Homework 4: Ruby, Erlang and Message Passing Programming

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will briefly look into Ruby, and then explore programming in Erlang, doing a bit of functional programming, but concentrating on programming in the message passing model [UseModels] [Concepts].

Directions

We will take some points off for: code with the wrong name, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. Avoid duplicating code by using helping functions, or library functions (when not prohibited in the problems). It is a good idea to check your code for these problems before submitting.
For this homework we suggest that you work individually. (However, per the course's grading policy you can work in a group if you wish, provided that carefully follow the policy on cooperation described in the course's grading policy.)
Don't hesitate to contact the staff if you are stuck at some point.

What to Turn In

For English answers, please paste your answer into the assignment as a "text answer" in the problem's "assignment" on Webcourses.
For each problem that requires code, turn in (on Webcourses) your code and output of testing with our test cases. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.erl` (that is, do not give us a Word document or a PDF file for the code). Also paste the output from our tests into the Comment box for that "assignment".
For all Erlang programs, you must run your code with Erlang/OTP. See the course's Running Erlang page for some help and pointers on getting Erlang installed and running. Your code should compile properly; if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.
You are encouraged to use any helping functions you wish, and to use Erlang library functions, unless the problem specifically prohibits that.

What to Read

For Ruby, see ruby-lang.org and attend our guest lecture on Ruby.
Our recommended book is Joe Armstrong's *Programming Erlang, Second Edition* [Arm13], in which we recommend reading chapters 1-6 and 8-10. There is also an online tutorial Getting Start with Erlang available for free online. Also the tutorial book *Learn you some Erlang for great good!* is free online (or you can order a print version).
For details on Erlang see The Erlang Reference Manual (Feb. 2013).
For the type notation used in the problems, see Types and Function Specifications, chapter 6 in the Erlang Reference Manual.
See also the course code examples page and the course resources page.

Testing Code

The testing code for Erlang is given in the `testing.erl` file supplied with this homework's zip file. Note that to run our tests, all your modules involved must first be compiled. Then to execute our tests in a module named `m_tests`, type `m_tests:main().` to the Erlang prompt.

Problems

Programming Language Concepts in Ruby

1. (5 points) [Concepts] Does Ruby make closures (for blocks and lambdas)? Answer "yes" or "no".
2. (5 points) Which does Ruby use for scoping for variable identifiers: static scoping or dynamic scoping?
3. (5 points) What kind of type checking does Ruby use: static type checking or dynamic type checking?

Sequential Programming in Erlang

In this section the problems review what you have learned using Haskell, but now using Erlang.

4. (10 points) [UseModels] In Erlang, without using any functions from Erlang's `lists` module, write a function `concat/1`, whose type is given by the following.

   ```
   -spec concat(Lists :: [[T]]) -> [T].
   ```

   The function `concat` takes a list of lists of elements (of some type `T`) and returns a list of the elements (of type `T`) formed by concatenating the inner lists together in order. The following are examples written using the Erlang testing module. You can use the built-in ++ in your code, or any other helping functions. Run these tests by compiling your file and the testing file, and then running `concat_tests:main().`

   ```
   % $Id: concat_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
   -module(concat_tests).
   -import(testing,[dotests/2,eqTest/3]).
   -export([main/0, tests/1]).
   main() ->
       compile:file(concat),
       dotests("concat_tests $Revision: 1.3 $", tests(fun concat:concat/1)).
   -spec tests(CFun :: fun(([[T]]) -> [T])) -> testing:testCase([atom()]).
   tests(CFun) ->
       [eqTest(CFun([]), "==", []),
        eqTest(CFun([[]]), "==", []),
        eqTest(CFun([[fee], [fie], [fo], [fum]]), "==", [fee, fie, fo, fum]),
        eqTest(CFun([[], [hmm], [], [okay]]), "==", [hmm, okay]),
        eqTest(CFun([[keep], [ancient, lands], [your, storied, pomp], [cries, she]]),
               "==", [keep, ancient, lands, your, storied, pomp, cries, she]),
        eqTest(CFun([[four, score, nd, seven], [years, ago], [our, ancestors],
                     [brought, forth, on, this, continent], [a, new, nation]]),
               "==", [four, score, nd, seven, years, ago, our, ancestors,
                      brought, forth, on, this, continent, a, new, nation]),
        eqTest(CFun([[[more],[nested]], [[than],[before]]]),
               "==", [[more], [nested], [than], [before]])
       ].
   ```

