Group Project Part 2 (Non-Recursive predictive parser with Error Recovery):

Assigned on: Monday march 25, 2024

Due Date: Thursday April 25, 2024 till 11:59 Mid Night

Consider the following CFG which generates all infix expressions:

E ⟶ E + T | T
T ⟶ T * F | F
F ⟶ (E) | id

(i) Remove left recursion from the above grammar. (Manually)
(ii) Find FIRST and FOLLOW sets for all non-terminals of the resultant grammar (Manually)
(iii) Construct Non-recursive predictive parse table with "Synch" entries (programmatically using the algorithm given at the end)
(iv) Finally Write a complete Java program (Using the algorithm given in Lab notes as well as shown below) show the actions of non-recursive predictive parser on the inputs given in the input file "input.txt".

Algorithm:
Construction of non-recursive predictive parse table
Input: Grammar G
Output: Parse table M [X, a], where X will be the non-terminals of the grammar against which their corresponding productions will be written under the columns 'a' which will be the terminals of the grammar or $.
Method:
Step 1: For each production A ⟶ α of the grammar, do steps 2 and 3.
Step 2: For each terminal 'a' in FIRST(α), add the production A ⟶ α to M[A, a].
Step 3: If ϵ is in FIRST(α), add the production A ⟶ α to M[A, b] for each terminal 'b' in FOLLOW(A). Moreover, if ϵ is in FIRST(α) and $ is in FOLLOW(A), add the production A ⟶ α to M[A, $].
Step 4: Write "Synch" in the FOLLOWs of every non-terminal (if it is not already filled).
Step 5: Make each undefined entry of M be an error.

Non-Recursive Predictive Parser:

Algorithm: Constructing non-recursive predictive parser
Input: A string 'W' and a parse table 'M' for grammar 'G'.
Output: If 'W' is in L(G) (i.e. W is a string of language L defined by grammar G), then left most derivation of the string 'W', otherwise an error indication.
Method:

- Initially the parser is in a configuration in which it has $S (where S is the start symbol) on the stack and W$ in the input buffer.
- Set 'ip' (input pointer) to point to the first symbol of W$.
- Repeat

```
Let 'X' be the top stack symbol and 'a' the symbol pointed to by 'ip';
if X is a terminal or $ then
        if X= = a then
                pop 'a' from stack;
                advance 'ip' to point to next input symbol;
        else
                error();
else if M[X, a] = X ⟶ Y1Y2 . . . . . Yk then
        pop X from stack;
        push YkYk-1 . . . . . Y2Y1; //Push the production in reverse order
        Output the production X ⟶ Y1Y2 . . . . . Yk ;
else
        error();
until X = = $
```

Note:

If the parser looks up the entry M[A, a] and finds that

1. It is blank, then the input symbol 'a' is skipped and pointer is advanced.
2. It is "Synch", then the non-terminal on top of the stack is popped in an attempt to resume parsing (except for the first time when start symbol is on top of the stack and there are some tokens on the input tape) and do not advance the pointer.
3. If a token on top of the stack does not match the input symbol, then pop the token from the stack and do not advance the input pointer.
4. If there is a non-terminal A on top of the stack and the pointer points to some terminal symbol that is not present in the language generated by the grammar, then skip the input token and do not pop the non-terminal.

Implementation Restrictions:

(i) After removing left recursion manually, **write the resultant grammar as comments** in the main class.

(ii) Write FIRST and FOLLOW sets of all non-terminals **as comments** in the main class.

(iii) You should use Hash Table or Two dimensional array to store the parse table.

(iv) You can **use built-in Stack** for your program.

(v) You should **read all arithmetic expressions** from the **file "input.txt"** and generate the **outputs for all these expressions on screen**.

(vi) As soon as parser finds some syntax error, it should display the message "Syntax Error" and resume the parsing for the remaining input.

(vii) Only **one member** of the group should upload the solution on Blackboard.

If the input file (input.txt) contains the following arithmetic expressions (tokens are separated with spaces):

(id + id) * id $

) id * + id $

the output displayed on screen should be as follows:

Left most derivation for the arithmetic expression (id + id) * id $**:**

STACK	INPUT	OUTPUT
$E	(id+id)*id$	E →TE'
$E'T	(id+id)*id$	T → FT'
$E'T'F	(id+id)*id$	F → (E)
$E'T')E (	(id+id)*id$	
$E'T')E	id+id)*id$	E →TE'
$E'T')E'T	id+id)*id$	T → FT'
$E'T')E'T'F	id+id)*id$	F → id
$E'T')E'T'id	id+id)*id$	
$E'T')E'T'	+id)*id$	T' → ε
$E'T')E'	+id)*id$	E' → +TE'
$E'T')E'T+	+id)*id$	
$E'T')E'T	id)*id$	T → FT'
$E'T')E'T'F	id)*id$	F → id
$E'T')E'T'id	id)*id$	
$E'T')E'T'	)*id$	T' → ε
$E'T')E'	)*id$	E' → ε
$E'T')	)*id$	
$E'T'	*id$	T' → *FT'
$E'T'F*	*id$	
$E'T'F	id$	F → id
$E'T'id	id$	
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	parsing successfully halts

Left most derivation for the arithmetic expression) id * + id $:

Stack	Input	Output
$E	)id * + id$	Error, Skip)
$E	id * + id$	E ⟶ T E'
$E'T	id * + id$	T ⟶ F T'
$E'T'F	id * + id$	F ⟶ id
$E'T'id	id * + id$	
$E'T'	* + id$	T' ⟶ *F T'
$E'T'F*	* + id$	
$E'T'F	+ id$	Error, Pop F
$E'T'	+ id$	T' ⟶ ϵ
$E'	+ id$	E' ⟶ + T E'
$E'T+	+ id$	
$E'T	id$	T ⟶ FT'
$E'T'F	id$	F ⟶ id
$E'T'id	id$	
$E'T'	$	T' ⟶ ϵ
$E'	$	E' ⟶ ϵ
$	$	**Parsing successfully halted**

17 Lines