实验3 语义分析

202011081001 卢皓宇 2022年11月14日

实验要求

基于前面的实验，编写一个程序对使用 C—语言书写的源代码进行语义分析，输出语义分析中发现的错误（涉及 17 种错误类型）并完成实验报告，实验中主要使用 C 语言。

- 基本要求
    1. 对程序进行语法分析，输出语法分析结果；
    2. 能够识别多个位置的语法错误。
- 附加要求
    1. 修改假设3在文法中加入函数声明（需要修改文法）并能够检查函数声明涉及的两种错误：函数声明了但没定义、函数多次声明冲突；
    2. 修改假设4在假设“变量的定义可受嵌套作用域的影响，外层语句块中定义的变量可以在内层语句块中重复定义，内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内定义的局部变量可以相互重名”；
    3. 修改前面的C--语言假设5，将结构体间的类型等价机制由名等价改为结构等价。在新的假设5下，完成对错误类型1-17的检查。

实验分工

- 卢皓宇：主要完成符号未定义与重定义问题（Error 1~4、Error 13~17）以及附加要求1（Error 18~19）和附加要求2，协助对类型匹配问题（Error 5~12）进行debug；设计以Functional Style维护RBT栈以模拟作用域的思路；GitHub项目管理
- 姜潮涌：完成类型匹配问题（Error 5~12）的识别与报错，程序鲁棒性测试
- 段欣然：完成基于Linux kernel中rbtree的符号表封装及相应说明文档、函数实参形参不匹配问题（Error 9）、结构体赋值改进（附加要求3），完成栈模拟作用域等思路的实现。
- 杜隆清：协助完成附加要求1（Error 18~19）的识别与报错代码设计

实验环境

- Bison: v3.5.1
- Flex: v2.6.4
- 本地程序编写与测试
    - Windows Subsystem for Linux 2: Ubuntu 20.04 LTS
    - gcc: 9.4.0
- 云主机测试
    - BNUCLOUD云主机: Ubuntu 18.04 LTS
    - gcc: 7.5.0

实验设计

程序功能

在实验1词法分析和实验2语法分析的基础上，本次实验程序一共实现了对如下19种语义错误的识别：

1. 变量未定义
2. 函数未定义
3. 变量重定义
4. 函数重定义
5. 赋值号两侧类型不匹配
6. 赋值号左侧出现右值表达式
7. 操作数类型不匹配
8. Return类型不匹配
9. 函数实参形参不匹配
10. 对非数组变量进行数组访问
11. 对普通变量调用函数
12. 数组访问符中出现非整数
13. 对非结构体变量使用“.”
14. 访问结构体中未定义域
15. 结构体域名重复
16. 结构体名字重复
17. 使用未定义结构体定义变量
18. 函数声明但未定义（附加要求）
19. 函数声明冲突（附加要求）

以及附加要求2（变量定义与嵌套作用域）和附加要求3（结构体由名等价改结构等价）。由于涉及的工作量很大，为了节省篇幅、提升阅读体验~~绝对不是因为懒得写~~，下面我们仅以部分具有代表性的语义错误识别的实现为例，整体介绍我们的思路与实现过程。

功能实现

符号表

为了提高查找效率，我们采用**红黑树**作为存储变量、结构体和函数声明的数据结构。同时也是为了便于开发，我们基于Linux kernel 3.0的红黑树实现代码（rbtree.h & rbtree.c，添加宏offsetof和container_of）进行二次开发，自行编写RBTtest.c文件对红黑树的插入、删除和查找等基础维护操作进行封装，并基于我们维护符号表、结合多层作用域的需求，自定义了其他相关函数。

下面即符号表中结点MyType结构体定义。其中rb_root是rbtree.h中定义的结构体

```
typedef struct mytype {
    int def;  // 是否被定义
    char name[20];   // 变量名，主键
    int isvariable;  // 是否为VARIABLE
    char type[20];  // 类型
    int isstruct;  // 是否为STRUCT
    int isfunc;  // 是否为FUNCTION
    int isarr;  // 是否为ARRAY
    int dimension;   // 数组维度
    char return_type[20];  // func返回类型
    struct rb_root varilist; // 结构体和函数的属性/参数列表
} MyType, *Mylink;

MyType MyType_default = {0, "", 0, "", 0, 0, 0, 0, ""};
```