5. (20 points) [UseModels] This problem is about "sales data records." There are two record types that are involved in the type `salesdata()`, which are defined in the file `salesdata.hrl` (note that file extension carefully, it's `hrl` with an "h") shown below and included with the testing files.

```
% $Id: salesdata.hrl,v 1.1 2013/04/11 02:24:12 leavens Exp $
% Record definitions for the salesdata() type.
-record(store, {address :: string(), amounts :: [integer()]}).
-record(group, {gname :: string(), members :: [salesdata:salesdata()]}).
```

The type `salesdata()` itself is defined by the following, which says that a sales data values is either a store record or a group record.

```
% $Id: salesdata.erl,v 1.4 2015/03/12 12:15:20 leavens Exp $
-module(salesdata).
-include("salesdata.hrl").
-export_type([salesdata/0]).

-type salesdata() :: #store{} | #group{}.
```

Your task is to write, in Erlang, a function

```
-spec substaddr(salesdata:salesdata(), string(), string()) -> salesdata:salesdata().
```

that takes a sales data record `SD`, two strings `New` and `Old`, and returns a sales data record that is just like `SD` except that all store records in `SD` whose address field's value is `Old` in `SD` are changed to `New` in the result.

Your solution must follow the grammar; we will take points off for not following the grammar! In particular, you should never call `substaddr` with a list argument; be sure to use a helping function for lists instead.

Figure 1 on the following page has tests for this problem. To run our tests, run `substaddr_tests:main()`. Note how the testing file does `-include("salesdata.hrl")`; your solution file (`substaddr.erl`) must also include this directive if you want to use the Erlang record syntax.

To be clear, your solution should go in a file `substaddr.erl`, as the tests import `substaddr/3` from that file. Thus to use the record syntax and typing, your solution file `substaddr.erl` should start out as follows. (Note that you cannot import the type `salesdata/0` from `salesdata.erl`, as Erlang does not currently permit type imports.)

```
-module(substaddr).
-export([substaddr/3]).
-include("salesdata.hrl").
-import(salesdata, [store/2, group/2]).

-spec substaddr(salesdata:salesdata(), string(), string()) -> salesdata:salesdata().
```

6. (15 points) [UseModels] [Concepts] In Erlang, without using the lists module or any list comprehensions, write a tail-recursive function

```
-spec count_matching (fun((T) -> boolean()), list(T)) -> non_neg_integer().
```

that takes a predicate, `Pred`, and a list, `Lst`, and returns the number of elements in `Lst` that satisfy `Pred`. Examples are shown in Figure 2 on page 5.

Note that your code must use tail recursion and is not allowed to use any functions from the list module or list comprehensions.

Hint: In Erlang, you can call a fuction closure, say `Pred` on an argument `X` using the syntax `Pred(X)`. However, you cannot call an arbitrary function in the guard of an if expression, as the syntax only allows guard sequences there. See Section 8.17 and Section 8.25 of the Erlang Reference Manual User's Guide for details. Instead of using an if-expression, try using a **`case`** expression instead.

```
% $Id: substaddr_tests.erl,v 1.4 2015/03/12 13:50:59 leavens Exp leavens $
-module(substaddr_tests).
-include("salesdata.hrl").
-import(salesdata,[salesdata/0]).
-import(substaddr,[substaddr/3]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0]).

main() ->
    compile:file(substaddr),
    dotests("substaddr_tests $Revision: 1.4 $", tests()).
tests() ->
    [eqTest(substaddr(#group{gname = "StartUP!", members = []},
                      "Downtown", "50 Washington Ave."),
            "==", #group{gname = "StartUP!", members = []}),
     eqTest(substaddr(#store{address = "The Mall", amounts = [10,32,55]},
                      "110 Main St.", "The Mall"),
            "==", #store{address = "110 Main St.", amounts = [10,32,55]}),
     eqTest(substaddr(
              #group{gname = "Target",
                     members = [#store{address = "The Mall", amounts = [10,32,55]}]},
                     "Substaddr", "OldAddress"),
            "==",
            #group{gname = "Target",
                   members = [#store{address = "The Mall", amounts = [10,32,55]}]}),
     eqTest(substaddr(
              #group{gname = "Target",
                     members = [#store{address = "The Mall", amounts = [10,32,55]},
                                #store{address = "Downtown", amounts = [4,0,2,0]}]},
              "253 Sears Tower", "The Mall"),
            "==",
            #group{gname = "Target",
                   members = [#store{address = "253 Sears Tower", amounts = [10,32,55]},
                              #store{address = "Downtown", amounts = [4,0,2,0]}]}),
     eqTest(substaddr(
              #group{gname = "ACME",
                     members =
                         [#group{gname = "Robucks",
                                 members = [#store{address = "The Mall", amounts = [99]},
                                            #store{address = "Maple St.", amounts = [32]}]},
                          #group{gname = "Target",
                                 members = [#store{address = "The Mall", amounts = [10,55]},
                                            #store{address = "Downtown", amounts = [4]}]}]},
       "High St.", "The Mall"),
            "==", #group{gname = "ACME",
                         members =
                             [#group{gname = "Robucks",
                                     members = [#store{address = "High St.", amounts = [99]},
                                                #store{address = "Maple St.", amounts = [32]}]},
                              #group{gname = "Target",
                                     members = [#store{address = "High St.", amounts = [10,55]},
                                                #store{address = "Downtown", amounts = [4]}]}]})
    ].
```

