BNF – Backus Naur Form

Praseed Pai K.T.

BNF for date

<date> ::= <month> "/" <day> "/" <year>
<month> ::= "1" [0-2] | "0" [1-9]
<day> ::= "0" [1-9] | [1-2] [0-9] | "3" [0-1]
<year> ::= [0-9] [0-9] [0-9] [0-9]

BNF for integer

<integer> ::= <digit> | <digit> <integer>

BNF for floating point

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

A BNF for specifying Postal Address

<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL> | <personal-part> <name-part>

<personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

<opt-apt-num> ::= <apt-num> | ""

A note about Slang Compiler Infrastructure

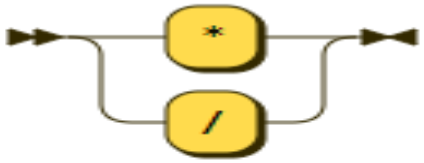
- An Open Source Compiler (to learn Compiler Engineering)
- Available @ https://github.com/praseedpai/SlangForDotNet
- Ports available to Java,C++,VB.net,JS,Python
- A JavaScript Port of the Compiler is available @ http://shinexavier.github.io/SlangJS/
- A Pascal Syntax Diagram for the BNF available @ http://shinexavier.github.io/SlangJS/SyntaxDiagram.xhtml

Pascal Syntax Diagram (based on BNF) for Slang

<expr> ::= <BExpr>
<BExpr> ::= <LExpr> LOGIC_OP <BExpr>
<LExpr> ::= <RExpr> REL_OP <LExpr>
<RExpr> ::= <Term> ADD_OP <RExpr>
<Term>::= <Factor> MUL_OP <Term>
<Factor> ::= <Numeric> | <String> | TRUE | FALSE | <variable> | '(' <expr> ')' | {+|-|!}<Factor> | <callexpr>
<callexpr> ::= funcname '(' actuals ')'
<LOGIC_OP> := '&&' | '||'
<REL_OP> := '>' |' < '|' >=' |' <=' |' <>' |' =='
<MUL_OP> := '*' |' /'
<ADD_OP> := '+' |' -'

Pascal Syntax Diagram for Slang (based on BNF)

MUL_OP:

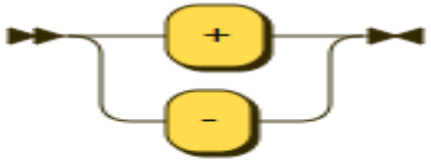



```
MUL_OP    ::= '*'
            | '/'
```

referenced by:

- Term

ADD_OP:

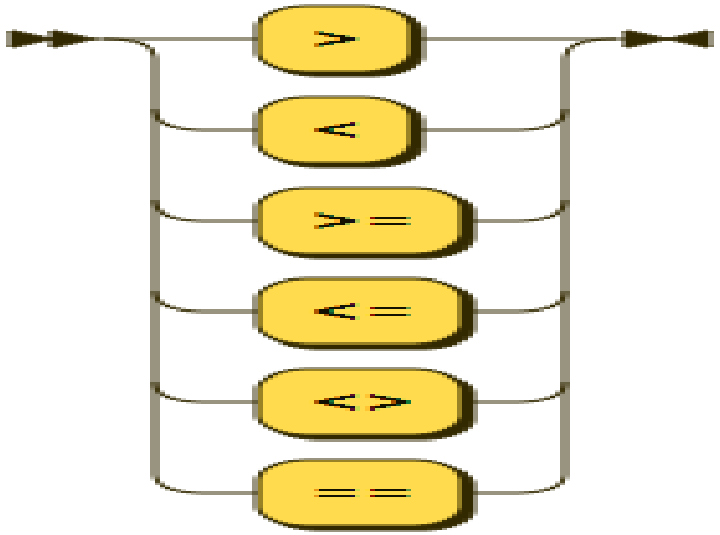



```
ADD_OP    ::= '+'
            | '-'
```

referenced by:

- RExpr

REL_OP:




```
REL_OP    ::= '>'
            | '<'
            | '>='
            | '<='
            | '<>'
            | '=='
```

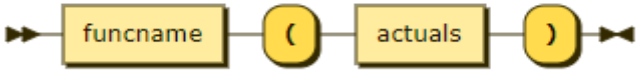
referenced by:

- LExpr

referenced by:

- Factor
- Term

callexpr:

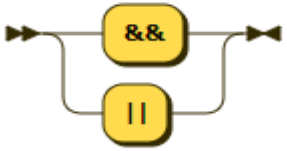



```
callexpr ::= funcname '(' actuals ')'
```

referenced by:

- Factor

LOGIC_OP:



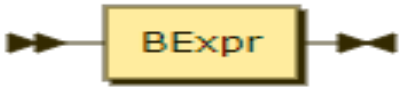

```
LOGIC_OP ::= '&&'
           | '||'
```

referenced by:

- BExpr

Pascal Syntax Diagram for Slang (based on BNF)

expr:

