# 2019～2020年春季学期

**序号：**

# 《编译原理》课程

## 实验报告

# 实验二 语法分析器构造

# 专业班级：

# 学生姓名：

# 学生学号：

**一、目的和要求**

借助于词法分析程序提供的分析结果，编写一个算符优先语法分析程序，程序能进行语法结构分析和错误检查，并产生相应的归约信息。同时给出出错信息和错误类型，从而加深对语法分析的理解。

**二、实验内容**

给定文法G和算符优先分析法，构造其算符优先分析程序。文法G：

语句→赋值语句｜条件语句｜转移语句｜带标号的赋值语句

带标号的赋值语句→<标号><赋值语句>

赋值语句→变量=算术表达式

条件语句→TF <布尔表达式> THEN 语句

 ｜TF <布尔表达式> THEN语句 ELSE 语句

转移语句→GOTO标号

变量→标识符

标识符→字母｜<标识符><数字>

字母→A｜B｜…｜Z｜a｜b｜…｜z

数字→0｜1｜…｜9

算术表达式→项｜算术表达式+项｜算术表达式-项

项→因子｜项\*因子｜项/因子｜因子↑项

因子→变量｜常数｜(表达式)

布尔表达式→<算术表达式><关系符><算术表达式>

关系符→>｜<｜>=｜<=｜=｜<>

标号→常数

常数→数字｜<常数><数字>

**三、实验的实现**

1. 主要设计思想

**a. 优先关系表**

在程序中用一个二维数组table []来存储算符间的优先关系。table[a][b]=1表示a>b; 。table [a][b]=0表示a=b; 。table [a][b]=-1表示a<b; table[a][b]=-2，表示出错

**b.归约栈和剩余输入串的设计**

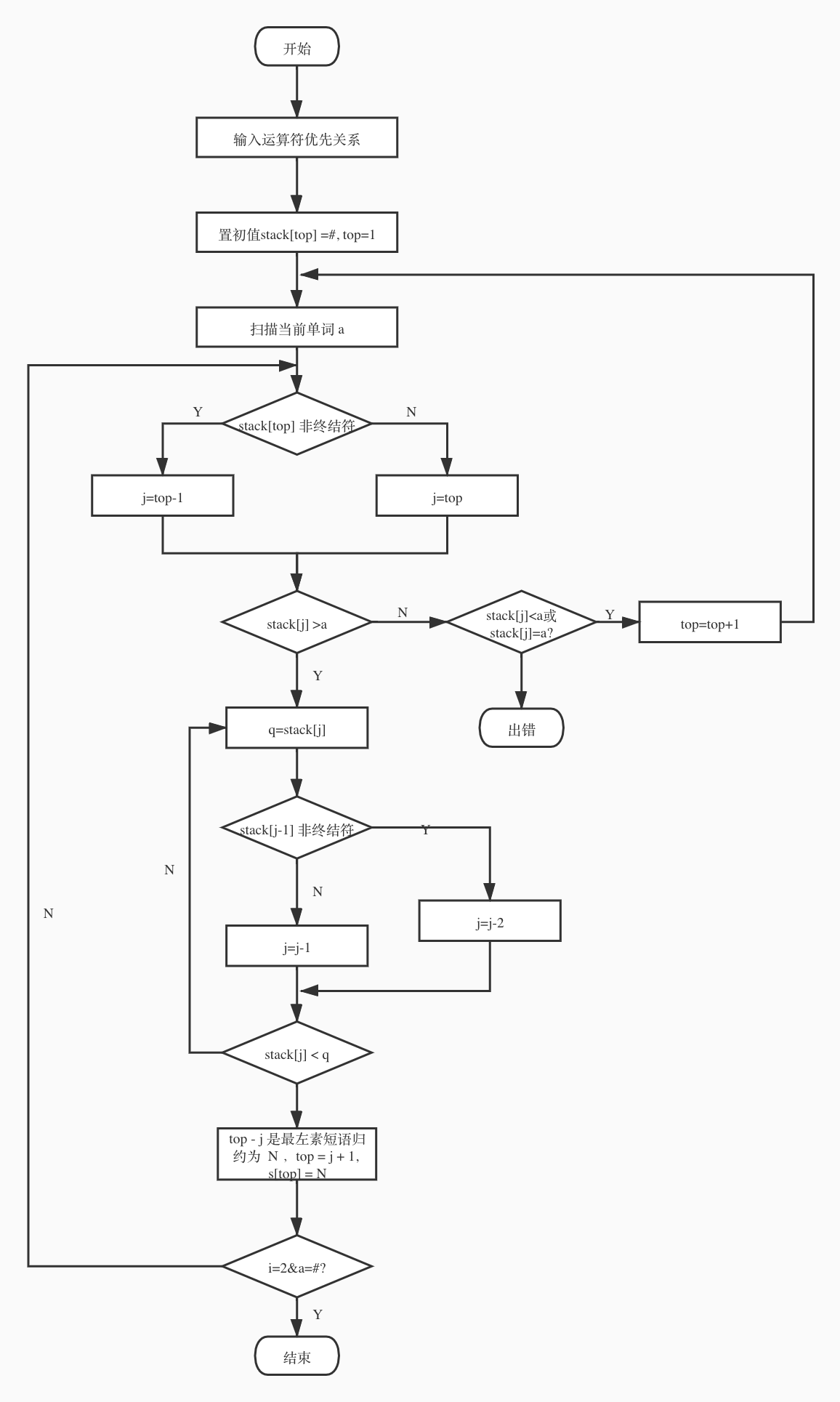
　　使用数组stack[]来代替栈，top代表栈顶的索引，stack[top]就代表栈顶的元素；Surplus[]代表剩余输入串，surplus[0] 代表当前分析符。

