编译原理实验报告-语法分析程序

刘帅 2020212267

1 - 实验内容

编写语法分析程序，实现对算术表达式的语法分析。要求所分析算术表达式由如下的文法产生。

$$E \to E+T$$
$$E \to E-T$$
$$E \to T$$
$$T \to T*F$$
$$T \to T/F$$
$$T \to F$$
$$F \to (E)$$
$$F \to num$$

2 - 实验方法

2.1 方法一 递归调用

2.1.1 消除左递归

$$S \to E$$
$$E \to TA$$
$$A \to +TA|-TA|\varepsilon$$
$$T \to FB$$
$$B \to *FB|/FB|\varepsilon$$
$$F \to (E)|num$$

2.1.2 - 代码描述

2.1.2.1 - 数字转换

将数字转换为n，方便后续进行识别

```
void transfer(string a,string &b){//将输入字符串中的所有数字转换为n 方便语法分析
    int length=a.length();
    for(int i=0;i<length;i++){
        if('0'<=a[i]&&a[i]<='9'){
            while('0'<=a[i]&&a[i]<='9')
                i++;
            if(a[i]=='.'){
                i++;
                while('0'<=a[i]&&a[i]<='9')
                    i++;
            }
            b+="n";
            i--;
            continue;
        }
```

```
        else
            b.push_back(a[i]);
    }
}
```

2.1.2.2- 递归函数E()

此时调用E，则输出$E->TA$,并递归调用$T()$与$A()$。

```
void E()
{
    cout<<"E->TA"<<endl;
    T();
    A();

}
```

2.1.2.3- 递归函数T()

此时调用T，则输出$E->FB$,并递归调用$F()$与$B()$。

```
void T()
{
    cout << "T->FB" << endl;
    F();
    B();
}
```

2.1.2.4- 递归函数A()

此时调用A，判断目前str中的符号为$+$或$-$，分别输出$A->\pm TA$,并递归调用$T()$与$A()$。

```
void A()
{
    if (str[i] == '+') {
        i++;
        cout << "A->+TA" << endl;
        T();
        A();
    }
    else if (str[i] == '-')
    {
        cout<<"A->-TA"<<endl;
        i++;
        T();
        A();
    }

}
```

2.1.2.5- 递归函数F()

此时调用F，判断括号和数字，为递归的结尾。若未识别到数字或右括号，说明语法出错，isarith设置为0，在主程序中报错。

```
void F()
{
    if (str[i] == '(')
    {
        i++;
        E();
        if (str[i] == ')')
        {
            i++;
            cout<<"F->(E)"<<endl;
        }
        else
            isarith = false;
    }
    else if (str[i] =='n')
    {
        cout<<"F->num"<<endl;
        i++;
    }
    else
        isarith = false;
}
```

2.1.2.6 - 递归函数B()

```
void B() {

    if (str[i] == '*')
    {
        cout << "B->*FB" << endl;
        i++;
        F();
        B();
    }
    else if (str[i] == '/')
    {
        cout << "B->/FB" << endl;
        i++;
        F();
        B();
    }

}
```

2.1.2.7- 主函数

```
int main()
{
    bool flag = false;
    while (flag == false) {
        cout << "请输入待分析语句，用$结束" << endl;
        cin >> str;
        if (str[str.length() - 1] == '$' && str.length() != 1) {
            break;
        }
        else {
            cout << "输入错误，请重新输入！" << endl;
        }
    }
    string empty = "";
    transfer(str, empty);
    str = empty; //将str转换为n+n的形式
    i = 0;
    E();
    if (str[i] == '$' && isarith == true)
    {
        cout<<"语句合法"<<endl;
    }
    else
    {
        cout<<"不合法"<<endl;
    }
    return 0;
}
```

2.1.3 输出结果

```
请输入待分析语句，用$结束
12+24*(32-14/2)+43/2+(1-3)$
E->TA
T->FB
F->num
A->+TA
T->FB
F->num
B->*FB
E->TA
T->FB
F->num
A->-TA
T->FB
F->num
B->/FB
F->num
F->(E)
A->+TA
T->FB
F->num
B->/FB
F->num
A->+TA
T->FB
E->TA
T->FB
F->num
A->-TA
T->FB
F->num
F->(E)
语句合法
```

```
请输入待分析语句，用$结束
1()2+*3/4(($
E->TA
T->FB
F->num
不合法
```

2.2 方法二 LL(1)文法

2.2.1 FIRST集/FOLLOW集

	E	A	T	B	F
FIRST	$(, num$	$+, -, \varepsilon$	$(, num$	$*, /, \varepsilon$	$(, num$
FOLLOW	$\$,)$	$\$,)$	$+, -, \$,)$	$+, -, \$,)$	$*, /, +, -, \$,)$

