# 编译原理

# 实验二

## 1 程序功能

程序在实验一词法分析和语法分析的基础上，实现了对 C--源代码的语义分析与类型检查，并打印出分析结果。

本实验除了完成最基础的 17 个错误类型检测之外，还在要求 2.2 的假设之下，实现了对变量作用域的识别，允许不同语句块中重名变量的出现。

为了实现对源程序的语义分析，要使用符号表来记录源程序中各种名字（如变量名和函数名等）的特性信息（包括具体类型、维数和参数个数等）。实验首先便是对符号表的数据结构进行设计。

本实验将散列表用于符号表数据结构中，并使用指导书上 P.J.Weinberger 所提出的 hash 函数进行映射，将 hash 值相同的各个名字链接起来。此外，为了实现要求 2.2 的假设，还需要记录各个名字的嵌套层数，并维护当前名字所对应的所有外层名字，于是又引入了栈这一结构，将各名字按所在嵌套层数进行链接。所以，本实验最终的符号表是基于十字链表和散列表的，具体数据结构如下：

```
typedef struct tableItem {
    int symbolDepth;
    pFieldList field;
    pItem nextSymbol;
    pItem nextHash;
} TableItem;

typedef struct hashTable {
    pItem* hashArray;
} HashTable;

typedef struct stack {
    pItem* stackArray;
    int curStackDepth;
} Stack;

typedef struct table {
    pHash hash;
    pStack stack;
    int unNamedStructNum;
} Table;
```

对于类型表示，参考了指导书上的结构体定义，将类型分为基本类型、数组类型、结构体类型和函数类型，具体如下：

```
typedef enum kind { BASIC, ARRAY, STRUCTURE, FUNCTION } Kind;
typedef enum basicType { INT_TYPE, FLOAT_TYPE } BasicType;
typedef struct type {
    Kind kind;
    union {
        BasicType basic;

        struct {
            pType elem;
            int size;
        } array;

        struct {
            char* structName;
            pFieldList field;
        } structure;

        struct {
            int argc;
            pFieldList argv;
            pType returnType;
        } function;
    } u;
} Type;
```

其中，对于结构体和函数，其内部可能会有多个需要记录在符号表中的名字（如结构体中的域和函数的形参），于是又引入一个结构体来存放这些信息：

```
typedef struct fieldList {
    char* name;
    pType type;
    pFieldList tail;
} FieldList;
```

同时，程序中为上述各个结构体都封装了一些函数，例如最基础的 new（创建新的结构体）和 delete（释放内存空间）等，还有一些各个结构体特有的函数，例如 Type 的 checkType（用于检查两个类型是否相同）和 Table 的 checkTableItemConflict（用于检查当前名字是否与已存在于符号表中的名字冲突）等。通过对这些常用函数的封装，能直观地看出各个结构体所对应的功能，且能大大方便后续的语义分析。

在进行语义分析时，需要遍历实验一中已构造出来的语法分析树，每当遇到 ExtDef 节点便意味着有变量、结构体或函数的定义出现，需要从该节点开始进行分析，对于无错误的定义，要创建相应的 tableItem 并插入到符号表中；对于有错误的定义，需要进行错误类型的报告。在对 ExtDef 节点进行分析时，会根据其子节点情况调用各种非终结符节点对应的处理流程，充分利用各个节点的综合属性与继承属性获取当前定义所对应的名字、类型和嵌套层数等信息。

现用下面代码简单说明错误类型 1 其中一个检查过程的实现：

```
// Generate symbol table functions
void ExtDef(pNode node) {
    assert(node != NULL);
    // ExtDef -> Specifier ExtDecList SEMI
    //         | Specifier SEMI
    //         | Specifier FunDec CompSt
    pType specifierType = Specifier(node->child);
    char* secondName = node->child->sibling->name;

    // ExtDef -> Specifier ExtDecList SEMI
    if (!strcmp(secondName, "ExtDecList"))
        ExtDecList(node->child->sibling, specifierType);

    // ExtDef -> Specifier FunDec CompSt
    else if (!strcmp(secondName, "FunDec")) {
        FunDec(node->child->sibling, specifierType);
        CompSt(node->child->sibling->sibling, specifierType);
    }
    if (specifierType)
        deleteType(specifierType);

    // ExtDef -> Specifier SEMI
    // this situation has no meaning
    // or is struct define(have been processe in Specifier())
}

void ExtDecList(pNode node, pType specifier) {
    assert(node != NULL);
    // ExtDecList -> VarDec
    //             | VarDec COMMA ExtDecList
    pNode temp = node;
    while (temp) {
        pItem item = VarDec(temp->child, specifier);
        if (checkTableItemConflict(table, item)) {
            char msg[100] = {0};
            sprintf(msg, "Redefined variable \"%s\".", item->field->name);
            pError(REDEF_VAR, temp->line, msg);
            deleteItem(item);
        }
        else
            addTableItem(table, item);
        if (temp->child->sibling)
            temp = temp->sibling->sibling->child;
        else
            break;
    }
}
```

首先调用 ExtDef 函数，将其第一个节点作为参数传给 Specifier 函数，获得当前定义对应的类型 specifierType。接下来发现当前节点的第二个子节点名字为 ExtDecList 时，将该子

节点和 specifierType 一起作为参数传给 ExtDecList 函数。在 ExtDecList 函数中，先是调用 VarDec 函数查找出第一个变量定义，接下来调用 checkTableItemConflict 函数查找该定义是否与符号表中已有定义存在冲突，若有冲突，则报告“Redefiend variable”的错误；否则将该变量定义插入符号表中。

对于更多的非终结符节点对应处理流程，其构造思路和上述函数一致，即通过对当前节点各个子节点名字的判断，调用与名字对应的子函数，在层层调用和返回之后，获得每条定义所对应的名字、类型和嵌套层数等信息。当信息获取完整后，进行冲突查询，并根据查询结果进行符号表的插入或错误类型报告。

## 2 程序运行

在源程序所在文件夹下进入终端，输入以下指令，即可得到编译结果 parser2:

```
zr@ubuntu:~/compilerlab/lab2$ gcc main.c syntax.tab.c semantic.c -lfl -o parser2
```

将测试用例也置于该文件夹下，测试时，在终端输入“./parser2 ”加上待测试文件名即可得到结果。例如，若要对 test1.cmm 文件进行测试，输入指令及结果如下:

```
zr@ubuntu:~/compilerlab/lab2$ ./parser2 test1.cmm
Error type 1 at Line 4: Undefined variable "j".
```