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.

3. Including expressions with parentheses

The next change is proposed to see how to modify the code when the grammar is expanded. We are going to limit ourselves to operate with expressions including parentheses.

The modification is highlighted in the new grammar:

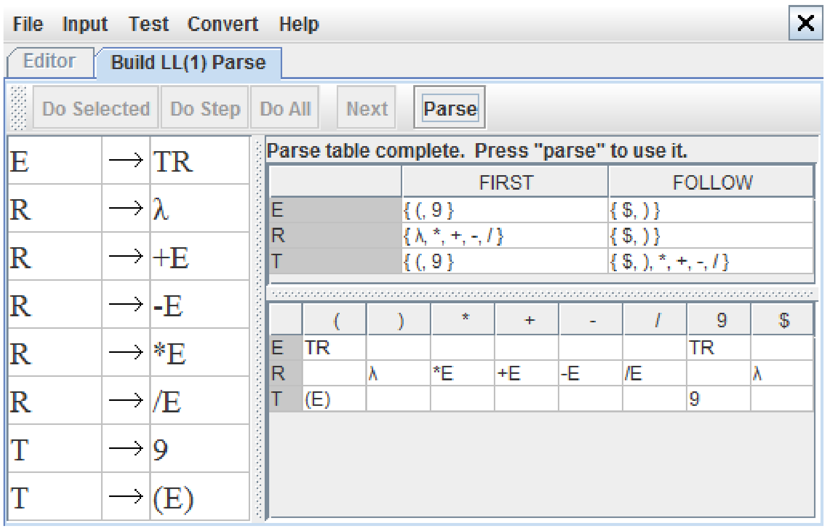
```
Expression ::=        Term ExpressionRest

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

Term ::=              Number |
                      ( Expression )
```

The changes will only affect the *ParseTerm*() function. We see that the grammar fulfils the requirements for a Recursive Descending Parser (without left recursion, without productions that generate Non-Determinism). But now, Term generates two alternative productions, which forces us to calculate its FIRST set. In this case, the calculation is simple, since both alternatives begin with a Terminal Symbol in the first derivation: FIRST (Term) = {Number, (}

Term is not nullable symbol, therefore there is no need to calculate the FOLLOING (Term) set. But you must always be aware that any changes in the grammar can influence the calculation of all sets!



E	→	TR
R	→	λ
R	→	+E
R	→	-E
R	→	*E
R	→	/E
T	→	9
T	→	(E)

Parse table complete. Press "parse" to use it.

	FIRST	FOLLOW
E	{ (, 9 }	{ $,) }
R	{ λ, *, +, -, / }	{ $,) }
T	{ (, 9 }	{ $,), *, +, -, / }

	(	)	*	+	-	/	9	$
E	TR						TR	
R		λ	*E	+E	-E	/E		λ
T	(E)						9	

We can see that almost all sets change, although the FOLLOWING(ExpressionRest) is the only set to consider, since for the other Non-Terminals there are no alternative or nullable productions.

The necessary changes are indicated in the following fragments. *ParseTerm*() is modified to process the alternative of an expression within parentheses. First, it checks if the token is any of the symbols in FIRST(Term). If it is a token of numeric type, *ParseNumber*() is called, otherwise it is assumed to be an expression within parentheses and *ParseLParen*() followed by *ParseExpression*() and *ParseRParen*() are called. *ParseLParen*() is responsible for giving an error if the token is not the opening parenthesis (contained in FIRST(Term)). The recursive reference in advance to *ParseExpression*() requires a prototype.

```
int ParseExpression () ;                    // Prototype for forward reference

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(Expression)
                    val = ParseNumber () ;
          } else {
                    ParseLParen () ;
                    val = ParseExpression () ;
                    ParseRParen () ;
          }

          return val ;
}
```

ParseLParen() and *ParseRParen*() should have their corresponding functions, but in this case we have redefined them as macros based on *MatchSymbol*(*XXX*). This is a common practice for writing more efficient code, since each call to a function (in this case to *ParseLParen*() and *ParseRParen*()) implies an overhead in generating space for parameters, local variables, etc.

```
#define ParseLParen()        MatchSymbol ('(') ; // More concise and efficient definitions
#define ParseRParen()        MatchSymbol (')') ; // rather than using functions
                                                 // This is only useful for matching Literals
```

The only additional change is in *ParseExpression*() where we need to check if lambda is being derived.

```
int ParseExpression () {                 // E ::= TE'    U    E' ::= lambda | E
{
. . .
        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
. . .
}
```

Now we can test the program on the following sequences:

```
1+2*3
Value 7
3*2+1
Value 9
3
Value 3
1+(2*3)
Value 7
3*(2+1)
Value 9
(3)*(2)+(1)
Value 9
(3)
Value 3

ERROR in line 9 token 10 expected, but 40 was read
```

We see that the operator precedence of * with respect to + is not met. 3 * 2 + 1 should give 7, as well as for 1 + 2 * 3.

In the following example, it can also be seen that the left associativity is not fulfilled (the result should be -1). Although the theory says that a recursive descending parser applies left to right associativity (the correct one in this case), in this implementation it happens that the evaluations are performed in reverse when returning from the recursive calls. Hence, the expressions are evaluated from right to left.

```
1-1-1
Value 1

ERROR in line 3 token 10 expected, but 40 was read
```