Figure 1: Tests for problem 5.

```
% $Id: count_matching_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
-module(count_matching_tests).
-import(count_matching,[count_matching/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() ->
    compile:file(count_matching),
    dotests("count_matching_tests $Revision: 1.3 $", tests()).
tests() ->
    [eqTest(count_matching(fun (X) -> X == 3 end, [3,2,1]), "==", 1),
     eqTest(count_matching(fun (X) -> X == 3 end, [2,1]), "==", 0),
     eqTest(count_matching(fun (X) -> X == 3 end, []), "==", 0),
     eqTest(count_matching(fun (X) -> X > 4 end, [5,5,1,2,7,9,8,4,20]), "==", 6),
     eqTest(count_matching(fun ({X,Y}) -> X == Y end,
                           [{3,3},{7,5},{4,3},{4,4}]), "==", 2),
     eqTest(count_matching(fun ({X,Y}) -> 3*X+1 == Y end,
                           [{3,3},{3,10},{4,13},{4,4},{27,82}]), "==", 3),
     eqTest(count_matching(fun ({X,Y}) -> X*2 == Y end,
                           [{3,6},{7,15},{7,14},{7,13},{92,184},{1,2}]), "==", 4)
    ].
```

Figure 2: Tests for problem 6.

Concurrent Actor Programming in Erlang

7. (10 points) [UseModels] In Erlang, write a stateless server in a module named `power`. This server responds to messages of the form `{Pid, power, N, M}`, where `Pid` is the sender's process id, `N` and `M` are non-negative integers. When the server receives such a message, it responds by sending a message of the form `{answer, Res}` to `Pid`, where `Res` is N^M, that is `N` raised to the `M`th power. In your solution you can use the library function `math:pow`, which is defined so that `math:pow(N,M)` returns N^M. Figure 3 has tests for this problem. To run our tests, run `power_tests:main().`

```
% $Id: power_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
-module(power_tests).
-export([main/0]).
-import(power,[start/0]).
-import(testing,[dotests/2,eqTest/3]).
main() ->
    compile:file(power),
    dotests("power_tests $Revision: 1.3 $", tests()).
tests() ->
    PS = start(),
    [eqTest(compute_power(PS, 0,0),"==",1.0),
     eqTest(compute_power(PS, 22,0),"==",1.0),
     eqTest(compute_power(PS, 1,1),"==",1.0),
     eqTest(compute_power(PS, 6,1),"==",6.0),
     eqTest(compute_power(PS, 2,3),"==",8.0),
     eqTest(compute_power(PS, 3,8),"==",6561.0),
     eqTest(compute_power(PS, 3,4),"==",81.0),
     eqTest(compute_power(PS, 3,3),"==",27.0),
     eqTest(compute_power(PS, 3,2),"==",9.0),
     eqTest(compute_power(PS, 5,10),"==",9765625.0),
     eqTest(compute_power(PS, 5,2),"==",25.0)].
%% helper for testing, NOT for you to implement.
compute_power(PS, N, M) ->
    PS ! {self(), power, N, M},
    receive {answer, Res} -> Res end.
```

Figure 3: Tests for problem 7.