> 当然红黑树有一个主要的问题，就是为了保证查找效率会在插入时自行维护（着色 or 旋转），因而不能直接通过某种遍历顺序来“还原”符号在原代码中的顺序。
>
> 这里涉及到的最典型的问题是函数形参列表与调用时实参的对比（Error 9），如声明函数int func(int a, float b)，我们认为在调用该函数时func的第1个参数就必须是int型，第二个参数必须是float型，不可调换。后面的部分我们会详细说明如何实现

组织方式与多层作用域

- 为了便于以作用域为界区分各声明符号，我们将**一个作用域内的所有符号**组织成一张符号表（一棵红黑树）
- 在此基础上，为了支持多层作用域的符号声明，我们采用**Functional Style**，即维护一个符号表栈（红黑树栈，下称RBT栈）。具体而言：
  - 每遇到{}包含的语句块，就会先往RBT栈中压栈一棵空RBT
  - 语句块结束后弹出该RBT，即删除该局部作用域
  - 声明的符号直接插入栈顶的RBT

    > 注意：为了结构体可能存在的嵌套定义和函数可能存在的递归调用，我们在匹配到这两者时会将其插入栈顶的RBT和栈顶的上一级RBT（对应结构体/函数内部的作用域和上一级更接近全局的作用域），经实测不会影响其他错误的识别与处理

相应结构体定义如下

```
typedef struct table_stack {
    struct rb_root my_root;  // 当前作用域
    int top;   // 作用域编号，原本是头指针，现在debug用
    struct table_stack *last;  // link上一个作用域
}VariTables, *VariLink;
```

符号表的建立与使用

建立

- 在syntax.y文件的开始声明一个VariLink类型的全局变量this_scope，即符号表栈
- 在每一处涉及语义检查的产生式中定义MyType类型的局部变量tmp用以存储相应的变量名/结构体名/函数名以及相应的类型/域名/形参列表

使用

- 对于符号的声明，我们在每一次匹配到声明时都**只在当前作用域**（即目前栈顶的符号表/RBT）搜索是否存在同名的符号。若存在则说明在同一作用域里重复声明了同名符号，报重复定义错；否则向栈顶符号表插入该符号结点
- 对于符号的使用，我们会对**整个RBT栈**进行搜索，查找是否存在同名符号。若存在则说明被声明过，没有问题；否则报未定义错

符号的属性参数传递

之所以写这个部分，是因为我们在多人协作写不同部分的报错处理代码时，某一次merge后发现实验2中实现的**没有错误时打印语法树的功能会段错误**；经过大家反复调试和探讨后确定，是因为一位同学在Exp部分（较多依赖符号的使用问题）的错误检查代码中，没有采用孩子兄弟指针（child-bro）去自顶向下地找到对应结点的属性复制给一个临时符号的思路，而是自底向上的传递结点属性。例如产生式

```
Exp : ID
```

的错误检查代码中，如果写

```
MyType* mt = search(this_scope, tmp); // 寻找该变量的声明结点
if(mt != NULL) {
    $$->name = mt->type; // 将变量的类型 (char*) 传递至其父结点Exp的"类型" (char*)
    tmp.def = 1;
}
```

那么就会导致父结点Exp的name发生变化，何况C并不允许char*类型的变量在声明后单独用=赋值，所以导致了段错误。

当然这种自底向上的传递方式并不是导致段错误的直接原因，直接原因应该是赋值语句的问题。但是这种做法依然会打乱甚至破坏先前的语法树打印功能，兼容性相对较差。因此我们的实际做法是

```
| ID {
        $$ = insNode($1, "Exp", @1.first_line, NON_TERMINAL);

        MyType tmp = MyType_default;
        strcpy(tmp.name, $1->id);
        MyType* mt = search(this_scope, tmp); // 寻找该变量的声明结点
        if(mt != NULL) {
            tmp.def = 1;
        }
        else { // 变量未定义
            char msg[100];
            sprintf(msg, "Error %d at line %d : Undefined variable \'%s\'", UNDEFINED_VARIABLE, last_row, tmp.name);
            myerror(msg);
        }
    }
```

> 另外值得一提的是，原本我们所有的“数组”类型变量都是以指针形式声明的，符号表结点MyType类型变量也是全局声明的，在每一次用完后需要“清空”。但后来我们发现了结构体指针的深拷贝与浅拷贝问题，所以我们将数组改为一般的定义形式（如char name[20]），符号表结点MyType类型变量也换成了局部声明

