University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2020

Exercise 2: Parsing

(Deadline for uploading solutions: May 22, 2020, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the folder structure and the templates that must be used for the submission**.

The folder structure is shown in Figure 1.

```
exercise2
├─ task1.csv
├─ task2
│  ├─ Grammar_A.csv
│  ├─ Grammar_B.csv
│  ├─ Grammar_C.csv
│  └─ Grammar_Example.csv
└─ task3.zip
```

Figure 1: Folder tree provided.

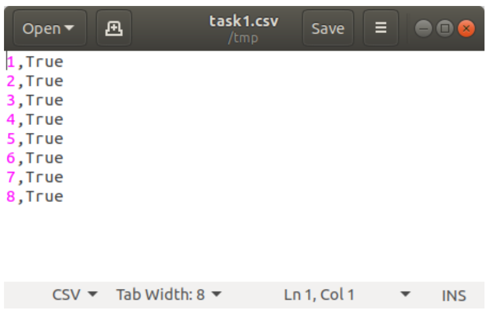
The submission must be compressed in a zip file using the given folder structure. The name of folders and files must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (**not rar**, 7z, etc.) ;
- Your zip file should contain exactly one top-level directory *"exercise2/"*, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions;
- Before submitting, compile and run the Java code of Task 3.

Notes about the symbols used in this exercise:

- `blue font` is for non-terminals;
- `black font` is for terminals;
- $*$ (green colour) is the Kleene star symbol (arbitrary number of repetitions, including zero repetitions);
- $*$ (black colour) is a terminal symbol (e.g., multiplication symbol);
- $|$ means "or";
- ε is the empty word.



```
1,True
2,True
3,True
4,True
5,True
6,True
7,True
8,True
```

Figure 2: Example of the file containing the answer of Task I.

1 Task I (10% of total points of the exercise)

Which of the following statements are correct? Your answer, either **True** or **False**, must be written into the file *exercise2/task1.csv* one string per line as described in Figure 2. The statements are listed below

1. Top-down parsers are starting from the root of the to-be-constructed parse tree.
2. Parsers work only with non-recursive grammars.
3. Parsers transform a token sequence into a parse tree.
4. In a recursive descent parser, the match function will consume arbitrary tokens without raising an error.
5. FOLLOW sets can include EOF for "End of file".
6. The FIRST set of a non-terminal is always a subset of the FOLLOW set of the non-terminal.
7. FIRST sets may contain ϵ (the empty word).
8. FOLLOW sets never contain ϵ (the empty word).

2 Task II (30% of total points of the exercise)

You are given three context-free grammars. For each grammar, compute the **FIRST** and **FOLLOW** sets.

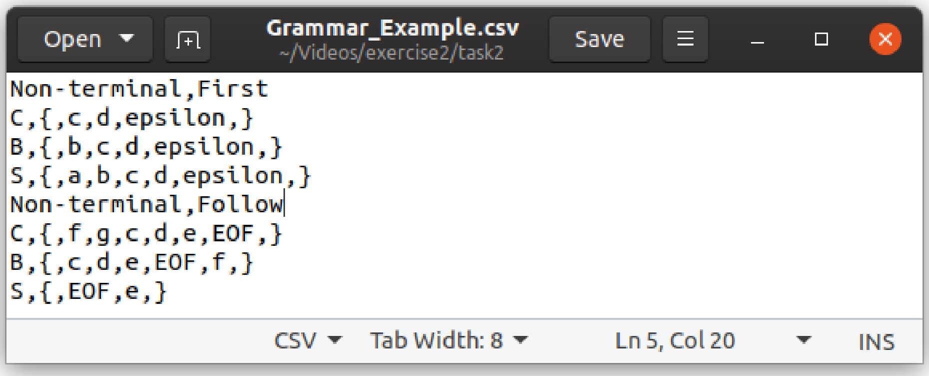
Write the following into the files *exercise2/task2/Grammar_A.csv* as shown in Figure 3.
Specify the Non-terminal and Set type before each set solution in the file. In-order to specify ϵ in the csv file, kindly use **epsilon** keyword and **EOF** for end-of-file.

Note: The grammars are case sensitive. Finally, each set should begin with '{' and end with '}' symbol. The order of the non-terminal and the order of terminals in the sets do not matter.

2.1 Example Grammar

This is only an example to show the expected output.

$$
\begin{aligned}
S &\longrightarrow a\ S\ e \mid B \\
B &\longrightarrow b\ B\ C\ f \mid C \\
C &\longrightarrow c\ C\ g \mid d \mid \epsilon
\end{aligned}
$$



```
Non-terminal,First
C,{,c,d,epsilon,}
B,{,b,c,d,epsilon,}
S,{,a,b,c,d,epsilon,}
Non-terminal,Follow
C,{,f,g,c,d,e,EOF,}
B,{,c,d,e,EOF,f,}
S,{,EOF,e,}
```

Figure 3: Example of the file containing the answer of task 2.

2.2 Grammar A

```
start ⟶ menu
menu ⟶ pizza $ price | pizza $ price ; menu
pizza ⟶ margherita | diavola | napoletana
price ⟶ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.3 Grammar B

```
start ⟶ ACB | CbD | DA
A ⟶ aC | BC
B ⟶ DgC | Af | ϵ
C ⟶ gC | DhD | ϵ
D ⟶ i | ϵ
```

2.4 Grammar C

In this case ”`input_end`” is a terminal token, so it appears in the FIRST and FOLLOW sets. As indicated in the example in Figure 3, use EOF for end-of-file.