2.2.2 LL(1)预测分析表

	$+$	$-$	$*$	$/$	$($	$)$	num	$\$$
E	$error$	$error$	$error$	$error$	$E \to TA$	$synch$	$E \to TA$	$synch$
A	$A \to +TA$	$A \to -TA$	$error$	$error$	$error$	$A \to \varepsilon$	$error$	$A \to \varepsilon$
T	$synch$	$synch$	$error$	$error$	$T \to FB$	$synch$	$T \to FB$	$synch$
B	$B \to \varepsilon$	$B \to \varepsilon$	$B \to *FB$	$B \to /FB$	$error$	$B \to \varepsilon$	$error$	$B \to \varepsilon$
F	$synch$	$synch$	$synch$	$synch$	$F \to (E)$	$synch$	$F \to num$	$synch$

2.2.3 代码实现

2.2.3.1 变量声明

```
char grammar[10][8];//产生式集
char terminal[9];//终结符
char noterminal[6];//非终结符
char first[5][8];
char follow[5][8];
int table[5][8];//预测分析表
string str; //输入字符串
stack<char> s,tempstack;
```

2.2.3.2 打印产生式

```
void printgrammar(int flag) {
    if (flag == 0) cout << "E->TA";
    else if (flag == 1) cout << "A->+TA";
    else if (flag == 2) cout << "A->-TA" ;
    else if (flag == 3) cout << "A->ε" ;
    else if (flag == 4) cout << "T->FB" ;
    else if (flag == 5) cout << "B->*FB" ;
    else if (flag == 6) cout << "B->/FB";
    else if (flag == 7) cout << "B->ε";
    else if (flag == 8) cout << "F->(E)";
    else if (flag == 9) cout << "F->num" ;
    else if (flag == -1) cout << "error";
    else if (flag == -2) cout << "synch";
}
```

2.2.3.3 初始化

将产生式填入grammar之中，前后使用"#"进行分隔，从而便于后续反向入栈的识别。

```
void init() {
    s.push('$');
    s.push('E');
    strcpy_s(grammar[0], "E#TA#");
    strcpy_s(grammar[1], "A#+TA#");
    strcpy_s(grammar[2], "A#-TA#");
    strcpy_s(grammar[3], "A#e#");
    strcpy_s(grammar[4], "T#FB#");
    strcpy_s(grammar[5], "B#*FB#");
    strcpy_s(grammar[6], "B#/FB#");
```

```
    strcpy_s(grammar[7], "B#e#");
    strcpy_s(grammar[8], "F#(E)#");
    strcpy_s(grammar[9], "F#n#");

    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 8; j++) {
            table[i][j] = -1;
        }
    }
    strcpy_s(terminal, "+-*/()n$");  //初始化终结符表
    strcpy_s(noterminal, "EATBF");  //初始化非终结符表
    strcpy_s(first[0], "(n#");
    strcpy_s(first[1], "+-e#");
    strcpy_s(first[2], "(n#");
    strcpy_s(first[3], "*/e#");
    strcpy_s(first[4], "(n#");

    strcpy_s(follow[0], ")$#");
    strcpy_s(follow[1], ")$#");
    strcpy_s(follow[2], "+-)$#");
    strcpy_s(follow[3], "+-)$#");
    strcpy_s(follow[4], "+-*/)$#");
}
```

2.2.3.4 寻找非终结符的下标，便于后面加入表格

```
int getnoterminal_index(char ch) {//找出非终结符下标的index
    for (int i = 0; i < 5; i++) {
        if (ch == noterminal[i])
            return i;
    }
    return -1;
}
```

2.2.3.5寻找终结符的下标，便于后续加入表格

```
int getterminal_index(char ch) {
    for (int i = 0; i < 8; i++) {
        if (ch == terminal[i])
            return i;
    }
    return -1;
}
```

2.2.3.6 判断目前的字符ch是否在left(左产生式)的first集中，便于后续构造分析表。

```
bool infirst(char left, char ch) {//查看ch是否在非终结符的first集中
    int index = getnoterminal_index(left);
    for (int i = 0; first[index][i] != '#'; i++) {
        if (first[index][i]==ch)
            return true;
    }
    return false;

}
```