**c归约过程**

使用 Stack[top] 和 surplus[0] 进行算符优先级判断，当 table[a][b]=1，进入规约的逻辑，归约的时候，找到最左素短语的种别码匹配到 word [] 可规约字串中的某一项，就替换成 N。

2. 实现算法及程序流程图

比较栈顶和当前分析符的优先级，如果是小于、等于则进行移进操作，将当前分析符推入符号栈中；如果是大于，就找到归约字串，将找到的字串替换为 N，推入栈顶。



3. 主要技术问题的处理方法

在实际过程中使用栈来实现符号栈不是很方便，所以使用了数组进行代替，主要是因为 nodejs对数组的 api 封装的更完善，让我能更专注于业务逻辑代码的编写。

4. 程序代码：实验实现的源程序，要求符合一般的程序书写风格，并包括必要的注释。

const fs = require("fs");

const path = require("path");

// 1表示 >, -1表示 <, 0表示 =, -2表示错误

const table = [

[1, 1, -1, -1, -1, 1, -1, 1],

[1, 1, -1, -1, -1, 1, -1, 1],

[1, 1, 1, 1, -1, 1, -1, 1],

[1, 1, 1, 1, -1, 1, -1, 1],

[-1, -1, -1, -1, -1, 0, -1, -2],

[1, 1, 1, 1, -2, 1, -2, 1],

[1, 1, 1, 1, -2, 1, -2, 1],

[-1, -1, -1, -1, -1, -2, -1, 0],

];

// 可归约字符串

const word = ["N+N", "N-N", "N\*N", "N/N", ")N(", "i"];

// 算数表达式

let expStrings = [];

// 输入串

let inpStrings = [];

// 判断是表达式

let isExp = false,

isChar = false,

i = -1,

j = -1;

init("./预处理.txt");

function init(filename) {

const str = fs.readFileSync(path.join(\_\_dirname, filename), {

encoding: "utf-8",

});

const txt = str.split("");

// 处理算术表达式

txt.forEach((c, index) => {

if (c !== "=" && c !== "#" && !isExp) return;

if ((txt[index - 1] === "<" || txt[index - 1] === ">") && c === "=") return;

if (c === "#") return;

if (isExp) {

expStrings[i] = ((expStrings[i] || "") + c).trim();

if (isalnum(c) && !isChar) {

inpStrings[j] = ((inpStrings[j] || "") + "i").trim();

isChar = true

} else if (!isalnum(c)) {

inpStrings[j] = ((inpStrings[j] || "") + c).trim();

isChar = false;

}

}

if (c === "=") {

isExp = true;

i++;

j++

return;

