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.

3. Including expressions with parentheses

dr_calc3.c

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

#define T_NUMBER 1001
#define T_OP     1002

int ParseExpression () ;                    // Prototype for forward reference

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 ;
```


Juan Manuel Alonso Weber

```
	return (token) ;	// returns a literal
}

void rd_syntax_error (int expected, int token, char *output)
{
	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") ;
	}
}


#define ParseLParen()		MatchSymbol ('(') ; // More concise and efficient definitions
#define ParseRParen()		MatchSymbol (')') ; // rather than using functions
					// This is only useful for matching Literals
int ParseNumber ()			// Parsing Non Terminals and some Tokens require more
{					// complex functions
	MatchSymbol (T_NUMBER) ;
	return number ;
}


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 () ;
	} else {
		ParseLParen () ;
		val = ParseExpression () ;
		ParseRParen () ;
	}
	return val ;
}

int ParseExpression ()			// E ::= TE' + E' ::= lambda | E
{					// returns the numeric value of the Expression
	int val, val2, operator ;

	val = ParseTerm () ;

//	ParseExpressionRes t() ;			// we expand this function into ParseExpression()

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

	switch (token) {		// ExpressionRest derives in alternatives
		case '+' :		// requires checking FIRST(ExpressionRest))
		case '-' :
		case '*' :
```

```
                case '/' :  operator = token ;
                            break ;
                default :   rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                            break ;
        }
        val2 = ParseExpression () ;
                                                // At this point the input has been parsed correctly
        switch (operator) {                     // This part is for the Semantic actions
                case '+' :  val += val2 ;
                            break ;
                case '-' :  val -= val2 ;
                            break ;
                case '*' :  val *= val2 ;
                            break ;
                case '/' :  val /= val2 ;
                            break ;
                default :   rd_syntax_error (operator, 0, "Unexpected error in ParseExpressionRest for
operator %c\n") ;
                            break ;
        }
        return val ;
}

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


Juan Manuel Alonso Weber