举例说明

产生式存在递归：结构体声明

这个问题其实很常见，比如函数声明时的形参列表、结构体声明的域名等，他们基本对应到具有右递归的产生式

```
VarList : ParamDec COMMA VarList {
    ...
}
| ParamDec {
    ...
}

Mid : Def Mid {
    ...
}
| Stmt Mid {
```

```
    ...
}
| {
    ...
}
```

解决这个问题的基本思路是：while(还有剩余符号): 循环插入符号

以结构体声明为例，如test5.cmm中

```
struct sa{
        int a;
        float b;
        int b;          // error 15
}c;
```

整个声明应该被规约为StructSpecifier(STRUCT OptTag LC Mid RC) ExtDecList SEMI，其中**Mid**对应的就是结构体sa中的域名声明。进一步有

1. Mid -> Def Mid
2. Def -> Specifier DecList SEMI
3. DecList -> Dec COMMA DecList

对于test5这一样例，我们只需考虑带右递归产生式1即可满足要求。具体来说，要判断是否还有剩余域名没有插入该结构体的域名列表varilist，我们只需要看Mid->child是否为空指针

- Mid->child为空说明这里Mid -> ε，域名的定义结束了
- 否则说明Mid -> Def Mid，还有未插入域名列表varilist的域名结点

为此，我们引入临时的node类型指针newnode，在while循环中维护它使其始终指向Mid，即

```
if(newnode->child) { // strcmp(newnode->child->bro->name, "Mid") == 0
     newnode = newnode->child->bro;
}
else break;
```

由此循环插入域名即可实现结构体域名列表的存储。但是如果我们不局限于test5的情况，写如下C语言代码：

```
struct sa{
        int a, i, t;
        float b;
        int b;          // error 15
}c;
```

即同一类型标识符后多个同类型域名，也是可以编译通过的。所以为了严谨，我们也考虑了右递归产生式3，也就是DecList -> Dec COMMA DecList的情况。该情况思想与上述思路基本一致，故不赘述。下面是结构体声明对应的插入符号与错误处理代码

```
StructSpecifier : STRUCT OptTag LC Mid RC {
        $$ = insNode($1, "StructSpecifier", @1.first_line, NON_TERMINAL);
        $1->bro = $2;
        $2->bro = $3;
        $3->bro = $4;
```

```
        $4->bro = $5;

        MyType tmp = MyType_default;
        strcpy(tmp.name, $2->child->id);
        strcpy(tmp.type, $1->id);
        if(search(this_scope, tmp)) { // 结构体名字重复
            char msg[100];
            sprintf(msg, "Error %d at line %d : Duplicate name \'%s\'", REDEFINED_STRUCT, last_row, tmp.name);
            myerror(msg);
        }
        else {
            tmp.def = 1;
            tmp.isstruct = 1;
            // printf("%s\n", tmp.name);
            struct node* newnode = $4; // newnode保持指向Mid

            do {
                MyType temp = MyType_default;
                temp.def = 1;
                temp.isvariable = 1;
                strcpy(temp.type, newnode->child->child->child->id);

                struct node* newnew = newnode->child->child->bro; // newnew保持指向DecList
                do {
                    strcpy(temp.name, newnew->child->child->child->id);
                    // printf("Struct \'%s\' has variable \'%s\' of type \'%s\'\n", tmp.name, temp.name, temp.type);
                    int result = my_insert(&tmp.varilist, temp);
                    // printf("Whether successful: %d\n", result);

                    if(newnew->child->bro != NULL) {
                        newnew = newnew->child->bro->bro;
                    }
                    else break;
                } while(newnew != NULL);

                if(newnode->child) { // strcmp(newnode->child->bro->name, "Mid") == 0
                    newnode = newnode->child->bro;
                }
                else break;
            } while(newnode->child);

            this_scope = insert(this_scope, tmp); // 插入当前作用域
            int result = my_insert(&this_scope->last->my_root, tmp); // 插入上一级作用域
            // 这里是为了pop结构体的子作用域时不会扔掉结构体这个结点

            tmp.def = 0;
            tmp.isstruct = 0;
            // flgStruct = 1;
        }