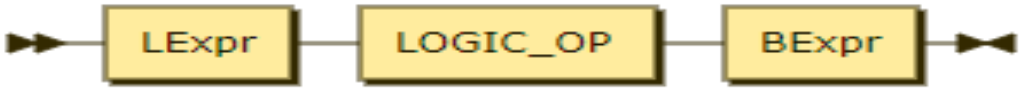



```
expr      ::= BExpr
```

referenced by:

- Factor
- ifstmt
- printstmt
- returnstmt
- whilestmt

BExpr:

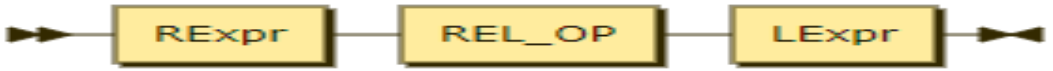



```
BExpr     ::= LExpr LOGIC_OP BExpr
```

referenced by:

- BExpr
- expr

LExpr:




```
LExpr     ::= RExpr REL_OP LExpr
```

referenced by:

- BExpr
- LExpr

RExpr:


```
RExpr     ::= Term ADD_OP RExpr
```

referenced by:

- LExpr
- RExpr

Pascal Syntax Diagram for Slang (based on BNF)

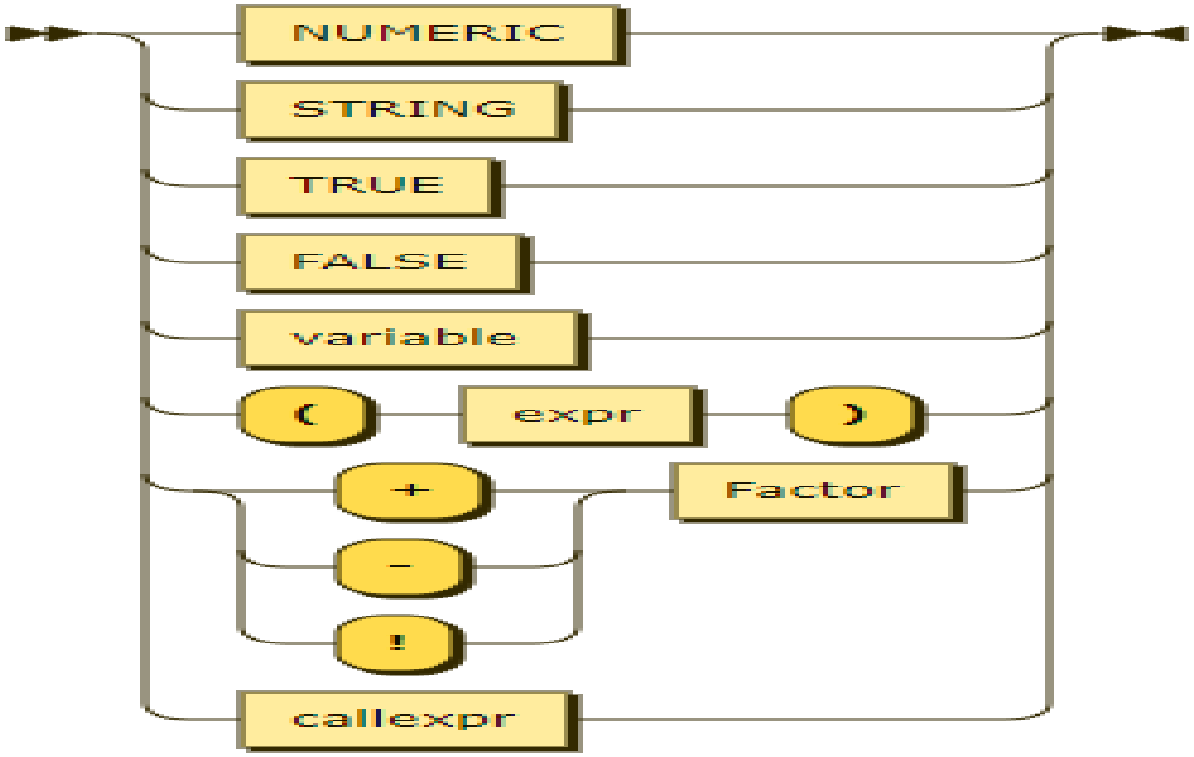
Term:


```
Term     ::= Factor MUL_OP Term
```

referenced by:

- RExpr
- Term

Factor:




```
Factor     ::=  NUMERIC
              |  STRING
              |  TRUE
              |  FALSE
              |  variable
              |  '(' expr ')'
              |  ( '+' | '-' | '!' ) Factor
              |  callexpr
```