8. (15 points) [UseModels] Write, in Erlang, a module `var`, whose `start/1` function returns the process id of a server. The server's state contains a value, which is initially the value given to `start/1` as its argument. The server responds to the following messages:

   - **`{`**`assign, NewVal`**`}`**, which makes the server continue with `NewVal` as its new value.
   - **`{`**`Pid, fetch`**`}`**, which causes the serve to send the message **`{`**`value, Value`**`}`** to `Pid`, where `Value` is the server's current value. The server's value is unchanged by this message.

   Do *not* use the `ets`, `dets`, or `mnesia` modules in your solution. (This is to keep the problem simple.) Instead, store the value in an argument to the server's loop, as we have shown in class.

   There are tests in Figure 4.

```
% $Id: var_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
-module(var_tests).
-import(var, [start/1]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0,vfetch/1,bassign/2]).

main() ->
    compile:file(var),
    dotests("var_tests $Revision: 1.3 $", tests()).

-spec tests() -> [testing:testCase(integer())].
tests() ->
    B1 = var:start(1),
    B2 = var:start(2),
    [eqTest(vfetch(B1),"==",1),
     eqTest(vfetch(B2),"==",2),
     eqTest(bassign(B1,99),"==",99),
     eqTest(vfetch(B1),"==",99),
     eqTest(vfetch(B2),"==",2),
     eqTest(bassign(B2,3),"==",3),
     eqTest(bassign(B2,5),"==",5),
     eqTest(bassign(B2,5),"==",5),
     eqTest(vfetch(B2),"==",5),
     eqTest(vfetch(B1),"==",99),
     eqTest(vfetch(B2),"==",5)
     ].

% The following functions are used for testing purposes.
% You don't have to implement them again.
vfetch(Pid) ->
    Pid!{self(), fetch},
    receive
        {value, Value} ->
            Value
    after 3000 ->
            io:format("timeout waiting for {value, Value} message~n"),
            exit(wrong_message)
    end.

bassign(Pid, Value) ->
    Pid!{assign, Value},
    Value.
```

Figure 4: Tests for problem 8.

9. (20 points) [UseModels] In an Erlang module named `logger`, write a function `start/0`, which creates a log server and returns its process id. A server created by `logger:start()` keeps track of a list of log entries. The entries are simply Erlang values (of any type). The server responds to two types of messages:

   - `{Pid, log, Entry}`, where `Pid` is the sender's process id, and `Entry` is a value. This message causes the server to remember `Entry` in its list. The server responds by sending to `Pid` a message of the form `{SPid, logged}`, where `SPid` is the server's process id.
   - `{Pid, fetch}`, where `Pid` is the sender's process id. The server responds by sending a message to `Pid` of the form `{SPid, log_is, Entries}`, where `SPid` is the server's process id, and `Entries` is a list of all the entries that have been previously received by the log server (`SPid`), in the order in which they were received (oldest first).

   Figure 5 contains tests for this problem. To run our tests, run `logger_tests:main().`

```
% $Id: logger_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
-module(logger_tests).
-import(logger,[start/0]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0,logThenFetch/2,log/2,fetch/1]).
main() ->
    compile:file(logger),
    dotests("logger_tests $Revision: 1.3 $", tests()).
tests() ->
    L1 = logger:start(),
    L2 = logger:start(),
    [eqTest(fetch(L1),"==",[]),
     eqTest(fetch(L2),"==",[]),
     eqTest(logThenFetch(L1,[starting,middle,ending]),"==",[starting,middle,ending]),
     eqTest(logThenFetch(L2,[start,between,last]),"==",[start,between,last]),
     eqTest(logThenFetch(L1,[final]),"==",[starting,middle,ending,final]),
     eqTest(logThenFetch(L1,[really]),"==",[starting,middle,ending,final,really]),
     eqTest(logThenFetch(L2,[ultimate]),"==",[start,between,last,ultimate]),
     eqTest(fetch(L1),"==",[starting,middle,ending,final,really])
    ].
% helpers for testing (client functions), NOT for you to implement
logThenFetch(Logger, []) ->
    fetch(Logger);
logThenFetch(Logger, [Entry|Entries]) ->
    log(Logger, Entry),
    logThenFetch(Logger, Entries).
log(Logger, Entry) ->
    Logger ! {self(), log, Entry},
    receive {Logger, logged} -> logged end.
fetch(Logger) ->
    Logger ! {self(), fetch},
    receive {Logger, log_is, Entries} -> Entries end.
```

Figure 5: Tests for problem 9.

