University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2021

Exercise 2: Parsing

(Deadline for uploading solutions: May 14, 2021, 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
 │   └─ Task1.csv
 ├─ Task2
 │   └─ Task2.zip
 └─ Task3
     └─ Task3.zip
```

Figure 1: Folder tree provided. Keep the exact same structure when submitting your solution.

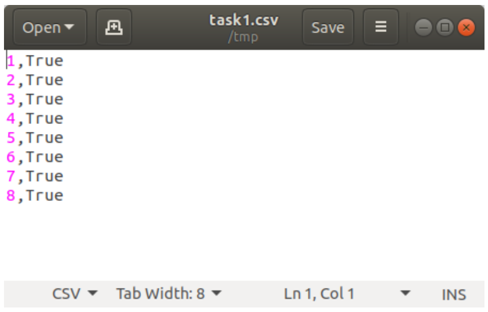
The submission must be compressed in a zip file using the given folder structure. The name of folders and files must not be changed or moved, otherwise the homework will not be evaluated.

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

- Use only the zip format for the archive (other compression formats will make your submission invalid: e.g. not rar, 7z, etc.;
- Your zip file should contain exactly one top-level directory *"Exercise2/"*, as in the zip file provided by us.
- Keep the name of all files or folders unchanged, simply open the files provided and put your solutions;
- Before submitting, compile and run the Java code of Task 2 and Task 3.

Notes about the symbols used in this exercise:

- EOF represents the end of file;
- a^n means the character or group of characters a should be repeated n times;
- | 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/Task1.csv* one answer per line as described in Figure 2.

1. The main output artifact of a parser is the parse tree.
2. FIRST sets cannot contain ϵ (the empty word).
3. A top-down parser always applies the one and only rule that matches the next non-terminal.
4. Bottom-up parsers start from the root of the to-be-constructed parse tree.
5. In a recursive descent parser, each function implements the derivation rules that have a specific non-terminal on the left-hand side.
6. In LL(k) parsers, k stands for the number of tokens the method can inspect ahead of the current token.
7. The FOLLOW set of the start symbol always contains EOF for "End of file".
8. The action table is used by LL parser.

2 Task II (60% of total points of the exercise)

For this task you have to implement an algorithm that, given an input grammar, computes the FIRST and FOLLOW sets of that specific grammar for all terminals and non-terminals. Please refer to the definitions of these sets in the lecture "Syntax (Part 5)". As a further guideline, we recommend to follow the algorithm given in the "Programming Language Pragmatics", which is also available in Ilias. Note that, in addition to what the algorithm describes, the starting symbol should always have EOF in its FOLLOW set.

2.1 Starting Point

The Eclipse project for the algorithm implementation is provided: *Exercise2/Task2/Task2.zip*. You can import the project template (.zip) into Eclipse as follows: *File-Import-General-Existing project into Workspace-Select archive file-Finish*.

As a skeleton of the implementation, there are four Java classes: `Algorithm`, to implement your algorithm, `Ex2Task2Test`, for testing, and two auxiliary classes:

- `Rule`, which represents a single grammar rule that you receive as an input. You can access the left and right part or a rule as a string and a list of strings, respectively. For example, in the rule **A** $\rightarrow$ **B C**, the left part is a string `"A"` and the right part is a list of two strings `"B"` and `"C"`. Terminals and non-terminals will never start or end with a space, because we trim them.
- `Result`, which represents the output of your program. Each `Result` object represents a single symbol name, its FIRST set and its FOLLOW set.

Refer to the code itself to understand how to initialize these classes, and how to access and update their fields.

Your task is to implement the `computeFirstFollow` method in class `Algorithm`. The input is given as the string representing the starting symbol and the list of rules.

2.2 Testing

We provide some test cases for testing your implementation in file Task2/src/task2/Ex2Task2Test.java. Feel free to create additional tests to further validate your implementation. We strongly suggest to solve the test case by hand to get a deeper understanding of the algorithm before proceeding with the implementation.
To create a new tests remember to initialize all the terminals with a FIRST set that contains the character itself and an empty FOLLOW set, e.g., `new Result("a", "a", "")` where "a" is a terminal symbol. To write new test you also need new grammars. To create a grammar rule, you need a string like `"A -> b C d"`, where the `->` divides the left and the right side. Check out the given examples for more details.

2.3 Hints

The following points are a checklist to read before starting with the implementation:

1. All symbols are represented by a corresponding string. E.g., a non-terminal A is represented as `"A"`. The string `"EPSILON"` represents ϵ (the empty word). The string `"EOF"` represents the end of the input file.
2. Modify only Algorithm.java, and no other files of the project. We will extract only this file from your submission for evaluating the solution.
3. Our test code uses Java 8, and we recommend testing your solution with this version.

For the submission, your project must be exported into a **.zip archive** (a .rar will be considered invalid) using Eclipse: *File-Export-General-Archive File*, and then added to the tree structure in Figure 1.

3 Task III (30% of total points of the exercise)

Consider the following language:

$$L = \{a^n b^{3n} | n \geq 1\} \cup \{a^n (bbbc)^n | n \geq 1\}$$

This language is composed of all string belonging to two sublanguages, i.e., the language contains the strings of both sublanguages. For example, the language includes all of the following strings: abbb, aabbbbbb, aaabbbbbbbbb, etc. (coming from the first sublanguage); it also includes all of the following strings: abbbc aabbbcbbbc, aaabbbcbbbcbbbc, etc. (coming from the second sublanguage).

Do you remember ANTLR? We used ANTLR in Exercise 1 to create a scanner (also called lexer). In this exercise, you will use ANTLR to specify the grammar of the above language and to generate a parser for the language. More specifically, your task is to:

1. Write the grammar, composed of production rules, from which one can derive all and only the strings of the language L. Assume that the starting symbol is `start`.

```
start ⟶ ...
... ⟶ ...
... ⟶ ...
```

2. Translate the rules into the ANTLR format and add them to the file *Task3/src/Task3.g4*. Add them at the end of the file, making sure to use the tokens defined in the lexer part of the file: `A`, `B`, `C`. Consult the ANTLR documentation to learn how to specify production rules. Another good way to learn about ANTLR is to explore existing examples of .g4 files for real programming languages.
3. Compile the grammar into a parser implemented in Java by running the following command from directory Task3:
`java -jar antlr-4.9.2-complete.jar src/Task3.g4`.
4. Validate the generated parser by running the test cases in the *Task3/src/TestCases.java*. We encourage you to add additional test cases for further testing.

For the submission, your Eclipse project must be exported into a **.zip archive** (a .rar will be considered invalid) using Eclipse: *File-Export-General-Archive File*, and then added to the tree structure in Figure 1. The `Task3.g4` file and the generated `Task3Parser.java` will used for grading.