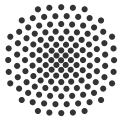


University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2021

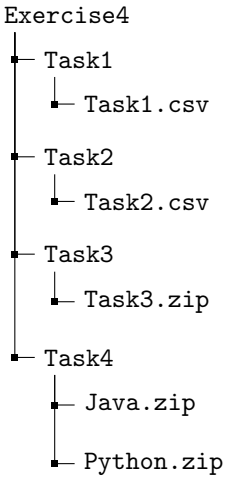
Exercise 4: Control Flow

(Deadline for uploading solutions: June 18, 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:



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 *"Exercise4/"*, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions;

Operator	Rules 1		Type	
	Assoc.	Prec.	Operands	Result
**	R	2	numeric	numeric
*	L	3	numeric	numeric
/	L	3	numeric	numeric
+	L	4	numeric	numeric
-	L	4	numeric	numeric
>>	L	5	numeric	numeric
<<	L	5	numeric	numeric
>	L	6	numeric	boolean
<	L	6	numeric	boolean
!=	L	7	numeric	boolean
&&	L	8	numeric	boolean
||	L	8	numeric	boolean

(a) Ruleset 1.

Operator	Rules 2		Type	
	Assoc.	Prec.	Operands	Result
**	L	2	numeric	numeric
+	L	2	numeric	numeric
-	R	3	numeric	numeric
*	L	3	numeric	numeric
!=	L	4	numeric	boolean
&&	L	5	numeric	boolean
/	L	5	numeric	numeric
>>	R	5	numeric	numeric
<<	R	5	numeric	numeric
>	R	6	numeric	boolean
<	R	6	numeric	boolean
||	L	8	numeric	boolean

(b) Ruleset 2.

Figure 1: Sets of rules for precedence and associativity. A lower number in the "Prec." column means they should be applied first. The "Assoc." column indicates whether the operator is left-associative ("L") or right-associative ("R").

1 Task I (25% of total points of the exercise)

The goal of this task is to evaluate the given expressions under different sets of rules for precedence and associativity. Figures 1a and1b define two sets of rules.
Task: Using the rules in Figure 1a, evaluate the following expressions:

1. `1 > 5 && 5 < 15 / 5 || 2 ** 3 ** 2 > 10`
2. `3 + 1 ** 4 ** 2 > 3`
3. `9 / 3 + 2 << 8 / 4`

Using the rules in Figures 1b, evaluate the following expressions:

4. `9 ** 2 - 18 + 24 - 3 / 1`
5. `1 < 5 || 20 < 10 + 22`
6. `4 != 2 ** 2 && true`

To submit your answer, please fill in the file *Exercise4/Task1/Task1.csv*. Each line in the file correspond to one expression, i.e., there should be six lines in total. In each line, enter only the result of evaluating the expression. The meaning of individual operators is the same as in Java. Additionally, '`**`' represents the exponential (power) operation (e.g., `2 ** 3` is `8`). To indicate the result of evaluating an expression, use the syntax specified by Java literals (e.g., the number five is `5` and the Boolean values are `true` and `false`). Furthermore, all literals are integer and as such consider integer division. Note that the result can either be an integer or a Boolean.
Tip: If you want to check what is the meaning of an individual operation in Java you can just run it here: `https://www.w3schools.com/java/tryjava.asp?filename=demo_compiler`.
Evaluation Criteria: Your solution will be compared against the correct result of evaluating each expression under each set of rules.

2 Task II (25% of total points of the exercise)

This task is about understanding the concept of continuations. Consider the following Ruby code:

Ruby Snippet

```
1   require "continuation"
2   def factorial ( i )
3     (cc, f, i) = callcc{|cc| [cc, 1, i]}
4     if ( i == 0) then
5       return f
6     else
7       cc.call(cc, i * f, i - 1)
8     end
9   end
10  puts(factorial(5))
```

Task: You have to write the sequence in which the lines are executed, the value of the variable `f`, and the value of the variable `i` after each line has been executed. Do not list lines 1, 6, 8, and 9, as they are either irrelevant for understanding continuations or purely syntactic. Submit the correct sequence in the solution file *Exercise4/Task2/Task2.csv* in using the following format:

Task2.csv format

```
1   10, undefined, undefined
2   2, undefined, 5
3   ...
```

The above example means that at first, line 10 is executed, where both `f` and `i` are undefined. Next, line 2 is executed, where `f` is still undefined and `i` has value 5.
Evaluation Criteria: Your solution will be compared against the correct sequence of executed lines and variable values.

3 Task III (20% of total points of the exercise)

This task is about understanding the semantics of loops and recursion. You are provided with Java code that implements an algorithm using a while loop, and your task is to implement the same algorithm using recursion. The algorithm takes an arbitrary tree and turns it into a binary tree, i.e., a tree where each node has at most two children. To this end, the algorithm introduces additional, intermediate nodes whenever a node has more than two children.
The original algorithm is given in the `transformToBinary` method of class `Algorithm`. Please put your solution into the same method, i.e., replacing the given code. Using the `for` and `while` keywords is forbidden in your solution.
An Eclipse project for the implementation is provided: *Exercise4/Task3/Task3.zip*. You can import the project template into Eclipse as follows: *File-Import-General-Existing project into Workspace-Select archive file-Finish*.
You can find a JUnit test in folder *Task3/src/*. Method `testSimpleTreeTransformation()` is provided for understanding of existing code. You should use this method to check whether your code works, and you may want to add additional tests to further validate it.
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 specified above.

Evaluation Criteria: Your code will be evaluated against a set of test cases that exercise the refactored code and that check whether it is semantically equivalent to the provided loop-based code. During the evaluation, we will use Java 8.

4 Task IV (30% of total points of the exercise)

This task is about different kinds of iterators available in popular languages. You are given two implementations of a connected social graph, in Java and Python. Your task is to implement an iterator in each language that returns the list of all the people in the graph, where each person is represented as an object of class `Person`. The order of iteration is up to you.

1. For Java, please extend the `SocialGraph` class to enable iterating over the graph by implementing the `Iterable<Person>` interface.

2. For Python, please extend the given skeleton of a "true" iterator, i.e., the `iterate_nodes()` method of the `SocialGraph` class.

Note: The implementation of the iterator must be yours, i.e., you are not allowed to call into any third-party library.
For Java, you can import the Eclipse project provided under *Exercise4/Task4/Java.zip*. For Python, you can find the Python code under *Exercise4/Task4/Python.zip*. To execute the test cases you have to implement the iterator first. The Python test case requires *pytest*, which you can install, e.g., via *pip install pytest*

Evaluation Criteria: Your code will be executed with different social graphs, and we will check whether the iterators enumerate the correct set of people. During the evaluation, we will use Java 8 and Python 3.7.