}

if (c === " ") isExp = false;

});

inpStrings = inpStrings.map(item => "#" + item + "#");

priorAnalysis();

}

// 分析

function priorAnalysis() {

inpStrings.forEach((inp, index) => {

console.log("\n");

console.log("算数表达式：", expStrings[index]);

console.log("转换为输入串：", inp.slice(1));

console.log("步骤号 符号栈 优先关系 当前分析符 剩余输入串 动作");

let step = 1;

const stack = []; // 符号栈

const surplus = Array.from(inp); // 剩余输入串

let cur = surplus.shift();

stack.push(cur);

console.log(`${step++} ${stack.join("")} < null ${surplus.join("")} 移进`);

while (surplus.length) {

let top = stack.length - 1;

cur = surplus[0];

let j = stack[top] === "N" ? top - 1 : top;

while (table[chDir(stack[j]) - 1][chDir(cur) - 1] === 1) {

let q;

let ch = "";

// 找到最左素短语（当stack[j] 的优先级小于 q 时，停止）

do {

q = stack[j];

if (stack[j - 1] !== "N")

j = j - 1;

else

j = j - 2;

} while (table[chDir(stack[j]) - 1][chDir(q) - 1] !== -1);

while ((top - j) != 0) { // 提取最左素短语（可归约的子串）

ch += stack.pop();

top = stack.length - 1;

}

// 将上一步得到的子串替换为 N，推入栈顶

for (let m = 0; m <= 5; m++) {

if (word.includes(ch))

str = 'N';

}

stack.push(str);

console.log(`${step++} ${stack.join("")} > ${cur} ${surplus.join("")} 归约`);

}

if (table[chDir(stack[j]) - 1][chDir(cur) - 1] === -1) {

cur = surplus.shift();

stack.push(cur);

console.log(`${step++} ${stack.join("")} < ${cur} ${surplus.join("")} 移进`);

} else {

if (table[chDir(stack[j]) - 1][chDir(cur) - 1] === 0) {

cur = surplus.shift();

stack.push(cur);

console.log(`${step++} ${stack.join("")} = ${cur} ${surplus.join("")} 移进`);

} else {

error(j);

stack.push("#");

break;

}

}

}

console.log("\n");

});

}

function chDir(ch) {

switch (ch) {

case "+":

return 1;

case "-":

return 2;

case "\*":

return 3;

case "/":

return 4;

case "(":

return 5;

case ")":

return 6;

case "i":

return 7;

case "#":

return 8;

default:

return 0;

}

}

function error(j) {

if (chDir(stack[t]) == 6 || chDir(stack[t]) == 7) console.log("\n错误e2:缺少运算符!");

else if (ch\_di(stack[t]) == 5) console.log("\n错误e1:非法左括号!");

else console.log("\n错误e3:非法右括号!");

}

function isalnum(char) {

return /[a-zA-Z\d]/.test(char);

