Who Doesn’t Love Emoji 💖

By Ken Friis Larsen (`kflarsen@diku.dk`)

Last revision: October 6, 2020

Part 1: Warm up

- Finish the following declaration of the function `move` from the Erlang intro slides, so that a position also can be moved to in the directions `south` and `east` (at least).

  ```
  move(north, {X, Y}) -> {X, Y+1};
  move(west, {X, Y}) -> {X-1, Y}.
  ```

- A binary search tree is either:

  - the atom `leaf`
  - a tuple `{node, Key, Value, Left, Right}`

    where `Left` and `Right` are binary search trees, and all the keys in `Left` are smaller than `Key` and all the keys in `Right` are larger than `Key`

  Implement the two functions:

  - `insert` that inserts a key and a value into a binary search tree. If the key is already there the value is updated.
  - `lookup` that find the value associated to a key in a binary search tree. Returns `{ok, Value}` if the key is in the tree; or `none` if the key is not in the tree.

Part 2: Emoji Server

Objective

The learning objectives of this assignment are:

- Gain hands-on programming experience with Erlang, how to structure modules and splitting code into functions.
- Learn how to implement a simple server.
- Dipping the toes into making robust code.

The assignment is about making an *Emoji server* that allow looking up emojis

Terminology

Emoji (aka emoticons) are ideograms and smileys used in electronic messages and web pages.

An *emoji shortcode* is an ASCII string starting and ending with `:` (colon) that denotes an emoji. For instance, `:smiley:` denotes 😃, and `:facepalm:` denotes 🤦.

We represent emojis as binaries with the UTF-8 encoding of the emoji. For instance the binary `<<240,159,152,131>>` represents 😃 and `<<240,159,164,166>>` represents 🤦.

An analytics function is a function that takes a shortcode and some state and returns an updated state. Analytics functions may raise exception and may fail to terminate. Your code should be robust against this. Robust here means that an emoji server should, ideally, continue to run and answer requests, even if an analytics function throws exceptions or takes a long (potentially infinite) time to terminate.

Types:

```
-type shortcode() :: string().
-type emoji() :: binary().
-type analytic_fun(State) :: fun((shortcode(), State) -> State).
```

In this assignment we shall not be concerned with enforcing a specific form of shortcodes, such as starting and ending with :, this a job for a parser using our module. Any string will do. Likewise, it’s out of scope to enforce that the binaries are valid UTF-8 encoded emoticons.

The Emoji module

Implement an Erlang module `emoji` with the following API, where `E` always stands for an Emoji server:

- `start(Initial)` 😄 for starting an Emoji server, with the initial shortcodes definition given by `Initial`. The type of `Initial` should be a list of pairs, where the first component is a string and the second is a binary. All shortcodes in `Initial` should be unique, otherwise it is an error.

  Returns `{ok, E}` on success, or `{error, Reason}` if some error occurred.

- `new_shortcode(E, Short, Emo)` 😃 for registering that the shortcode `Short` denotes the emoji `Emo`. It is an error to try to register the same shortcode more than once.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `alias(E, Short1, Short2)` 🙌 for registering `Short2` as an alias for `Short1` . It is an error if `Short1` is not registered, or if `Short2` is already registered.

  After this function succeeds `Short2` should be a valid shortcode in every sense.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `delete(E, Short)` 💥 removes any denotations for `Short`. This is a *non-blocking* function that should always succeed (thus the return value is unspecified). Should delete all aliases for `Short` as well.

- `lookup(E, Short)` 👀 for looking up the emoji for `Short`.

  Returns `{ok, Emo}` on success, or `no_emoji` if no shortcode (including aliases) are found.

- `analytics(E, Short, Fun, Label, Init)` 📈 for registering an analytics function for `Short` (and aliases). Where `Fun` is an analytics function, `Init` is the initial state for `Fun` (thus `Init` could be anything), and `Label` is a string.

  A shortcode can have multiple analytics functions registered, but each must have a unique label. Analytics functions are evaluated when a shortcode (including aliases) is looked up.

