

Exercise 1: Programming Distributed Systems (SS 2018)

Submission

- After you have registered for the exercise via Mail to Peter Zeller (p_zeller@cs.uni-kl.de), you will get access to a new Git repository. If you want to work in a Team, you can add your team members using the Gitlab project settings.
- Submit your solution to the programming exercise via your group's repository to a folder named "ex1".
- Test your submission with the provided test cases. Feel free to add more tests, but do not change the existing test cases.

1 Tool Setup (0 points)

Follow the instructions at <https://softech.cs.uni-kl.de/homepage/de/teaching/SS18/progdist/software.html> to install the required software for the exercises.

2 Erlang Basics

Create a new Rebar project: Open a terminal and navigate to the folder, where you want to create the project. Then run "rebar3 new app ex1". This will create a new application in the folder "ex1".

2.1 The Shell

To open a shell in your project, navigate your project folder and run "rebar3 shell". You can use the shell to evaluate Erlang expressions, for example:

```
1> 7*6.  
42  
2> hd([5,4,6]).  
5
```

You can bind the result of expressions to variables. As variables can only be assigned once, you can use `f/1` command to remove a binding in the shell or the `f/0` command to remove all current bindings.

```
3> X = 5.  
5  
4> Y = X + 1.  
6  
5> X = 3.  
** exception error: no match of right hand side value 3  
6> f(X).  
ok  
7> X = 3.  
3
```

The shell is useful to experiment with code you have written. To try this feature, create a new file named "warmup.erl" in your projects src-folder. Then add the following content to the file:

```
1 -module(warmup).  
2 -export([add/2]).  
3 add(X,Y) -> X + Y.
```

Now you can use the command `r3:do(compile).` to load the new code into your shell. As the function is exported from module `warmup`, you can now call the function from the shell to see that it can correctly add 2 numbers:

```
8> r3:do(compile).
Verifying dependencies...
Compiling ex1
ok
9> warmup:add(4,3).
7
```

2.2 Testing

Download the file `ex1_tests.erl` from the course material and put it into a folder named `test` into your project. This file contains the test cases which you can use to test your solutions for this exercise. You can run them from the terminal with the command `rebar3 eunit`. The results are printed on the terminal.

You can also load the tests in an interactive shell by starting the shell with the test profile:

```
rebar3 as test shell
```

Then you can directly call the test functions from the test module or use the generated `test` function to run all tests in the module:

```
1> ex1_tests:maximum_test().
ok
2> ex1_tests:test().
All 10 tests passed.
ok
```

2.3 Numbers, Lists, and Tuples

Implement the functions described below in the module named `warmup`. Use the provided test suite to test your implementation. Avoid using functions from the standard library and try to implement your own version of the functionality.

- a) Write a function `maximum/2`, which takes two numbers and returns the maximum of the two. Do not use the built-in `max` function. Hint: You can use the `if`-expression, `case`-expression or guards.

```
> warmup:maximum(3, 7).
7
> warmup:maximum(5, 4).
5
```

- b) Write a function `list_max/1`, which takes a nonempty list of numbers and computes the maximal element in the list. Do not use the built-in function `lists:max`. Use recursion to implement the function.

```
> warmup:list_max([4,7,5,2]).
7
```

- c) Write a function `sorted/1`, which takes a list of numbers and checks, whether it is sorted in ascending order.

```
> warmup:sorted([4,7,5,2]).
false
> warmup:sorted([2,4,5,7]).
true
```

- d) Write a function `swap/1`, which takes a pair and returns a pair where the two components are swapped.

```
> warmup:swap({ok, 100}).
{100, ok}
```

- e) Write a function `find/2`, which takes a key and a list of key-value pairs. The function should return `{ok, x}`, if `x` is the value of the first pair in the list that has the given key. If no entry with the given key exists, the function should return `error`.

```
> warmup:find(d, [{c, 5}, {z, 7}, {d, 3}, {a, 1}]).
{ok, 3}
> warmup:find(x, [{c, 5}, {z, 7}, {d, 3}, {a, 1}]).
error
```

