Exam 2012-04-13
Advanced Functional Programming

Uppsala University
Department of information technology

Sven-Olof Nyström

April 4, 2012

Please read the following instructions carefully:

- The exam is five hours.
- This is a closed book exam. No literature or notes are allowed.
- Use a separate sheet for each question, and write your anonymous exam code (*not* your name) on each sheet. Leave a margin. Don't write anything important in the top-left corner.
- Please write clearly. Keep in mind that if I cannot read an answer, I cannot give any points for it.
- If something is unclear, try to make reasonable assumptions and write them down. I will *not* come in and answer questions, but the invigolators (the persons responsible for watching you while you write the exam, *skrivvakterna*) have my phone number if there is any problem or question.

  I'll be prepared for calls between 9.30 and 10.30, so make sure you have read the exam by then. (I'll try to be available for phone calls during the entire exam period, but it may be harder to reach me at other times.)
- The last pages contain some simple program examples in Erlang, Common Lisp and Haskell.

Maximum points: 60. Normally, 30 points are required for a 3 (passing) grade, 40 for a 4, and 50 for a 5 (very good) grade.

Read the whole exam before you start. If you get stuck on one question, proceed to the next—the questions are not necessarily ordered in level of difficulty.

1. **Walking (10p)**

   Your task is to simulate a random walk. A process starts at position 0, receives commands 'left' or 'right' and increments or decrements the position accordingly. If a process reaches position -1 it becomes 'stuck' and will not move again.

   Examples:

   ```
   Input: right, right, right

   Final position: 3.


   Input: right, left

   Final position: 0


   Input: left, right

   Final position: -1
   ```

   There must also be a way to ask the process for its current position.

   As part of this problem you'll need to define an interface to the process, i.e., a set of functions that are to be used when one wants to start a process or communicate with the process.

2. **Messenger (10p)**

   Consider the Erlang program below (some library functions used are explained after the program).

   One node in the network of Erlang nodes runs a server which maintains data about the logged on users. The server is registered as `messenger`. Each node where there is a user logged on runs a client process registered as `mess_client`.

   Among the operations defined are

   - `logon(Name)`
     Allows a user to logon and choose a suitable Name. If the Name is already logged in at another node or if someone else is already logged in at the same node, login will be rejected with a suitable error message.
   - `logoff()`
     Logs off anybody at at node
   - `message(ToName, Message)`
     sends Message to ToName. Error messages if the user of this function is not logged on or if ToName is not logged on at any node.

   Questions:

   (a) There are two types of processes in this program. Which?

   (b) Which messages do the two types of processes accept?

(c) Describe the communication that occurs when a client

   i. logs on,
   ii. sends a message to another client
   iii. logs off

   (You may assume that the server and the other client are already spawned and the other client has logged on.)

```
-module(messenger).
-export([start_server/0, server/1, logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    server_node.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger,
             spawn(messenger, server, [[]])).


%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node},  %reject logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).
```

```
%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {From, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.
%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
  case lists:keysearch(To, 2, User_List) of
    false ->
      From ! {messenger, receiver_not_found};
    {value, {ToPid, To}} ->
      ToPid ! {message_from, Name, Message},
      From ! {messenger, sent}
    end.

%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
             ok
end.

%%% The client process which runs on each server node
client(Server_Node, Name) ->
  {messenger, Server_Node} ! {self(), logon, Name},
  await_result(),
  client(Server_Node).

client(Server_Node) ->
  receive
    logoff ->
      {messenger, Server_Node} ! {self(), logoff},
      exit(normal);
    {message_to, ToName, Message} ->
      {messenger, Server_Node} ! {self(), message_to, ToName, Message},
      await_result();
    {message_from, FromName, Message} ->
      io:format("Message from ~p: ~p~n", [FromName, Message])
  end,
  client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} ->  % Normal response
            io:format("~p~n", [What])
    end.
```

Some library functions.

- `lists:keysearch(To, 2, User_List)` looks for a tuple with second element equal to `Name` in `User_List`.
- `lists:keymember(Name, 2, User_List)` Returns `true` if such a tuple exists, otherwise `false`.
- `lists:keydelete(From, 1, User_List)` Deletes a tuple in `User_List` if its first element is equal to `From`.

3. if-let (10p)

Your task is to implement a macro `if-let`. An expression

```
(if-let x e1 e2 e3)
```

should behave like an ordinary `if`, i.e., `(if e1 e2 e3)`, but when the condition `e1` does *not* evaluate to `nil`, the variable `x` should be bound to the value of the condition.

Example: If we assume that the function `lookup` looks for a key in some table and returns `nil` if the key was not found, the following two functions should behave the same.

```
(defun foo (key table)
  (if-let x (lookup key table)
          (list x x)
          nil))

(defun foo1 (key table)
  (let ((x (lookup key table)))
    (if x (list x x)
        nil)))
```

4. dolist (10p)

Common Lisp has a simple construct for iterating over a list called `dolist`. `dolist` binds a variable to the elements of a list in order and stops when it hits the end of the list.

