Universität Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart, Winter 2019/2020

Exercise 2: Parsing

(Deadline for uploading solutions: Nov 10, 2019, 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.

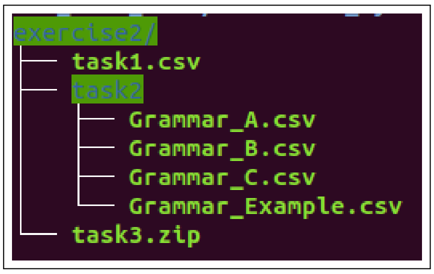

Figure 1: Folder structure to be used for the solution.

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.

There are three tasks, which contribute a specific percentage to the overall points for this exercise.
Notes about the symbols used in this exercise:

- *blue tokens* are **non-terminals**;
- *black tokens* are **terminals**;
- $*$ is the Kleene star symbol;
- | means "or";

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. Bottom-up parsers must predict which grammar rule to apply next.
2. Parsers transform raw code into token sequences.
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. To compute FOLLOW sets, one needs to compute FIRST sets first.
6. The FIRST set of a non-terminal is always a subset of the FOLLOW set of the terminal.
7. FIRST sets may contain ϵ (the empty word).
8. FOLLOW sets may contain ϵ (the empty word).

Figure 2: Example of the file containing the three valid strings.

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 [Header] 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 file is case sensitive, i.e., terminals should be represented in lower-case, while non-terminals and end-of-file in upper-case. Finally, each set should begin with '{' and end with '}' symbol.

2.1 Example Grammar

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

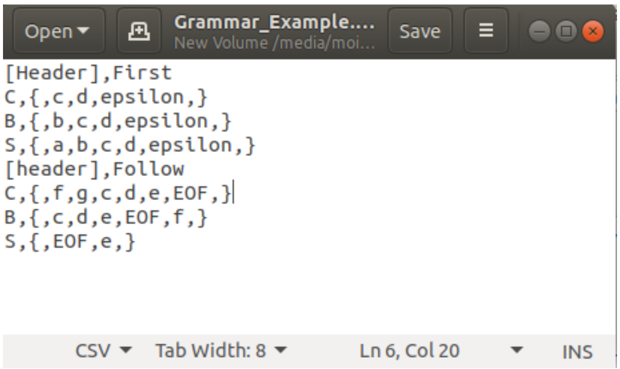

Figure 3: Example of the file containing the answer of task 2.

2.2 Grammar A

$$
\begin{aligned}
&start \longrightarrow wishlist \\
&wishlist \longrightarrow product\ \$\ price \mid product\ \$\ price\ ;\ wishlist \\
&product \longrightarrow car \mid computer \mid smartphone \\
&price \longrightarrow non\_zero\_digit\ digit* \\
&non\_zero\_digit \longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
&digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

2.3 Grammar B

$$
\begin{aligned}
Start &\longrightarrow ACB \mid CbD \mid DA \\
A &\longrightarrow daC \mid BC \\
B &\longrightarrow DgC \mid Af \mid \epsilon \\
C &\longrightarrow gC \mid DhD \mid \epsilon
\end{aligned}
$$

$D \longrightarrow i \mid \epsilon$

2.4 Grammar C

$start \longrightarrow stmt\_list$ $$

$stmt\_list \longrightarrow stmt\ stmt\_list \mid \epsilon$

$stmt \longrightarrow id := expr \mid read\ id \mid write\ expr$

$expr \longrightarrow term\ term\_tail$

$term\_tail \longrightarrow add\_op\ term\ term\_tail \mid \epsilon$

$term \longrightarrow factor\ factor\_tail$

$factor\_tail \longrightarrow mult\_op\ factor\ factor\_tail \mid \epsilon$

$factor \longrightarrow (expr) \mid id \mid literal$

$add\_op \longrightarrow + \mid -$

$mult\_op \longrightarrow \times \mid \div$

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

The aim of this exercise 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 illegal. You must implement the parser by hand, i.e., do not use a parser generator tool.

For the task you must use the grammar for a calculator provided in Task 2.4. The program allows values to be read into numeric variables, which can be used in expressions. The end-marker "**$$**" signifies the end of the input.

An Eclipse project for the parser implementation is provided: *exercise2/task3*. 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 exercise2/task3/src/parser/TokenParser.java.

The input to the parser is a $List < String >$ containing a sequence of tokens, as returned by a scanner. 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 $Node$. In-order to add a child node to the tree use $public\ void\ add\_children(Node\ child)$ method. In case of illegal tokens, the output must be a single $Node$ object with zero children and a label "Illegal".

You find some JUnit tests in folder *exercise2/task3/src/parser/*. 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 archive using Eclipse: *File-Export-General-Archive File*, and then added to the tree structure in Figure 1.