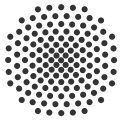

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2022

Exercise 1: Scanners and Parsers

(Deadline for uploading solutions: Apr 27, 2022, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this document);
- a zip file with **the templates file and folder structure you must use for the submission**.

```
exercise1.zip
├─ task_1
│  ├─ src/taskscanner
│  │  ├─ Ex1Task1Test.java
│  │  └─ TokenScanner.java (Substitute this file with your solution)
│  ├─ .classpath
│  └─ .project
└─ task_2
   ├─ src/taskfirstfollow
   │  ├─ Ex1Task2Test.java
   │  ├─ Algorithm.java (Substitute this file with your solution)
   │  ├─ Rule.java
   │  └─ Result.java
   ├─ .classpath
   └─ .project
```

Figure 1: Folder structure to use for submission.

The submission must be compressed in a zip file (not rar, 7z, gz, or anything else) using exactly the folder structure and names in Figure 1. To ensure that your submission has the expected structure, check it with the validator.jar tool provided on the course page, using the following command (where exercise1.zip is the original file provided to you, and your_exercise1.zip is your modified version of it):

java -jar validator.jar exercise1.zip your_exercise1.zip

Make sure that your IDE is configured to use Java 8, as described by the .project files in the template, or alternatively, make sure that your code can be compiled and executed with Java 8.

Note: Submissions that do not comply with the exact folder structure and names in Figure 1, or that do not work with Java 8, cannot be graded.

1 Task 1 (40% of total points of the exercise)

For this task you have to implement a scanner in Java that transforms a string in the given language into a sequence of tokens, or reports an error if the string is invalid. You must implement the scanner by hand, i.e., do not use a scanner generator tool.
The legal tokes of the target toy language (inspired by HTML) are expressed as regular expressions:

```
HTML ⟶ <html> | </html>
SPAN ⟶ <span> | </span>
OPENDIV ⟶ <div
CLOSEDIV_SHORT ⟶ >
CLOSEDIV ⟶ </div>
CLASS ⟶ class
EQUAL ⟶ =
IDENTIFIER ⟶ letter+     (+ means one or more letters, but at least one.)
STRING ⟶ 'letter+'       (+ means one or more letters, but at least one.)
```

Figure 2: Tokens of a toy HTML-like language.

Note that `letter` means a lowercase or uppercase letter (English alphabet only, for example, ä, ê, ì etc. are invalid characters.

New lines *(\n, \r and \t)* and spaces in the input string are allowed between tokens and should be skipped during scanning. That is, these characters do not cause an error and they should not appear in the output token sequence. If two tokens are unambiguously concatenated without any whitespace in between, e.g., two equal signs "==", then they should be identified by the scanner as two separate tokens.

Note that a scanner does not check whether an input string parses into a valid parse tree or abstract syntax tree. All it does is to split the input string into a sequence of valid tokens (or determine that this is impossible).

1.1 Starting Point

In the provided zip file you will find the template Java project to use for this task: *exercise1/task_1.zip*. The 'src' folder contains the only source files you must use for this task, together with '.project' and '.classpath' files to help you configure the project with the correct Java version used also for evaluation: Java 8. Your solution must be entirely implemented following this template, using a single source file (`TokenScanner.java`) and no other files. You are free to use the IDE of your choice; be sure to configure the correct Java version used for evaluation.

Examples of correct input-output pairs for the scanner:

Input	Output
`"<div classIC==>"`	`"<div", "classIC", "=", "=", ">"`
`"<div class='study'>PASS>"`	`"<div", "class", "=", "'study'", ">", "PASS", ">"`
`"<html> Happy Programming Paradigms </html>"`	`"<html>", "Happy","Programming", "Paradigms", "</html>"`
`"<html> Fail/div> </html>"`	`null`
`"<html> mail@UniStuttgart </html>"`	`null`

The input string in the first row of Table 1 contains `classIC` because it is a single token, as expressed by the regular expression of IDENTIFIER. Remember to return always the longest possible matching token, e.g., in this case you return `classIC` and not `class` because the first one is longer. If the input would contain `class IC` with a space instead, the scanner would return two tokens `"class"` and `"IC"`. In the second row you see that it is not a "correct" HTML document because of the missing closing tag. However, since the scanner does not check grammatical correctness beyond identifying tokens, it produces a valid token sequence. The fourth example is invalid because it contains an invalid token: `Fail/div`. The last example is illegal because it contains an invalid character: `@`.

Your task is to implement the `scanner` method in class `TokenScanner.java`. The input to the scanner is a `String` containing a snippet of code in our toy language. The output is a `List<String>` with all the tokens found by the scanner. In case of invalid tokens, your implementation must return `null`.

1.2 Testing

You find some JUnit tests in file `Ex1Task1Test.java`. Use them to check whether your scanner works as expected. We will use additional tests to evaluate your solution, and you are advised to also add further tests for your own testing.

1.3 Evaluation Criteria

Your code will be evaluated against a held-out set of test cases that exercise your code. During the evaluation, we will use Java 8.

2 Task 2 (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" book, 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

In the provided zip file you will find the template Java project to use for this task: *exercise1/task_2.zip*.
The 'src' folder contains the only source files you must use for this task, together with '.project' and '.classpath' files to help you configure the project with the correct Java version used also for evaluation: Java 8. Your solution must be entirely implemented following this template, using a single source file (`Algorithm.java`) and no other files. You are free to use the IDE of your choice; be sure to configure the correct Java version used for evaluation.

As a skeleton of the implementation, there are four Java classes: `Algorithm`, to implement your algorithm, `Ex1Task2Test.java`, 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 `Ex1Task2Test.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 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 tests 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.

2.4 Evaluation Criteria

Your code will be evaluated against a held-out set of test cases that exercise your code. During the evaluation, we will use Java 8.