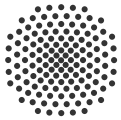

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2020

Exercise 1 Syntax: Regular Expressions, Grammars and Scanners

(Deadline for uploading solutions: May 8, 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:

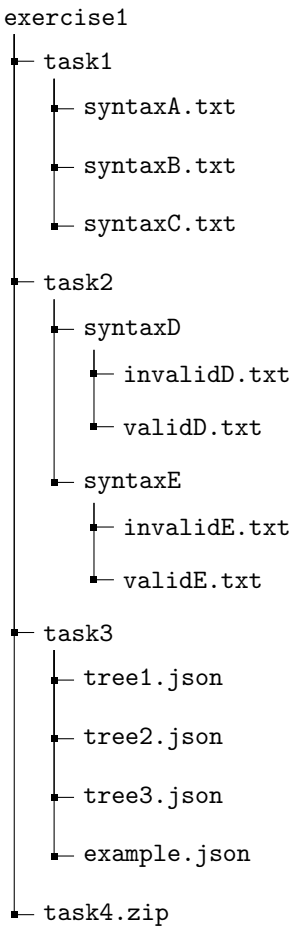

Figure 1: Folder tree provided.

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 *"exercise1/"*, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions;
- Inside the .txt files provided for Task 1 and Task 2 remove the word "SOMETHING" and put your answer;
- Before submitting, compile and run the Java code of Task 4.

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

In the following subtasks, we will present you different grammars, given as a list of production rules.

For each grammar you have to decide whether it describes a regular expression (i.e., is there a regular expression that recognizes the exact same language like the grammar presented) or if the grammar is context-free (i.e., there cannot be a regular expression that recognizes the same language).

The first rule of each grammar is the production rule of the `start` symbol.

1.1 Syntax A

```
start ⟶ statement
statement ⟶ block | stmt
block ⟶ begin list_of_statement end
list_of_statement ⟶ statement ; list_of_statement | ε
```

Write the following into the file *exercise1/task1/syntaxA.txt* as show in Figure 2: ***CONTEXT-FREE*** if the above is context-free or ***REGEX*** if it is a regular expression (in capital letters).

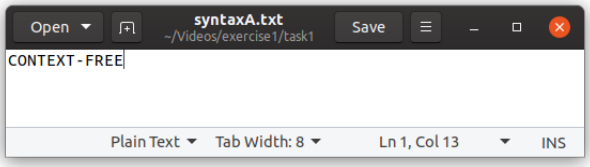

Figure 2: Example of the file containing the answer of Task 1.

1.2 Syntax B

```
start ⟶ menu
menu ⟶ pizza $ price | menu, pizza $ price
pizza ⟶ margherita | diavola | napoletana
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
```

Write the following into the file *exercise1/task1/syntaxB.txt* as show in Figure 2: ***CONTEXT-FREE*** if the above is context-free or ***REGEX*** if it is a regular expression (in capital letters).

1.3 Syntax C

```
start ⟶ sentence
sentence ⟶ The qualified_noun verb | pronoun verb
qualified_noun ⟶ adjective noun
noun ⟶ student | programmer | engineer
pronoun ⟶ he | she
verb ⟶ talks | studies | works
adjective ⟶ expert | smart | new
```

Write the following into the file *exercise1/task1/syntaxC.txt* as show in Figure 2: ***CONTEXT-FREE*** if the above is context-free or ***REGEX*** if it is a regular expression (in capital letters).

2 Task II (10% of total points of the exercise)

Two grammars are given. Your task is to write three valid strings and three invalid strings for each of them.

2.1 Syntax D

```
start ⟶ expression
expression ⟶ term | expression + term
term ⟶ factor | term * factor
factor ⟶ 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
```

where * is a terminal symbol and spaces are allowed, that is, you are allowed to insert spaces between terminals.

The three valid strings must be written into the file *exercise1/task2/**syntaxD/validD.txt***, one string per line. An example is shown in Figure 3.

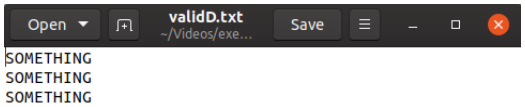

Figure 3: Example of the file containing the three valid strings.

The three invalid strings must be written into the file *exercise1/task2/**syntaxD/invalidD.txt***, one string per line.

2.2 Syntax E

```
start ⟶ pathfilename
pathfilename ⟶ drive : / pathname* filename fileextension
drive ⟶ upper
pathname ⟶ id /
filename ⟶ id
fileextension ⟶ .txt | .json | .csv
id ⟶ letter letter*
upper ⟶ 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
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
```

where * is the Kleene star symbol.
None of the rules contain a space terminal symbol, so spaces are not allowed in the strings of the language.

The three valid strings must be written into the file *exercise1/task2/**syntaxE/validE.txt***,
one string per line.

The three invalid strings must be written into the file *exercise1/task2/**syntaxE/invalidE.txt***,
one string per line.

