哈尔滨工业大学　编译系统 2023 春

实验二 语义分析

学院 计算学部 | 姓名 王炳轩 | 学号 120L022115 | 指导教师 陈鄞

一、 实验目的

a) 巩固对词法分析与语法分析的基本功能和原理的认识
b) 能够应用自动机的知识进行词法与语法分析
c) 理解并处理词法分析与语法分析中的异常和错误

二、 实验内容

(一) 实验环境

Ubuntu 12.04, kernel version 3.2.0-29; GCC version 4.6.3
GNU Flex version 2.5.35; GUN Bison version 2.5

(二) 实验过程

本次实验实现了所有的必做要求以及选做要求 2.2。

1. 数据结构设计

设计了符号表、栈。栈用于存放哈希表的指针，用以保持标识符的作用域。
由于本次实验只需要做分析，无需生成，因此符号表被生成、使用之后就会被舍弃，无需有级联的指针指向。
每个符号表和栈的元素都是一个叫做 MemberList 的单向链表（包括 3 个成员：name, type, next），next 指向了下一个节点，name 为该标识符的名字，type 为该标识符的类型和相关参数。

Type：包括枚举值 kind 和对应 kind 的参数。

BASIC	basic：枚举 INT_TYPE, FLOAT_TYPE;
ARRAY	elem：Type*，元素类型； size：数组大小
STRUCTURE	structName：char*名称 field：MemberList*单向链表
FUNCTION	argc：参数个数 argv：MemberList* 参数列表（单向链表） returnType：Type* ，返回值类型

同时定义相关函数，如 initTable、deleteTable，checkTableItemConflict、addTableItem、deleteItem 等。

2. 语义分析

语义分析基于词法和语法分析。这里我们设置词法错误、语法错误的标志位，如果没有出现错误，即成功建立语法树，之后再按照深度优先的顺序遍历一遍树，遍历树的过程中计算属性并检查语义错误。

```
yyrestart(f);
yyparse();
if (!lexError && !synError) {
    table = initTable();
    // printTreeInfo(root, 0);
    traverseTree(root);
    deleteTable(table);
}
delNode(&root);
return 0;
```

语义分析是根据语法树的遍历而来的，而语法

树的构建来自于产生式。这里我们曾在词法分析中构造的语法树结点中存在 Name 字段，我们使用该 Name 字段来对文法符号进行判断，并逐步执行对应于语法动作的函数（将当前语法树的结点指针传入）来计算各类属性。

```
Exp -> Exp ASSIGNOP Exp
     | Exp AND Exp
     | Exp OR Exp
     | Exp RELOP Exp
     | Exp PLUS Exp
     | Exp MINUS Exp
     | Exp STAR Exp
     | Exp DIV Exp
     | LP Exp RP
     | MINUS Exp
     | NOT Exp
     | ID LP Args RP
     | ID LP RP
     | Exp LB Exp RB
     | Exp DOT ID
     | ID
     | INT
     | FLOAT
```

下面，我将以**“表达式分析”**简述如何进行语义分析。

如语法单元 Exp 的定义如右图所示。当进行语义分析时，将会在 Exp(pNode node)函数中进行分析，传入的 node 结点既是当前的 Exp 语法树结点，我们要对该结点的子结点进行分析和计算，来保证符合语法规则要求。

该函数的框架如下图所示。我们通过 if-else-if...进行级联判断，判断当前首个子结点的类型，然后具体产生式具体分析。

```
//二值运算
if (!strcmp(t->name, "Exp")) {
    // 基本数学运算符
    if (strcmp(t->next->name, "LB") && strcmp(t->next->name, "DOT"))
    // 数组和结构体访问
    else {
        // Exp -> Exp LB Exp RB
        if (!strcmp(t->next->name, "LB")) {…
        // Exp -> Exp DOT ID
        else {…
    }
}
//单目运算符
// Exp -> MINUS Exp
//      | NOT Exp
else if (!strcmp(t->name, "MINUS") || !strcmp(t->name, "NOT")) {…
} else if (!strcmp(t->name, "LP")) {…
// Exp -> ID LP Args RP
//      | ID LP RP
else if (!strcmp(t->name, "ID") && t->next) {…
// Exp -> ID
else if (!strcmp(t->name, "ID")) {…
} else {
    // Exp -> FLOAT
    if (!strcmp(t->name, "FLOAT")) {…
    // Exp -> INT
    else {…
```

