实验二报告 语义分析

王逊 131220134

一、实验目的：

在词法分析和语法分析程序的基础上编写一个程序，对 C——源代码进行语义分析和类型检查，并打印分析结果。

二、测试环境（编译步骤）：

GNU linux Release:Ubuntu 12.04.2　　GCC version: 4.6.3

GNU Flex version 2.5.35　　GNU Bison version2.5

提交的代码中包括 lexical.l,syntax.y tree.h,tree.c,main.c 五个文件。

编译步骤：

A) flex lexical -l 编译 Flex 源码，生成 lex.yy.c 文件

B) bison -d syntax.y 编译 Bison 源码，生成 syntax.tab.c 和 syntax.tab.h 文件

C) gcc main.c tree.c syntax.tab.c -lfl -ly -o parser （lex.yy.c 已经被 syntax.tab.c 引用了）生成可执行文件 parser

D)./parser test.c 执行测试代码，进行语义分析并打印结果。

三、实验内容：

必做部分：完成 1-12 种类型检查（普通变量、数组、函数）。

选做部分：完成 13-19 种类型检查（结构体和函数声明）。

实现方法：

（1）符号表的设计（详见 tree.h）：普通变量、数组、函数的类型检查大多只需要通过维护符号表和查找符号表来实现，所以符号表结构的设计显得至关重要。这里我的思路是采用哈希表的结构来维护符号表，所有变量（除结构体的参数之外）都直接作为表项存在符号表中。

```
typedef struct {
        char key[32];
        int is_func;
        char func_name[32];
        enum{Var,Struct,Func} type;
        union{
                var_symbol var_val;
                struct_symbol struct_val;
                func_symbol  func_val;
        }u;
}symbol_node;
```

插入的结点类型

```
typedef struct hash_node_{
        symbol_node *data;
        struct hash_node_ *prev,*next;
}hash_node;

#define NR_SYMBOL_TABLE 0X4000
hash_node symbol_table[NR_SYMBOL_TABLE];
```

存储的哈希表类型

所以符号表的每个表项需要记录变量名、变量类型，其中各种可能类型用联合表示节省空间。

（2）结点遍历和类型检查（详见 tree.c 中的 DFS 函数）：由于我并没有选择 SDT 的方式进行语义分析，而是深搜语法树，所以在结点间传递属性时，我采用的是继承属性加综合属性的

方式进行。整个 DFS 的过程主要就是访问不同类型的文法结点，根据具体情况计算不同的属性，只考虑父亲结点和子结点的继承属性和综合属性的计算。

```
L_dfvisit(n)
{
    for m=从左到右的n的每个子节点 do
    {
            计算m的继承属性；
            L_dfvisit(m);
    }
    计算n的综合属性。
}
```

（L-属性文法的语法翻译方案）

（3）普通变量、数组类型变量的实现：普通变量（例如 int a;float b）应该是需要插入符号表中结构最简单的部分了。在整个语言的文法中，普通变量是由 VarDec->ID 定义的，其类型是由 Specifier 给出，所以采用 L-属性文法，将变量的类型通过继承属性 inh_kind 传递到 ID 结点，在 VarDec->ID 这层使用综合属性 syn_kind 将变量类型传回父亲结点，并在上一层创建结点，插入符号表；数组变量（例如 int a[10]的实现在借鉴 ppt 上的语法翻译树之后，就很好实现了。数组是由 VarDec->VarDec LB INT RB 定义的，这是一个左循环文法表达式，采用的方法就是构建一个数组类型，包含当层个数 size 和子结点类型 children[0]->syn_kind，作为父亲结点的综合属性 syn_kind 返回给上面一层，如此就实现了多维数组类型。

```
else if(root->type==34) // 当前结点为VarDec
{
        if(root->children_number==1){ // VarDec -> id 普通类型
                root->syn_kind=root->inh_kind;
                strcpy(root->string_value,root->children[0]->string_value);
        }
        else if(root->children_number==4) // VarDec -> VarDec LB INT RB 数组类型
        {
            root->children[0]->inh_kind=root->inh_kind;
            DFS(root->children[0]);
            strcpy(root->string_value,root->children[0]->string_value);
            Type* new_kind=(Type*)malloc(sizeof(Type));
            new_kind->kind=Array;//构造新的数组类型作为综合属性返回值传回上一层
            new_kind->u.array.size=root->children[2]->int_value;
            new_kind->u.array.elem=root->children[0]->syn_kind;
            root->syn_kind=new_kind;
        }
}
```

