uc3m | Universidad **Carlos III** de Madrid

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

Including the reading of the next *token* allows to eliminate low-level operations in the more complex parser functions that deal only with Non-Terminals rather than with *tokens*. Thus, the *ParseExpression*() function is transformed into:

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

        val = ParseTerm () ;

//      ParseExpressionRest () ;                    // we expand this function into ParseExpression()

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

        operator = ParseOperator () ;

        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, "Error in ParseExpressionRest for operator %c\n") ;
```


Juan Manuel Alonso Weber

```
                    break ;
      }

      return val ;
}
```

The low-level operations that cannot be eliminated are the query of FOLLOW(E '), and everything related to the semantics associated with the production.

We use the *ParseExpression*() function to integrate the two functions corresponding to E (Expression) and E' (ExpressionRest). Although the grammar cannot reflect this fusion, we can represent it with the EBNF notation:

```
Expression ::=        Term [Operator Expression]*
```

Which is equivalent to:

```
Expression ::=        Term ExpressionRest
ExpressionRest ::=    Operator Expression |
                      lambda
```

The [Operator Expression] * fragment indicates that it is a nullable sequence, so we need to insert a previous check on the FOLLOW set to determine if the analysis process should be terminated.

Juan Manuel Alonso Weber