编译原理实验二: 语义分析

组长 161130118 尹浚宇
组员 171850524 刘扬

一、实现功能

总体而言，我们完成了必做内容和选做 2.2，实现了如下功能：

1. 能够在实验一建立的语法树(程序无词法语法错误)上进行语义分析.
2. 在语义分析的过程中能够检测并输出程序中存在的语义错误.
3. 支持嵌套作用域对于变量定义的影响，即选做 2.2.

二、实现方法

(1) 类型系统的数据结构和实现细节

我们所使用的数据结构是由教程里给出 Type_和 FieldList_改造而来，详细见下图.

```
struct Type_
{
    enum { BASIC, ARRAY, STRUCTURE, FUNCTION } kind;
    union
    {
        enum { VOID_TYPE, INT_TYPE, FLOAT_TYPE } basic;
        struct { Type elem; int size; } array;
        Structure structure;
        Function function;
    } u;
};
```

具体而言，这里我们将所遇到的变量分为 4 类，分别对应基本类型，数组类型，结构体类型和函数类型. 对于每一种类型，在联合体 u 中又有专门的分量对其进行封装处理. 特别地，这里用基本类型中的 VOID_TYPE 给那些在识别类型时出错的标识符的类型赋值.

我们这里没有对教程 FieldList_类型做修改， 对于封装的结构类型和函数类型见下图.

```
struct Structure_ //结构体类型
{
    char* name;//结构体名字
    FieldList content;
};
struct Function_ //函数类型
{
    char* name;
    Type retType;
    FieldList paraType;
};
```

具体而言，结构体的 name 对应了结构体名字，content 存储了结构体各个域的信息，这两个部分足以描述一个结构体. 函数的 name 同样对应了函数名字，retType 用于存储函数返回值类型信息，paraType 存储了函数的形参信息，这些信息构成了一个完整的函数签名，足以对一个函数进行描述.

(2) 符号表的数据结构和实现细节

这里为了实现嵌套作用域的识别，我们采用了教程中提供的基于十字链表和 open

hashing 散列表的 Imperative Style 的符号表．具体实现如下：

```
struct SymNode //十字链表的节点
{
    char name[NAME_SIZE];
    Type type;
    SymList down;
    SymList right;
};
```

该变量对应于程序中可能遇到的符号，也就是符号表的结点．

```
struct Table_ //符号表
{
    SymList stack;//横向的栈
    SymList hashTable;//纵向的哈希表
    int stackTop;//栈顶
    Structure* structTable;
    int structTop;
};
```

同时，我们将整个符号表单独进行了一层封装，以结构体 Table_表示．除了教程中提到的栈和哈希表外，我们认为结构体定义是一种比较特殊的符号，它既是符号，又是类型声明符号，故这里为结构体定义单独设置了一个存储结构 structTable.

(3) 语义分析的实现方法

这里我们为 C--语言中的每一个产生式都设置了单独的函数进行处理， 在语法树上进行语义分析的过程可以视作一个自顶向上和自底向上相结合的过程． 具体而言， 对于需要向下传递的信息， 如变量的类型， 变量是否处于结构体中等，我们以参数的形式进行传递．对于需要向上传递的信息，如变量的名字等，我们通过返回值的方式进行传递．同时，为了提高整个编译器的鲁棒性， semantic.c 里大部分内容都使用了本地声明， 避免之后继续扩充内容可能导致的重名问题.每个函数的函数签名如下：

```c
// High-level Definitions (except Program because it has been declared in .h)
static void ExtDefList(Node *root);
static void ExtDef(Node *root);
static void ExtDecList(Type type, Node *root);

// Specifier
static Type Specifier(Node *root);
static Type StructSpecifier(Node *root);

// Declarators
static FieldList VarDec(Type type, Node *root, int isStruct);
static SymList FunDec(Type type, Node *root);
static FieldList VarList(Node *root);
static FieldList ParamDec(Node *root);

// Statements
static void CompSt(Type type, Node *root);
static void StmtList(Type type, Node *root);
static void Stmt(Type type, Node *root);

// Local Definitions
static FieldList DefList(Node *root, int isStruct);
static FieldList Def(Node *root, int isStruct);
static FieldList DecList(Type type, Node *root, int isStruct);
static FieldList Dec(Type type, Node *root, int isStruct);

// Expressions
static Type Exp(Node *root);
static int Args(Node *root, FieldList paramList);
```

三、编译方式

使用自带的 `makefile` 进行编译.