（4）函数的实现：函数的定义在 ExtDef->Specifier FunDec CompSt,Specifier 是要提取的返回类型，FunDec 是要提取的形参个数和每个具体形参，Compst 是函数具体实现，可能需要提取的就是 RETURN Exp SEMI 返回的类型和 Specifier 是否一致。同之前一样，还是采用继承属性 inh_kind 传递返回类型给 FunDec 和 CompSt 结点，在 FunDec 中用于构造符号表的新表项，在 CompSt 中用于对比返回类型是否一致。FunDec->ID LP Deflist RP，将 Deflist 中的各个形参存在链表 args 中（同时也要插入符号表确保唯一性）。在调用函数时，通过编写的 cmp 函数比较调用的形参和定义的形参是否一致。

选做部分允许函数的多次声明，支持函数声明的产生式为 ExtDef->Specifier FunDec SEMI（在 bison 文件中添加），声明和定义标识不同的就是函数结构中定义的 is_defined 成员。具体实现时则只需修改 is_defined 成员，并且为了使得声明或定义过程中不再向符号表中添加重复形参导致报错，我的做法是通过结点的 is_func 成员和 func_name 成员来记录符号表中每个形参对应存在于哪个函数，额外定义或声明时会先查表是否存在这样的形参，如果有则不重复插入，否则对应的会报错。并且额外维护一个进行过函数声明的函数名称的链表，在

遍历结束时，依次查表看这些函数是否已经定义，没有就报错。

（5）结构体的实现：结构体的实现类似于函数和数组，大致思想就是每个成员变量按照之前实现之后，形成一个域（类似符号表结点的形式），再将这些域以链表的形式串起来，作为类型返回给 ID，构造成结构体结点插入符号表。主要是由 StructSpecifier -> STRUCT OptTag LC DefList RC（结构体自身的定义）、StructSpecifier -> STRUCT Tag（用已定义的结构体定义新的结构体变量）。值得一提的是，每个结构体成员变量并没有插入符号表中（不同结构体的成员变量可以相同），所以查看成员函数是否重名，就只好通过遍历域组成的链表（void redefined_field(Type* type)所实现的功能）来判断。

函数和结构体的实现且复杂，具体详见代码中函数和结构体定义部分。

实现了上述类型之后，符号表的插入和查询也就可以完全进行下去了，接下来比较复杂的就是类型检查部分。

（6）表达式类型检查：表达式有关的产生式以 Exp 为头。在进行类型检查过程中，最重要的是将表达式各部分相同的类型作为综合属性 syn_kind 传递上去。对于四则运算表达式，运算符两边类型不同，则操作符不匹配报错；对于逻辑关系表达式，返回的类型总是 Int 型；对于最基本的表达式 Int ，Float，ID,返回类型就是相应类型或符号表中存储类型；对于函数调用表达式，传递类型即返回类型；对于数组表达式，返回类型即数组元素类型；对于结构体表达式，返回类型即相应成员类型。如果一旦发生类型错误，为了不影响后续检测，在类型中多加入了一种 Error 类型，遇此类型一律返回 Error 类型并且不再类型不匹配错误。（表达式的实现在当前节点为 Exp 中，具体详见代码中的相应部分，这里不再赘述）。

（7）实现了上述所有功能之后，检测错误类型 1-19 基本都是通过填表和查表操作完成，具体实现详见 tree.c。

四、实验总结：

这次实验实现的内容多且烦，我就遵循着实现基本变量、数组、函数、结构体的顺序实现，数组类型的实现对其他类型的实现有着重要的指导作用。在实验的过程中对语法制导的翻译，尤其是 L 属性文法有了深刻的认识。由于情况较多，可能有一些没有考虑到的复杂情况没有实现，希望能在后面的实验之前改正过来。