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

The simplest approach corresponds to an input sequence of integer operands and arithmetic operators (+, -, * /) that ends by a line break. No precedence or associativity will be considered. Neither will we try to obtain the numerical result of the expression. We just want to check the syntax correction of the input. It should be able to recognize the following sequences:

```
1+2*3
       OK
3*2+1
       OK
3
       OK
```

Our initial grammar is:

```
Expression ::=  Expression + Expression |
                Expression - Expression |
                Expression * Expression |
                Expression / Expression |
                Number
```

Where *Number* is a *token* that represents an integer value.

We know that in the grammar for a recursive descendent parser no left-recursion is allowed. We choose to redesign the grammar:

```
Expression ::=  Term + Expression |
                Term - Expression |
                Term * Expression |
                Term / Expression |
                Term
```

```
Term ::=        Number
```

It is also not allowed for the same Non-Terminal to derive in productions with the same beginning (which produces a Non-Determinism). So, we left-factor the arithmetic productions:

```
Expression ::=        Term ExpressionRest

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

Term ::=              Number
```

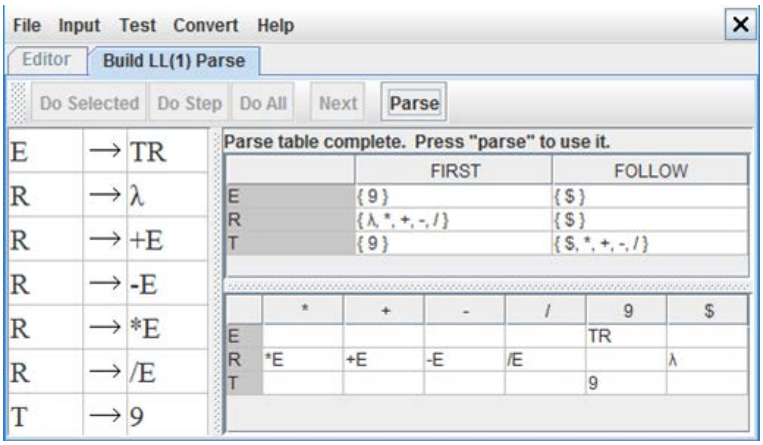
We see that ExpressionRest has several alternative productions, and, additionally, derives lambda. In the first case, it means that to decide which production to apply we will need to check the token read at the given moment. It must belong to the FIRST set of the ExpressionRest symbol. In this case it is easy to see which Terminals belong to this one. FIRST(ExpressionRest) = {+, -, *, /, □} since they are the first symbols of each of the five alternative productions.

Since ExpressionRest derives *lambda*, it is a nullable symbol. Specifically, it serves to generate a finite sequence of operator-operand pairs. To detect when this sequence ends, we need to know the Follow set (ExpressionRest). This set will contain only the final symbol of the sequence. It is usually represented by the $ symbol. We had specified that the arithmetic sequence ends with a line break. That is what the $ symbol will represent.

We can use a tool to calculate the First and Following sets. For example, with JFLAP, we can enter the grammar representing:

```
Expression      ➔    E
ExpressionRest  ➔    R
Term            ➔    T
Number          ➔    9
```

We select <Input> → <Build LL (1) Table> and we obtain the FIRST and FOLLOW sets.



Now we can address the design of the parser in the next step.

We begin with the initial definitions and the Lexical Analyzer:

```
#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
}
```


Juan Manuel Alonso Weber

The *rd_lex*() Lexical Analyzer reads the entry character-by-character, skipping the white spaces and tabulators. If a given character is a digit, it returns it to the input stream (with *ungetc*), and reads a whole integer with *scanf*. The number is stored in the global variable number (although this value is not used in this section). And it returns the token *T_Number*, which is also stored in the variable token.

In any other case, it returns the character read as literal (it is also stored in the variable token).

In case of line break, the *line_counter* counter is incremented to give an approximate information in case of error.

The variable token has the function of storing the lexeme last read, and number stores the value of the integer last read (when appropriate).

Syntax errors are handled with the rd_syntax_error() function (which can be improved). In case of invoking this function, a message is printed, and the program ends:

```
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) ;
}
```

The main function calls the parser inside an infinite loop. The parser will terminate every time a line break is introduced. If there is an error, the parser outputs a message and the program ends. If all goes well, the loop in main prints OK and repeats the process.

```
int main (void)
{
	while (1) {
		ParseExpression () ;
		printf ("OK\n") ;
	}

	system ("PAUSE") ;
}
```

The ParseExpression () function is the parser's main function. We see that it follows faithfully the grammatical production for Expression:

```
Expression ::= Term ExpressionRest
```

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

Each Non-Terminal of the Grammar will have associated a function that is responsible for analyzing the input assuming that it corresponds to the Non-Terminal.

For example, ParseTerm() checks what its production indicates

```
Term ::= Number
```

Since *Number* is a *Token*, and not another *Non-Terminal*, we will have to read something concrete from the input (calling rd_lex ()). The verification is done by calling ParseNumber ().

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

ParseNumber() uses the generic function *MatchSymbol(token)* to check that the lexeme read is of type *T_NUMBER.*

```
void ParseNumber ()
{
        MatchSymbol (T_NUMBER) ;
}
```

MatchSymbol() checks that the current token corresponds to the last argument. Otherwise, it outputs an error message and finishes.

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

It is understood that when a *ParseXXX*() function finishes, the analysis process has been correct.

We have to comment on the most elaborate function, which is *ParseExpressionRest*(), since this has to deal with alternative and null-able productions.

Since there is mutual recursion between *ParseExpression*() and *ParseExpressionRest*() we will need to include at least one prototype, to solve the reference in advance.

ParseExpressionRest() expects either an operator (followed by an operand) or *lambda*. This means that it has to work at the level of terminal symbols, so it has to perform a read with *rd_lex*().

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

void ParseExpressionRest ()  // E' ::= lambda | Op E
{
        rd_lex () ;                             // ExpressionRest is a nullable Non Terminal
        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
}
```

The first point we need to ensure is that there will be an operator-operating pair in the input. For this we check that the current token is not in the NEXT set (of ExpressionRest). This set only contains the symbol '\ n' (line break == $). If we had read the line break, it is understood that ExpressionRest derives in lambda, and we must return from the current function.

If this is not true, we need to verify that the current token corresponds to one of the symbols of FOLLOW(ExpressionRest) other than lambda. In this case, these can be one of the four arithmetic operators (+, -, * and /). We use a switch/case to verify this. Obviously, if a different symbol (eg,%) has been read, there will be a syntactic error. There is no more code in this block since checking the four cases is enough to verify the syntactic correction. In addition, the end of the four productions is the same

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

So, we can analyze the Expression symbol independently of the read operator. We use a recursive call to *ParseExpression*().

With this, we are now able to operate on the following sequences:

```
1+2*3
OK
3*2+1
OK
3
OK

ERROR in line 4 token 10 expected, but 1001 was read
```