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.

2. A calculator for very simple operations (Parser + Semantic Actions)

We now need to elaborate Parser Analyzer that is able to check the syntactic correction of the input, and in addition, to evaluate the sequence and return the numeric value of the expression.

Let's address a simple solution. It implies that some functions return a value, for example *ParseNumber*, *ParseTerm*, *ParseExpression*, and that the latter one is able to operate with these values. One complication is that Expressions are split in two (*Expression* and *ExpressionRest*), and we have to operate with values obtained in both. So, we would need a mechanism to transmit a value from the first function to the second one (for example, passing arguments). This will be avoided with a design decision.

There will be no change in the initial code. The changes start in *ParseNumber*(). Specifically, its type is changed from *void to int* and at the end the value stored in the global variable *number* is returned. This can be done if *ParseNumber*() verifies that there is a current token *T_Number*. The same applies for *ParseTerm*().

```
int ParseNumber ()
{
        MatchSymbol (T_NUMBER) ;
        return number ;
}

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

        rd_lex () ;
        val = ParseNumber () ;

        return val ;
}
```


Juan Manuel Alonso Weber

For *ParseExpression*() and *ParseExpressionRest*() we take the decision to integrate the code of the second function into the first one (instead of making the call to the second). The changes involve the management of the values returned by *ParseTerm*() and *ParseExpression*(). In the case of lambda derivation, the value of the term read last must be returned. It is also necessary to store the read operator so it can be used later to operate. For this the variables *val*, *val2* and *operator* are defined.

```
int ParseExpression ()                          // E ::= TE'    U    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') {                    // Therefore, we check FOLLOW(Expression)
                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 ;  // remember the operator for later
                            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
. . .
```

At the point after:

```
        val2 = ParseExpression () ;             // At this point the input has been parsed correctly
```

we know that the parsing has been satisfactory, with which we can proceed to calculate the value of the expression. Therefore, the switch / case is repeated for calculating the corresponding operations. The default option has been included to detect an "unlikely" error.

```
. . .
        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 ;
        }
```


Juan Manuel Alonso Weber

```
	return val ;
}
```

A last minimal change corresponds to the *main*() function that will now print the value of the evaluated expression:

```
int main (void) {

	while (1) {
		printf ("Value %d OK\n", ParseExpression ()) ;
	}

	system ("PAUSE") ;
}
```

Now we can test the program on the following sequences:

```
1+2*3
Value 7 OK
3*2+1
Value 9 OK
3
Value 3 OK

ERROR in line 5 token 10 expected, but 1001 was read
```