例如，在大儿子结点是“Exp”且二儿子结点时“LB”时，对应于产生式 Exp→Exp LB Exp RB，即数组引用。这时候，将大儿子结点、三儿子结点传入 Exp 函数进行细节分析，分别得到结果 p1\p2，之后检测 p1 是一个数组，p2 是一个 int，如果不是则报错。

```
// Exp -> Exp LB Exp RB
if (!strcmp(t->next->name, "LB")) {
    //数组
    pType p1 = Exp(t);
    pType p2 = Exp(t->next->next);
    pType returnType = NULL;

    if (!p1) {// 第一个exp为null，上层报错，这里不用再管
    } else if (p1 && p1->kind != ARRAY) {//报错，非数组使用[]运算符…
    } else if (!p2 || p2->kind != BASIC ||
               p2->u.basic != INT_TYPE) {//报错，不用int索引[]…
    } else {…
    if (p1) deleteType(p1);
    if (p2) deleteType(p2);
    return returnType;
}
```

在上例中的细节分析时，直到最终结点 ID，我们搜索我们之前创建的符号表是否包含这个域，如果有则说明之前已经定义该变量名，同时返回该变量的 Type 用于后续检验。如果符号表中没有该信息，我们则报错“变量未定义”。

下面时第二个例子，变量定义，使用 Dec 函数，传入当前结点、继承的类型、是否处于结构体内三个信息。然后针对这两条产生式分别处理。

```
void Dec(pNode node, pType specifier, pItem structInfo) {
    assert(node != NULL);
    // Dec -> VarDec
    //      | VarDec ASSIGNOP Exp

    // Dec -> VarDec
    if (node->child->next == NULL) {
        if (structInfo != NULL) { // 结构体内，将VarDec返回的Item中的filedList…
        } else {
            // 非结构体内，判断返回的item有无冲突，无冲突放入表中，有冲突报错delete
            pItem decitem = VarDec(node->child, specifier);
            if (checkTableItemConflict(table, decitem)) {//出现冲突，报错…
            } else {
                addTableItem(table, decitem);
            }
        }
    }
    // Dec -> VarDec ASSIGNOP Exp
    else {
        if (structInfo != NULL) {…
        } else {
            // 判断赋值类型是否相符
            //如果成功，注册该符号
            pItem decitem = VarDec(node->child, specifier);
            pType exptype = Exp(node->child->next->next);
            if (checkTableItemConflict(table, decitem)) {…
            if (!checkType(decitem->field->type, exptype)) {…
            if (decitem->field->type && decitem->field->type->kind == ARRAY) {
            } else {
                addTableItem(table, decitem);
            }
            // exp不出意外应该返回一个无用的type，删除
            if (exptype) deleteType(exptype);
        }
    }
}
```

如第二个产生式有赋初值的定义，要先获得大儿子和三儿子的结点信息，然后去检查是否和当前的符号表存在冲突、检查赋值类型匹配、是否针对数组或结构体赋值（不支持，报错），如果都没问题则添加到符号表。

