Assignment 1

Advanced Functional Programming, 2019
(Avancerad funktionell programmering, 2019)

due 24 November 2019, 23:59

1 Stack Permutations (`perm.erl` & `perm.pdf`, 4 + 2 points)

Let us consider an input queue, a stack, and an output queue. You are asked to use the stack to transfer all elements from the input to the output queue. At each step, you are allowed to either dequeue one or more elements from the front of the input queue and push them on the top of the stack, or pop one or more elements from the top of the stack and enqueue them at the back of the output queue. This is repeated until both the input queue and the stack are empty and all the elements have been transferred to the output queue.

Depending on the order you choose for the dequeue and pop operations, the elements in the output queue will form different permutations of the input queue. These are called *stack permutations*. While there is a large number of stack permutations possible, not all permutations of the input elements are valid stack permutations. Your goal is to determine if, given an input and an output queue, the latter can be generated by performing a stack permutation on the former.

Example

Let us present an example of one stack permutation. We start with the input queue containing, starting from the back of the queue, the elements 1, 2, and 3. We then perform the following five steps (cf. Figure 1):

1. Dequeue 3 and push it on the stack.
2. Dequeue 2 and push it on the stack.
3. Pop 2 from the stack and enqueue it in the output queue.
4. Dequeue 1 and push it on the stack.
5. Pop 1 and 3 and enqueue them in the output queue.

The output queue now contains the elements 3, 1, 2.

Task

Write an Erlang program (`perm.erl`) that returns if the output list is a stack permutation of the input list. To get the maximum number of points for this part (**4 points**), your program needs to be *efficient*: it should return an answer in one second or less, for input lists of up to 1 million elements. There are lots of possible stack permutations, so you cannot just try them all.

Your solution should include tests, including property-based tests, for appropriately chosen parts of the implementation. Your submission must also include a report (`perm.pdf`) explaining the algorithm you used and the properties you tested. **Note that the PDF is required for receiving any of the additional 2 points for writing good property-based tests.**

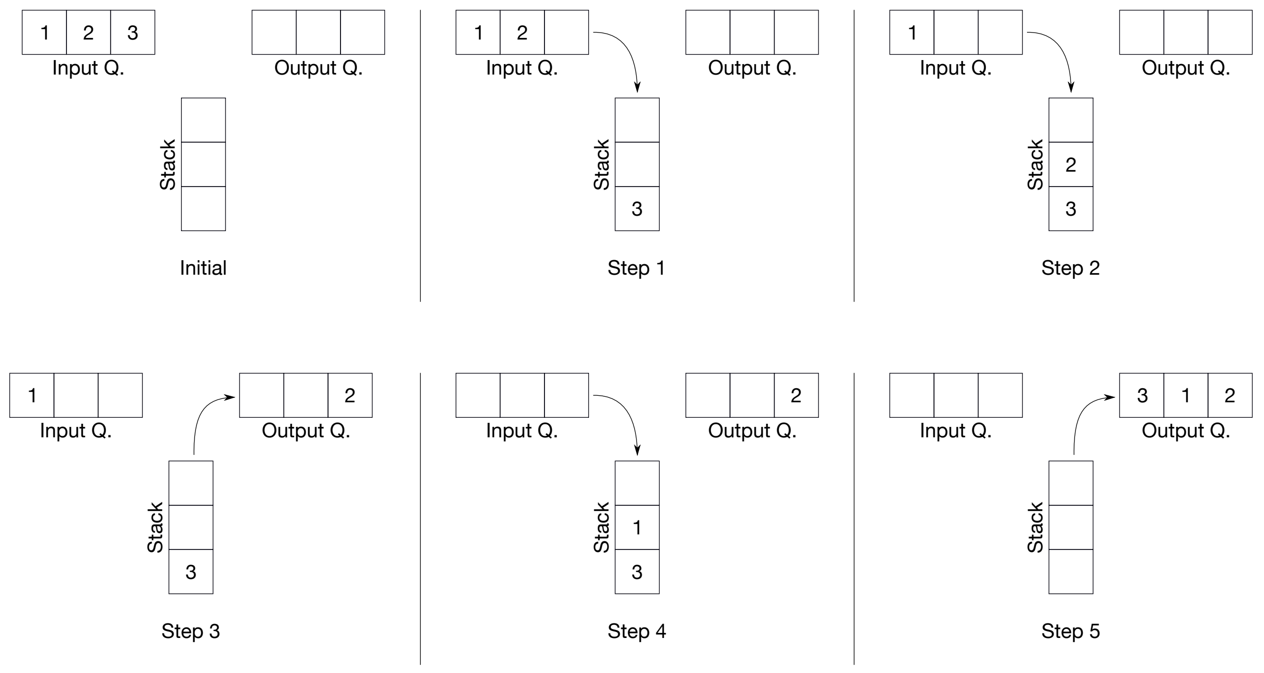


Figure 1: Stack permutation example.