2.2.3.7 构造分析表

如果右边的符号为非终结符，则将非终结符的first集对应的表格中填入产生式；若右边符号为终结符，则直接将产生式放入表中;若右边符号为空产生式，则将其左部符号的follow集加入到表格中。最后将error且在follow集中的终结符对应的表格用synch进行填充，表示遇到时进行弹栈操作。

```
void maketable() {
    for (int i = 0; i < 10; i++) {//对每条产生式遍历
        char left = grammar[i][0]; //产生式左部
        char ch = grammar[i][2];//右边第一个符号 终极符or非终结符or空
        if (ch == 'E' || ch == 'T' || ch == 'A' || ch == 'B' || ch == 'F') {
            for(int j=0;j<8;j++){
                if (infirst(ch, terminal[j])) {
                    int index1 = getnoterminal_index(left);
                    table[index1][j] = i;
                    // cout << left << terminal[j] << i<<endl;
                }
            }
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '('
|| ch == ')' || ch == 'n' || ch == '$') {
            int index1 = getnoterminal_index(left);
            int index2 = getterminal_index(ch);
            table[index1][index2] = i; //如果是终结符就直接把产生式放进表里
            //cout << left << terminal[index2] << i << endl;
        }
        else if (ch == 'e') {
            int index1 = getnoterminal_index(left);
            for (int j = 0; follow[index1][j] != '#'; j++) {//从他的follow集里面找
对应
                int index2 = getterminal_index(follow[index1][j]);
                table[index1][index2] = i;
                //cout << left << terminal[index2] << i << endl;
            }
        }
    }
    for (int i = 0; i < 5; i++) {
        for (int j = 0; follow[i][j] != '#'; j++) {
            int index1 = getterminal_index(follow[i][j]);
            if (table[i][index1] == -1) {
                table[i][index1] = -2;//如果此时表里是空白，则将follow集里的元素对应的表
替换为synch
            }
        }
    }
```

```
}
```

2.2.3.8 分析表可视化

```
void showtable() {
    for (int i = 0; i < 72; i++) {
        cout << '-';
    }cout << endl;
    for (int i = 0; i < 8; i++) {
        cout <<setw(8)<<terminal[i]<<'|';
    }
    cout << endl;
    for (int i = 0; i < 5; i++) {
        for (int i = 0; i < 72; i++) {
            cout << '-';
        }cout << endl;
        cout << noterminal[i];
        for (int j = 0; j < 8; j++) {
            cout << setw(8);
            printgrammar(table[i][j]);
            cout << '|';
        }
        cout << endl;
    }
    for (int i = 0; i < 72; i++) {
        cout << '-';
    }

}
```

2.2.3.9 数字转化

同递归调用

```
void transfer(string a,string &b){//将输入字符串中的所有数字转换为n 方便语法分析
    int length=a.length();
    for(int i=0;i<length;i++){
        if('0'<=a[i]&&a[i]<='9'){
            while('0'<=a[i]&&a[i]<='9')
                i++;
            if(a[i]=='.'){
                i++;
                while('0'<=a[i]&&a[i]<='9')
                    i++;
            }
            b+="n";
            i--;
            continue;
        }
        else
            b.push_back(a[i]);
    }

}
```

2.2.3.10 LL1分析

如果栈顶为终极符，则和str目前所指的字符进行匹配，匹配成功，栈顶元素弹出，str指针后移。

若栈顶为非终结符，则将其出栈，将其对应产生式倒叙入栈。

错误处理：若此时显示error(即-1)，则进入恐慌模式，跳过当前字符

若此时显示为synch(即-2)，则对弹出栈内符号。

```
void ll1analysis() {
    int ip = 0;//输入指针
    string b = "";
    transfer(str, b);
    str= b;
    //cout << str<<endl;
    int step = 0;
    while (!s.empty() || ip < str.length()) {
        step++;
        cout << "top=" << s.top()<<endl;
        if(s.top()!='$') cout << "step" << step << "  ";
        if (isterminal(s.top())) {//如果栈顶是终极符，则和str进行匹配
            if (s.top() == str[ip])
            {
                s.pop();
                ip++;
            }
            else
            {
                s.pop();
                ip++;
            }
        }
        else//当栈顶是非终结符
        {
            int index1 = getnoterminal_index(s.top());
            int index2 = getterminal_index(str[ip]);
            if (table[index1][index2] != -1 && table[index1][index2] != -2)
            {
                s.pop();
                //cout << "index1=" << index1 << "index2=" << index2 << endl;
                int index = table[index1][index2];
                if (grammar[index][2] != 'e')
                {
                    int j = 0;
                    for (j = 2; grammar[index][j] != '#'; j++);
                    for (j--; j >= 2; j--)//将右部倒序写入stack
                    {
                        s.push(grammar[index][j]);
                        tempstack = s;
                    }
                    printgrammar(index);
                    cout << "  ";
                }
                else {
                    printgrammar(index); cout << "  ";
```

```
                }

            }
            else if (table[index1][index2] == -1)//error
            {
                ip++;
                cout << "输入字符错误，跳过" <<" ";
            }
            else if (table[index1][index2] == -2) {
                    s.pop();
                    cout << "弹栈" << " ";
                    if (s.empty()) break;

            }
        }
        tempstack = s;
        showstack(tempstack);
        cout << "  ";
        showstr(ip);
        cout << endl;
    }
}
```

2.2.3.11 可视化

```
void showstack(stack<char>&tempstack) {
    while (!tempstack.empty()) {
        cout<<tempstack.top();
        tempstack.pop();
    }
}
void showstr(int ip) {
    for (ip; ip < str.length(); ip++) {
        cout << str[ip];
    }
}
```

