实验二报告

171860513 刘天宇　171860573 宾向荣

一、实验目标

通过手写代码，在实验一的基础上，在已经构建好语法树上实现语义分析，包括17种错误的检测和恢复、符号哈希表、类型定义表的维护。

二、实验亮点

1、类型丰富、性能优越的数据结构（**总体架构图见第2页**）

（1）符号栈×哈希表（对应选做2.2）

在符号栈外，构建快速索引符号栈中表项的哈希表，不仅能够实现**按名快速查询，**还能够：
① 在离开复合语句，局部变量集体消亡（符号栈退栈）时支持**惰性删除**，即索引后打上标签为删除，而不是见一个删一个。
② 每个哈希链表使用**头插法**，查询最内层复合语句定义的（局部变量）时，找到相应链表第一个同名符号即可。

（2）（结构体）类型定义哈希表

由于结构体数量可能较多，也采用了哈希表实现，以保证找到结构体定义的高效。

2、良好的**错误检测—恢复机制**（举例）

我们使用traverseTree(Node* root)后序遍历语法树。在每个结点调用traverseTree时，如果某个结点node产生一个语义错误，该结点会立刻停止处理，并根据实际需要返回TRUE或FALSE给自己的父节点parent，以此反馈这个问题是否需要被高层文法符号“重视”。

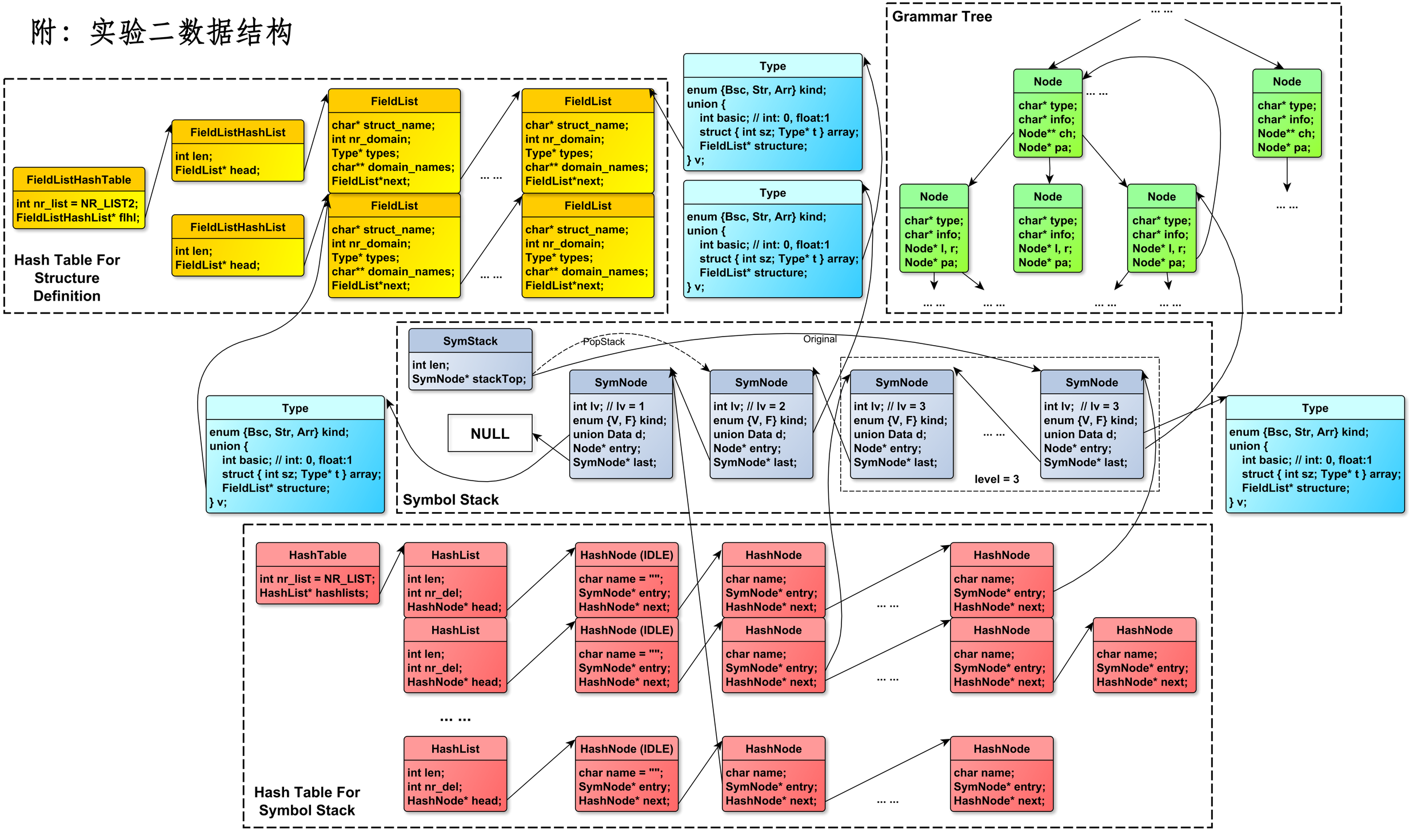
举一个例子。现有产生式：ExtDef → Specifier ExtDefList SEMI；Specifier → StructSpecifier；StructSpecifier → STRUCT OptTag LC DefList RC。如果我们在StructSpecifier结点处理中发现结构体名已被使用，那么该结构体不能被定义；对应的Specifier类型说明符也是无效的。为此，StructSpecifier结点应向上层的Specifier返回FALSE，告知其停止执行。同理ExtDef也应被Specifier告知停止，因为类型不完整的全局定义是无效的。