Input / Output

The function takes two lists filled with integers as the input:

1. The first list is the input queue. It can contain from one to one million (1000000) elements. Consider the front of the queue to be on the tail (right side) of the list. Assume that there are no duplicate elements in the list.

2. The second list is the output queue. Once again, consider the front of the queue to be on the tail. The output list will always have the same number of elements as the input list.

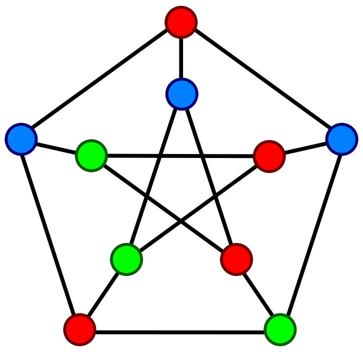
The function should return the atoms `true` or `false` indicating if the output queue is a stack permutation of the input queue or not.

Below we show two sample calls for the function:

```
1> perm:perm([1, 2, 3], [3, 1, 2]).
true
2> perm:perm([1, 2, 3], [2, 3, 1]).
false
```

2 Graph Coloring (`kcolor.erl` & `kcolor.pdf`, 2 + 3 points)

Graph coloring is a common problem in graph theory, where a set of vertices needs to be colored so that there are no adjacent vertices with the same color. If the number of colors is constrained to a number k, then the problem is known as a k-coloring problem. While graph coloring with an unconstrained number of colors can be done in linear time, the k-coloring problem is in NP. Below you can see an example graph colored with 3 colors:



Task

Your task is to implement a *functional* Graph data structure in Erlang, test it, and then use it to implement a graph k-coloring algorithm:

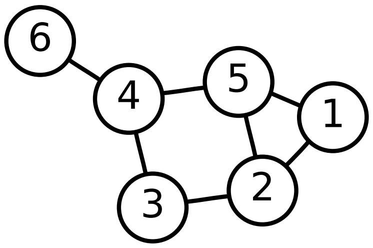
1. **Implement a basic Graph data structure in Erlang.** The Graph should be undirected and without any weights. The data structure should support constructing a Graph by inserting individual nodes and edges. You should also be able to get all the adjacent vertices from a given vertex. It should also support any other operations you need for the next steps. You are *not* allowed to use the built-in Graph implementation in Erlang, which is not purely functional, or any other graph libraries in Erlang you may find on the web. Explain how you implemented your graph structure and why you did it this way. (**1 point**)
2. **Write property-based tests for the Graph data structure.** It is up to you to select appropriate properties to test for. Make sure that you have good code coverage. (**1 point**)
3. **Implement a graph k-coloring algorithm** using the Graph data structure you have created. You are free to choose any algorithm you want, including the backtracking, brute-force approach. (**2 points**)
4. **Test your algorithm.** Same as (2). (**1 point**)

Since graph k-coloring is a problem in NP, you are free to use the backtracking, brute-force algorithm. Other, more efficient (still NP) algorithms exist as well, but it is up to you if you want to use them. If you decide to use another algorithm, make sure that you use an *exact* algorithm, not an approximate one. Keep in mind that given a graph and a number of colors k, it might not be possible to color the graph with only k colors. Even though the algorithm implemented is in NP and we do not have a strict time limit in place, you should still try and write code that is reasonably efficient.

All your code should be submitted in one module (`kcolor.erl`). Additionally, include a PDF where you briefly describe your Graph implementation and why you chose it, which algorithm you used, and what testing you did and why (`kcolor.pdf`). **Note that the PDF is required for receiving any points for the graph implementation and for testing.**

Input / Output

Your `kcolor` module should export a `kcolor` function that takes two inputs: A graph as a list of adjacency lists, and the number of colors k. Each vertex in the input is represented by a positive integer value. For each vertex, the adjacency list will be given as a tuple, with the first element being the vertex label and the second being a list of adjacent vertices. You can see an example graph and its representation here:



```
[
  {1, [2, 5]},
  {2, [1, 5, 3]},
  {3, [2, 4]},
  {4, [3, 5, 6]},
  {5, [1, 2, 4]},
  {6, [4]}
]
```

