# 计算器作业

## 1.计算器描述

计算器接受四则运算表达式为输入（如下图所示）。如果表达式语法正确，则输出计算结果，否则报错，指出错误位置及原因。

|  |
| --- |
| Input 1:  float a; int b;  a = (10.44\*356+1.28) / 2 + 1024 \* 1.6;  b = a \* 2 + a/2;  write(b);  write(a).  Output 1:  8743.4  3497 |

|  |
| --- |
| Input 2:  float a;  int b;  a = (10.44\*356+1.28) / 2 + 1024 \* 1.6;  b = a \* 2 – c/2;  write(b).  Output 2:  Error(line 4): undefined identifier . |

1. 支持浮点数和整数两种类型，浮点数可以转换成整数；整数不可以转换成浮点数；
2. 每个语句需要以“；”结束，以“.”表示全部输入的结束
3. 变量需要先声明再使用；
4. 变量名可以是由数字和字母组成，但首字符必须是字母。
5. 每个表达式中使用的变量需要在之前已经有赋值。
6. 输出语句使用write(a)，输出并换行，其中a为int类型或者float类型的变量名。

## 2.文法定义

### 2.1 词法

#### 2.1.1 单词表(粗体作保留字)

保留字先判断为ID，再进入保留字表查询

|  |
| --- |
| id |
| **float** |
| **int** |
| **num** |
| **write** |
| = |
| + |
| - |
| \* |
| / |
| ( |
| ) |
| ; |
| . |

#### 2.1.2 RE正则表达式

Letter -> [A-Za-z]

Digit -> [0-9]

Digits -> digit+

Id -> letter(letter|digit)\*

Float -> float

Int -> int

Num -> digits(.digits)?

Write -> write

ASSIGN -> =

PLUS -> +

MINUS -> -

TIMES -> \*

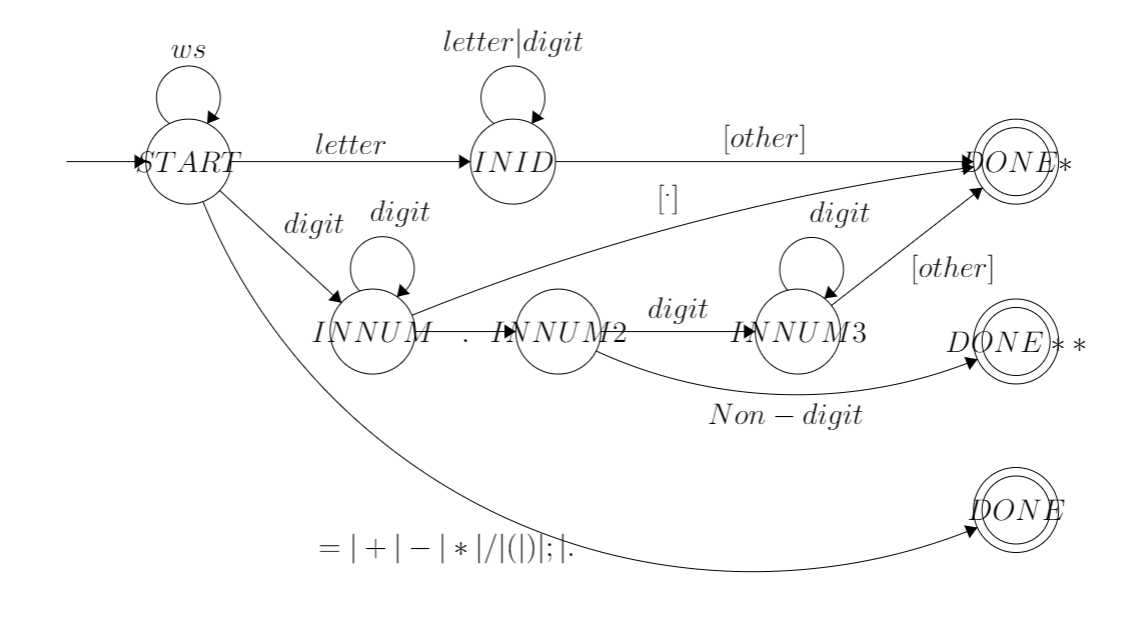
OVER -> /

LPAREN -> (

RPAREN -> )

DOT -> .

