Programming Exercise 03

Non-Recursive Predictive Parsing

OBJECTIVE

To be able to implement a non-recursive predictive parser that checks if a given sequence of tokens is valid based on the given syntactic rules.

INSTRUCTIONS

1. You are to work on this activity by group.
2. You are to write a code for a program that follows the details in the PROGRAM SPECIFICATION section. There should only be one program for this PE.
3. You are to develop your program following the structured programming approach at the very least.
4. All your program files (source code files, etc.) should be together with your main program file (the one that contains the main function). There should be no need to put certain files in certain folders just so your program will compile and run.
5. Follow the details specified in the SUBMISSION REQUIREMENT section.

PROGRAM SPECIFICATION

The program is a simple non-recursive predictive parser that uses the information from a parse table and the corresponding productions to check if a given string (sequence of tokens) is valid or not. The details of the program are specified in the succeeding texts.

PROGRAM DETAILS

1. The program must read a production file (*.prod* extension) containing the list of productions for the grammar.
    - The first column is the line number of a particular production.
    - The second column is the list of all the non-terminal symbols.
    - The third column is the production for the corresponding non-terminal symbol.
    - Instead of using the OR symbol "|" for non-terminals having two or more productions, the productions are now placed on separate lines.
    - Each symbol in the production is separated by at least one horizontal space. This will make the process of reading the file easier for you.
    - The empty symbol is represented by the symbol e.

2. It must also read a parse table file (*.ptbl* extension) containing the parse table content for the grammar.
    - The first column is the list of all the non-terminal symbols.
    - The second column and each succeeding columns are labeled individually with all the terminal symbols plus the input right end marker.

- The content of each cell will be either a number that corresponds to the production number from the productions file or a blank.

3. The program must check if the string of tokens specified in the input field is valid under the grammar represented by the input parse table and the productions list. The tokens in the input field are separated by some whitespace (trailing, in between; one or more horizontal/vertical spaces).

4. The series of steps while checking the input must be placed in an output file (***.prsd*** extension).
   - The output filename (*outname* in the format) should be asked from the user and then appended with the productions filename (*prodname* in the format): ***outname_prodname***
   - The first row of the solution table shows the initial state of the parsing process. The succeeding rows correspond to the steps in the parsing process.
   - The first column shows the content of the stack at every step in the process.
   - The second column shows the content of the input buffer at every step in the process.
   - The third column shows the action performed for the process.

5. After checking the input, the result should be **VALID** if the input is indeed correct under the grammar, and should be **INVALID** otherwise.

6. The productions, parse table, and output files content will be in CSV format.

PROGRAM FLOW

Once program is running, the main UI should show up immediately. The program should allow the following functions:

- Checking of validity of input can be done only when a valid file has been loaded for the parse table, a valid file has been loaded for the productions, and an input has been specified in the input field. The parse table and the productions files should be of the same name.
- Whenever a new file is to be loaded (successful/ unsuccessful) in the program, the Parsing solution display should be set to empty.
- Whenever an unsuccessful attempt (cancelled or invalid file encountered) is made in loading a file, the previously successfully loaded file should still be valid for use. This is for the files with the same extension.
- For the displays, the content of the successfully loaded file should replace the previously loaded content of the file with the same extension.
- Any time during program execution, the program user should be able to load files and process.
- The output file is to be saved to the same directory as the input file.
- Loading of file from any accessible directory.

SAMPLE INPUT

The file inputs to the program are the parse table (*.ptbl* extension) and the productions (*.prod* extension). To show what the expected content of each input file is, the following examples are provided:

rules.ptbl

```
,id,+,*,(,),$
E,1,,,1,,
E’,,2,,,3,3
T,4,,,4,,
T’,,6,5,,6,6
F,8,,,7,,
```

rules.prod

```
1,E,T E’
2,E’,+ T E’
3,E’,e
4,T,F T'
5,T’,* F T’
6,T’,e
7,F,(E)
8,F,id
```

In these examples, the file *rules.ptbl* is for and contains the parse table while the file *rules.prod* is for and contains the productions.

