Assignment 1

due 15 November 2013, 16:00

Advanced Functional Programming
(Avancerad funktionell programmering)
2013

1 Car Rally (`rally.sml`, `rally.erl`, 5 + 5 points)

(From MIUP'2006, available online here.)

You are about to participate as a co-driver in the next edition of the "Rallye de Monte-Carlo" (organized by ACM - Automobile Clube de Monaco). Before the actual race all drivers are allowed to run on the tracks of the course. In these reconnaissance drives, the co-drivers, who sit next to the drivers, write down shorthand notes on how to best drive the stage.

Based on your observations on these notes, you were able to create a map with the advisable speed limits of track sections. Passing by these locations over the speed limit is not recommended, since it can make your car crash. To assist your pilot, you need to devise a winning strategy based on the speed limits. And a nice computer program could be handy for this task...

Given a car rally track marked with speed limits at specific locations, your goal is to devise a strategy to reduce and increase the car speed such that you will run the track at the fastest possible time without ever going over the speed limit.

For simplicity the track is divided into section units, each one of them with a specific speed limit. At start position your car marks speed 0 Km/h. You can increase your speed or decrease it by multiples of 10. For each 10 Km/h your car advances 1 unit. For example, if you have a current speed of 50 Km/h your car advances 5 units making what we call a move. After each move, you can change again the car speed. You can for instance accelerate to 70 Km/h making your car advance 7 units more, or you can brake to 40 Km/h making your car advance 4 units.

While you are running at a determined speed (in a single move), you can only pass track units with speed limit equal or bigger than your current speed. For calculations purposes, a move starts on the unit immediately after the current position. Due to mechanical limitations, rally cars have a maximum acceleration and braking speed. For example a car with a maximum acceleration speed of 30 Km/h and maximum break speed of 20 Km/h, can only make an increment to the current speed of 30, 20, 10, 0, -10 or -20 Km/h.

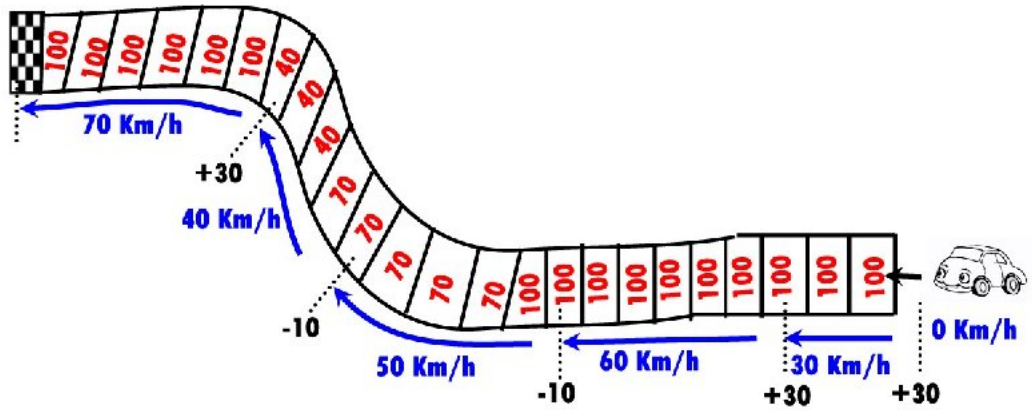
Your car starts the race before the first unit of the track (a "virtual" zero unit) and to finish the race it must pass over the last unit of the track, passing the finish line. You can cross the finish line at any speed. Note that arriving at the last unit is not considered terminating the race!

Task

Write two programs, one in ML (`rally.sml`) and one in Erlang (`rally.erl`), which calculate the number of moves that a car needs to make to finish a track, assuming it uses an optimal strategy, which is one that minimizes the total number of moves, without ever exceeding the speed limit.

Example

The following figure illustrates a track example and its corresponding optimal strategy (5 moves). This track is also described in one of the Samples below.



Input - Output

The programs take as input two integers A and B and a list of integer tuples N, V. Integers A and B indicate the maximum values of acceleration and braking and each tuple indicates a section of N units with speed limit V. The end of the track is indicated by the pair 0 0.

Limits

A and B are always multiples of 10, with $10 < A, B < 240$. V is always a positive multiple of 10 smaller than 250. The maximum number of units in one track is 10.000.

Samples

Here are some sample calls for the two programs:

SML

```
- rally 30 10 [(10,100),(5,70),(3,40),(6,100),(0,0)];
val it = 5 : int
- rally 40 50 [(15,100),(0,0)];
val it = 3 : int
- rally 40 20 [(1,50),(1,40),(1,30),(1,20),(1,10),(1,20),(1,30),(1,40),(1,50),(0,0)];
val it = 5 : int
```

Erlang

```
1> rally:rally(30, 10, [{10,100},{5,70},{3,40},{6,100},{0,0}]).
5
2> rally:rally(40, 50, [{15,100},{0,0}]).
3
3> rally:rally(40, 20, [{1,50},{1,40},{1,30},{1,20},{1,10},
                        {1,20},{1,30},{1,40},{1,50},{0,0}]).
5
```

2 Vector Calculator Server (`vector_server.erl`, 6 points)

Task

Following the tutorial available at `http://www.manning.com/logan/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

⟨*top*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*vector*⟩
| {⟨*vector-op*⟩, ⟨*expr*⟩, ⟨*expr*⟩}
| {⟨*scalar-op*⟩, ⟨*int-expr*⟩, ⟨*expr*⟩}

⟨*vector*⟩ ::= [⟨*integer*⟩, ...]

⟨*int-expr*⟩ ::= ⟨*integer*⟩
| {⟨*norm*⟩, ⟨*expr*⟩}

⟨*vector-op*⟩ ::= 'add'
| 'sub'
| 'dot'

⟨*scalar-op*⟩ ::= 'mul'
| 'div'

⟨*norm*⟩ ::= 'norm_one'
| 'norm_inf'

Vector language semantics

- The binary vector operations are: addition, subtraction and dot product.
- **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 $\|\mathbf{x}\|_\infty := \max(|x_1|, \ldots, |x_n|)$.
- 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.
  - 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. Use a call like `io_lib:fwrite("Res: ~w~n", [Result])` to print the response to the socket.

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 it 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+]`.

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

The module `vectors.beam`, which you can download from Studentportalen, 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` 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.

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:

- 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.
- For this assignment you must submit a single `afp13_assignment1.zip` file at the relevant section in Studentportalen.
- `afp13_assignment1.zip` should contain 5 files (wihout any directory structure):
  - the 4 programs requested (`rally.sml`, `rally.erl`, `vector_server.erl`, `bughunt.erl`), that should conform to the specified interfaces regarding exported functions, handling of input and format of output.
  - 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.

Have fun!