The input graph will always be connected, you do not need to worry about disconnected graphs. The vertices might not necessarily be ordered like they are in the example. Your function should then return a list of tuples where the first element is the vertex label and the second is the color assigned to it. Use letters, starting from 'a', to represent the colors (assume we will not ask you for more than 26 colors). If it is not possible to color the graph with the given number of colors, then return `false` instead. Here are two example calls:

```
1> kcolor:kcolor([{1, [2, 3]}, {2, [1, 3]}, {3, [1, 2]}], 3).
[{1, 'a'}, {2, 'b'}, {3, 'c'}]
2> kcolor:kcolor([{1, [2, 3]}, {2, [1, 3]}, {3, [1, 2]}], 2).
false
```

3 Vector Calculator Server (`vector_server.erl`, 5 points)

Task

Following the tutorial of Chapter 3 of the book “Erlang and OTP in Action”, which is available at `https://manning-content.s3.amazonaws.com/download/0/8c4c508-6e21-4e8b-ab22-ba9ff22f27e2/sample_Ch03_Erlang.pdf`, implement in Erlang a simple RPC server that evaluates vector expressions given in the language described below. After a connection has terminated, the server should wait for a new connection.

Language

$$
\begin{array}{lcl}
\langle \textit{top} \rangle & ::= & \langle \textit{expr} \rangle \\
\\
\langle \textit{expr} \rangle & ::= & \langle \textit{vector} \rangle \\
 & | & \{\langle \textit{vector-op} \rangle, \langle \textit{expr} \rangle, \langle \textit{expr} \rangle\} \\
 & | & \{\langle \textit{scalar-op} \rangle, \langle \textit{int-expr} \rangle, \langle \textit{expr} \rangle\} \\
\\
\langle \textit{vector} \rangle & ::= & [\langle \textit{integer} \rangle, \ldots] \\
\\
\langle \textit{int-expr} \rangle & ::= & \langle \textit{integer} \rangle \\
 & | & \{\langle \textit{norm} \rangle, \langle \textit{expr} \rangle\}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{vector-op} \rangle & ::= & \text{'add'} \\
 & | & \text{'sub'} \\
 & | & \text{'dot'} \\
\\
\langle \textit{scalar-op} \rangle & ::= & \text{'mul'} \\
 & | & \text{'div'} \\
\\
\langle \textit{norm} \rangle & ::= & \text{'norm\_one'} \\
 & | & \text{'norm\_inf'}
\end{array}
$$

Vector language semantics

- The binary vector operations are: addition, subtraction, and a **non-aggregated** “dot product”-like operator which is just a pairwise multiplication of the vectors’ elements.
- **mul** is multiplication and **div** is integer division of all vector elements with an integer.
- **norm_one** or “Taxicab norm” for vectors is defined as $\|\mathbf{x}\|_1 := \sum_{i=1}^{n} |x_i|$.
- **norm_inf** or “Maximum norm” for vectors is defined as $\|\mathbf{x}\|_\infty := \max\left(|x_1|, \ldots, |x_n|\right)$.
- Integers are not bounded.

Evaluation rules

- The evaluation results in a vector, unless it fails.
- The evaluation **must fail** if:
  - An integer division with 0 is attempted.
  - Any vector in the input has a number of elements that is not between 1 and 100.
  - An expression is nesting deeper than 100 levels. (For example, `{'add', [1], [2]}` has level 0.)
  - The sizes of vectors in a binary vector operation (i.e., ’add’, ’sub’, ’dot’) are not equal.

Input / Output

- The server’s input is a string from the language above. Whitespace is not important. Consider parsing the string as an Erlang term before evaluating it.
- The response should be the result of the evaluation: a vector if the evaluation is successful or the message “error” if the evaluation has failed. Printing response to the socket **must** be done using `io_lib:fwrite("Res: ~w~n", [Result])`, assuming `Result` is what you want to print.

Sample

Similar to the tutorial's example, the server should be started from the Erlang shell with:

```
1> vector_server:start_link().
{ok,<0.35.0>}
```

After the server has started, you can use `telnet` to communicate:

```
$ telnet localhost 1055
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[1,2,3,4,5]
Res: [1,2,3,4,5]
{'dot', [6,6,6], [7,7,7]}
Res: [42,42,42]
{'mul', {'norm_one', [1,-1,2,-2]}, [7,-7,7]}
Res: [42,-42,42]
{'div', 0, [1,2,3,4,5]}
Res: error
```

Connection termination

The server described in the tutorial cannot handle connection termination correctly. Your implementation should take care of the messages received when the client closes the socket and wait for a new connection to be established.

```
$ telnet localhost 1055
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[1,2,3,4,5]
Res: [1,2,3,4,5]
{'dot', [6,6,6], [7,7,7]}
Res: [42,42,42]
^]

telnet> quit
Connection closed.
$ telnet localhost 1055
{'mul', {'norm_one', [1,-1,2,-2]}, [7,-7,7]}
Res: [42,-42,42]
{'div', 0, [1,2,3,4,5]}
Res: error
^]

telnet> quit
Connection closed.
$
```

Notice that the character `^]` is produced by `Ctrl+]`.