Backus Naur Form for a Four function Calculator

<Expr> ::= <Term> | Term { + | - } <Expr>

<Term> ::= <Factor> | <Factor> {*|/} <Term>

<Factor>::= <number> | (<expr>) | {+|-} <factor>

Tokens and Lexical Analysis

```
TOK_PLUS – '+' TOK_MUL - '*' TOK_SUB - '-' TOK_DIV - '/'
TOK_OPAREN – '(' TOK_CPAREN – ')' TOK_DOUBLE – [0-9]+

/// Enumeration for Tokens

public enum TOKEN {
        ILLEGAL_TOKEN = -1, // Not a Token
        TOK_PLUS = 1, // '+'
        TOK_MUL, // '*'
        TOK_DIV, // '/'
        TOK_SUB, // '-'
        TOK_OPAREN, // '('
        TOK_CPAREN, // ')'
        TOK_DOUBLE, // '('
        TOK_NULL // End of string
}
```

```
while ( there is input ) {
switch(currentchar) {
case Operands:
advance input pointer
return TOK_XXXX;
case Number:
Extract the number( Advance the input )
return TOK_DOUBLE;
default:
error
}
}
```

Converting <Expr> Production to the Code

```
// <Expr> ::= <Term> { + | - } <Expr>
Void Expr() {
        Term();
        if ( Token == TOK_PLUS || Token == TOK_SUB ){
                // Emit instructions
                // and perform semantic operations
                Expr(); // recurse
        }
}
```

Converting <Term> Production to the Code

```
// <Term> ::= <Factor> { * | / } <Term>
Void Term() {
        Factor();
        if ( Token == TOK_MUL || Token == TOK_DIV ){
                // Emit instructions
                // and perform semantic operations
                Term(); // recurse
        }
}
```

Converting <Factor> Production to the Code

```
// <Factor> ::= <TOK_DOUBLE> | ( <expr> ) | { + |- } <Factor>
//
Void Factor() {
        switch(Token)
                case TOK_DOUBLE:
                // push token to IL operand stack return
                case TOK_OPAREN:
                        Expr(); //recurse
                // check for closing parenthesis and return
                case UNARYOP:
                        Factor(); //recurse
                default:
                //Error
}
```

A “Naïve” Stack Class

```
class Stack {
        double[] stk;
        int top_stack = 0;
        public Stack() { stk = new double[256]; top_stack = 0; }
        public void Clear(){ top_stack = 0; }
        public void push(double dbl ){
                if ( top_stack == 255 ){ throw new Exception();}
                stk[top_stack++] = dbl;
        }
        public double  pop(){
                if ( top_stack == 0 ){ throw new Exception(); }
                return stk[--top_stack];
        }
}
```

Recursive Descent Parsing

```
class RDParser : Lexer {
      TOKEN Current_Token; Stack ValueStack = new Stack() ;
      public RDParser(String str):base(str){}
         public double CallExpr(){
           ValueStack.Clear(); Current_Token= GetToken(); Expr();
        double nd = ValueStack.pop(); return nd;
         }
         public void Expr(){
           TOKEN l_token; Term();
           if (Current_Token == TOKEN.TOK_PLUS || Current_Token == TOKEN.TOK_SUB){
             l_token = Current_Token; Current_Token = GetToken();
             Expr();
         double x = ValueStack.pop(); double y = ValueStack.pop();
         ValueStack.push( (l_token == TOKEN.TOK_PLUS ) ? (x + y) : (y-x) );
       }
      }
         public void Term(){
        TOKEN l_token; Factor();
           if (Current_Token == TOKEN.TOK_MUL || Current_Token==TOKEN.TOK_DIV) {
             l_token = Current_Token; Current_Token = GetToken(); Term();
             double x = ValueStack.pop();double y = ValueStack.pop();
```

Recursive Descent Parsing (Contd…)

```
            if ( x == 0 ) { throw new Exception();}
            ValueStack.push( (l_token == TOKEN.TOK_MUL ) ? (x * y) :  (y/x) );
          }
      }
         public void Factor(){
        TOKEN l_token;
          if ( Current_Token == TOKEN.TOK_DOUBLE ){
           ValueStack.push(GetNumber());Current_Token = GetToken();
          } else if ( Current_Token == TOKEN.TOK_OPAREN ) {
            Current_Token = GetToken();   Expr();  // Recurse
         if ( Current_Token != TOKEN.TOK_CPAREN ){ throw new Exception();}
            Current_Token = GetToken();
           } else if (Current_Token == TOKEN.TOK_PLUS||Current_Token == TOKEN.TOK_SUB ){
         l_token = Current_Token; Current_Token = GetToken();
            Factor(); double  x = ValueStack.pop();
            if ( l_token == TOKEN.TOK_SUB ) { x = -x; }
            ValueStack.push(x);
          } else { throw new Exception();}
        }
}
```

Run and Evaluate

https://github.com/praseedpai/ElementaryMathForProgrammingSeries/blob/master/AlgebraNArith/ExprEval/RDParserStack.cs

```
D:\cpp>RDParserStack "2+3"
The Evaluated Value is 5

D:\cpp>RDParserStack "2+3*4"
The Evaluated Value is 14

D:\cpp>RDParserStack "2+3*-4"
The Evaluated Value is -10
```

Q&A

- If any!