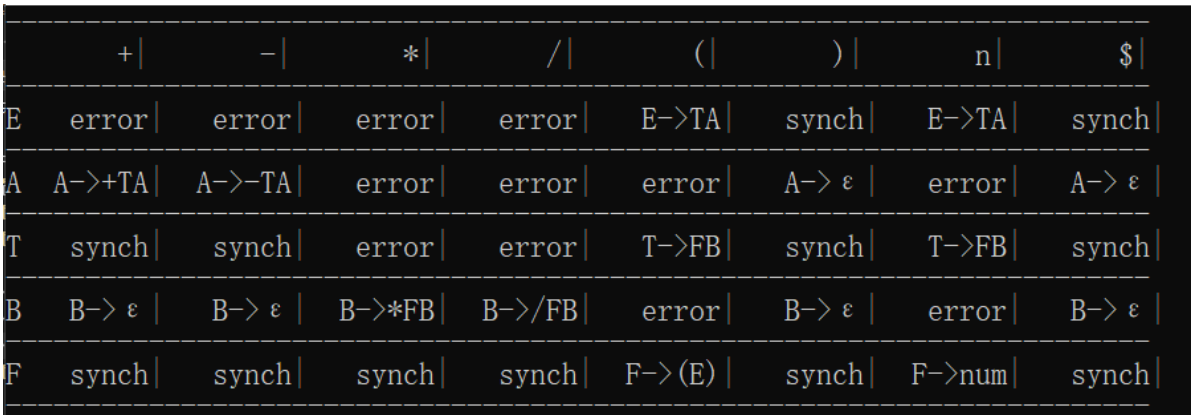
2.2.3.12 主函数

```
int main() {
    init();
    maketable();
    showtable();
    cout << endl;
    bool flag=false;
    while (flag == false) {
        cout << "请输入待分析语句，用$结束"<<endl;
        cin >> str;
        if (str[str.length() - 1] == '$' &&str.length()!=1) {
            break;
        }
        else {
            cout<<"输入错误，请重新输入！"<<endl;
        }
    }
```

```
    ll1analysis();
    tempstack = s;
}
```

2.2.4 输出结果

2.2.4.1 分析表可视化



	+	-	*	/	(	)	n	$
E	error	error	error	error	E->TA	synch	E->TA	synch
A	A->+TA	A->-TA	error	error	error	A-> ε	error	A-> ε
T	synch	synch	error	error	T->FB	synch	T->FB	synch
B	B-> ε	B-> ε	B->*FB	B->/FB	error	B-> ε	error	B-> ε
F	synch	synch	synch	synch	F->(E)	synch	F->num	synch

2.2.4.2 分析过程可视化

```
请输入待分析语句，用$结束
1+2$
top=E
step1  E->TA  TA$  n+n$
top=T
step2  T->FB  FBA$  n+n$
top=F
step3  F->num  nBA$  n+n$
top=n
step4  BA$  +n$
top=B
step5  B->ε  A$  +n$
top=A
step6  A->+TA  +TA$  +n$
top=+
step7  TA$  n$
top=T
step8  T->FB  FBA$  n$
top=F
step9  F->num  nBA$  n$
top=n
step10  BA$  $
top=B
step11  B->ε  A$  $
top=A
step12  A->ε  $  $
top=$
```

