编译原理实验二 实验报告

151220152 殷乔逸

程序功能

- [x] 实验代码已实现对 C 语言代码的简单词法分析及语法分析，能够识别基本的词法错误和语法错误，例如：a[..]属于对数组声明的错误等。
- [x] 代码中词法分析部分采用了 GNU Flex 方法，语法分析部分采用了 GNU Bison 方法并且自行设计了多叉树结构。
- [x] 代码中符号表采用了基于十字链表和 Open hashing 的散列表结构。
- [x] 代码中语义分析部分采用深度优先搜索的方法遍历语法树，并且查填表以找到错误类型。
- [x] 本次实验中进一步实现了对 C 语言代码的语义分析，能够输出符合 C 语言文法语法但是并不符合 C 语言语义要求的多种错误类型(包括所有必做和选做要求)。

数据结构

1. 首先针对变量类型，设计 Type 数据结构，存储不同类型的变量，比如函数、数组和结构体等，具体设计如下：

```
typedef struct Type_ {
    enum {BASIC, ARRAY, STRUCTURE, FUNCTION, UNKNOWN} kind;
    /* Different types use different groups of variables */
    union {
        enum {TYPE_INT, TYPE_FLOAT, TYPE_OTHERS} basic;
        struct {
            /* Array elments' type */
            struct Type_* elem;
            /* Array size list */
            struct ArraySizeList_* array_size;
        } array;
        /* Structure type */
        struct Structure_* structure;
        struct {
            /* Type of return value of a function */
            struct Type_* rtn;
            /* Parameters list of a fuction*/
            struct FuncParamList_* param_list;
        } func;
        char unknown[128];
    };
} Type;
```

2. 其次针对不同变量类型，采用不同的结构体来存储信息，比如针对结构体，需要有存放成员变量的域链表，具体数据结构设计如下：

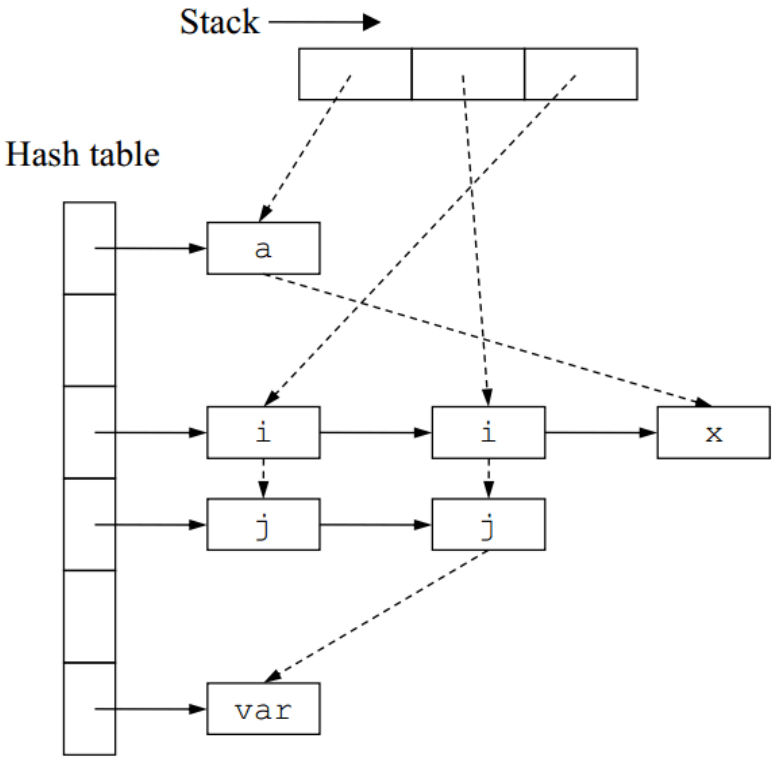
```
/* Structure for different fields in programs */
    typedef struct FieldList_ {
        char* field_name;
        /* Type of the field */
        struct Type_* field_type;
        /* prev: for convenience of deletion
         * next: for embedded fields
         */
        struct FieldList_* prev;
        struct FieldList_* next;
    } FieldList;

    /* Structure for STRUCTURE in syntax tree */
    typedef struct Structure_ {
        struct FieldList_* fields;
    } Structure;
```