10. (25 points) [Concepts] [UseModels] In Erlang, write a barrier synchronization server in a module `barrier`. In barrier synchronization, a group of processes wait until all of them have are done executing up to a certain point (the barrier). You will write a function `start/1`, which takes a positive integer, which is the size of the group of processes, and creates a barrier synchronization server, returning its process id. The barrier synchronization server tracks in its state the number of processes that are still running (are not yet done), and the process ids of all processes that have reached the barrier. The server responds to messages of the following forms:

    - `{Pid, done}`, where `Pid` is the process id of the sender. The server responds sending a message to `Pid` of the form `{SPid, ok}`, where `SPid` is the server's own process id. What it does next depends on whether `Pid` was the last process in the group to finish. If `Pid` is the last process in the group to be done (i.e., if there are no other running processes), then `Pid` and all the processes that have previously sent such a `done` message are sent a message of the form `{SPid, continue}`, which lets them continue past the barrier. If there are other running processes, then the server just remembers `Pid` in the list of processes that have reached the barrier (and are thus waiting).
    - `{Pid, how_many_running}`, where `Pid` is the process id of the sender. The server responds by sending a message to `Pid` of the form `{SPid, number_running_is, Running}`, where `SPid` is the server's own process id and `Running` is the number of processes in the group that have not yet reached the barrier. The server continues with an unchanged state.

    You can assume that each process in the group only sends a `done` message to the server once. (But despite this, the server does not "reset" or start over, but keeps running once all the processes in the group are done.)

    Figure 6 on the next page contains tests for this problem. To run our tests, run `barrier_tests:main().`

```
% $Id: barrier_tests.erl,v 1.3 2015/03/12 13:50:59 leavens Exp leavens $
-module(barrier_tests).
-export([main/0]).
-import(barrier,[start/1]).
-import(testing,[dotests/2,eqTest/3]).
-import(lists,[map/2,foreach/2]).
main() ->
    compile:file(barrier),
    dotests("barrier_tests $Revision: 1.3 $", tests()).
tests() ->
    Br = barrier:start(4),
    Workers = map(workerCreator(Br),[1,2,3,4]),  %% start the workers
    [eqTest(num_running(Br), "==", 4),
     begin
         send_finish(hd(Workers)),
         eqTest(num_running(Br), "==", 3)
     end,
     eqTest(num_released(), "==", 0),
     begin
         send_finish(hd(tl(Workers))),
         eqTest(num_running(Br), "==", 2)
     end,
     eqTest(num_released(), "==", 0),
     begin
         foreach(fun(W) -> send_finish(W) end, tl(tl(Workers))),
         eqTest(num_running(Br), "==", 0)
     end,
     eqTest(num_released(), "==", 4)
    ].
%% helpers for testing (client functions), NOT for you to implement
workerCreator(Barrier) -> fun(_Num) ->
                              TPid = self(),
                              spawn(fun() -> worker_fun(Barrier, TPid) end)
                          end.
%% worker_fun acts under control of the testing code's finish message, telling
%% it when the barrier acknowledges its done message and when it's released.
worker_fun(Barrier, TestingPid) ->
    receive {TestingPid, finish} -> ok end,
    Barrier ! {self(), done},
    receive {Barrier, ok} -> TestingPid ! {self(), ok} end,
    receive {Barrier, continue} -> ok end,
    TestingPid ! {self(), released}.
%% send finish to the worker process and get an ack (for testing purposes).
send_finish(Pid) ->
    Pid ! {self(), finish},
    receive {Pid, ok} -> ok end.
%% How many processes are not finished (still running)?
num_running(Barrier) ->
    Barrier ! {self(), how_many_running},
    receive {Barrier, number_running_is, Num} -> Num end.
%% How many released messages have been received by the testing process?
num_released() -> num_released(0).
num_released(N) -> receive {_Pid, released} -> num_released(N+1)
                   after 0 -> N
                   end.
```

Figure 6: Tests for problem 10.

11. (25 points) [UseModels] In this problem you will write an election server and two client functions. The three functions you are to implement are the following.

    ```
    -spec start() -> pid().
    ```

    The `start/0` function which creates a new election server and returns its process id.

    ```
    -spec vote(ES::pid(), Candidate::atom()) -> ok.
    ```

    The `vote/2` function takes as arguments the process id of an election server, and a candidate's name (an atom). By sending messages to the election server, this function casts a single vote the candidate. After the server has received the vote, this function returns the atom ok to the caller.