  Returns `ok` on success or `{error, Reason}` if some error occurred.

- `get_analytics(E, Short)` 📉 for getting the result of all analytics functions associated with a specific shortcode (including aliases).

Returns `{ok, Stat}` on success, where `Stat` is a list `[{string, any()}]` with the label and the (updated) state for each analytics function registered for `Short`; otherwise returns `{error, Reason}` if some error occurred.

- `remove_analytics(E, Short, Label)` ✏️ for removing the analytics function registered under `Label` for `Short` (including aliases). This is a non-blocking function.

- `stop(E)` :godmode: for stopping an emoji server, and all associated processes (if relevant) . Any use of `E` after this function has been called is an error that is outside the scope of the library.

  Returns `ok` once all processes associated with `E` has terminated; or `{error, Reason}` if some error occurred.

Example use of Emoji

The following example module shows how to use the `emoji` module:

```erlang
-module(love_emoji).
-export([try_it/0]).

hit(_, N) -> N+1.
accessed(SC, TS) ->
  Now = calendar:local_time(),
  [{SC,Now} | TS].

setup() ->
    {ok, E} = emoji:start([]),
    ok = emoji:new_shortcode(E, "smiley", <<240,159,152,131>>),
    ok = emoji:new_shortcode(E, "poop", <<"\xF0\x9F\x92\xA9">>),
    ok = emoji:alias(E, "poop", "hankey"),
    ok = emoji:analytics(E, "smiley", fun(_, N) -> N+1 end, "Counter", 0),
    ok = emoji:analytics(E, "hankey", fun hit/2, "Counter", 0),
    ok = emoji:analytics(E, "poop", fun accessed/2, "Accessed", []),
    E.

print_analytics(Stats) ->
    lists:foreach(fun({Lab, Res}) -> io:fwrite("  ~s: ~p~n", [Lab, Res]) end,
                  Stats).

try_it() ->
    E = setup(),
    {ok, Res} = emoji:lookup(E, "hankey"),
    io:fwrite("I looked for :hankey: and got a pile of ~ts~n", [Res]),
    {ok, Stats} = emoji:get_analytics(E, "poop"),
    io:fwrite("Poppy statistics:~n"),
    print_analytics(Stats),
    io:fwrite("(Hopefully you got a 1 under 'Counter')~n").
```

Testing `emoji`

You tests should be in a module called `test_emoji` in the `tests` directory, this module should export at least one function, `test_all/0`. We will run your tests against our special `emoji` module(s), and OnlineTA might not do any testing of your code unless it is minimally satisfied with your tests.

How to get started

The two difficult parts of this assignment are how to deal with aliases and how to deal with analytics function. Thus, the recommended course of action, is to first get a minimal version working that doesn't include aliases nor analytics function. After that, to deal either aliases or analytics functions, but not both at the same time.

Somewhat counterintuitive, analytics functions can be easier to start with before trying to deal with aliases.

Hand-in Instructions

You should hand in two things:

1. A short report, `report.pdf`, explaining the main design of your code, and **an assessment** of your implementation, including what this assessment is based on.

   In this assignment, it is of particular interest that you discuss how you deal with unreliable analytics functions. That is, you should shortly discuss what issues they cause, and how you have dealt with them.

   Also, if you have thoughts on how the API could be improved, feel free to share them.

2. A ZIP archive `code.zip`, mirroring the structure of the handout, containing your source code and tests.

   Your own tests should be separated in one or more test files that must be in the `tests` directory, so that OnlineTA can treat them right.

Make sure that you adhere to the types of the API, and test that you do.

To keep your TA happy, follow these simple rules:

1. The Erlang compiler, with the parameter `-Wall`, should not yield any errors or warnings for your code.
2. You should comment your code properly, especially if you doubt its correctness, or if you think there is a more elegant way to write a certain piece of code. A good comment should explain your ideas and provide insight.
3. Adhere to the restrictions set in the assignment text, and make sure that you export all of the requested API (even if some functions only have a dummy implementation).
4. Describe clearly what parts of the assignment you have solved.