编译原理实验二报告

张旭钦 201240058

实验二是对实验一内容的延申，本次实验在上一次实验的语法树上遍历，完成语义分析。我使用的方法是树上dfs，更加准确地说是树的后序遍历。

对于文法中的每一个非终结符，我基本上都设计了对应的$handler$函数处理。比如`Program_handler`函数等。

（1）符号表的设计

我采用了散列表和Functional Style的方式。（为了方便，我没有使用union将不同的Symbol类型放在一起处理）

```
typedef struct symbol {
    char *name;             // 符号名字
    enum SymType symType;   // 符号类型:  =VAR, FUNC, STRUCTURE
    int scope;              // 符号作用域
    int lineno;             // 符号所在行号，只是为了更好的报错
    //------------------------//
    Type vartype;    // type=VAR时有效
    int isAssigned;  // type=VAR时有效，是否已经被赋值
    //------------------------//
    Type returnType;                // type=FUNC时有效，函数返回值类型
    struct ParamList_ *paramList;   // type=FUNC时有效，函数参数列表
    int isFundef;                   // type=FUNC时有效，函数是否定义过
    // ----------------------- //
    Type structType;  // type=STRUCTURE时有效，结构体类型
} Symbol;

typedef struct symbol_table_entry {
    Symbol *symbol;
    struct symbol_table_entry *next;
} SymbolTableEntry;

typedef struct symbol_table {
    int scope;                  // 当前作用域，0 全局作用域
    struct symbol_table *prev;  // 上一个作用域的符号表
    struct symbol_table *next;  // 下一个作用域的符号表
    int size;                   // 符号表大小
    SymbolTableEntry **table;   // 哈希表
} SymbolTable;
```

符号表的辅助函数为：

```
// 查找符号表中的符号
Symbol *lookup_symbol(SymbolTable *table, char *name, int flag, enum SymType symType);

// 向符号表中添加符号
void add_symbol(SymbolTable *table, Symbol *sym);
```

（2）作用域问题

进入新的作用域，创建新的符号表，同时将旧的符号表链入新符号表的pre指针。退出当前作用域，则将通过pre指针取出上一层作用域，修改当前作用域指针即可。这两个功能我通过两个函数实现：

```
SymbolTable *enter_scope(SymbolTable *table);
SymbolTable *exit_scope(SymbolTable *table);
```

需要重点考虑的其实是什么时候创建并进入新的作用域的问题

一个错误的做法是在`CompSt_handler`函数内部处理，但是这样是不合理的，因为函数定义中的参数和函数体{}应该位于同一个作用域。因此是否进入新的作用域应该由比较上层的节点控制函数决定。下层的节点只是“负责搬砖和完成上层节点分派的任务”。

> 考虑到函数和结构体的作用域为全局，因此我用了一个额外的变量`global_table`保存全局表，这两者的符号插入global表

（3）节点之间的信息传递

比较上层的处理函数需要负责在子节点之间传递好信息。比如`Def -> Specifier  DecList SEMI`。我的处理方案是先将`Specifier`与一个特定的类型绑定，其handler函数会返回一个type。这个type作为参数传递给declist。核心代码如下：

```
void def_handler(ASTNode *def, FieldList *fields) {
    // Def -> Specifier DecList SEMI
    Panic_on(strcmp("Def", def->node_name), "Def");
    ASTNode *specifier = def->child_list[0];
    Type type = specifier_handler(specifier); // 提取对应的Type
    decList_handler(def->child_list[1], type, fields); // 将Type作为参数传递给declist
}
```

类似的处理方式还体现在函数定义处理上，返回值类型需要作为参数传递给函数体{}，以方便进行返回值类型是否匹配的判断。其它的地方此处不再赘述。