COP 4020 — Programming Languages I November 1, 2013

Homework 6: Erlang and Message Passing Programming

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will 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

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 show in Figure 1 on the following page and Figure 2 on page 4. Note that to run our tests, all your modules involved must first be compiled.

Problems

Sequential Programming in Erlang

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

1. (10 points) [UseModels] In Erlang, write a function `censor/2`, whose type is given by the following.

   ```
   -spec censor([[atom()]], [atom()]) -> [[atom()]].
   ```

   The function `censor` takes a list of lists of atoms, `Document`, and a list of atoms, `BadWords`, and returns a list of lists of atoms that is just like `Document` except that the result omits all the words that are in `BadWords`.

   Examples written using the testing module are found in Figure 3 on page 5.

```
% $Id: testing.erl,v 1.6 2013/11/01 14:17:22 leavens Exp leavens $
-module(testing).
-import(io, [write/1, nl/0, put_chars/1]).
-export([eqTest/3, gTest/4, assertTrue/1, assertFalse/1,
         run_test/1, run_tests/1, dotests/2,
         run_test_list/2, startTesting/1, doneTesting/1]).
-export_type([testCase/1]).

-type testCase(A) :: {'test',fun((A,A) -> boolean()),fun(() -> A),string(),A}.

% eqTest makes an equality test.
-spec eqTest(A,string(),A) -> testCase(A).
eqTest(Code,Connective,Expected) ->
    {test, fun(X,Y) -> X == Y end, Code, Connective, Expected}.

% gTest constructs a test case (in the most general way).
-spec gTest(fun((A,A)->boolean()), A, string(), A) -> testCase(A).
gTest(Comp,Code,Connective,Expected) ->
    {test, Comp, Code, Connective,Expected}.

% Assertions, which make test cases
-spec assertTrue(boolean()) -> testCase(boolean()).
-spec assertFalse(boolean()) -> testCase(boolean()).
assertTrue(B) -> {test, fun(X,Y) -> X == Y end, B, "==", true}.
assertFalse(B) -> {test, fun(X,Y) -> X == Y end, B, "==", false}.

% For running a single test case, use the following.
% The number returned is the number of test cases that failed.
-spec run_test(testCase(_)) -> integer().
run_test({test,Comp,Code,Connective,Expected}) ->
    Result = Comp(Code,Expected),
    case Result of
        true -> writeln(Code);
        false -> put_chars(failure()),
                 writeln(Code)
    end,
    put_chars("      "),
    put_chars(Connective),
    put_chars(" "),
    writeln(Expected),
    case Result of
        true -> 0;
        false -> 1
    end.
```

Figure 1: Testing module for Erlang, part 1 of 2.

```
% The following will run an entire list of tests.
-spec run_tests([testCase(_)]) -> integer().
run_tests(Ts) ->
    Errs = run_test_list(0,Ts),
    doneTesting(Errs).

% A version of run_tests with more labeling.
-spec dotests(string(),[testCase(_)]) -> integer().
dotests(Name,Ts) ->
    startTesting(Name),
    run_tests(Ts).

% A way to run a list of tests
-spec run_test_list(integer(),[testCase(_)]) -> integer().
run_test_list(Errs_so_far,[]) -> Errs_so_far;
run_test_list(Errs_so_far,[T|Ts]) ->
    Err_count = run_test(T),
    run_test_list(Errs_so_far + Err_count, Ts).


% Print a newline and a message that testing is beginning.
-spec startTesting(string()) -> ok.
startTesting(Name) -> nl(),
                      put_chars("Testing "),
                      put_chars(Name),
                      put_chars("..."),
                      nl().

% Print information about the tests.
-spec doneTesting(integer()) -> integer().
doneTesting(Fails) ->
    put_chars("Finished with "),
    write(Fails),
    put_chars(" "),
    put_chars(case Fails of
              1 -> "failure!";
              _ -> "failures!"
          end),
    nl(),
    Fails.

% hidden functions below

% write Term out, followed by a newline
writeln(Term) ->
    io:format("~p~n",[Term]).

% the failure string
failure() -> "FAILURE: ".
```

Figure 2: Testing module for Erlang, part 2 of 2.

```
-module(censor_tests).
-import(censor,[censor/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("censor_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(censor([],[sword, fword]), "==", []),
     eqTest(censor([[], [fword], []],[sword, fword, pword]),
            "==", [[], [], []]),
     eqTest(censor([[fword, it]], [sword, fword]), "==", [[it]]),
     eqTest(censor([[this, is, sword, see], [fword, it]], [sword, fword]),
            "==", [[this, is, see], [it]]),
     eqTest(censor([[inceptis, grauibus, plerumque, et, magna, professis],
                    [purpureus, late, qui, splendeat], [unus, et, alter],
                    [adsuitur, pannus, cum, lucus, et, ara, dianae]],
                   [purpureus, et, inceptis, dianae, unus, alter]),
            "==", [[grauibus,plerumque,magna,professis],
                   [late,qui,splendeat], [], [adsuitur,pannus,cum,lucus,ara]]),
     eqTest(censor([['@#*!+', '@#*!+', '@#*!+'], ['*^$@!', '*^$@!'],
                    [flowers, birds, trees], ['@#*!+', '*^$@!']],
```

Figure 3: Tests for problem 1.