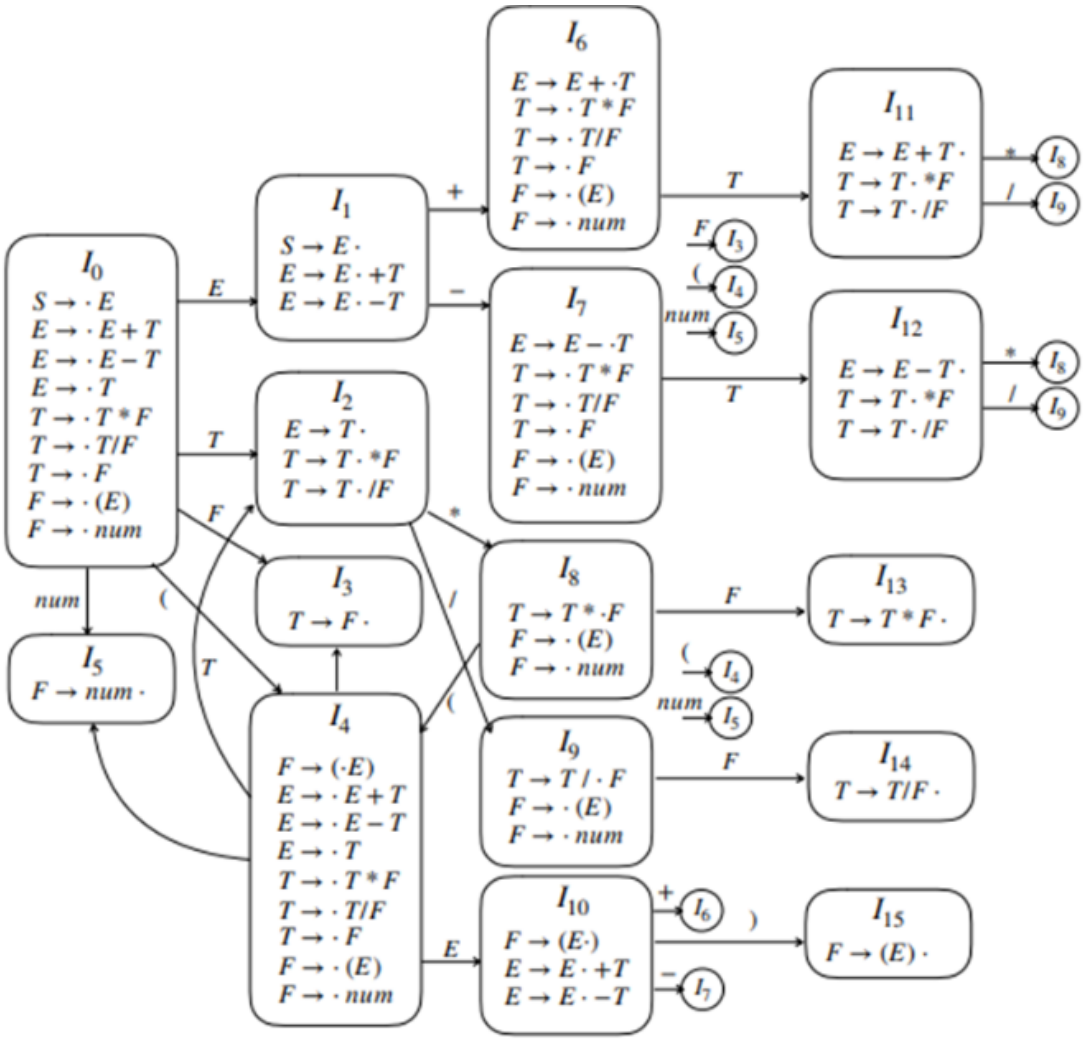
```
请输入待分析语句，用$结束
1+2*+3$
top=E
step1  E->TA  TA$  n+n*+n$
top=T
step2  T->FB  FBA$  n+n*+n$
top=F
step3  F->num  nBA$  n+n*+n$
top=n
step4  BA$  +n*+n$
top=B
step5  B-> ε  A$  +n*+n$
top=A
step6  A->+TA  +TA$  +n*+n$
top=+
step7  TA$  n*+n$
top=T
step8  T->FB  FBA$  n*+n$
top=F
step9  F->num  nBA$  n*+n$
top=n
step10  BA$  *+n$
top=B
step11  B->*FB  *FBA$  *+n$
top=*
step12  FBA$  +n$
top=F
step13  弹栈 BA$  +n$
top=B
step14  B-> ε  A$  +n$
top=A
step15  A->+TA  +TA$  +n$
top=+
step16  TA$  n$
top=T
step17  T->FB  FBA$  n$
top=F
step18  F->num  nBA$  n$
top=n
step19  BA$  $
top=B
```

2.3 方法三 SLR(1)文法

2.3.1 拓广文法

$$S \to E$$
$$E \to E+T$$
$$E \to E-T$$
$$E \to T$$
$$T \to T*F$$
$$T \to T/F$$
$$T \to F$$
$$F \to (E)$$
$$F \to num$$

2.3.2 识别该文法所有活前缀的DFA



2.3.3 SLR（1）分析表

状态	+	-	*	/	(	)	num	$	E	T	F
0					S4		S5		1	2	3
1	S6	S7						ACC			
2	R3	R3	S8	S9		R3		R3			
3	R6	R6	R6	R6		R6		R6			
4					S4		S5		10	2	3
5	R8	R8	R8	R8		R8		R8			
6					S4		S5			11	3
7					S4		S5			12	3
8					S4		S5				13
9					S4		S5				14
10	S6	S7				S15					
11	R1	R1	S8	S9		R1		R1			

状态	+	-	*	/	(	)	num	$	E	T	F
12	R2	R2	S8	S9		R2		R2			
13	R4	R4	R4	R4		R4		R4			
14	R5	R5	R5	R5		R5		R5			
15	R7	R7	R7	R7		R7		R7			

2.3.4 代码实现

2.3.4.1 结构体定义及变量声明

```
struct gram {
    int num=0;//右部字符长度
    string out="";//产生式输出
    char in = 'n';//规约后左部标识
};

struct node {
    char type='n';
    int num=0;
};
node action[16][8];          //action表
int goTo[16][3];             //goto表
gram grammar[9];
map<char, int> terminal = {
    {'+',0},
    {'-',1},
    {'*',2},
    {'/',3},
    {'(',4},
    {')',5},
    {'n',6},
    {'$',7},
};
map<char, int> nonterminal = {
    {'E',0},
    {'T',1},
    {'F',2},
};
```