    ```
    -spec results(ES::pid()) -> [{atom(), non_neg_integer()}].
    ```

    The `results/1` function takes the election server's process id as an argument. It returns a list of pairs of the form `{Candidate, Vote}`, where `Candidate` is a candidate's name, and `Vote` is the total number of votes cast for the candidate. The returned list sorted in the order given by `lists:sort/1`, i.e., in non-decreasing order by the candidate's name. This function does not change the state of the server.

    Figure 7 contains tests for this problem. To run our tests, run `electionserver_tests:main()`.

```
% $Id: electionserver_tests.erl,v 1.2 2015/03/12 19:14:01 leavens Exp leavens $
-module(electionserver_tests).
-import(electionserver,[start/0, vote/2, results/1]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() ->
       compile:file(electionserver),
       dotests("electionserver_tests $Revision: 1.2 $", tests()).
tests() ->
    ES = start(), E2 = start(),
    [eqTest(results(ES),"==",[]),
     begin vote(ES, clinton), vote(ES, clinton), vote(ES, christy),
           eqTest(results(ES),"==",[{christy,1},{clinton,2}])
     end,
     begin vote(ES, christy), vote(ES, christy), vote(ES, christy),
           vote(ES, abel), vote(ES, baker), vote(ES,clinton),
           eqTest(results(ES),"==",[{abel,1},{baker,1},{christy,4},{clinton,3}])
     end,
     eqTest(results(E2),"==",[]),
     begin vote(E2, ucf), vote(E2, usf), vote(E2, fiu), vote(E2, uf),
           vote(E2, ucf), vote(E2, ucf), vote(E2, fsu),
           eqTest(results(E2),"==",[{fiu,1},{fsu,1},{ucf,3},{uf,1},{usf,1}])
     end
    ].
```

Figure 7: Tests for problem 11.

12. (30 points) [UseModels] [Concepts] In Erlang, write a module `eventdetector`, which has a function `start/2` that takes two arguments: `InitialState`, and `TransitionFun`. For each type of state, `S`, the `InitialState` has type `S` and the `TransitionFun` has type `fun((S,atom()) ->{S,atom()})`; that is, it is a function that takes a state and an atom and returns a pair of a state and an atom. A call to `start/2` makes a server process that tracks a state and a list of observers. The observers are remembered by remembering their process ids. An event detector server responds to the following messages.

    - `{Pid, add_me}`, which sends the message `{added}` (i.e., a singleton tuple containing the atom `added`) to the process id `Pid`, and then adds the process id `Pid` to the head of the list of observers that the event detector is remembering. (The remembered state is unchanged.)
    - `{Pid, add_yourself_to, EDPid}`, which sends the message `{self(), add_me}` to `EDPid`, and then waits for the server `EDPid` to respond with the message `{added}`, which it sends to `Pid`. (The remembered state and list of observers of this event detector itself are unchanged.)
    - `{Pid, state_value}`, which sends the current state of the server to `Pid` in a message of the form `{value_is, State}`, where `State` is the event detector's remembered state. (The remembered state and list of observers of this event detector are unchanged.)
    - an atom, which causes the event detector to call `TransitionFun` on the current state and the atom received, which yields a pair `{NewState, Event}`. If `Event` is the atom `none`, then no observers are sent messages; otherwise, if `Event` is any other atom (other than `none`), then `Event` is sent to all observers in the server's list of observers (in the order of that list). After handling any needed notification of the observers, then the event detector continues with the state `NewState` and an unchanged list of observers.

    Figure 8 on the next page through Figure 10 on page 14 contain some examples from the file `eventdetector_tests.erl`. To run our tests execute `eventdetector_tests:main().`

Points

This homework's total points: 185.

