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.

1. A Parser for very simple operations

dr_calc1.c

```
#include <stdio.h>
#include <stdlib.h>

#define T_NUMBER 1001
#define T_OP     1002

int token ;                             // Here we store the current token/literal
int number ;                            // and the value of the number

int line_counter = 1 ;

int rd_lex ()
{
        int c ;

        do {
                c = getchar () ;
        } while (c == ' ' || c == '\t') ;

        if (isdigit (c)) {
                ungetc (c, stdin) ;
                scanf ("%d", &number) ;
                token = T_NUMBER ;
                return (token) ;        // returns the Token for Number
        }

        if (c == '\n')
                line_counter++ ;        // info for rd_syntax_error()

        token = c ;
        return (token) ;        // returns a literal
}

void rd_syntax_error (int expected, int token, char *output)
{
```


Juan Manuel Alonso Weber

```
		fprintf (stderr, "ERROR in line %d ", line_counter) ;
		fprintf (stderr, output, token, expected) ;

		exit (0) ;
}

void MatchSymbol (int expected_token)
{
		if (token != expected_token) {
			rd_syntax_error (expected_token, token, "token %d expected, but %d was read") ;
		}
}

void ParseNumber ()
{
		MatchSymbol (T_NUMBER) ;
}

void ParseTerm ()		// T ::= N
{
		rd_lex () ;
		ParseNumber () ;
}


void ParseExpression () ;	// required prototype for forward reference in mutual recursion

void ParseExpressionRest ()	// E' ::= lambda | OpE
{				// ExpressionRest is a nullable Non Terminal
		rd_lex () ;
		if (token == '\n') {			// Therefore, we check FOLLOW(ExpressionRest)
			return ;			// This means that lambda has been derived
		}

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

		ParseExpression () ;		// Forward reference in mutual recursion requires a previous prototye
}


void ParseExpression ()			// E ::= TE'
{
		ParseTerm () ;
		ParseExpressionRest () ;
}

int main (void) {

		while (1) {
			ParseExpression () ;
			printf ("OK\n") ;
		}

		system ("PAUSE") ;
}
```