- f) Write a function `find_all/2`, which takes a list of keys and a list of key-value pairs. The function should use the `find`-function above to lookup every key from the first in the second list. The result should be a list of all key-value pairs that were found with the same order as they appeared in the given list of keys.

```
> warmup:find_all([d, x, c], [{c, 5}, {z, 7}, {d, 3}, {a, 1}]).
[{d, 3}, {c, 5}]
```

2.4 Higher Order Functions

- g) Use `lists:filter/2` to write a function `positive/1`, which takes a list of numbers `L` and returns a list of all numbers in `L`, which are greater or equal to 0.

```
> warmup:positive([6, -5, 3, 0, -2]).
[6, 3, 0]
```

- h) Use `lists:all/2` to write a function `all_positive/1`, which takes a list of numbers and checks whether all numbers in the list are greater or equal to 0.

```
> warmup:all_positive([1, 2, 3]).
true
> warmup:all_positive([1, -2, 3]).
false
```

- i) Use `lists:map/2` to write a function `values/1`, which takes a list of key-value pairs and returns a list of only the values.

```
> warmup:values([{c, 5}, {z, 7}, {d, 3}, {a, 1}]).
[5, 7, 3, 1]
```

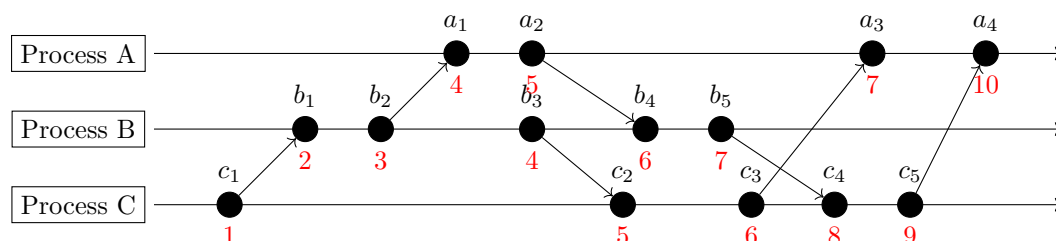
- j) Use `lists:foldl/3` to write a function `list_min`, which computes the minimal element of a nonempty list.

```
> warmup:list_min([7, 2, 9]).
2
```

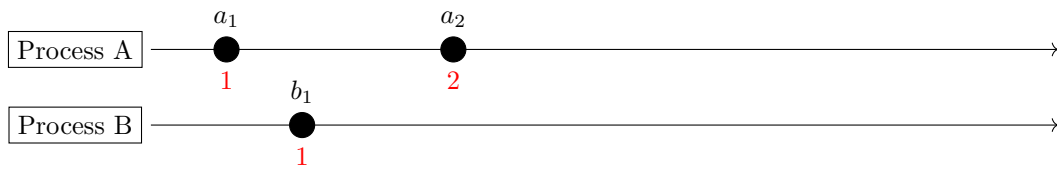
3 Logical Clocks

In the lecture we learned about logical clocks, which assign an integer $t(e)$ to every event e .

- a) Calculate the timestamps $t(e)$ for all events in the following execution.



- b) Give an example execution that shows: $t(e_1) < t(e_2)$ does not imply that $e_1 \rightarrow e_2$.



$t(b_1) = 1 < 2 = t(a_2)$, but $b_1 \not\rightarrow a_2$.

c) Prove that the other direction is valid: If $e_1 \rightarrow e_2$, then $t(e_1) < t(e_2)$.

Assume $e_1 \rightarrow e_2$ and show $t(e_1) < t(e_2)$.

Proof by induction over the inductive definition of the happens before-relation:

Case 1: If e_1 and e_2 are events in the same process and e_1 comes before e_2 :

Since l_p is strongly monotonically increasing for each event, we have $t(e_1) < t(e_2)$.

Case 2: If e_1 is the sending of a message by one process and e_2 is the receipt of the same message by another process:

Then the message must include $t(e_1)$. As $t(e_2) = \max(t(e_1), l_p) + 1$, we have $t(e_2) > t(e_1)$.

Case 3: Transitivity: There is an event e' , such that $e_1 \rightarrow e'$ and $e' \rightarrow e_2$. By induction hypothesis, we have $t(e_1) < t(e')$ and $t(e') < t(e_2)$ and because $<$ is transitive on natural numbers, we get $t(e_1) < t(e_2)$.