Note: The parse table is generated from the productions. Each file is to follow the CSV format.

SAMPLE OUTPUT

The file output of the program contains the solution to parsing the given input sequence of tokens. To show what the output file looks like, the following example is provided:

test_rules.prsd

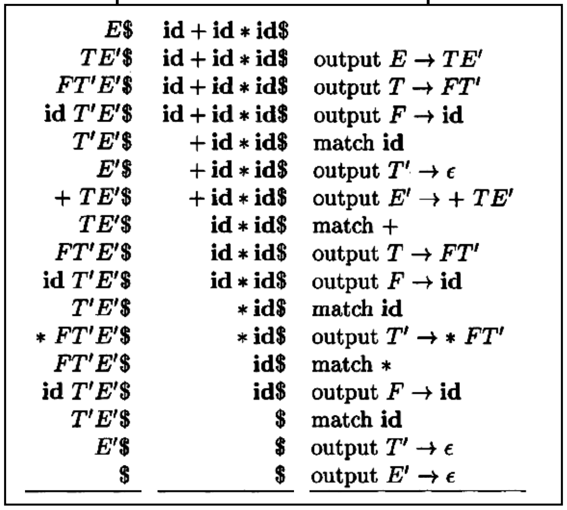
```
E $,id + id * id $
T E’ $,id + id * id $,Output E > T E’
F T’ E’ $,id + id * id $,Output T > F T’
id T’ E’ $,id + id * id $,Output F > id
T’ E’ $,+ id * id $,Match id
E’ $,+ id * id $,Output T’ > e
+ T E’ $,+ id * id $,Output E’ > + T E’
T E’ $,id * id $,Match +
F T’E’ $,id * id $,Output T > F T’
id T’ E’ $,id * id $,Output F > id
T’ E’ $,* id $,Match id
* F T’ E’ $,* id $,Output T’ > * F T’
F T’ E’ $,id $,Match *
id T’ E’ $,id $,Output F > id
T’ E’ $,$,Match id
E’ $,$,Output T’ > e
$,$,Output E’ > e
,,Match $
```

NOTE: If the result of parsing is VALID, the output file should contain the complete solution of the parsing. If the result of parsing is INVALID, the output file should contain the partial solution of the parsing to the point where parsing can no longer proceed. Then add another line stating an error (under the Action column).

The input sequence of tokens used for the example is: **id + id * id**

The content of the output file is to be written following the CSV format.

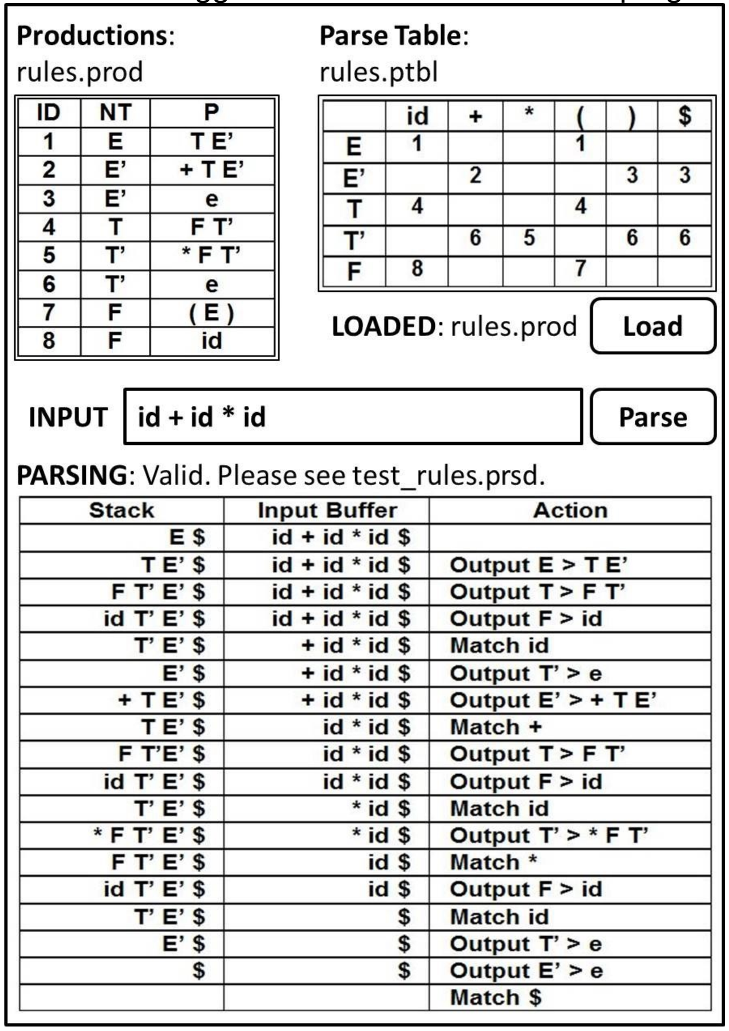
The output file content is interpreted as follows (example for *test_rules.prsd*):