如第一个产生式，单纯的变量定义，先判断是否在结构体内，如果在，需要在符号表的结构体内判断、注册；如果不在则需要先计算儿子，检查冲突，没有问题

添加到符号表。

3. 编译过程

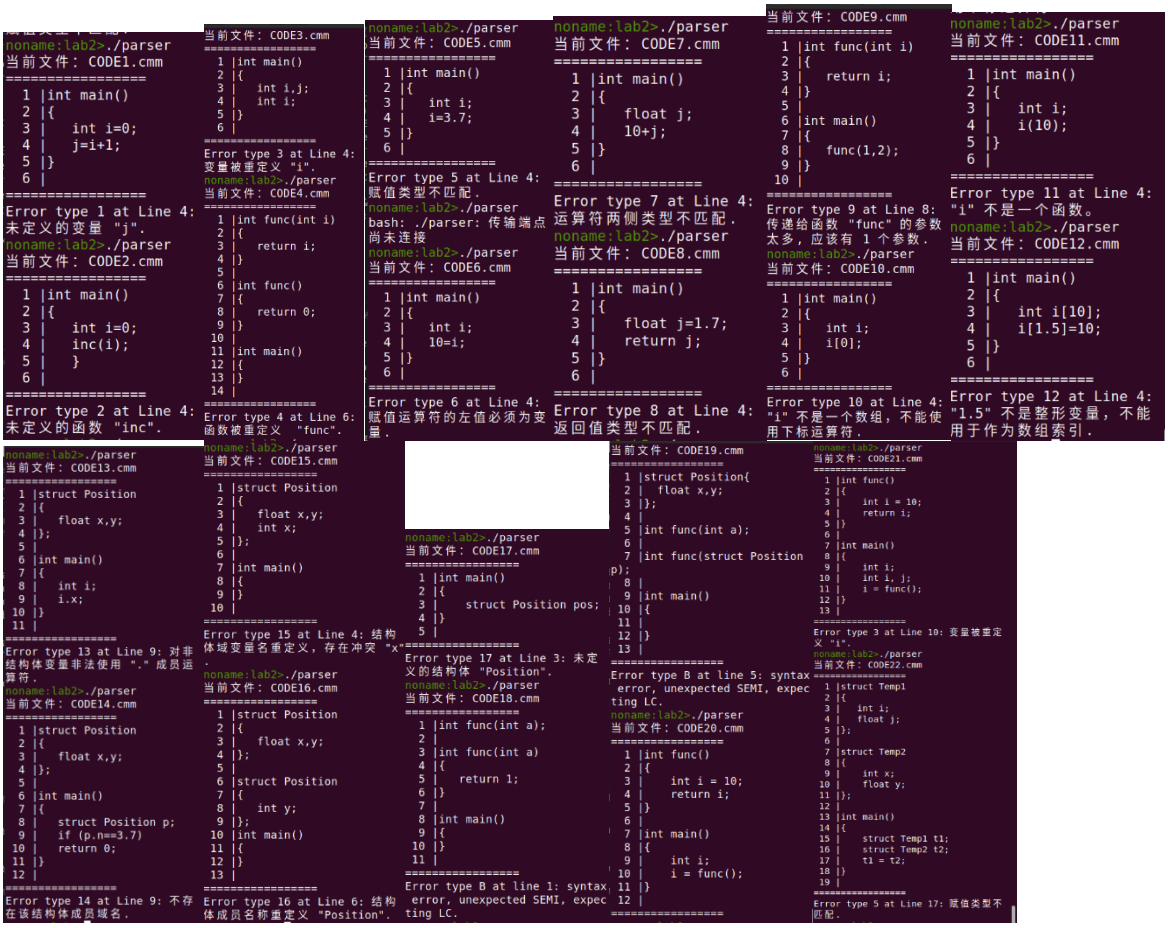
本次实验采用 Makefile 进行编译。具体内容如右图所示：

```
# GNU make手册：http://www.gnu.org/software/make/manual/make.html
# ************ 遇到不明白的地方请google以及阅读手册 ************

# 编译器设定和编译选项
CC = gcc
FLEX = flex
BISON = bison
CFLAGS = -std=c99

# 编译目标：src目录下的所有.c文件
CFILES = $(shell find ./ -name "*.c")
OBJS = $(CFILES:.c=.o)
LFILE = $(shell find ./ -name "*.l")
YFILE = $(shell find ./ -name "*.y")
LFC = $(shell find ./ -name "*.l" | sed s/[^/]*\\.l/lex.yy.c/)
YFC = $(shell find ./ -name "*.y" | sed s/[^/]*\\.y/syntax.tab.c/)
LFO = $(LFC:.c=.o)
YFO = $(YFC:.c=.o)

parser: syntax $(filter-out $(LFO),$(OBJS))
    $(CC) -o parser $(filter-out $(LFO),$(OBJS)) -lfl

syntax: lexical syntax-c
    $(CC) -c $(YFC) -o $(YFO)

lexical: $(LFILE)
    $(FLEX) -o $(LFC) $(LFILE)

syntax-c: $(YFILE)
    $(BISON) -o $(YFC) -d -v $(YFILE)

-include $(patsubst %.o, %.d, $(OBJS))

# 定义的一些伪目标
.PHONY: clean test
test:
    ./parser ../Test/test1.cmm
clean:
    rm -f parser lex.yy.c syntax.tab.c syntax.tab.h syntax.output
    rm -f $(OBJS) $(OBJS:.o=.d)
    rm -f $(LFC) $(YFC) $(YFC:.c=.h)
```

本次实验，我还针对验收进行了细节优化。我先准备了所有的测试用例文件分别叫 CODExx.cmm，xx 从 1 到 21。然后，我们可以以命令行输入测试用例文件，也可以不指定测试用例文件，此时将自动选择测试用例文件，从 CODE1.cmm 开始，每次自动增加 1，序号保存在 current.txt 中。

同时，还将先打印测试用例文件中的所有代码+行号，后续再跟上语法分析的错误，以方便核对。

（三） 实验结果



三、 实验总结

1. 学会了如何实现简单的语义分析任务。
2. 学会使用各种哈希表和栈进行符号表的设计。