```

```
        this_scope = pop_scope(this_scope);
    }
```

> **注意**：这里我们没有处理域名重复的问题，是因为DecList的符号插入与错误处理代码中已经实现在同一作用域重复定义的报错了。我们的具体做法是
>
> - 设置一个全局变量int flgStruct作为结构体声明的标识符，默认为0，在yf.l中定义匹配到token STRUCT就将其置为1，在整个声明的产生式规约完成后置回0
> - 在DecList/ExtDecList中，如果查找到同一作用域存在同名符号，则需要判断一下flgStruct：若为1则是struct内的域名重复，输出"Redefined field"；若为0则是一般作用域内的变量重复，输出"Redefined variable"

红黑树的“顺序”问题：函数形参实参对比

除了函数形参实参对比，在结构体赋值（附加要求）中也需要顺序访问红黑树，我们用一组约定好的、“不合法”的变量名有序地保存函数形参和结构体内部变量。下面以函数形参实参对比为例具体阐述。

在保存形参列表时，我们每次保存形参，将同时保存一个虚拟参数。该参数的name为“XX_varifunc”，其中XX为自增的顺序编号。

顺序保存

```
struct node* n = $2->child->bro->bro;
char varifunc[12] = {"00_varifunc"};
do {
    MyType t = MyType_default;
    t.def = 1;
    t.isvariable = 1;
    strcpy(t.type, n->child->child->child->id);
    strcpy(t.name, n->child->child->bro->child->id);
    int result = my_insert(&tmp.varilist, t); // 插入形参

    strcpy(t.name, varifunc);
    /*约定虚拟参数变量名自增*/
    varifunc[1] += 1;
    if(varifunc[1] > '9'){
        varifunc[0] += 1;
        varifunc[1] = '0';
    }

    result = my_insert(&tmp.varilist, t);  // 插入虚拟参数

    if(n->child->bro) {
        n = n->child->bro->bro;
    }
    else break;
} while(n);
```

顺序访问

对于产生式`Exp : ID LP Args RP`，我们通过如下方式顺序对比函数形参实参。

```
struct node* newnode = $3;
char varifunc[12] = { "00_varifunc" };
char Parameter[10][10]; // 形参数组，用于输出报错
```

```
char Arguments[10][10]; // 实参数组，用于输出报错
int right = 1, vari_num = 0, para_num = 0;  // right标识合法性 1.合法 0.非法，vari_num，para_num用于输出
MyType parameter = MyType_default; // 形参
do { // 函数的参数列表 实参
    strcpy(parameter.name, varifunc);
    struct my_node* ttp = my_search(&(mt->varilist), parameter);
    if (ttp != NULL) {
        parameter = ttp->info;
    }  // 为NULL在后面判
    /* 分别讨论实参的类型 */
    if (strcmp(newnode->child->child->name, "ID")) {  // 非变量名
        char argu[20];
        if (strcmp(newnode->child->child->name, "INT") == 0) {
            strcpy(argu, "int");
        }
        else if (strcmp(newnode->child->child->name, "FLOAT") == 0) {
            strcpy(argu, "float");
        }
        else {
            printf("没听说这种变量类型QAQ\n");
            exit(0);
        }
        strcpy(Arguments[vari_num++], argu); // 用于输出
        if (ttp == NULL || strcmp(parameter.type, argu)) {  // 对比形参实参
            right = 0;
        }
        if (ttp != NULL) {  // 用于输出
            strcpy(Parameter[para_num++], parameter.type);
        }
    }
    else { // 变量名
        MyType argu = MyType_default;
        strcpy(argu.name, newnode->child->child->id);
        Mylink tttp = search(this_scope, argu);  // 搜索实参以获得其类型
        if (tttp != NULL) {
            argu = *tttp;
            strcpy(Arguments[vari_num++], argu.type);
            if (ttp == NULL || strcmp(parameter.type, argu.type)) {
                right = 0;
            }
            if (ttp != NULL) {
                strcpy(Parameter[para_num++], parameter.type);
            }
        }
    }
    /*约定虚拟参数变量名自增*/
    varifunc[1] += 1;
    if (varifunc[1] > '9') {
        varifunc[0] += 1;
        varifunc[1] = '0';
    }

    if (newnode->child->bro != NULL) {
        newnode = newnode->child->bro->bro;
```

```
    }
    else break;
} while (newnode != NULL);
strcpy(parameter.name, varifunc);
/* 检查函数剩余形参 */
struct my_node* tpp = my_search(&(mt->varilist), parameter);
while (tpp) {  // 若该情形下仍有剩余形参，加入数组用于输出并将right置0
    parameter = tpp->info;
    strcpy(Parameter[para_num++], parameter.type);
    varifunc[1] += 1;
    if (varifunc[1] > '9') {
        varifunc[0] += 1;
        varifunc[1] = '0';
    }
    strcpy(parameter.name, varifunc);
    tpp = my_search(&(mt->varilist), parameter);
    right = 0;
}
if (right == 0) {  // 非法输出报错，errors++
    printf("Error %d at line %d : Function \'%s(", FUNCTION_MISMATCH,
last_row, mt->name);
    for (int i = 0; i < vari_num; i++) {
        if (i != vari_num - 1)
            printf("%s,", Arguments[i]);
        else
            printf("%s", Arguments[i]);
    }
    printf(")\' is not applicable for arguments \'%s(", mt->name);
    for (int i = 0; i < para_num; i++) {
        if (i != para_num - 1)
            printf("%s,", Parameter[i]);
        else
            printf("%s", Parameter[i]);
    }
    printf(")\'");
    printf("\n");
    errors++;
}
```

