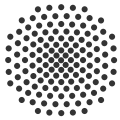

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2021

Exercise 1: Regular Expressions, Grammars and Scanners

(Deadline for uploading solutions: May 7, 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.

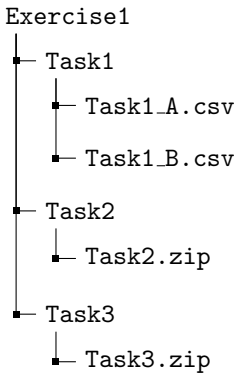

Figure 1: Directory structure in the provided .zip file and in your uploaded solution file.

The submission must be compressed in a zip file using the given folder structure. The names 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, gz, or anything else).
- Your zip file should contain exactly one top-level directory *"Exercise1/"*, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions.

Notes about the symbols used in this exercise:

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

1 Task I (20% of total points of the exercise)

This task is about regular expressions. The goal to determine if the given strings are part of the language described by the given regular grammar.

1.1 Regular Expression A

```
start ⟶ human | robot
human ⟶ traditional_id | electronic_id
robot ⟶ serial_id
traditional_id ⟶ letter digit digit digit digit digit
electronic_id ⟶ letter digit digit digit letter
serial_id ⟶ location year letter digit digit
location ⟶ LON | STR | MIL
year ⟶ yy digit digit | digit digit digit digit
digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter ⟶ a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
```

Figure 2: Regular expression A for Task I.

Figure 2 shows the grammar rules for a regular expression. Which of the following strings is accepted by the given regular expression? Provide your answer in file *Exercise1/Task1/Task1_A.csv* by replacing the `True or False` with `True` or `False`. You must update all eight lines, one for each of the following strings:

1. h244856
2. z184e
3. w29382
4. LONyy35E99
5. ah342b
6. MIL657r74
7. k67223
8. PAR24m72

1.2 Regular Expression B

```
start ⟶ wishlist
wishlist ⟶ product $$* price
product ⟶ car | computer | smartphone
price ⟶ non_zero_digit digit*
non_zero_digit ⟶ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 3: Regular expression B for task 1.

Figure 3 shows another grammar describing a regular expression. The task is analogous to the above. Provide your solution in file *Exercise1/Task1/Task1_B.csv* for each of the following strings:

1. car$0544
2. computer$999
3. smartphone$499.9

4. smartphone$$$$$499
5. tv$450
6. car$1
7. $999computer
8. smartphone$$$$$4992483238573298723

2 Task II (40% of total points of the exercise)

You are given several regular expressions that each describe the legal tokens of a toy programming language (inspired by SQL):

```
SELECT_ ⟶ SELECT | select
DELETE_ ⟶ DELETE | delete
UPDATE_ ⟶ UPDATE | update
WHERE_ ⟶ WHERE | where
FROM_ ⟶ FROM | from
SCOL ⟶ ;
COMMA ⟶ ,
STAR ⟶ *
IDENTIFIER ⟶ letter letter*    (except for keywords)
NUMBER ⟶ digit*
```

Figure 4: Tokens of a toy SQL-like language.

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

The goal of this exercise is to implement a scanner in Java that transforms a string in the 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.

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 commas ",,", 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).

An Eclipse project for the scanner implementation is provided: *Exercise1/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*. Alternatively, using any other IDE or editor is fine, as long as you preserve the structure of the zip package.

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`.

Examples of correct input-output pairs for the scanner:

Input	Output
`"SELECT * FROM SELECTion;"`	`"SELECT", "*", "FROM", "SELECTion", ";"`
`"from SELECT where UPDATE"`	`"from", "SELECT", "where", "UPDATE"`
`"ADD v@r;"`	`null`

The input string in the first row of Table 1 contains `SELECTion` because it is a single token. If the input would contain `SELECT ion` instead, the scanner would return two tokens `"SELECT"` and `"ion"`. In the second row you see that it is not a ”correct” SQL query. However, since the scanner does not check grammatical correctness beyond identifying tokens, it produces a valid token sequence. The third example is illegal because it contains an invalid character.

Implement your scanner in the method `scanner` of class `TokenScanner`, which you find in file TokenScanner.java. Do not create new Java files.

You find some JUnit tests in file Exercise1Task2Test.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 additional tests for your own testing.

For the submission, your Eclipse 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. Alternatively, you can update the given Task2.zip file with your updated Java files.

3 Task III (40% of total points of the exercise)

In this task we use a tool called ANTLR4[1]. ANTLR4 (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a grammar, ANTLR4 generates a scanner and a parser. Parsers will be the topic of Exercise 2, and we here focus on a scanner generated with ANTLR4. Note that the terminology used by ANTLR4 is to say *lexer*, instead of *scanner*, but the meaning is the same.

As for Task 2, start by importing the Eclipse project given in *Exercise1/Task3/Task3.zip* into Eclipse or your favorite IDE. File *Task3/src/Task3.g4* shows an example of a grammar written using the ANTLR4 syntax. Your goal is to modify the grammar into the the grammar shown in Figure 4 using ANTLR4 syntax. Please read about the ANTLR syntax for lexer rules in their GitHub repository[2]. The main rules are:

- Every rule must end with a semicolon (*;*).
- Every terminal token is surrounded by single quotes, for example *'select'*.
- *or* is written with the symbol | , for example *SELECT* | *select*.
- ANTLR4 syntax is case sensitive. Keep the same cases show in Figure 4, e.g., *SELECT*_ all upper case and *select* all lower case.

To generate a lexer from the grammar, run the following command from directory Task3:
`java -jar antlr-4.9.2-complete.jar src/Task3.g4`
It will create several Java classes. For this exercise, we are interest on `Task3Lexer.java`, which is a generated scanner that recognizes the tokens described in the grammar.

To test if your grammar, and hence, your generated lexer/scanner are correct, execute the given JUnit tests. Initially, only the first test will pass. Your goal is to pass all tests. As for Task II, it is recommended that you add further tests for additional testing.

When editing the grammar file, your only modification should be to insert rules into the marked location. Do not change the other lines, and do not rename the file.

For the submission, your Eclipse 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. The submitted project should include your updated `Task3.g4` file and the generated `Task3Lexer.java`.

[1] https://www.antlr.org/
[2] https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md