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

In this task, you will do parsing by hand, as shown in the lecture. You will not have to implement a parser. Instead, you will build the parse tree yourself. For that, we give you a grammar (Syntax F) and three valid strings from the language. For each input string, you shall create a parse tree according to the grammar and encode it as a JSON [1] file.

3.1 Syntax F

```
start ⟶ command
command ⟶ convert | run | comparison | exit now
exit ⟶ shutdown | reboot
convert ⟶ convert argument filename
argument ⟶ RAWtoPNG | PNGtoJPEG | JPEGtoPNG
run ⟶ python filename
comparison ⟶ diff filename filename > filename
filename ⟶ letter letter*
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
```

where * is the Kleene star symbol.
`RAWtoPNG`, `PNGtoJPEG` and `JPEGtoPNG` are written without spaces. Spaces are allowed between terminals, but skipped during parsing.

An example for a valid string from the language above is: `shutdown now`
The parse tree generated from this string is shown in Figure 4. The (only) correct JSON encoding of the parse tree shown in Figure 4 is the JSON code given in Figure 5.

Please use the JSON format shown in Figure 5 to create the three requested parse trees. You can find it in the file *exercise1/task3/**example.json***

[1] `https://en.wikipedia.org/wiki/JSON`

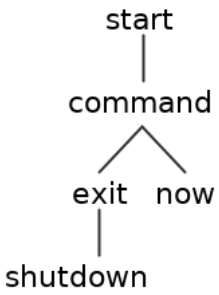

Figure 4: Example of the parse tree created from the `shutdown now` string.

```
{
    "id": "start",
    "children": [
        {
            "id": "command",
            "children": [
                {
                    "id": "exit",
                    "children": [
                        {
                            "id": "shutdown",
                            "children": []
                        }
                    ]
                },
                {
                    "id": "now",
                    "children": []
                }
            ]
        }
    ]
}
```

Figure 5: Example of the JSON tree created from the `shutdown now` string.

The goal of this exercise is to create the parse tree of the following three commands:

1. `convert RAWtoPNG lunch`
2. `python main`
3. `diff old new > out`

Write the parse trees into the following JSON files:

- parse tree of command 1 in the file *exercise1/task3/**tree1.json***
- parse tree of command 2 in the file *exercise1/task3/**tree2.json***
- parse tree of command 3 in the file *exercise1/task3/**tree3.json***

Tips for encoding the parse trees as valid JSON files:

- Do not copy and paste from Figure 5 of the PDF into a text editor. This will often result in invalid or missing characters. Use directly the files provided in the folders;
- Make sure that your JSON file is correct by using a JSON linting tool (like JSONlint [2]) or use a text editor with JSON syntax highlighting, e.g., Sublime text, VS Code, Notepad++, IntelliJ IDEA or many others.

[2] `https://jsonlint.com/`

4 Task IV (50% of total points of the exercise)

You are given regular expressions that describe the legal tokens of a toy programming language:

```
keyword ⟶ if | else | for | while | return
punctuator ⟶ ( | ) | [ | ] | { | } | . | ; | ,
comparison ⟶ < | >
assign ⟶ =
op ⟶ + | − | ∗ | / | && | ||
id ⟶ letter letter∗    (except for keywords)
number ⟶ digit∗
```

where `letter` is a lowercase or uppercase letter (English alphabet only, for example, ä, ê, ì etc. are invalid characters. The *keywords* are case-sensitive.

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. Your 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.

Note that a scanner does not check whether an input string parses into a valid parse tree or abstract syntax tree. All it does check, is that the input string can be split into a valid sequence of tokens.

An Eclipse project for the scanner implementation is provided: *exercise1/task4/*. You can import the project template (.zip) into Eclipse as follows: *File - Import - General - Existing project into Workspace - Select archive file - Finish* .

The input to the scanner is a $String$ containing a snippet of code. The output is a $List < String >$ with all the tokens found by the scanner. In case of invalid tokens, the output must be a $List < String >$ with a single $String$ element: ”invalid”.

Examples of the correct input-output pairs:

Input	**Output**
`while (ifvar1 <0){\n y= 7;}`	`"while","(","ifvar","1","<","0",")","{","y","=","7",";","}"`
`if (k>&& 0 ){\n var = 0;`	`"if","(","k",">","&&","0",")","{","var","=","0",";"`
`return v@r;`	`"invalid"`

The input string in the first row of Table 1 contains `ifvar` because it is a single token. If the input would contain ”`if var`” instead, the scanner would return two tokens ”if” and ”var”. In the second row you see that the input misses a closing } at the end. However, since the scanner does not check grammatical correctness beyond identifying tokens, it produces a valid token sequence.

Please implement your scanner in the method $public\ static\ List < String >\ scanner(String\ input)$, which you find in file exercise1/task4/scr/scanner/TokenScanner.java. Do not create new Java files; use only *TokenScanner.java*.

You find some JUnit tests in folder *exercise1/task4/scr/scanner/*. Use them to check whether your scanner works. They can be run using Eclipse. 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.