#### 2.1.3 FA图（不包括保留字）



### 2.2 语法

|  |  |
| --- | --- |
| 产生式 | 语义规则 |
| program -> stmt\_seq ‘.’ ENDFILE |  |
| Stmt\_seq -> stmt stmt\_seq\_tail |  |
| Stmt\_seq\_tail -> ; stmt\_seq | ε |  |
| Stmt -> **declaration\_stmt**|assign\_stmt|write\_stmt 注：C语言中声明不符合语句规则，此处自己规定 |  |
| declaration\_stmt -> float|int id | insertSymbolTable(float|int, id) |
| assign\_stmt -> id = expr | Id.syn = expr.syn |
| expr -> term expr\_tail | Expr\_tail.inh = term.syn  Expr.syn = expr\_tail.syn |
| expr\_tail -> +term expr\_tail1 | -term expr\_tail1 |ε | Expr\_tail1.inh = expr\_tail.inh +|- term.syn  Expr\_tail.syn = expr\_tail1.syn |
| term -> factor term\_tail | Term\_tail.inh = factor.syn  Term.syn = term\_tail.syn |
| term\_tail -> \*factor term\_tail1 | /factor term\_tail1 | ε | Term\_tail1.inh = term\_tail.inh \*|/ factor.syn  Term\_tail.syn = term\_tail1.syn  情况为ε:  Term\_tail.syn =term\_tail.inh |
| factor -> -number | number | Factor.syn = -number.syn | number.syn |
| number -> (expr)|id | NUM | Number.syn = expr.syn | id.val | num.lexval |
| write\_stmt -> write(expr) |  |

|  |  |  |
| --- | --- | --- |
| 非终结符 | First | Follow |
| program | { FLOAT,INT,ID,WRITE } | { $} |
| Stmt\_seq | { FLOAT,INT,ID,WRITE } | { . } |
| Stmt\_seq\_tail | { ;,ε } | { . } |
| Stmt | { FLOAT,INT,ID,WRITE } | { ; , . } |
| declaration\_stmt | { FLOAT,INT } | { ; , . } |
| assign\_stmt | { ID } | { ; , . } |
| expr | { -,NUM,ID,( } | { ; , . ,) } |
| expr\_tail | { +,-,ε } | { ; , . , ) } |
| term | { -,NUM,ID,( } | {+,-, ; , . , ) } |
| term\_tail | { \*,/,ε } | {+,-, ; , . , ) } |
| factor | { -,NUM,ID,( } | { \*, / , +,-, ; , . , ) } |
| number | { NUM,ID,( } | { \*, / , +,-, ; , ., ) } |
| write\_stmt | { WRITE } | { ; , .} |

## 3.程序实现

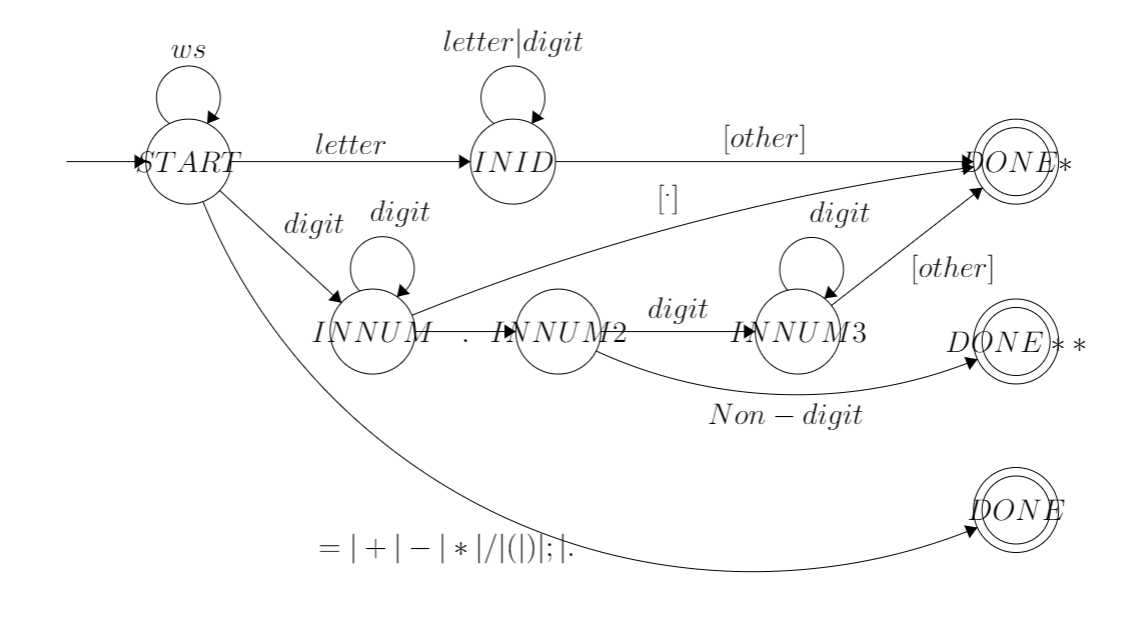
整体实现思路为递归下降的翻译器。

### 3.1 词法分析

linebuf

1.设计一个行缓冲区，读入文本每一行。

2.逐个字符扫描，同时应用自动机实现状态转变。



根据当前状态和当前获取的字符进行状态转变。

state[currentState][ch]=NextState.