2. (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 1.3 2013/10/30 03:13:40 leavens Exp leavens $
-module(salesdata).
-include("salesdata.hrl").
-export_type([salesdata/0]).

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

Your task is to write, in Erlang, a function

```
-spec newaddress(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!

Figure 4 on the next page has tests for this problem. To run our tests, run `newaddress_tests:main()`. Note how the testing file does `-include("salesdata.hrl")`; your solution file (`newaddress.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 `newaddress.erl`, as the tests import `newaddress/3` from that file. Thus to use the record syntax and typing, your solution file `newaddress.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(newaddress).
-export([newaddress/3]).
-include("salesdata.hrl").
-import(salesdata, [store/2, group/2]).

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

3. (10 points) [UseModels] [Concepts] In Erlang, write a tail-recursive function

```
-spec count (char(), string()) -> integer().
```

that takes a `char()`, `What`, and a `string()`, `S`, and returns the number of times that `What` occurs in `S`. Examples are shown in Figure 5 on page 8.

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

```
% $Id: newaddress_tests.erl,v 1.3 2013/10/30 03:13:40 leavens Exp leavens $
-module(newaddress_tests).
-include("salesdata.hrl").
-import(salesdata,[salesdata/0]).
-import(newaddress,[newaddress/3]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0]).

main() ->
    dotests("newaddress_tests $Revision: 1.3 $", tests()).

tests() ->
    [eqTest(newaddress(#group{gname = "StartUP!", members = []},
                       "Downtown", "50 Washington Ave."),
            "==", #group{gname = "StartUP!", members = []}),
     eqTest(newaddress(#store{address = "The Mall", amounts = [10,32,55]},
                       "110 Main St.", "The Mall"),
            "==", #store{address = "110 Main St.", amounts = [10,32,55]}),
     eqTest(newaddress(
              #group{gname = "Target",
                     members = [#store{address = "The Mall", amounts = [10,32,55]}]},
                     "NewAddress", "OldAddress"),
            "==",
            #group{gname = "Target",
                   members = [#store{address = "The Mall", amounts = [10,32,55]}]}),
     eqTest(newaddress(
              #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(newaddress(
              #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 4: Tests for problem 2.

```
% $Id: count_tests.erl,v 1.1 2013/10/30 03:13:40 leavens Exp leavens $
-module(count_tests).
-import(count,[count/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("count_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(count($c,[]), "==", 0),
     eqTest(count($i, [$M,$i,$s,$s,$i,$s,$s,$i,$p,$p,$i]), "==", 4),
     eqTest(count($p, "Mississippi"), "==", 2),
     eqTest(count($p, "principles of programming parallel programs"), "==", 5),
     eqTest(count($., ".........................."), "==", 26)
    ].
```

Figure 5: Tests for problem 3.

Concurrent Actor Programming in Erlang

4. (15 points) [UseModels] [Concepts] Write, in Erlang, a module, `future`, that exports the functions `makeFuture/2` and `futureValue/1` and a type `future(T)`. It should thus start as follows.

```
-module(future).
-export([makeFuture/2,futureValue/1]).
-export_type([future/1]).
-type future(T) :: {future, pid(), T}.
```

We have selected a representation for the datatype `future:future(T)` for you. This tuple type contains the atom `future`, a process id of the process that is running the computation, and a value of type `T` to be used when exceptions, exits, or errors occur in the computation.

The two exported functions you must implement are described below.

- The function `makeFuture/2` takes a zero-argument function, `P` of type `fun(() ->T)` and value of type `T`, `ErrValue`, and returns a `future(T)` value. The process id in the returned future is the pid of a process that is computing the result of the call `P()` in parallel with the process that called `makeFuture`. (That is, `makeFuture/2` must spawn a new process to compute the value of the call `P()`.)
- The function `futureValue/1` takes a `future:future(T)`, `F`, as an argument, and returns the value that the function call in the process computed (when that process finishes computing it). However, if an exception, exit, or error occurs in the computation, it returns instead the `ErrValue` that is the third element of `F`.

