Practical Exercise: Development of a Recursive Descent Interpreter

In this guided practical exercise, we will approach the design of an Interpreter with basic resources to review the main concepts of a Recursive Descent Parser. To avoid dealing with a large and complicated grammar, we will restrict the domain to the typical arithmetic expression calculator. This way, we can obtain results with a reduced number of production rules.

We will begin with a very elementary approach, and complicate it in successive steps:

1. A parser for very simple operations.
2. A calculator for very simple operations (Parser + Semantic Routines).
3. Inclusion of expressions with parentheses.
4. Inclusion of operator precedence, and unary signs.

4. Inclusion of operator precedence, and unary signs

The last section proposes the extension of the calculator to

- Accept unary signs.
- Apply the adequate precedence to the arithmetic operators.

This is proposed as homework for the students.

Some suggestions:

- Unary + and - signs can be limited to the *token Number*.
- To assign greater precedence to one operator (*) over another (+), a new level must be included in the grammar. It is proposed to follow a scheme like the following one (keep Term as in the original grammar, and introduce a new one, like F).

```
E ::= F + E | …

F ::= T * F | …

T ::= ( E ) | …
```

We start with some changes to improve the presentation and consistency of some details of the parser.

Until now the *tokens* are read "on demand" just before making a match with some expected syntactic element. The change that is proposed is to read the next token when a match has just been completed. In this way, we can perform these read operations in the parse operations at the lowest level, avoiding the need to read the next token in situations in which the NEXT set must be checked.

The changes are the following:

```
#define ParseLParen() MatchSymbol ('(') ; rd_lex () ;       //More concise and efficient definitions
#define ParseRParen() MatchSymbol (')') ; rd_lex () ;       // rather than defining functions
                                                            // This is only useful for matching Literals

int ParseNumber ()          // Parsing Non Terminals and some Tokens requires more complex functions
{
        MatchSymbol (T_NUMBER) ;

        rd_lex () ;

        return number ;
}
```

We can eliminate:

```
int ParseTerm ()                // T ::= N | ( E )    returns the numeric value of the Term
{
        int val ;

        rd_lex () ;
        if (token == T_NUMBER) {                // T derives alternatives, requires checking FIRST( E )
                val = ParseNumber () ;
. . .
}

int ParseExpression ()                  // E ::= TE' + E' ::= lambda | E
{                                       // returns the numeric value of the Expression
. . .
        rd_lex () ;
        if (token == '\n' || token == ')') {    // ExpressionRest is a nullable Non Terminal
                                                // Therefore, we check FOLLOW(ExpressionRest)
. . .
```

Now, the parser starts reading the first token.

```
int main (void)
{
        while (1) {
                rd_lex () ;
                printf ("Value %d\n", ParseExpression ()) ;
        }

        system ("PAUSE") ;
}
```

Another change is to produce a left-factored production of the arithmetic operators in:

```
ExpressionRest ::=    + Expression |
                      - Expression |
                      * Expression |
                      / Expression |
                      lambda
```


Juan Manuel Alonso Weber

The result is:

```
ExpressionRest ::=    Operator Expression |
                      lambda

Operator ::=          + | - | * | /
```

And this modifies the function *ParseOperator()*:

```
int ParseOperator ()
{
        int operator ;

        switch (token) {                // Operator derives in alternatives
                case '+' :              // requires checking FIRST(Operator)
                case '-' :
                case '*' :
                case '/' :  operator = token ;
                            break ;
                default :   rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                            break ;
        }

        rd_lex () ;

        return operator ;
}
```