Example:

```
> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

Dolist always returns nil. Note that the value of `x` in the above example was never nil: the `NIL` below the `C` was the value that `dolist` returned, printed by the read-eval-print loop.

Here is another example of the use of `dolist`:

```
[49]> (defun my-reverse (list)
          (let ((new-list nil))   ;initially nil, or empty list
            (dolist (x list)
              (setf new-list (cons x new-list)))
            new-list))
MY-REVERSE
[50]> (my-reverse '(a b c d e f g))
(G F E D C B A)
```

Your task is to implement dolist as a macro.

5. Random walk, revisited (10p)

Again, your task is to implement a simulation of a random walk. Here, a function takes an initial position and a list of commands (`LEFT`, `RIGHT`) and produces a list of positions.

As previously. position `-1` is "sticky". Once the process has reached position `-1` it will remain there.

Make sure that you also handle infinite lists of input.

Note that you also need to definie a datatype with the values `LEFT` and `RIGHT`.

Example:

```
*Main> walk 0 [RIGHT, RIGHT]
[0,1,2]
*Main> walk 0 [RIGHT, RIGHT, LEFT]
[0,1,2,1]
*Main> walk 0 [LEFT, LEFT]
[0,-1,-1]
*Main> walk 0 [LEFT, LEFT,RIGHT]
[0,-1,-1,-1]
*Main> walk 0 [RIGHT, LEFT, RIGHT, RIGHT]
[0,1,0,1,2]
```

6. Some functions on infinite lists (10p)

Define the following functions so that they work even for lists that may be infinite.

(a) A function that takes two sorted lists of integers without duplicates and computes their *union*, i.e., the list of integers that occur in either list.

(b) A function that takes two sorted lists of integers without duplicates and computes their *intersection*, i.e., the list of integers that occur in both lists.

(c) A function that takes a sorted list of integers that may contain duplicates and returns a list with the duplicates removed.

Good Luck!

Sven-Olof

```
% Simple definitions of map, quicksort and factorial, to illustrate
% Erlang's list syntax, list comprehensions, if-expressions and case
% expressions.

map1(_F, []) ->
    [];
map1(F, [X|L]) ->
    [F(X) | map1(F, L)].

map2(F, L) ->
    [F(X) || X <- L].

quick([]) ->
    [];
quick([X|L]) ->
    quick([Y || Y <- L, Y<X ])
          ++ [X]
          ++  quick([Y || Y <- L, X=<Y ]).

fac1(X) ->
    if
        X<1 ->
            1;
        true ->
            X*fac1(X-1)
    end.

fac2(X) ->
    case X<1 of
        true ->
            1;
        false ->
            X*fac2(X-1)
    end.

% A simple example with processes
-module(counter).

-export([start/0, loop/1, increment/1, value/1, stop/1]).

start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(), value},
    receive
{Counter, Value} ->
    Value
    end.

stop(Counter) ->
```

```
    Counter ! stop.

loop(Val) ->
    receive
increment ->
    loop(Val+ 1);
{From, value} ->
    From ! {self(), Val},
    loop(Val);
stop ->
    true;
_ ->
    loop(Val)
    end.

;; Factorial written in a rather clumsy imperative style.

(defun fac1 (n)
  (let ((p 1))
    (block my-loop
      (loop
         (when (= n 0)
           (return-from my-loop p))
         (setf p (* p n))
         (setf n (- n 1))))))

;; A slightly less clumsy imperative factorial. (dotimes (i n) ...)
;; iterates over the values 0 .. n-1.

(defun fac2 (n)
  (let ((p 1))
    (dotimes (i n)
      (setf p (* p (+ i 1))))
    p))


;; Append two lists

(defun appendx (x y)
  (cond
    ((null x) y)
    (t (cons (car x) (appendx (cdr x) y)))))

;; Efficient list reversal using a help function.

(defun myrev (l)
  (labels
      ((rev-help (l acc)
         (cond
           ((null l) acc)
           (t (rev-help (cdr l) (cons (car l) acc))))))
    (rev-help l nil)))

;; A simple macro.
```

```
(defmacro my-if (condition then else)
  `(cond (,condition ,then)
         (t ,else)))
```

```
fak :: Integer -> Integer
fak 0 = 1
fak n = n * fak (n-1)

quicksort :: Ord a => [a] -> [a]
quicksort []     =  []
quicksort (x:xs) =  quicksort [y | y <- xs, y<x ]
                 ++ [x]
                 ++ quicksort [y | y <- xs, y>=x]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  xs     ys    = []

fib :: [Integer]
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

```
class Weight a where
    weight :: a -> Integer

instance Weight Integer where
    weight x = x

instance Weight Int where
    weight x = toInteger(x)

instance Weight [a] where
    weight l = toInteger(length l)
```