编译原理实验

第二次试验报告

孙楷文 学号：111100027 （5班）

1. 代码文件

实验源代码文件包含：lexical.l、syntax.y、main.c、Makefile、semantic.c、semantic.h、syntaxtree.c、syntaxtree.h。另有用 Makefile 生成的部分文件也在提交的代码中。

本次实验的新增内容主要在 semantic.h 和 semantic.c 中。代码中注释非常详细。

用 make 或 make analysis 命令即可编译。生产可执行程序 parser。

2. 代码组成元素

2.1. 本次代码主要的数据结构有三个：

struct Type 主要用于表示定义的结构体类型，也可表示数组和 int、float 数据类型

struct Prop 用于表示变量或函数，其内有子结构 VarProp 和 FunProp。

struct FieldList 主要用于表示结构体中的域，内容与 Prop 相似。但因初期设计失误，导致部分应返回 Prop 属性的函数以 FieldList 作为了返回值。

2.2. 两个符号表（链表形式的栈）及作用域的表示：

stack_prop_top（链表头）记录了所有已经声明/定义了的变量和函数，以 struct Prop 为节点

stack_type_top（链表头）记录了所有已经定义了的结构体类型，以 struct Type 为节点

Prop 节点中有一个 int layer 项（其实 Type 中也有）用于记录表示该变量/函数的作用域。layer 的值由一个全局变量 int semantic_layer 赋予，semantic_layer 初值为0，在遇到“{”时加1，遇到“}”时减1。同时，在遇到“}”时把符号表栈顶的作用域失效的变量/函数弹栈，从而使得符号表栈中的元素实时对应当前作用域有效的变量/函数，使得栈中的元素的 layer 由栈底到栈顶递增，同时还可以根据 layer 支持嵌套定义。（函数定义中形参的作用域在函数体内，函数声明中形参的作用域转瞬即逝，这两者在代码中做了特殊处理。）

2.3. semantic.h 中的函数

本次实验新增的函数近50个，在 semantic.h 中有每个函数功能的说明，在 semantic.c 中有每个函数的详细说明，在此就不一一列举了。我把主要的函数分为了4个大类：

2.3.1. 基本元素操作：

包括与上述三种数据结构相关的复制、释放空间(free)、压栈（符号表）、弹栈（符号表）、判断两个元素结构/内容是否相同、在栈（符号表）中根据名称和指定的作用域查找元素、判断数据类型等。

2.3.2. 其它函数

（在.h 文件中这些“其他函数”被放在了第二的位置，再次姑且就先叫做“其他”吧）

“其他函数”包括驱动遍历整个语法树、打印语义错误信息、申请左值类型的空间、查找已声明未定义的函数、计算参数列表个数等等。

2.3.3. 语义节点属性构造函数

这些函数是驱动整个语义分析的重要组成部分。

它们的命名规则是“analyze_node_语义节点名”。运用了继承属性和综合属性的思想，把要进一步分析的子节点和所需的继承属性作为参数传给函数，把综合属性（如果有）作为函数的返回值。此外，部分属性构造函数会在不同的场景下被调用（如结构体内的 DefList 和 CompSt 中的 DefList），会影响到语义分析的细节，这种上层的场景状态通过参数 flag 传给子节点构造函数。

在语义分析的过程中，如何简便地确定语法树中当前节点的子节点的结构呢？（同一个符号可能

有多个产生式。）为了快速看出以当前节点为根的子树的结构，我的在代码中为语法树中每个节点信息(node->info)增添了一个成员项——产生式编号（node->info.ruleId），每个 ruleId 对应一个产生式（图1）：

```
ExtDef  :       Specifier ExtDecList SEMI
                {
                        struct Node* node = getNewNode();
                        strcpy(node->info.name,"ExtDef");
                        node->info.lineno = $1->info.lineno;
                        node->info.nodetype = STX;
                        node->info.ruleId = 4;
                        InsertNode(node,$1);
                        InsertNode(node,$2);
                        InsertNode(node,$3);
                        $$ = node;
                }
        |       Specifier SEMI
                {
                        struct Node* node = getNewNode();
                        strcpy(node->info.name,"ExtDef");
                        node->info.lineno = $1->info.lineno;
                        node->info.nodetype = STX;
                        node->info.ruleId = 5;
                        InsertNode(node,$1);
                        InsertNode(node,$2);
                        $$ = node;
                }
        |       Specifier FunDec CompSt
                {
                        struct Node* node = getNewNode();
                        strcpy(node->info.name,"ExtDef");
                        node->info.lineno = $1->info.lineno;
                        node->info.nodetype = STX;
                        node->info.ruleId = 6;
```

