遇到ExtDef就停止递归并跳出读取，进入进一步的deal函数。在deal函数内部在层层往下传递分析。

比如，遇到Specifier ExtDecList SEMI这样的变量/结构体定义语句，要先调用deal_Specifier函数处理前面的Specifier，返回一个Type*类型，将得到的变量类型传递回来，接下来才能在处理ExtDecList时将得到的类型与变量名结合生成一个完整的符号，并插入符号表。

基本要求

下面列举部分。

- 错误类型6：等号左边出现非左值

观察Exp生成式，只有 Exp DOT ID | ID | Exp LB Exp RB 三种才是可以被赋值的左值（关于 [] 和 . 的正确使用与否已在其它函数中检查），所以只需要递归检查即可。

- 错误类型8：return返回类型不匹配

return语句处于compst->stmtlist->stmt中，而函数类型处于specifier中，所以在一开始处理Specifier FunDec CompSt 语句时，就要将处理specifier得到的类型传递给deal_compst，依次往下传，这样才能比较类型。

- 错误类型15：结构体定义时对成员变量初始化

因为结构体和函数的内部变量都属于语法单元DefList，因此需要传入一个flag帮助区分。当flag=1即结构体时，Dec 只能被解释为 VarDec 而不是 VarDec ASSIGNOP Exp（赋值）。

附加要求

- 要求2.1

  改变语法规则：在ExtDef = Specifier ExtDecList SEMI | Specifier SEMI | Specifier FunDec CompSt
  后面添加一条 Specifier FunDec SEMI （函数声明）。

  - 错误类型19：声明冲突

    只需要在遇到一个新的声明时，按照函数名寻找之前已声明过的函数，进行比较即可

  - 错误类型18：函数已声明未定义

    这一点比较特殊，因为在遇到一个声明时尚不清楚后面是否会有定义，而报错又需要定位到函数声明的一行，因此只能先全部处理完语法树后，即添加了所有的函数定义，再检查声明符号表announce_table中的所有函数符号，看在定义符号表define_table中有没有相应的定义。

    ```
    int main(){
        ...
        read(semantic_tree);//进行语法树的读入和检查，除error18外其它所有错误检查都在其中被调用
        check_error18();//符号添加完后才能检查哪些只有声明
    }
    ```

- 要求2.2

  变量可嵌套定义：将原来两个变量，声明符号表和定义符号表改为指针数组，用栈来实现作用域的嵌套。

  ```
  AVL_node *announce_stack[101];
  AVL_node *define_stack[101];
  int announce_top = 0;
  int define_top = 0;
  //符号表栈，用于嵌套定义
  ```

  每次调用table_push就会使top上移，相当于新建一个符号表，当前根节点为NULL。每次调用stack_pop就会使top下移，意为退出现在的作用域返回上一层。

  ```
  insert_listnode(p, &define_stack[define_top]);
  //假如p是生成的符号节点，就可以调用insert函数把它加入到当前的定义符号表中
  ```

  在查找符号的时候分为两种情况：全范围和当前范围。因为作用域可以嵌套，所以检查是否重复定义只能在当前范围内检查；而检查是否未定义需要从内到外查找所有作用域。

- 要求2.3

  结构体的结构等价：只需要将结构体判定条件改为依次比较成员变量即可

总结与反思

- 结构体与变量的定义域
  - 要明确一点：符号=变量+结构体+函数，变量指基本类型BASIC和数组ARRAY，它们是可以嵌套定义的，内层变量与外层变量可以同名，但结构体是全局的，也就是说只要有一个结构体被定义，那里层无论是变量还是结构体都不能再使用相同的名字。故变量是否重定义应该写成如下形式：
  - ```
    bool check_error3(char *name, int line_num)
    {
        ListNode *tmp1 = search_listnode(define, name, BASIC, true);
        //最后一个参数为true，BASIC基本类型作用域可以嵌套，所以只在本层寻找是否有重定义
        ListNode *tmp2 = search_listnode(define, name, ARRAY, true);
        ListNode *tmp3 = search_listnode(define, name, STRUCTURE, false);
        //最后一个参数为false，结构体定义是全局的，所以要从里到外查看是否有重定义
        if (tmp1 != NULL || tmp2 != NULL || tmp3 != NULL)
        {
            error(3, line_num, "Redefined variable", name);
            return false;
        }
        return true;
    }
    ```
- 被解释为empty的语法节点
  - 比如函数的具体定义CompSt = LC DefList StmtList RC，其中DefList 是内部的变量定义，StmtList 是之后的执行语句，它们都可以被解释为empty（也就是可以没有）。在实验一中，我采用的语法分析规则类似如下：
  - ```
    DefList : Def DefList {$$=new_node(@1.first_line,"DefList",2,$1,$2);}
    | /* empty */ {$$=NULL;}
    ```
  - 也就是解释为empty时直接把该节点赋值为NULL，但这在实验二中引发了严重的错误。因为语法树是由child和next关系联结的，所以当联结 CompSt = LC DefList StmtList RC 时，程序发现DefList=NULL，就会直接把后面的StmtList接到LC之后（因为Deflist=NULL，而NULL不能有next属性），这样CompSt = LC DefList StmtList RC 实际上分化出了四种可能：
    - CompSt = LC DefList StmtList RC
    - CompSt = LC StmtList RC
    - CompSt = LC DefList RC
    - CompSt = LC RC
  - 如果分类讨论，无疑是非常麻烦的操作且不符合语法树的结构。因此解释为empty的节点不能设置为NULL，而应该保留其存在并把它的child节点设置为NULL。为了和真正的叶子节点区分，设置child数目为1。这样在进行语义分析时就可以按照语法树结构进行检查了。
  - ```
    DefList : Def DefList {$$=new_node(@1.first_line,"DefList",2,$1,$2);}
    | /* empty */ {$$=new_node(0,"DefList",1,NULL);}
    ```
    ```