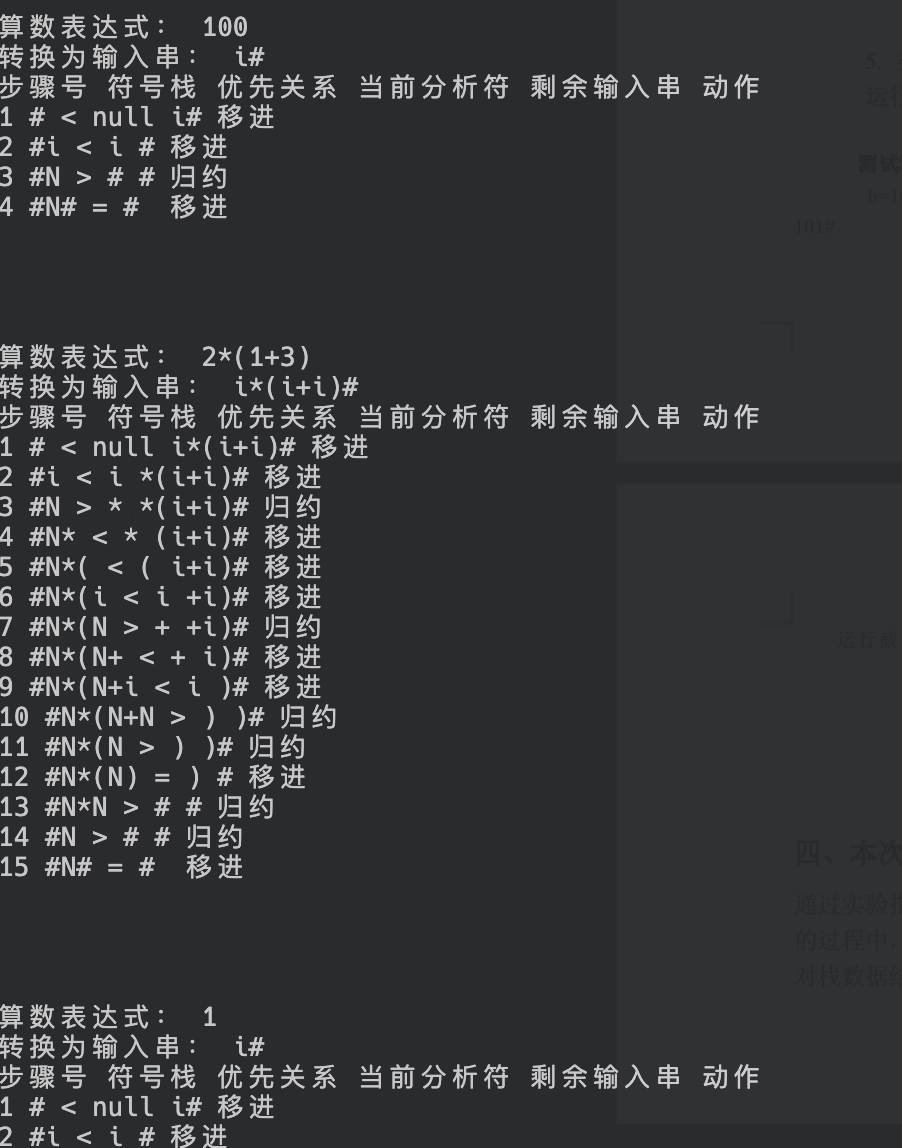
}

5. 实验运行结果及分析（包括测试用例源程序，运行结果截图，以及对运行结果的分析）

**测试程序：**

b=100 101:a=2\*(1+3) IF(b>10) THEN a=1 ELSE IF(b>=5) THEN a=2 ELSE GOTO 101#

**运行截图：**



**实验结果分析：**

通过比较栈顶元素和当前分析符来获取优先关系，当优先关系为 <、= 为移进，当 > 时为归约，当符号栈的栈顶元素为 #，为分析结束。

**四、本次实验总结体会**

通过实验指导如何通过算符优先表，比较两个终结符的优先级关系；在编写代码的过程中，加深了对算符优先分析法的执行过程；在解决程序的过程中，加强了对栈数据结构的操作和发现并解决问题的能力，

**五、对本实验过程及方法、手段的改进建议**

得到归约字串在代码的编写中较较难理解，最好在实验介绍的时候，能给予提示。

**六、说明与提示**

1. 本实验的优先表可以手工先设计好。

2. 本实验要求中提出的“产生相应的归约信息”意指在语法分析的过程中，一旦产生归约，在程序上产生并最终输出归约产生式序号。

3. 出错类型的产生可预先对应优先表中出错栏列表说明其出错类型，并分别编序，当分析中产生错误时以字符串输出相应表中错误信息。

4. 算法采用一个符号栈的数据结构，既用它存放终结符，也用它存放非终结符。设K为符号栈使用深度，其参考算法如下：

K:=1; S[K]:= '#';

Repeat

 把下一个输入符号读进a中；

If S[K]∈VT then j:=K else j:=K-1;

 while S［j］a do

 Begin

 Repeat

Q:=S[j];

If S[j-1]∈VT then j:=j-1 else j:=j-2

Until S[j]Q;

 把S[j+1]…S[K]归约为某个N；记录归约产生式序号；

 K:=j+1;

 S[K]:=N

end of while

If S[j]a OR S[j]a then

 Begin K:=K+1; S[K]:=a end

else ERROR {查表打印出错信息}

Until a='#'

输出归约产生式序列号。