$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$T'E'\$$	$+\mathbf{id} * \mathbf{id}\$$	match **id**
$E'\$$	$+\mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$+\ TE'\$$	$+\mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match +
$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$T'E'\$$	$* \mathbf{id}\$$	match **id**
$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$T'E'\$$	$\$$	match **id**
$E'\$$	$\$$	output $T' \to \epsilon$
$\$$	$\$$	output $E' \to \epsilon$

SUGGESTED INTERFACE

Here is the suggested user interface for the program:



Productions:
rules.prod

ID	NT	P
1	E	T E'
2	E'	+ T E'
3	E'	e
4	T	F T'
5	T'	* F T'
6	T'	e
7	F	(E)
8	F	id

Parse Table:
rules.ptbl

	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

LOADED: rules.prod

Load

INPUT id + id * id

Parse

PARSING: Valid. Please see test_rules.prsd.

Stack	Input Buffer	Action
E $	id + id * id $	
T E' $	id + id * id $	Output E > T E'
F T' E' $	id + id * id $	Output T > F T'
id T' E' $	id + id * id $	Output F > id
T' E' $	+ id * id $	Match id
E' $	+ id * id $	Output T' > e
+ T E' $	+ id * id $	Output E' > + T E'
T E' $	id * id $	Match +
F T'E' $	id * id $	Output T > F T'
id T' E' $	id * id $	Output F > id
T' E' $	* id $	Match id
* F T' E' $	* id $	Output T' > * F T'
F T' E' $	id $	Match *
id T' E' $	id $	Output F > id
T' E' $	$	Match id
E' $	$	Output T' > e
$	$	Output E' > e
		Match $

In the interface:

- The area where the text “rules.prod” is placed is a message area that displays the name of the productions file loaded. The area where “rules.ptbl” is placed is a message area that displays the name of the parse table file loaded.
- The area where the text “LOADED: rules.prod” is placed is a message area that displays the input file most recently loaded.
- The textbox on the right of the text “INPUT” is where the string (sequence of tokens) input is to be provided by the user of the program.
- The part that contains the text “Load” is the load button. There is only one load button. This button is to be used to load the input files one at a time.
- The part that contains the text “Parse” is the button that will trigger the parsing process. The button should only be enabled when the files for the parse table and the productions have been loaded.
- The area where the text “PARSING: Valid. Please see test_rules.prsd.” is placed is a message area that displays the result of the parsing and the name of the output file where the solution is saved.
- The table below the text “PARSING: Valid. Please see test_rules.prsd.” is for displaying the solution of the parsing process.

SUBMISSION REQUIREMENT

Your submission should contain the following archived in a zip file:

- Documentation file: includes description of your program, parts/modules/functions of the program and their corresponding description, control flow of the whole program, work distribution to members.
- Program source code files
- Program executable file (e.g., JAR file for Java programs)

Name the zip file using your surnames (alphabetical order, separated by underscore) followed by the PE # similar to this example: Bonifacio_Rizal_PE03.