4 Property-Based Bug Hunting (bughunt.erl, 4 points)

The module **vectors.beam**[1], contains 50 implementations of an evaluator for the language used in the previous task. Unfortunately 46 of them have bugs...!

Task

Write properties that can be used to test the evaluators. Identify those that do not conform to the specification, by giving an input, the expected output and the buggy evaluator's output.

Interface of vectors.beam

The contents of the corresponding **vectors.erl**[2] were the following:

```
-module(vectors).

-export([vector/1,
         vector_1/1,
         ...
         vector_50/1]).

-type vector()    :: [integer(),...].
-type expr()      :: vector()
                   | {vector_op(),     expr(), expr()}
                   | {scalar_op(), int_expr(), expr()}.
-type int_expr()  :: integer()
                   | {norm_op(), expr()}.
-type vector_op() :: 'add' | 'sub' | 'dot'.
-type scalar_op() :: 'mul' | 'div'.
-type norm_op()   :: 'norm_one' | 'norm_inf'.

-spec vector(integer()) -> fun((expr()) -> vector() | 'error').
vector(Id) when Id > 0, Id < 51 ->
  Name = list_to_atom(lists:flatten(io_lib:format("vector_~p", [Id]))),
  fun ?MODULE:Name/1.

-spec vector_1(expr()) -> vector() | 'error'.
vector_1(Expr) ->
  %% ???

...

-spec vector_50(expr()) -> vector() | 'error'.
vector_50(Expr) ->
  %% ???
```

As you can see, the function vector/1 can be used to get the evaluator corresponding to the provided Id. The evaluators can also be called directly using the vector_N/1 functions.

[1] http://www.it.uu.se/edu/course/homepage/avfunpro/h18/vectors.beam
[2] https://gist.github.com/aronisstav/626f0f8edd943c8ca998

Expected interface of `bughunt.erl`

Among other functions, the module `bughunt` should export a function `test/1` that takes as input an integer between 1 and 50 and returns one of the following within 5 seconds:

- if the input is the id of a correct evaluator, the atom `'correct'` is returned.
- if the input is the id of a buggy evaluator, the tuple {`Input, ExpectedOutput, ActualOutput, Comment`} is returned, where:
  - `Input` is an Erlang term of type `expr()`
  - `ExpectedOutput` is the expected output when evaluating the input
  - `ActualOutput` is the output returned by the buggy evaluator or the atom `'crash'` (if the evaluator crashes) and
  - `Comment` is a string that shortly describes a probable cause for the bug (you can leave it empty if you are not sure about the bug)

Sample

Assume that a hypothetical evaluator #51 is correct and evaluator #52 does not support addition.

```
1> vectors:vector_51({'div', {'norm_inf', [-1, 5, 10]}, [1, 10, 100, 9999]}).
[0, 1, 10, 999]
2> bughunt:test(51).
correct
3> vectors:vector_52({'add', [1], [1]}).
error
4> vectors:vector_52({'sub', [1], [1]}).
[0]
5> bughunt:test(52).
{{'add', [1], [1]}, [2], error, "The operation 'add' is not supported."}
```

Submission instructions

- Each student must send their own individual submission. You cannot work in groups.
- For this assignment, you must submit a single `afp_assignment1.zip` file at the relevant section in Studentportalen.
- `afp_assignment1.zip` should contain *seven* (7) files, wihout any directory structure:
  - the *four* modules requested (`perm.erl`, `kcolor.erl`, `vector_server.erl`, `bughunt.erl`) that should conform to the specified interfaces regarding exported functions, handling of input and format of output. Moreover, your programs should *not* produce any compiler warnings and, preferably, should also not produce any warnings from dialyzer.
  - the report `perm.pdf` explaining your algorithm and how you tested `perm.erl`.
  - the report `kcolor.pdf` explaining your graph implementation, your algorithm, and how you tested `kcolor.erl`.
  - a text file named `README.txt` whose first line should be your name. You can include any other comments about your solutions in this file.
- Remember that you have a total of *seven* (7) free "late" days for all assignments and the final project.

Have fun!