但若还有ExtDefList → ExtDef ExtDefList，由于一处无效全局定义并不影响其他全局定义，故ExtDef应返回TRUE，让ExtDefList继续处理其他全局定义。

三、实验收获

对语义分析的功能和过程有了初步认知；了解了语义分析所对应的不同错误类型；构建了错误检测—恢复机制；对符号表的作用有了初步了解。

附：实验二数据结构


FieldListHashTable
int nr_list = NR_LIST2;
FieldListHashList* flhl;
FieldListHashList
int len;
FieldList* head;
FieldListHashList
int len;
FieldList* head;
Hash Table For Structure Definition
FieldList
char* struct_name;
int nr_domain;
Type* types;
char** domain_names;
FieldList*next;
FieldList
char* struct_name;
int nr_domain;
Type* types;
char** domain_names;
FieldList*next;
FieldList
char* struct_name;
int nr_domain;
Type* types;
char** domain_names;
FieldList*next;
FieldList
char* struct_name;
int nr_domain;
Type* types;
char** domain_names;
FieldList*next;
Type
enum {Bsc, Str, Arr} kind;
union {
int basic; // int: 0, float:1
struct { int sz; Type* t } array;
FieldList* structure;
} v;
Type
enum {Bsc, Str, Arr} kind;
union {
int basic; // int: 0, float:1
struct { int sz; Type* t } array;
FieldList* structure;
} v;
Grammar Tree
Node
char* type;
char* info;
Node** ch;
Node* pa;
Node
char* type;
char* info;
Node** ch;
Node* pa;
Node
char* type;
char* info;
Node* l, r;
Node* pa;
Node
char* type;
char* info;
Node* l, r;
Node* pa;
Node
char* type;
char* info;
Node* l, r;
Node* pa;
SymStack
int len;
SymNode* stackTop;
PopStack
Original
NULL
SymNode
int lv; // lv = 1
enum {V, F} kind;
union Data d;
Node* entry;
SymNode* last;
SymNode
int lv; // lv = 2
enum {V, F} kind;
union Data d;
Node* entry;
SymNode* last;
SymNode
int lv; // lv = 3
enum {V, F} kind;
union Data d;
Node* entry;
SymNode* last;
SymNode
int lv; // lv = 3
enum {V, F} kind;
union Data d;
Node* entry;
SymNode* last;
level = 3
Type
enum {Bsc, Str, Arr} kind;
union {
int basic; // int: 0, float:1
struct { int sz; Type* t } array;
FieldList* structure;
} v;
Type
enum {Bsc, Str, Arr} kind;
union {
int basic; // int: 0, float:1
struct { int sz; Type* t } array;
FieldList* structure;
} v;
Symbol Stack
HashTable
int nr_list = NR_LIST;
HashList* hashlists;
HashList
int len;
int nr_del;
HashNode* head;
HashList
int len;
int nr_del;
HashNode* head;
HashList
int len;
int nr_del;
HashNode* head;
HashNode (IDLE)
char name = "";
SymNode* entry;
HashNode* next;
HashNode (IDLE)
char name = "";
SymNode* entry;
HashNode* next;
HashNode (IDLE)
char name = "";
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
HashNode
char name;
SymNode* entry;
HashNode* next;
Hash Table For Symbol Stack