Hint: use a simple server process, created by `makeFuture`, which does only the computation of `P()`, the transmission of the result, and which uses a `try` expression to catch exceptions, exits, and errors. This `try` expression should have the form:

```
try ...
catch
   throw:_ -> ...;
   exit:_ -> ...;
   error:_ -> ...
end
```

in order to catch all thrown exceptions, exits, and errors. (When you fill in the `...` parts, however, be sure to avoid code duplication.) The simple server will finish after it communicates the result back (in response to receiving a message); thus the process does not loop. You will need to use the RPC protocol shown in class to get the result from this process in `futureValue/1`.

There are tests shown in Figure 6 on the next page.

```
% $Id: future_tests.erl,v 1.4 2013/10/30 03:13:40 leavens Exp leavens $
-module(future_tests).
-import(future,[makeFuture/2,futureValue/1]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0]).

main() ->
    dotests("future_tests $Revision: 1.4 $", tests()).

slowfib(0) -> 0;
slowfib(1) -> 1;
slowfib(N) -> slowfib(N-1) + slowfib(N-2).

fibof30() ->
    makeFuture(fun() -> slowfib(30) end, -1).

-spec tests() -> [testing:testCase(integer())].
tests() ->
    F30 = fibof30(),
    [eqTest(futureValue(makeFuture(fun() -> 7 end,0)),"==",7),
     eqTest(futureValue(makeFuture(fun() -> slowfib(3) end,-1)),"==",2),
     eqTest(futureValue(makeFuture(fun() -> slowfib(4) end,-1)),"==",3),
     eqTest(futureValue(makeFuture(fun() -> slowfib(10) end,-1)),"==",55),
     begin
         F2 = makeFuture(fun() -> slowfib(13) end,-1),
         eqTest(futureValue(F2),"==",233)
     end,
     eqTest(futureValue(makeFuture(fun() -> 4000 + 20 end,0)),"==",4020),
     eqTest(futureValue(F30),"==",832040),
     % tests that use the error value (2nd argument to makeFuture):
     eqTest(futureValue(makeFuture(fun() -> throw(except) end,0)),"==",0),
     eqTest(futureValue(makeFuture(fun() -> 7/zero() end,99)),"==",99)
    ].

zero() -> 0.
```

Figure 6: Tests for problem 4.

5. (15 points) [UseModels] Write, in Erlang, a module `box`, 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:

   - **`{set, NewVal}`**, which makes the server continue with `NewVal` as its new value.
   - **`{Pid, get}`**, 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 7.

```
% $Id: box_tests.erl,v 1.4 2013/11/01 14:16:19 leavens Exp leavens $
-module(box_tests).
-import(box, [start/1]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0,bget/1,bset/2]).

main() ->
    dotests("box_tests $Revision: 1.4 $", tests()).

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

% The following functions are used for testing purposes.
% You don't have to implement them again.
bget(Pid) ->
    Pid!{self(), get},
    receive
        {value, Value} ->
            Value
    end.

bset(Pid, Value) ->
    Pid!{set, Value},
    Value.
```

Figure 7: Tests for problem 5.

6. (20 points) [UseModels] In Erlang, write a module `ebay`, whose function `start/1` takes an atom as an argument, prints (using `io:format`) the output "`Starting auction for` " followed by the atom passed to `start/1` and a newline, and then returns a pid for a server that acts as a simple electronic auction server. To keep things simple, the server will only hold an auction for the one item described by the argument to `start/1`, but which will be referred to below as "the item."

   This server understands several messages:

   - Messages of the form **`{Pid, bid, Amt, Info}`** message places a bid on the item, where `Pid` is the bidder's process id, `Amt` is a non-negative integer (the number of dollars bid), and `Info` is some information about the bidder (e.g., their name).
   - The message **`{Pid, finish}`**, contains ends the auction and does all notification of the bidders. When this finish message is received, the `Pid` in the message is sent the message `auction_closed`. And then the server sends to the `Pid` received in the earliest received `bid` message with the highest amount bid the message **`{you_won, Amt}`**, and all other `Pid`'s of bidders are sent the message `sorry` to tell them that their bid did not win the auction. (If there are no bids, then no such messages are sent.) In case of a tie (two or more bids with the same highest amount), the first `bid` message received with the highest amount wins.

     If any `bid` message is received after the `finish` message, it does not win, whatever the amount bid is, and the sender is immediately sent the message `auction_is_over`. You can assume that only one `finish` message is ever sent to the server's pid.
   - The **`{Pid, who_won}`** message is only processed by the server after the `finish` message is received. The sender's `Pid`, contained in the message, is sent the `Info` from the winning `bid` message, or the atom `no_one` if there were no bids received by the server.

   Note that, to prevent bidders from gaining information before the auction is over, none of the `sorry` messages should be sent before the server receives the `finish` message.

