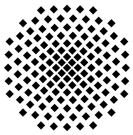

Universität Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart, Winter 2019/2020

Exercise 4: Control Flow

(Deadline for uploading solutions: Dec 08, 2019, 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:

```
exercise4
├── task1.csv
├── task2.csv
├── task3.zip
└── task4
    ├── Java.zip
    └── Python.zip
```

The submission must be compressed in a zip file (**not rar** or other formats) using the given folder structure. The names of folders and files must not be changed, otherwise the homework will not be evaluated. Late submissions will be not accepted.

There are five tasks, which contribute a specific percentage to the overall points for this exercise.

1 Task I (20% 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. Using each of these sets of rules, evaluate each of the following expressions. The meaning of individual operators is the same as in Java. 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`). Additionally, '`**`' represents the exponential (power) operation (e.g., `2 ** 3` is `8`). To submit your answer, please fill in the file *exercise4/task1.csv*.

1. `12 + 3 - 4 ** 2 / 8`
2. `1 > 2 + 3 && 4> 2*3`
3. `16 == 2 != 13 >= 6>>3`
4. `((60 << 2) + (6 ** 3)) / 2 / (6+18) >> 2`

Evaluation Criteria: Your solution will be compared against the correct result of evaluating each expression under each set of rules.

Operator	Rules 1		Rules 2		Rules 3	
	Assoc.	Prec.	Assoc.	Prec.	Assoc.	Prec.
++	–	2	–	2	–	2
--	–	2	–	2	–	2
**	R-L	2	R-L	3	R-L	2
!	–	2	–	4	–	3
%	L-R	3	L-R	6	L-R	4
*	L-R	3	L-R	6	L-R	4
/	L-R	3	L-R	6	L-R	4
+	L-R	4	L-R	5	L-R	3
-	L-R	4	L-R	5	L-R	3
>>	L-R	5	L-R	6	L-R	5
<<	L-R	5	L-R	6	L-R	5
>	L-R	6	L-R	7	L-R	6
>=	L-R	6	L-R	7	L-R	6
==	L-R	7	L-R	8	L-R	7
!=	L-R	7	L-R	8	L-R	7
&&	L-R	8	L-R	8	L-R	7
||	L-R	9	L-R	9	L-R	8
=	R-L	11	L-R	11	R-L	10
+=	R-L	11	R-L	11	R-L	10
-=	R-L	11	R-L	11	R-L	10

Table 1: Three sets of rules for precedence and associativity. A lower number of in the "Prec." column means higher precedence. The "Assoc." column indicates whether the operator is left-associative ("L-R") or right-associative ("R-L"). For unary operators, the associativity is not defined.

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

This task is about understanding the concept of **continuations**. You are provided with two incomplete **Ruby** functions and possible code fragments to be used to complete the code. Please fill in some of the blanks with some of the provided code fragments. 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.
Submit the correct assignments of blanks to code fragments in the solution file *exercise4/task2.csv*. In the provided skeleton file, for each blank either fill in either the number of the correct fragment or `0` to indicate that none of the fragments should be inserted.

Incomplete Code

```
1  def factorial(i)
2    __blank1__
3    __blank2__
4    if (i == 0) then
5      __blank3__
6      return f
7    else
8      __blank4__
9    end
10   __blank5__
11 end
12
13 def fibonacci(n)
14   __blank6__
15   if i < n then
16     __blank7__
17   else
18     __blank8__
19   end
20   __blank9__
21   return f
22 end
```

Options for code fragments to insert:

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

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

3 Task III (25% of total points of the exercise)

This task is about understanding the semantics of case/switch statements in C/C++/Java. You are provided Java code that uses a case/switch statement, and you should refactor it into equivalent code that uses only `if-else` statements.

An Eclipse project for the case/switch implementation is provided: *exercise4/task3*. You can import the project template (.zip) into Eclipse as follows: *File-Import-General-Existing project into Workspace-Select archive file-Finish*.

You find a JUnit test in folder *exercise4/task3/src/*. Use the test 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 an directed graph, in Java and Python. Your task is to implement an iterator on each language that enumerates all edges of the graph. The order of iteration is up to you. For Java, please extend the `Graph` class to implement the iterator object. For Python, please extend the given skeleton of a "true" iterator. 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*. For Python, you can find the Python code under *exercise4/task4/Python*. The graph must be represented as an instance of class `Graph`. To add *vertices* and *edges* into the graph use the $public\ void\ add\_vertex(Integer\ vertex)$ and $public\ void\ add\_edge(Edge\ edge)$ methods, respectively, for Java. To execute the test cases you have to implement the iterator first.

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