2.3.4.2 SLR(1)分析表及产生式初始化

```
void init() {

    grammar[0] = { 1,"S->E",'S'};
    grammar[1] = { 3,"E->E+T",'E'};
    grammar[2] = { 3,"E->E-T",'E'};
    grammar[3] = { 1,"E->T",'E'};
    grammar[4] = { 3, "T->T*F",'T'};
    grammar[5] = { 3,"T->T/F",'T'};
    grammar[6] = { 1,"T->F",'T'};
    grammar[7] = { 3,"F->(E)",'F'};
```

```
grammar[8] = { 1,"F->num",'F'};
action[0][terminal['(']] = { 'S',4 };
action[0][terminal['n']] = { 'S',5 };
goTo[0][nonterminal['E']] = 1;
goTo[0][nonterminal['T']] = 2;
goTo[0][nonterminal['F']] = 3;
action[1][terminal['+']] = { 'S',6 };
action[1][terminal['-']] = { 'S',7 };
action[0][7].type = 'F';
action[1][7].type = 'F';
action[2][terminal['+']] = { 'R',3 };
action[2][terminal['-']] = { 'R',3 };
action[2][terminal[')']] = { 'R',3 };
action[2][terminal['$']] = { 'R',3 };
action[2][terminal['*']] = { 'S',8 };
action[2][terminal['/']] = { 'S',9 };
action[3][terminal['+']] = { 'R',6 };
action[3][terminal['-']] = { 'R',6 };
action[3][terminal['*']] = { 'R',6 };
action[3][terminal['/']] = { 'R',6 };
action[3][terminal[')']] = { 'R',6 };
action[3][terminal['$']] = { 'R',6 };
action[4][terminal['(']] = { 'S',4 };
action[4][terminal['n']] = { 'S',5 };
goTo[4][nonterminal['E']] = 10;
goTo[4][nonterminal['T']] = 2;
goTo[4][nonterminal['F']] = 3;
action[5][terminal['+']] = { 'R',8 };
action[5][terminal['-']] = { 'R',8 };
action[5][terminal['*']] = { 'R',8 };
action[5][terminal['/']] = { 'R',8 };
action[5][terminal[')']] = { 'R',8 };
action[5][terminal['$']] = { 'R',8 };
action[6][terminal['(']] = { 'S',4 };
action[6][terminal['n']] = { 'S',5 };
goTo[6][nonterminal['T']] = 11;
goTo[6][nonterminal['F']] = 3;
action[7][terminal['(']] = { 'S',4 };
action[7][terminal['n']] = { 'S',5 };
goTo[7][nonterminal['T']] = 12;
goTo[7][nonterminal['F']] = 3;
action[8][terminal['(']] = { 'S',4 };
action[8][terminal['n']] = { 'S',5 };
goTo[8][nonterminal['F']] = 13;
action[9][terminal['(']] = { 'S',4 };
action[9][terminal['n']] = { 'S',5 };
goTo[9][nonterminal['F']] = 14;
action[10][terminal['+']] = { 'S',6 };
action[10][terminal['-']] = { 'S',7 };
action[10][terminal[')']] = { 'S',15 };
action[11][terminal['+']] = { 'R',1 };
action[11][terminal['-']] = { 'R',1 };
action[11][terminal['*']] = { 'S',8 };
action[11][terminal['/']] = { 'S',9 };
action[11][terminal[')']] = { 'R',1 };
```

```
    action[11][terminal['$']] = { 'R',1 };
    action[12][terminal['+']] = { 'R',2 };
    action[12][terminal['-']] = { 'R',2 };
    action[12][terminal['*']] = { 'S',8 };
    action[12][terminal['/']] = { 'S',9 };
    action[12][terminal[')']] = { 'R',2 };
    action[12][terminal['$']] = { 'R',2 };
    action[13][terminal['+']] = { 'R',4 };
    action[13][terminal['-']] = { 'R',4 };
    action[13][terminal['*']] = { 'R',4 };
    action[13][terminal['/']] = { 'R',4 };
    action[13][terminal[')']] = { 'R',4 };
    action[13][terminal['$']] = { 'R',4 };
    action[14][terminal['+']] = { 'R',5 };
    action[14][terminal['-']] = { 'R',5 };
    action[14][terminal['*']] = { 'R',5 };
    action[14][terminal['/']] = { 'R',5 };
    action[14][terminal[')']] = { 'R',5 };
    action[14][terminal['$']] = { 'R',5 };
    action[15][terminal['+']] = { 'R',7 };
    action[15][terminal['-']] = { 'R',7 };
    action[15][terminal['*']] = { 'R',7 };
    action[15][terminal['/']] = { 'R',7 };
    action[15][terminal[')']] = { 'R',7 };
    action[15][terminal['$']] = { 'R',7 };
}
```

