A simple Recursive Descent Parser (RDP)

A Recursive Descent Parser is so named because it consists of a collection of subprograms, many of which are recursive, and where each such subprogram implements one of the non-terminals of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

In an arithmetic expression, statements are often nested in other statements. Parenthesis in expressions must be properly nested. The syntax of these structures is naturally described with recursive grammar rules.

EBNF (Extended BNF) is ideally suited for recursive-descent parsers. The primary EBNF extensions are braces, which specify that what they enclose can appear zero or more times, and brackets, which specify that what they enclose can appear once or not at all. In both cases, the enclosed symbols are optional.

Consider the following EBNF description of simple arithmetic expression:

Expr -> Term **{**("+" | "-") Term**}**

Term -> Factor **{**("\*" | "/") Factor**}**

Factor -> number | "**(**" Expr "**)**"

The rule for Expr is either a Term or a Term followed by multiple occurrences of a ‘+’ or a ‘-‘ followed by another Term. Its equivalent in BNF it is shown below:

Expr -> Expr + Term

| Expr – Term

| Term

Term -> Term \* Factor

| Term / Factor

| Factor

Factor -> number

| “(” Expr “)”

Description of RDP

The pseudo code for RDP parser program, shown below, illustrates recursive descent parsing using a pure procedural approach.

The parser reads simple integer expressions using the operators **+**, **-**, **\***, **/**, and **(** **)** (a parenthesized expression) and numbers and calculates the answer.

The code has the following main components:

1. The Scanner. Provides Tokens. Java’s StringTokenizer can be used to break the expression string into tokens by specifying both numbers and operators as well as parenthesis.
2. The Parser – the subprograms corresponding to the non-terminals. The non-terminals in the given grammar are:
   1. Statement
   2. Expression
   3. Term
   4. Factor
3. The main driver; reads the expression from the console (not a file).
4. A class defining the token codes as named constants.

**The grammar**:

<Statement> -> {<Expression> ";"} "."

<Expression> -> <Term> {("+" | "-") <Term>}

<Term> -> <Factor> {("\*" | "/") <Factor>}

<Factor> -> number | "**(**" <Expression> "**)**"

A Statement is an expression ending in a semicolon. End of the input statements is indicated by a period (‘.’). The Lexical analyzer function calls the StringTokenizer to get the next lexeme and stores its code in the global variable nextToken.

For each terminal symbol in the RHS, that terminal symbol is compared with nextToken. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get the next input token.

For each non-terminal, the parsing subprogram for that non-terminal is called.

Recursive descent parsing subprograms are written with the convention that each one leaves the next token of input in nextToken. So, whenever a parsing function begins, it assumes that nextToken has the code for the leftmost token for the input that has not yet been used in the parsing process.

Also, notice that each parsing subprogram starts with a comment line showing the rule for that non-terminal.

The expression function parses one or more terms, separated by either plus or minus operators. Therefore, first it calls the function that parses terms (term). Then it continues to call that function as long as it finds ADD\_OP or SUB\_OP (which it passes over by calling the lexical analyzer)

The function for <factor> non-terminal has to choose between two RHSs. It also includes error detection. In the function for <factor>, when a syntax error is detected, the function calls the error() function.

class Parser {

private Scanner scanner;

public Parser(Scanner scanner) {

} // Parser

public void run ( ) {

// get Token

// call statement() to process the ROOT of the tree, namely,

// statement.

} // run

private void statement () {

/\* statement = {expression ";" } "." \*/

// While (scanner.token != Token.period) {

// call expression()

// Display the returned value of expression

// read the next token which is a “;”. Skip it.

}

} // statement

private int expression () {

/\* expression = term { ( "+" | "-" ) term } \*/

// Parse first term: int left = term();

// while ( nextToken is + or -, get the next token and parse the

// next term)

{

// save token in saveToken

// Read next token

// if it is *plusop*, parse the next term and

// add value to the value of the left term

// computed earlier.

// If it is a *minuop,* read the next term and

// subtract its value from the value of the left term

// computed earlier

} // while

// return result

} // end of expression

private int term () {

/\* term = factor {("\*" | "/") factor} \*/

// compute term by calling factor first

// While (token is a multOp or divOp {

// save the token

// get the next token

// if saved token is multOp

// call factor again.

// multiply its value by the value of the left token and

//save the result

// If the saved token is divOp

// call factor again.

// Divide the saved value of the first factor

// by the value of the second call to factor.

// Save the result.

} // while

// return result

} // end of term

private int factor () {

/\* factor = number | "(" expression ")" \*/

// int value = 0;

// Depending on the token read from the scanner, process it

// If it is a number, call the number method and save the value

// get the next token.

// return

// If the token is a lparen, read the next token

// Call expression procedure. Store the result

// If token read is not a rparen, display error message

// Get the next token and return

// Otherwise:

// Display error saying expecting “number” or “(“

// return value.

} // end of factor

} // end of class Parser