

Assignment 3 Recursive Descent Parsing

Due: March 25th at 11:59 PM

SEG2106 – Software Construction

Part A

Recursive Descent Parsing

Recursive Descent Parsing is a technique for parsing an LL(1) compatible grammar. The structure of a Recursive Descent Parser closely corresponds to that of the grammar, thus making it intuitive to understand and implement. An LL(1) grammar cannot be ambiguous or left recursive, and has to be left factored.

Consider the following grammar that we have seen in class:

```
<expr> ::= <term><expr'>
<expr'> ::= +<expr>
           | -<expr>
           | ε
<term> ::= <factor><term'>
<term'> ::= *<term>
           | /<term>
           | ε
<factor> ::= num
           | id
```

To produce a recursive descent parser, we first calculate the FOLLOW sets for non-terminals that can derive an *epsilon*. For instance, we have the following two rules in our grammar that derive an *epsilon*:

<expr'> ::= ε

<term'> ::= ε

Therefore, we calculate the FOLLOW sets of **<expr'>** and **<term'>**:

FOLLOW(expr') = {\$}

FOLLOW(term') = {\$, +, -}

Second, we produce a simple Recursive Descent Parser from the grammar by associating a procedure with each non-terminal. The pseudo code of such mechanism is shown below. Note that “token” is a global variable shared among all the given procedures.

```
Procedure: main ()
    token ← getNextToken();
    if (expr() == ERROR || token != "$") then
        return ERROR;
    else
        return OK;
```

```
Procedure: expr ()
    if (term() == ERROR) then
        return ERROR;
    else return expr_prime();
```

```
Procedure: expr_prime ()
    if (token == "+") then
        token ← getNextToken();
        return expr();
    else if (token == "-") then
        token ← getNextToken();
        return expr();
    // This is a special case where we need to process the epsilon
    // by checking against the terminal in the FOLLOW(expr') set
    else if (token == "$") then
        return OK;
    else return ERROR;
```

```
Procedure: term ()
    if (factor() == ERROR) then
        return ERROR;
    else return term_prime();
```

```
Procedure: term_prime()
    if (token == "*") then
        token ← getNextToken();
        return term();
    else if (token == "/") then
        token ← getNextToken();
        return term();
    // This is a special case where we need to process the epsilon
    // by checking against the terminals in the FOLLOW(term') set
    else if (token == "$" || token=="+" || token=="-") then
        return OK;
    else return ERROR;
```

```
Procedure: factor ()
    if (token == "num") then
        token ← getNextToken();
        return OK;
    else if (token == "id") then
        token ← getNextToken();
        return OK;
    else return ERROR;
```

Study the above given pseudo code very well. Make sure you understand all of its details, **especially how epsilon elements are handled.**

Part B

Given the following grammar for a **Very Simple Programming Language** (VSPL):

```
<program> ::= begin <statement_list> end
<statement_list> ::= <statement> ; <statement_list>
<statement_list> ::= <statement> ;
<statement> ::= id = <expression>
<expression> ::= <factor> + <factor>
<expression> ::= <factor> - <factor>
<expression> ::= <factor>
<factor> ::= id | num
```

The following grammar symbols are non-terminals:

```
program
statement_list
statement
expression
factor
```

The following grammar symbols are terminals:

```
begin
end
;
id
num
=
+
-
```

Below is a sample program written in VSPL:

```
begin
    a = 15;
    b = 20;
    c = a + b;
end
```

Exercise 1 (15 points)

Convert the **VSPL** context free grammar into an LL(1) grammar. Make all the necessary adjustments (if any are needed). Please assume that the grammar is not ambiguous. Therefore, you just need to eliminate left recursion and perform left factoring.

Exercise 2 (60 points)

Write a Java program that performs Recursive Descent Parsing on the LL(1) grammar you have previously produced. Your parser gets its tokens from a file (instead of receiving them from a scanner). When the parsing activity terminates, your program should display one of the following messages:

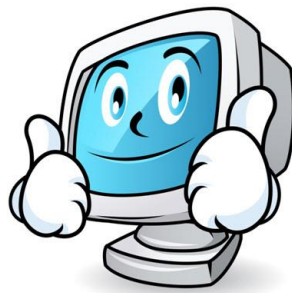
- **SUCCESS:** the code has been successfully parsed (*in case of success*)
- **ERROR:** the code contains a syntax mistake (*in case of failure*)

Three input files containing a list of tokens have been attached to this assignment. The first one is called `input1.txt` and contains no syntax errors. The second and third are called `input2.txt` and `input3.txt` and contain syntax errors. In these files, each token is written on a separate line. Therefore, in order to get a token from the file, you can simply use a `readline()` call.

Note that the developed Java program should receive the name of the file as an argument passed to its main method. Also, make sure to include your `.java` files as part of your assignment submission.

Exercise 3 (25 points)

1. Write the FIRST and FOLLOW sets for all the non-terminals of the LL(1) grammar produced in **Exercise 1**
2. Develop the parsing table for the LL(1) grammar produced in **Exercise 1**



That's it, good luck!