2.3.4.3 SLR(1)分析

如果此时识别到S，则继续进行推导，通过分析表进入下一个对应状态；若此时识别为R，则说明进行规约操作，同时，根据goto表进行状态跳转；如果识别到F，则说明状态跳转到ACC，表示识别成功；如果识别到其他表示，说明识别出错，即语法出现错误，程序结束。

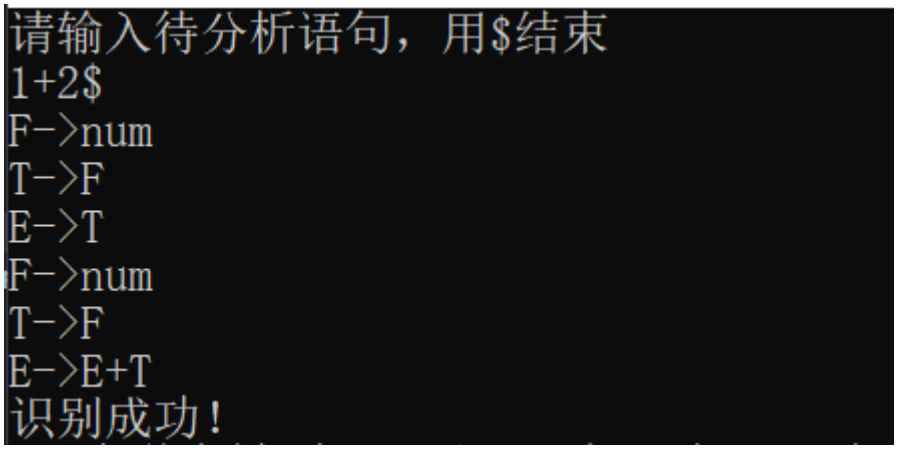
```
void analysis(string s) {
    int ptr = 0;//字符串指针
    stack<int> state;//状态栈
    state.push(0);
    stack<char>symbol;//目前字符串的栈
    while (true) {
        if (action[state.top()][terminal[s[ptr]]].type == 'F') {
            cout << "识别成功！";
            break;//识别到ACC代表分析结束
        }
        else if (action[state.top()][terminal[s[ptr]]].type =='S') {

            state.push(action[state.top()][terminal[s[ptr]]].num); //状态栈
            symbol.push(s[ptr]);
            ptr++;
        }
        else if (action[state.top()][terminal[s[ptr]]].type == 'R') {
            gram temp = grammar[(action[state.top()][terminal[s[ptr]]].num)];
            cout<<temp.out<<endl;
            int count = temp.num;
            for (int i = 0; i < count; i++) {
                symbol.pop();
```

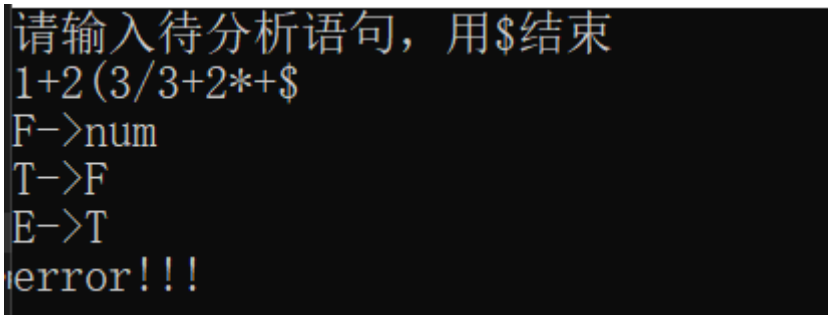
```
            }
            state.pop();
            symbol.push(temp.in);
            state.push(goTo[state.top()][nonterminal[symbol.top()]]);

        }
        else {
            cout << "error!!!" << endl;
            break;
        }
    }

}
```

2.3.4.5 主函数

```
int main(void) {
    init();
    string str;
    bool flag = false;
    while (flag == false) {
        cout << "请输入待分析语句，用$结束" << endl;
        cin >> str;
        if (str[str.length() - 1] == '$' && str.length() != 1) {
            break;
        }
        else {
            cout << "输入错误，请重新输入！" << endl;
        }
    }
    string empty = "";
    transfer(str, empty);
    str = empty;
    analysis(str);
    return 0;
}
```

2.3.5 实验结果





2.4 方法四 flex和bison实现

2.4.1 flex语法分析

2.4.1.1 外部定义和声明

这一部分被Lex翻译器处理后会全部拷贝到文件lex.yy.c中

```
%{
    #include "yacc.tab.h"
    void yyerror(char *);
    #define YYSTYPE double
    extern YYSTYPE yylval;
%}
```

2.4.1.2 设定词法规则

```
digit [0-9]
%option noyywrap /*不会调用yywrap 因为没有文件读写 */
```

2.4.1.3 正规定义和状态定义

```
%%
{digit}+(\.{digit}+)?    {yylval=atof(yytext);return NUMBER;}
[ \t\n]+ {}
.          {return yytext[0];} //通配符
%%
```