图 1：syntax.y 中，构造 ExtDef 节点的不同产生式有不同给的 ruleId

有了 ruleId，我的每个属性构造函数都是通过同一种模式在运行的：(1)声明返回值；(2)switch(node->info.ruleId)；（3）由 ruleId 判断子树（产生式）格式，并进一步分析子节点、分发继承属性、处理综合属性。图2是处理 Specifier 节点的属性构造函数。

```
struct Type* analyze_node_Specifier(struct Node* node_Specifier)
{
        /*
         * Specifier    :       TYPE (11)
         *              |       StructSpecifier (12)
         */
        struct Type* rtn_type = NULL;
        switch(node_Specifier->info.ruleId)
        {
                case 11:
                        {

                                rtn_type = analyze_node_TYPE(node_Specifier->child[0]);
                                break;
                        }
                case 12:
                        {

                                rtn_type = analyze_node_StructSpecifier(node_Specifier->child[0]);
                                break;
                        }
        }
        return rtn_type;
}
```

图 2：处理 Specifier 节点的属性构造函数。

2.3.4. 用于调试程序的函数

有三个，分别为：

```
void dbg_printPropStack(int need_detail);    打印当前变量/函数符号表中的信息
void dbg__printTypeStack(int need_detail);   打印当前结构体类型表中的信息
#define printf dbgprt                        用于 debug 的 printf，define 便于删除和使失效
```

3. 允许函数的预先声明，查找未已声明未定义的函数

若要允许函数在最外层声明，需要增加两个产生式

ExtDecList → FunDec
ExtDecList → FunDec COMMS ExtDecList

若要允许在函数内声明函数，需要增加两个产生式

DecList → FunDec DecList → FunDec COMMA DecList

但因为实验要求检查的错误不涉及函数内部的声明，所以我的 syntax.y 并没有采用后两个产生式。这也是为什么 ruleId 少了两个而不连贯。

对于函数节点，在 struct Prop 结构中的 FunProp 结构里，有这样两项：

```
int declared;                   //-1:未声明;+i:在第i行被声明(定义不算做声明)
int defined;                    //-1:未定义;+i:在第i行被定义;
```

语义分析最后，查找函数符号表中 declared>0 而 defined== -1 的函数节点即可发现已声明未定义的函数。

4. 在茂密的语法树的树形结构中不迷失

听闻身边许多同学都在抱怨“写着写着就不知道写到哪里了。”“写完这个子函数之后忘记了刚才上层被中断的写了一半的函数是哪个了。”本质上是大脑中记录了“上次写到哪里”的栈崩溃了。对此，我有“特别的技巧”。

按照继承属性、综合属性的思路构建代码的过程，也是一个对语法树做先序遍历的过程。先序遍历可以利用栈实现，于是我在写代码的过程中利用了 ruleId 和 《语法分析树进度.txt》图 3 这份文本，边写代码边写这份文本栈，避免了在庞大的树结构中迷路。

```
ExtDef
        ExtDef(4)
                Specifier(11)
                        TYPE(.)
                Specifier(12)
                        StructSpecifier(13)
                                OptTag(.)
                                DefList(34)
                                        Def(36)
                                                Specifier(recurs)
                                                DecList(37)
                                                        Dec(41)
                                                                VarDec(18)
                                                                VarDec(19)
                                                        Dec(42)
                                                DecList(38)
                                DefList(35)
                        StructSpecifier(14)
                ExtDecList(7)
                        VarDec(.)
                ExtDecList(8)
                        VarDec(.)
                        ExtDecList(.)
                ExtDecList(9)
                        FunDec(flag=1,20)
                                VarList(22)
                                        ParamDec(24)(.)
                                        VarList(.)
                        FunDec(flag=1,21)
                ExtDecList(10)
        ExtDef(5)
                Specifier(.)
        ExtDef(6)
                Specifier(.)
                FunDec(flag=2,20)(.)
                FunDec(flag=2,21)(.)
                CompSt(25)
                        DefList(2,34)
                                Def(2,36)
                                        Specifier(.)
                                        DecList(2,37/38)
```

图 3：语法分析树进度.txt