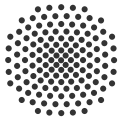


University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2020

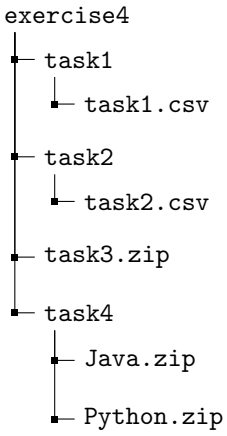
Exercise 4: Control Flow

(Deadline for uploading solutions: June 19, 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:



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;

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. Table 1 defines three sets of rules.
Task: Using each of these sets of rules, evaluate each of the following expressions. To submit your answer, please fill in the file *exercise4/task1/task1.csv*.
Note: 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.

1. `5 + 8 / 2 * 7 - 6`
2. `12 ** 3 >> 2 * 7 - 4 / 3`
3. `15 > 6 + 2 * 4 && 2 * 3 - 2 != 4`
4. `6 << 2 + 4 / 2 + 6 ** 3 / 18 - 6 >> 2`

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

Table 1: Three sets of rules for precedence and associativity. A lower number in the "Prec." column means higher precedence. The "Assoc." column indicates whether the operator is left-associative ("L-R") or right-associative ("R-L").

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**.
Task: You are provided with two incomplete **Ruby** functions and possible code fragments to be used to complete the code. Your task is to fill in some of the blanks with some of the provided code fragments.
Submit the correct assignments of blanks to code fragments in the solution file *exercise4/task2/task2.csv*.

Note: You can fill in at most one code fragment per blank. Once completed, the functions should compute the Factorial and the Fibonacci function of a given integer.
In the provided skeleton file, for each blank either fill in the number of the correct fragment or `0` to indicate that none of the fragments should be inserted.

Incomplete Code

```
def fibonacci(n)
  __blank1__
  if i < n then
    __blank2__
  else
    __blank3__
  end
  __blank4__
  return f
end

def factorial(i)
  __blank5__
  __blank6__
  if (i == 0) then
    __blank7__
    return f
  else
    __blank8__
  end
  __blank9__
end
```

Options for code fragments to insert:

- Fragment 1: `cc.call(cc, f + p, f, i + 1)`
- Fragment 2: `cc, f, p, i = callcc {|cc| [cc, 1, 0, 0]}`
- Fragment 3: `cc.call(cc, f + p, i - 1)`
- Fragment 4: `i = fibonacci(n - 1)`
- Fragment 5: `(cc, f, i) = callcc{|cc| [cc, 1, i]}`
- Fragment 6: `cc.call(cc, i * f, i - 1)`

Evaluation Criteria: Your solution will be compared against a correct assignment of blanks to code fragments.

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

This task is about understanding the semantics of case/switch statements. You are provided Java code that uses a case/switch statement in `activityRecorder(String[] activities, int year)` method.
You need to implement an equivalent code that uses only `if-else` statements in the provided `yourActivityRecorder(String[] activities, int year)` method.

An Eclipse project for the case/switch implementation is provided: *exercise4/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 `activityRecorderTest()` is provided for understanding of existing code. Where as, you should use method `yourActivityRecorderTest()` to check whether your code works. 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 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 equivalent to the provided case/switch code. During the evaluation, we will use Java 11.

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 tree, in Java and Python.
Your task is to implement an iterator in each language that enumerates all nodes of the tree. The order of iteration is up to you.

1. For Java, please extend the `Tree` class to enable iterating over the structure by implementing the `Iterable<Node>` interface.
2. For Python, please extend the given skeleton of a "true" iterator, i.e., the `iterate_nodes()` method.

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.

Evaluation Criteria: Your code will be executed with different trees, and we will check whether the iterators enumerate the correct set of nodes. During the evaluation, we will use Java 11 and Python 3.7.