对于产生式`Exp : ID LP RP`，只需判断函数声明时是否有形参即可，对应上述代码段中`/* 检查函数剩余形参 */`部分。

实验结果

test1.cmm

```
None!!!
```

test2.cmm

```
Error 3 at line 4 : Redefined variable 'i'
Error 1 at line 5 : Undefined variable 'j'
Error 2 at line 6 : Undefined function 'inc'
Error 8 at line 12 : Type mismatched for return
Error 4 at line 12 : Redefined function 'main'
```

test3.cmm

```
Error 5 at line 3 : Type mismatched for assignment
Error 5 at line 4 : Type mismatched for assignment
Error 6 at line 6 : The left-hand side of assignment must be a variable
Error 7 at line 7 : Type mismatched for operands
Error 8 at line 9 : Type mismatched for return
```

test4.cmm

```
Error 9 at line 8 : Function 'func(int)' is not applicable for arguments
'func()'
Error 9 at line 9 : Function 'func2(int,int)' is not applicable for arguments
'func2(int,float)'
Error 10 at line 10 : 'i' is not an array
Error 11 at line 11 : 'i' is not a function
Error 12 at line 12 : '1.500000' is not an integer
```

test5.cmm

```
Error 15 at line 4 : Redefined field 'b'
Error 16 at line 11 : Duplicate name 'sf'
Error 17 at line 15 : Undefined struct 'sk'
Error 14 at line 18 : Non-existing field 'f'
Error 13 at line 20 : Illegal use of '.'
```

ext1.cmm

```
Error 19 at line 3 : Incompleted definition of function 'f2'
Error 19 at line 4 : Incompleted definition of function 'f2'
Error 18 at line 1 : Undefined function 'f1'
Error 18 at line 2 : Undefined function 'f2'
```

eext2.cmm

```
Error 5 at line 14 : Type mismatched for assignment
```

实验反思

1. 红黑树的查找效率好于链表等顺序数据结构，但是因为其自身维护的特性，无法做到顺序读写，因此会给参数列表的顺序对比等问题带来麻烦；此外，链表等一般的顺序数据结构在本项目中应该会表现出更好的”复用性“：除了符号表可以用链表等存储之外，多维数组的索引涉及动态分配、多个函数的声明与实现等等都可以用链表更方便地存储与管理。当然，受限于C本身难以实现”泛型“（尤其是分配内存空间的问题上），实现这些功能可能需要对不同数据类型、甚至是自己定义的结构体类型各实现一种链表，这又会带来额外的麻烦；

2. 关于g++编译的尝试：我们在第1次实验，即词法分析实验中通过一些代码修改，成功使用g++编译出工作正常的scanner；但第2次实验以及本次实验，引入bison后语法分析器的g++编译实现却始终没能成功。时间精力有限，目前推测的原因是bison对语义值等存储的数据类型要求只能使用POD (Plain Old Data)类型，而char*——也就是C风格的"字符串"——并不是POD类型，因而不能通过编译。如果可以用g++编译的话，符号表就无需手写链表或是手动封装RBT、而是直接调用STL中的vector、map等容器，项目构建与维护都会更加高效。