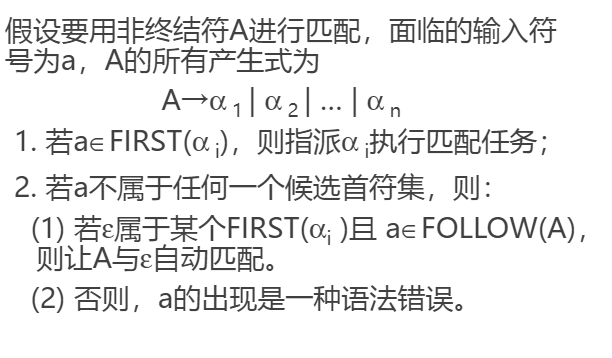
3.设计全局变量tokenString保留当前token字符串, token为当前token

### 3.2 语法分析

采用语法制导的递归下降翻译。

框架如下：

对每个非终结符设计一个函数。函数内容如下：



先设计好语法分析的框架。

然后考虑好翻译模式，在语法分析中加入翻译动作，边分析边翻译。

|  |  |
| --- | --- |
| 产生式 | 语义规则 |
| program -> stmt\_seq ‘.’ ENDFILE |  |
| Stmt\_seq -> stmt stmt\_seq\_tail |  |
| Stmt\_seq\_tail -> ; stmt\_seq | ε |  |
| Stmt -> **declaration\_stmt**|assign\_stmt|write\_stmt 注：C语言中声明不符合语句规则，此处自己规定 |  |
| declaration\_stmt -> float|int id | insertSymbolTable(float|int, id) |
| assign\_stmt -> id = expr | Id.syn = expr.syn |
| expr -> term expr\_tail | Expr\_tail.inh = term.syn  Expr.syn = expr\_tail.syn |
| expr\_tail -> +term expr\_tail1 | -term expr\_tail1 |ε | Expr\_tail1.inh = expr\_tail.inh +|- term.syn  Expr\_tail.syn = expr\_tail1.syn |
| term -> factor term\_tail | Term\_tail.inh = factor.syn  Term.syn = term\_tail.syn |
| term\_tail -> \*factor term\_tail1 | /factor term\_tail1 | ε | Term\_tail1.inh = term\_tail.inh \*|/ factor.syn  Term\_tail.syn = term\_tail1.syn  情况为ε:  Term\_tail.syn =term\_tail.inh |
| factor -> -number | number | Factor.syn = -number.syn | number.syn |
| number -> (expr)|id | NUM | Number.syn = expr.syn | id.val | num.lexval |
| write\_stmt -> write(expr) |  |

## 4.使用说明

1.以float id | int id形式声明变量。

2.用赋值语句对变量进行赋值，id = expr

3.用write(expr)打印结果

注意：

1.变量必须先声明，后使用。

2.必须先初始化，不可声明后第一次使用就出现在等号右侧。

3.可以仅使用write语句打印某个计算式。

4.赋值语句中，浮点数不可以转换为整数，整数可以转换为浮点数。

## 5.测试用例

Calculator.exe <srcFilename> <outFileName>

### Test1

float a;int b;

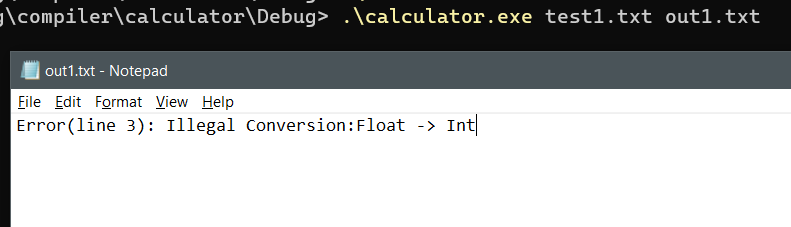
a=(10.44\*356+1.28)/2+1024\*1.6;

b=a\*2+a/2;

write(b);

write(a).

结果：



更正后

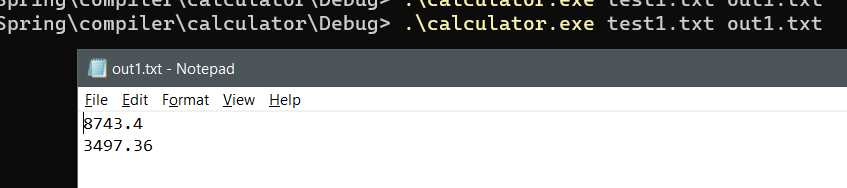
float a;float b;

a=(10.44\*356+1.28)/2+1024\*1.6;

b=a\*2+a/2;

write(b);

write(a).



### Test2

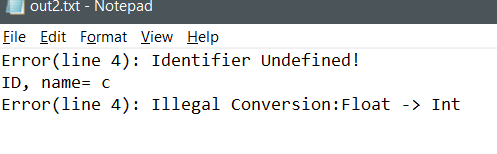
float a;

int b;

a=(10.44\*356+1.28)/2+1024\*1.6;

b=a\*2+c/2;

write(b).



### Test3

float a;

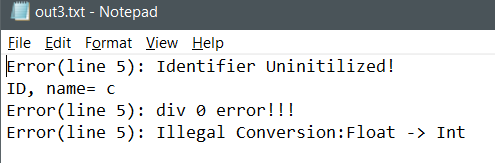
int b;

int c;

a=(10.44\*356+1.28)/2+1024\*1.6;

b=a\*2+c/0;

write(b).



## 6.实验总结

程序员三大浪漫之一：编译原理。

实践和理论结合的最好例子。