3. 最后每一个 symbol 都需要进行存储，所以最终针对符号的结构体设计如下,其中 scope_level 是针对嵌套作用域设计的作用域层数记录：

```
    /* Structure for symbols recorded in the symbol table */
    typedef struct SymbolRecord_ {
        /* Record name */
        char* record_name;
        /* Scope level */
        int scope_level;
        /* Type of scope or variables*/
        struct Type_* symbol_type;
        /* Syntax tree node for the scope of some functions */
        TreeNode* tree_node;
    } SymbolRecord;
```

实现思路

1. 首先是建立合适的数据结构，在这次实验的代码部分，亮点和完成度比较高的地方集中在基于十字链表和开散列的散列表结构。针对这个稍微复杂的结构，下面以图例具体描述，代码实现在第 2 点大致列出:



2. 除了建立好数据结构，还要实现基本的一些和该数据结构相关的方法，比如 `Push_Scope()`、`Pop_Scope()`、`Insert_Var_Symbol()`和 `Find_Var_Func_Symbol()`等方法，这样便于利用已经建立好的数据结构，一些方法的设计难点在第 4 点中列出。

3. 最后要实现深度优先遍历语法树，并且实现查表插表，对十字链表的符号表与作用域栈表进行动态的维护，对语义分析中的 19 个错误类型逐一分析。在设计深度遍历的时候可以根据终结符号和非终结符号来遍历，不同的非终结符号由于产生式不一样，遍历的方法也不一样，所以针对性的设计能够降低耦合度，使得代码更加模块化更好维护。

实验难点

本次实验主要有三大**难点**，具体如下：

1. 基于十字链表和 Open hashing 的散列表结构设计。这个可以参考上面的数据结构部分。

2. 从内层作用域重新进入外层作用域时，`Pop_Scope()`中有两大难点：
    1. 删除结点时分两步，先从 `var_func_symbol_hashtable`，也就是从存储变量或函数名的符号表中删除结点，这里要注意删除结点并不意味着要释放它的内存，这个时候如果释放内存，则该结点就无法再被访问，那么在 `symbol_scope_stack`，也就是作用域栈中就无法删除它，会发生

段错误。
所以删除的时候，将头结点指向它的下一个结点，原本的头结点本身的 node_next 和 node_prev 赋值为空，这样一来，这个结点相当于在符号表中消失了，没有任何其他结点可以访问到它。
然后同样的方法删除作用域栈中的结点，虽然这个结点在符号表中无法访问，但是依旧可以通过作用域栈访问，因为这个结点的 stack_next 和 stack_prev 依旧存在。按相同方法删除后，该结点的四个指针 node_next,node_prev,stack_next 和 stack_prev 全部置为 NULL，然后再释放这个结点的内存空间（当然不释放也是可以的，毕竟已经无法被访问到了）。

2. 上述难点是关于结点具体如何删除的，还有另一个和结点删除相关的难点。在作用域栈遍历相同作用域层级的结点链表进行删除整个作用域操作的时候，有一个很重要的关键点：可能会存在两个或两个以上的符号拥有相同的 Hash 值，同时这些符号在同一个作用域内。
这个时候当执行 Pop_Scope()操作的时候，有个隐含的信息：被 Pop_Scope()掉的内层作用域中符号应该都在符号表某 Hash 值链表的表头，因为插入结点的时候总是插在表头。所以如果存在上述特征的符号，那么意味着如果当前要删除的结点不是表头结点，必然存在其他结点和它在同一作用域且拥有相同 Hash 值，并且这些结点都在它前面。所以如果当前要删除的结点不是表头，那么记录该结点位置，先删除它前面的结点。当然直接顺次删除也是可行方法。

3. 深度优先搜索遍历语法树，按照错误类型要求输出语义错误。这个看起来是难点但是细致去写 DFS 这一块的代码的话出错的可能性也很小。一般 BUG 集中在上述两点。

解决方案

具体的解决方案详见代码内容，数据结构已在上述第 2 点简单介绍。

编译及运行方法

首先压缩包解压后得到的文件夹 Lab2,Test 文件夹中存储了 23 个测试用例，Code 文件夹存储了所有代码文件，results.txt 文件存储了 23 个测试用例的测试结果（包括错误提示的输出和无错的空输出）。

- 如果想要直接编译运行，则在 shell 命令行执行./run.sh，则会直接编译并且对 Test 文件夹下的用例进行测试，输出的结果按序存储在 results.txt 中。
- 如果仅仅想要编译，则进入 Code 文件夹执行 make parser，则会重新编译并在上层目录 Lab2 得到可执行文件 parser。
- 如果想要对某一个测试用例进行测试，则在 Lab2 目录下执行./parser xxx,其中 xxx 代表将要被测试的用例。