2.4.2 yacc语法分析

2.4.2.1 外部定义和声明

该部分直接添加到yacc编译的.c和.h头文件中，后续被lex.yy.c调用

```
%{
  #include <stdio.h>
  extern int yylex(void);
   void yyerror(char *);
   #define YYSTYPE double
%}
```

2.4.2.2 定义关系符和lex传过来的token

```
%token NUMBER /*由yylex通过lex1.l传过来的词法规则*/
%left '+' '-'
%left '*' '/'
```

2.4.2.3 正规定义和状态定义

```
%%
L : E '$'  {printf("Result=%f\n",$1);system("pause");return 0;}
;
E : T        {printf("Reduce by E-->T\n");$$=$1;}
    | E'+' T    {printf("Reduce by E-->E+T\n");$$=$1+$3;} /* 规则右部的文法相加赋值给
左部非终结符属性 */
    | E'-' T    {printf("Reduce by E-->E-T\n");$$=$1-$3;}
;

T : F         {printf("Reduce by T-->F\n");$$=$1;}
    | T'*' F    {printf("Reduce by T-->T*F\n");$$=$1*$3;}
    | T'/' F    {printf("Reduce by T-->T/F\n");$$=$1/$3;}
;

F : '('  E  ')' { printf("Reduce by F-->(E)\n");$$ = $2; }
    | NUMBER {printf("Reduce by F-->num\n"); $$ = $1; }
;
%%
```

2.4.2.4 主函数和异常处理

```
int main()
{
    printf("Please input an expression\n");
    yyparse();
    /*yyparse 函数调用一个扫描函数（即词法分析程序）yylex。
    yyparse 每次调用 yylex() 就得到一个二元式的记号<token,attribute> 。
    由 yylex() 返回的记号(如下 NUMBER 等)，必须事先在 YACC 源程序的说明部分用%token说明，
    该记号的属性值必须通过 YACC 定义的变量 yylval 传给分析程序。*/

}

void yyerror(char* s) {
    printf("\nExpression is invalid\n");
    system("pause");
}
```

2.4.2编译运行

```
E:\大学资料\大三上课程\编译原理\win_flex_bison-latest>win_bison yacc.tab.c
yacc.tab.c:44.1: error: invalid character: '#'
yacc.tab.c:44.2-7: error: syntax error, unexpected identifier

E:\大学资料\大三上课程\编译原理\win_flex_bison-latest>win_bison yacc.y

E:\大学资料\大三上课程\编译原理\win_flex_bison-latest>win_bison -d yacc.y

E:\大学资料\大三上课程\编译原理\win_flex_bison-latest>win_flex lex1.l

E:\大学资料\大三上课程\编译原理\win_flex_bison-latest>gcc -o yufa yacc.tab.c lex.yy.c
yacc.y: In function 'yyparse':
yacc.y:13:47: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
 L : E '$'  {printf("Result=%f\n",$1);system("pause");return 0;}
```

2.4.3 运行结果

```
Please input an expression
1+2/4*3-24/23+47*50$
Reduce by F-->num
Reduce by T-->F
Reduce by E-->T
Reduce by F-->num
Reduce by T-->F
Reduce by F-->num
Reduce by T-->T/F
Reduce by F-->num
Reduce by T-->T*F
Reduce by E-->E+T
Reduce by F-->num
Reduce by T-->F
Reduce by F-->num
Reduce by T-->T/F
Reduce by E-->E-T
Reduce by F-->num
Reduce by T-->F
Reduce by F-->num
Reduce by T-->T*F
Reduce by E-->E+T
Result=2351.456522
```

```
Please input an expression
1+2/4(32*24/3+2-1/2)$
Reduce by F-->num
Reduce by T-->F
Reduce by E-->T
Reduce by F-->num
Reduce by T-->F
Reduce by F-->num
Reduce by T-->T/F
Reduce by E-->E+T

Expression is invalid
```

3 - 实验总结

本次语法分析的实验是建立在充分理解词法分析的基础上的，本次实验，我采用了书中的四个方法进行实现，体会到了前人的智慧。在LL1文法分析的过程中，我体会到了”空间换时间”的思想，通过建立分析表，而后进行语法分析，极大程度降低了分析程序的时间复杂度。在SLR分析中，我个人认为仍有部分不足，因为提前构造了识别活前缀的DFA，并据此进行SLR分析表的建立，该过程可移植性较差，如遇到新的文法则需要手动进行DFA计算。在我学有余力和空余的情况下我会将其完善，使得DFA求解过程更加自动化。在实现flex和bison的过程中，让我对于flex语言有了更为深刻的认识，同时初步了解了flex和bison间的调用、依赖关系，体会到了以前研究者们的巧思。本次实验提升了我接触一门新计算机语言并进行应用的能力，培养了我面对新事物和新工程的设计能力和解决能力，使我收获颇丰。