```
start ⟶ stmt_list input_end
stmt_list ⟶ stmt stmt_list | ϵ
stmt ⟶ id := expr | read expr | write expr
expr ⟶ term term_tail
term_tail ⟶ add_op term term_tail | ϵ
term ⟶ factor factor_tail
factor_tail ⟶ mult_op factor factor_tail | ϵ
factor ⟶ (expr) | id | digit
add_op ⟶ + | −
mult_op ⟶ ∗ | /
```

3 Task III (60% of total points of the exercise)

The aim of this task is to implement a **Recursive Descent Parser** in Java that parses a string in the language into a parse tree, or reports an error if the input string is invalid. You must implement the parser by hand, i.e., do not use a parser generator tool.

For the task you must use the grammar C in Task 2.4. The end-marker **"input_end"** signifies the end of the input.

The input to the parser is a $List < String >$ containing a sequence of tokens. The output is a valid parse tree with all the tokens found by the a parser. The tree must be represented using instances of class $Node$ (see the code provided in *exercise2/task3.zip*) . To add a child node to the tree, use the $public\ void\ addChild(Node\ child)$ method.

In case of invalid tokens, the output must be a single $Node$ object with zero children and a label "invalid". For grading, we evaluate the shape of the tree and the *label* fields associated with each node, all remaining fields are provided for your convenience.

Note that "id", ":=", "(", ")", "read", "write" and "digit" are terminals, written in lower case, so if in the input there is a different token, the input is invalid.

Some examples:

- Input: "write", "a", "+", "9", "input_end" $\longrightarrow$ INVALID, because "a" and "9" are not valid tokens.
- Input: "write", "id", "+", "digit", "input_end" $\longrightarrow$ VALID.
- Input : "id", ":=", ":=", "digit", "input_end" $\longrightarrow$ INVALID, because it doesn't respect the grammar (double ":=").

ϵ is a terminal and it must to be represented with a node with the label "epsilon" (without quotes).

Note that "id" and ":=" are two separate terminals in the parse tree. Figure 4 and Figure 5 show two examples of the parse tree of valid inputs.

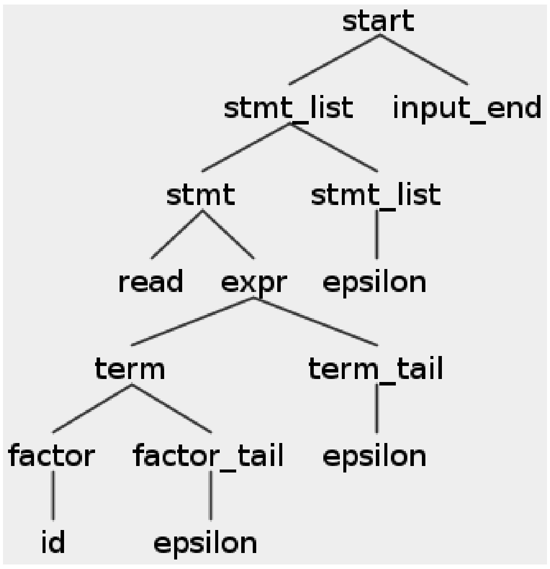


Figure 4: Input: "read", "id", "input_end" .

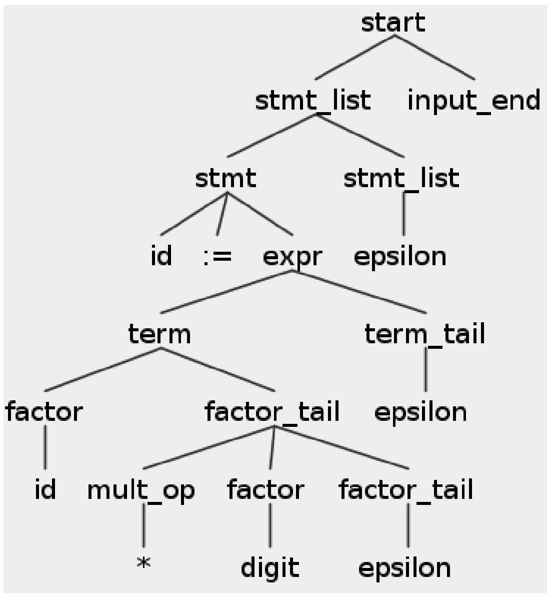


Figure 5: Input: "id", ":=", "id", "*", "digit", "input_end"

An Eclipse project for the parser implementation is provided: *exercise2/task3.zip*. You can import the project template (.zip) into Eclipse as follows: *File-Import-General-Existing project into Workspace-Select archive file-Finish*.

Please implement your parser in the method $public\ static\ Node\ functionParser(List < String >\ input)$, which you find in file task3/src/task3/TokenParser.java.

You find some JUnit tests in folder *task3/src/task3*. Use them to check whether your parser works. We will use additional tests to evaluate your solution, and you are advised to also add additional tests for your own testing.

For the submission, your project must be exported into a .zip **(NOT .rar)** archive using Eclipse: *File-Export-General-Archive File*, and then added to the tree structure in Figure 1.