- 学号: 221240093
- 姓名: 陈力峥
- 邮箱: 221240093@smail.nju.edu.cn

编译

编译实验一采用 `Makefile` ，在 `Code` 目录下执行 `make` 即可，随后生成 `parser` 可执行文件。

实验内容

完成实验二必做以及选做部分：

- `C--` 语言的语义分析，包含其中基本的类型检查，符号表设计以及选做部分的嵌套作用域、函数声明、结构体成员比较等。

实验细节

符号表

```
// 定义作用域链表节点结构体
typedef struct ScopeRBNode {
    RBNode root; // 当前作用域的红黑树根节点
    struct ScopeRBNode* next; // 指向下一个作用域
} ScopeRBNode;

// 定义全局符号表结构体
typedef struct {
    ScopeRBNode* currentScope; // 当前作用域
    RBNode globalStructRoot; // 全局结构体定义的根节点
    RBNode globalFuncRoot; // 全局函数定义的根节点
} SymbolTable;
```

- 对于符号表的设计，底层采用了红黑树来实现符号的存储。并且进行分表，将基本变量 `int`, `float` 以及 `struct` 变量（并非定义，为实际声明的变量，用 ID 字段与 INT FLOAT 区分， INT 为 0， FLOAT 为 1， STRUCT 类型 >= 2）并入同一张表，即 `currentScope` 这是一个链表，链表头部为当前作用域，每退出一个作用域，则删除头结点内的所有符号，并且更新头结点，来实现作用域的嵌套。
- 将 Struct 的定义视为一个符号，插入 `globalStructRoot` 表示的符号表中，将函数的定义和声明视为一个符号，插入 `globalFuncRoot` 表示的符号表中。

- 并且对外的 `Public API` 主要只提供以下几个操作， `insert` 内部会根据类型插入到对应的符号表。并且在进入新的作用域的时候需要调用 `enterScope()` 手动更新作用域。

```
void insert(Symbol symbol);
RBNode search(char* name, bool isDef);
void initSymbolTable();
void enterScope();
void exitScope();
```

结构体比较

- 主要通过实现下列函数来完成：

```
/* ================ Struct Compare ================ */
int compareStructTypes(Type t1, Type t2);
int compareFieldListsUnordered(FieldList fl1, FieldList fl2);
int compareFieldLists(FieldList fl1, FieldList fl2);
int compareFieldListsEquals(FieldList fl1, FieldList fl2);
int compareTypes(Type t1, Type t2);
int compareTypesEquals(Type t1, Type t2);
int getFieldListLength(FieldList fl);
int compareFieldsForSort(const void* a, const void* b);
FieldList* fieldListToArray(FieldList fl, int len);
```

- 主要步骤为将结构体中的 `member_fieldList` 转为数组，然后直接用 `qsort` 排序，并提供排序比较函数 `compareFieldsForSort()` 来实现。此后顺序扫描查看长度以及各个类型是否相同即可。

语义分析

- 通过阶段一实现的语法树来进行语义分析（并非边构造语法树边进行语义分析）。
- 为每一个语法结点符号设计一个语义分析函数，最后以 `DFS` 顺序遍历语法树，调用每一个结点的语义分析函数来实现。
- 简单举例 `check_compSt` 函数：
  - 选择忽略其中的 LC 和 RC 符号（无具体意义）
  - 先调用 `enterScope()` 进入新的作用域
  - 如果是函数类型，则将函数参数先插入符号表（意味着此时是函数定义内部，需要用到这些符号）
  - 然后递归调用 `check_defList` 和 `check_stmtList` 来遍历语法树解析
  - 最后调用 `exitScope()` 退出作用域

```
void check_compSt(Node *compSt, Type funcType) {
    // CompSt -> LC DefList StmtList RC
    assert(compSt != NULL);
    enterScope();

    if (funcType != NULL && funcType->kind == FUNCTION) {
        // Insert parameters into symbol table
        FieldList params = funcType->u.function.params;
        while (params != NULL) {
            Symbol symbol = createSymbol(params->name, params->type);
            insert(symbol);
            params = params->tail;
        }
    }

    Node *defList = compSt->child->next;
    Node *stmtList = defList->next;

    check_defList(defList, NULL, 0); // 解析局部变量定义
    check_stmtList(stmtList, funcType); // 解析语句列表

    exitScope();
}
```