   Figure 8 on the next page and Figure 9 on page 14 contain some examples.

```
% $Id: ebay_tests.erl,v 1.7 2013/10/30 03:13:40 leavens Exp leavens $
-module(ebay_tests).
-import(ebay,[start/1]).
-import(testing,[eqTest/3,dotests/2]).
-export([main/0,client/3,finishAuction/1,clientStatus/1]).

main() ->
    dotests("ebay_tests $Revision: 1.7 $",tests()).

tests() ->
    tests1() ++ tests2() ++ testempty().

tests1() ->
    Server = ebay:start('Florida'),
    Nelson = client(Server, 10, nelson),
    Rubio = client(Server, 8, rubio),
    Mica = client(Server, 11, mica),
    Obama = client(Server, 50, obama),
    finishAuction(Server),
    [eqTest(whoWon(Server),"==", obama),
     eqTest(clientStatus(Obama),"==",won),
     eqTest(clientStatus(Nelson),"==",lost),
     eqTest(clientStatus(Rubio),"==",lost),
     eqTest(clientStatus(Mica),"==",lost),
     eqTest(checkClosed(Server),"==",auction_is_over)].

tests2() ->
    Server = ebay:start(secret_formula),
    Agent007 = client(Server, 100, agent007),
    Agent86 = client(Server, 86, max),
    Agent99 = client(Server, 99, agent99),
    Agent992 = client(Server, 99, agent992),
    Agent100 = client(Server, 100, agent100),
    Agent1002 = client(Server, 100, agent1002),
    Agent1003 = client(Server, 100, agent1003),
    Agent1004 = client(Server, 100, agent1004),
    finishAuction(Server),
    [eqTest(checkClosed(Server),"==",auction_is_over),
     eqTest(whoWon(Server),"==",agent007),
     eqTest(clientStatus(Agent007),"==",won),
     eqTest(clientStatus(Agent86),"==",lost),
     eqTest(clientStatus(Agent99),"==",lost),
     eqTest(clientStatus(Agent992),"==",lost),
     eqTest(clientStatus(Agent100),"==",lost),
     eqTest(clientStatus(Agent1002),"==",lost),
     eqTest(clientStatus(Agent1003),"==",lost),
     eqTest(clientStatus(Agent1004),"==",lost)].

testempty() ->
    Server = ebay:start(junk),
    finishAuction(Server),
    [eqTest(checkClosed(Server),"==",auction_is_over),
     eqTest(whoWon(Server),"==",no_one)].
```

Figure 8: Testing for problem Problem 6 on the previous page, part 1 of 2.

```
% Helping functions for testing below, not for you to implement.
-spec client(pid(), non_neg_integer(), atom()) -> pid().
client(Server, Amt, Info) ->
    TestPid = self(),
    CPid = spawn(fun() -> clientaction(Server, Amt, Info, TestPid) end),
    receive % Make sure it has sent its bid in before returning...
        {CPid, sent_bid} -> ok
    end,
    CPid.

-spec clientaction(pid(), non_neg_integer(), term(), pid()) -> none().
clientaction(Server, Amt, Info, TestPid) ->
    Me = self(),
    Server ! {Me, bid, Amt, Info},
    TestPid ! {Me, sent_bid},
    receive
        {you_won, _BidAmt} ->
            clientloop(won);
        sorry -> clientloop(lost)
    end.

clientloop(Status) ->
    receive
        {Pid, status} ->
            Pid ! Status,
            clientloop(Status)
    end.

-spec whoWon(pid()) -> term().
whoWon(Server) ->
    Server ! {self(), who_won},
    receive
        Info ->
             Info
    end.

-spec clientStatus(pid()) -> won | lost.
clientStatus(Client) ->
    Client ! {self(), status},
    receive
        Status ->
            Status
    end.

checkClosed(Server) ->
    Server ! {self(), bid, 6000, checkingClosed},
    receive
        Msg ->
             Msg
    end.
```

Figure 9: Testing for problem Problem 6 on page 12, part 2 of 2.

7. (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` any other atom (other than none), then `Event` is sent to all observers in the server's list of observers. After handling any needed notification of the observers, then the event detector continues with the state `NewState` and an unchanged list of observers.

   Figure 10 on the next page and Figure 11 on page 17 contain some examples.

Points

This homework's total points: 120.

References

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

```
% $Id: eventdetector_tests.erl,v 1.6 2013/10/30 03:13:40 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() ->
    {CountGoGos, CountGadgets, CountGGGs,
     AccumGoGos, AccGGGs, AccumMatches} = setup(),
    dotests("eventdetector_tests $Revision: 1.6 $",
            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 10: Testing for problem Problem 7 on the previous page, part 1 of 2.

```
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
    end.

% Get the state value of an event detector.
-spec getValue(pid()) -> any().
getValue(Pid) ->
    Pid ! {self(), state_value},
    receive
        {value_is, State} -> State
    end.

% 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 11: Testing for problem Problem 7 on page 15, part 1 of 2.