References

[Arm13] Joe Armstrong. *Programming Erlang, Second Edition: Software for a Concurrent World*. Pragmatic Programmers, LLC, second edition, 2013.

```
% $Id: eventdetector_tests.erl,v 1.9 2015/03/12 13:50:59 leavens Exp leavens $
-module(eventdetector_tests).
-import(testing, [eqTest/3, dotests/2]).
-export([main/0, setup/0, tests/6, addObserver/2, getValue/1, feed/2]).

main() ->
    compile:file(eventdetector),
    {CountGoGos, CountGadgets, CountGGGs,
     AccumGoGos, AccGGGs, AccumMatches} = setup(),
    dotests("eventdetector_tests $Revision: 1.9 $",
            tests(CountGoGos, CountGadgets, CountGGGs,
                  AccumGoGos, AccGGGs, AccumMatches)).
setup() ->
    GoGo = eventdetector:start(zero, fun gogodetect/2),
    Gadget = eventdetector:start(init, fun gadgetdetect/2),
    GoGoGadget = eventdetector:start(start, fun gogogadget/2),
    Matcher = eventdetector:start(0, fun matchingdetect/2),
    CountGoGos = eventdetector:start(0, fun count/2),
    addObserver(GoGo, CountGoGos),
    addObserver(GoGo, GoGoGadget),
    addObserver(Gadget, GoGoGadget),
    AccumGoGos = eventdetector:start([], fun accumulate/2),
    addObserver(GoGo, AccumGoGos),
    CountGadgets = eventdetector:start(0, fun count/2),
    addObserver(Gadget, CountGadgets),
    CountGGGs = eventdetector:start(0, fun count/2),
    AccGGGs = eventdetector:start([], fun accumulate/2),
    addObserver(GoGoGadget, CountGGGs),
    AccumMatches = eventdetector:start([], fun accumulate/2),
    addObserver(Matcher, AccumMatches),
    feed(GoGo, [go,stop,go,stop,stop,go,go,go,go,stop,go,stop,go,go]),
    feed(Gadget, [gadget,trinket,blanket,gadget,gadget,widget,omlet,capulet,
                  gadget,gadget,gadget,gadget,trinket]),
    feed(Matcher, [left,right,left,left,right,right,right,left,left,right]),
    timer:sleep(200), % time for messages to be delivered... (hack)
    {CountGoGos, CountGadgets, CountGGGs,
     AccumGoGos, AccGGGs, AccumMatches}.

tests(CountGoGos, CountGadgets, CountGGGs,
     AccumGoGos, AccGGGs, AccumMatches) ->
    [eqTest(getValue(CountGoGos),"==",4),
     eqTest(getValue(CountGadgets),"==",7),
     eqTest(getValue(CountGGGs),"==",0),
     eqTest(getValue(AccumGoGos),"==",[gogo,gogo,gogo,gogo]),
     eqTest(getValue(AccGGGs),"==",[]),
     eqTest(getValue(AccumMatches),"==",[matched,matched,matched])
     ].

% Helpers for testing, not for you to implement.
% Some transition functions, for testing purposes only.
gogodetect(zero, go) -> {go, none};
gogodetect(go, go) -> {go, gogo};
gogodetect(_, _) -> {zero, none}.
```

Figure 8: Testing for problem Problem 12 on the previous page, part 1 of 3.

```
gadgetdetect(init, gadget) -> {init, gadget};
gadgetdetect(init,_) -> {init,none}.

gogogadget(start, gogo) -> {gogo, none};
gogogadget(start, _) -> {start, none};
gogogadget(gogo, gadget) -> {start, gogogadget};
gogogadget(gogo, _) -> {start, none}.

matchingdetect(N, left) -> {N + 1, none};
matchingdetect(1, right) -> {0, matched};
matchingdetect(N, right) -> {N-1, none}.

accumulate(Lst, Event) -> {[Event|Lst], none}.

count(N, _Event) -> {N+1, none}.

% Hook up an observer to an event detector
-spec addObserver(pid(), pid()) -> ok.
addObserver(EDPid, ObsPid) ->
    ObsPid ! {self(), add_yourself_to, EDPid},
    receive
        {added} -> ok
    after 3000 ->
            io:format("timeout waiting for {added} message~n"),
            exit(wrong_message)
    end.

% Get the state value of an event detector.
-spec getValue(pid()) -> any().
getValue(Pid) ->
    Pid ! {self(), state_value},
    receive
        {value_is, State} -> State
    after 3000 ->
            io:format("timeout waiting for {value_is, State} message~n"),
            exit(wrong_message)
    end.
```

Figure 9: Testing for problem Problem 12 on page 12, part 2 of 3.

```
% A testing helper.  Feeds a list of atoms to the Pid argument.
-spec feed(pid(),[atom()]) -> done.
feed(_Pid, []) -> done;
feed(Pid, [A|As]) ->
    Pid ! A,
    feed(Pid, As).
```

Figure 10: Testing for problem Problem 12 on page 12, part 3 of 3.