KTH Information and Communication Technology

ID1019
Johan Montelius

Programming II
2016-03-19 09:00-13:00
7.5 credits

Name: ______________________________________

Instructions

- You are not allowed to have any material besides pen and paper. Mobiles etc, should be left to the guards.
- All answers should be written in these pages, use the space allocated after each question to write down your answer.
- Answers should be written in English.
- You should hand in the whole exam.
- No additional pages should be handed in.

Grade

The exam is divided into a number of questions where some are a bit harder than others. The harder questions are marked with a star *points**, and will give you points for the higher grades. The exam is thus divided into basic points and points for higher grades. First of all make sure that you pass the basic points before engaging with the higher points.
Note that, of the 40 basic points only at most 34 are counted, the points for higher grades will not make up for lack of basic points. The limits for the grads are as follows:

- E: 24 basic points

- D: 30 basic points
- C: 34 basic points
- B: 34 basic points and 14 higher points
- A: 34 basic points and 20 higher points

The limits could be adjusted to lower values but not raised.

Gained points

Don't write anything here.

Question	1	2	3	4	5	6	Σ
Max B/H	4/-	10/2	2/6	4/2	4/4	16/10	40/24
B/H							

Total number of points:

Grade:

Name:____________________________________

1 Data structures and pattern matching

1.1 what is Y [2 points]

What is the resulting binding for `Y` i the following pattern matching expressions (each one by its own), in the case that the matching succeeds:

- `[X, Y|_] = [1,2,3]` **Svar:** `Y = 2`
- `[X,_ |Y] = [1,2]` **Svar:** `Y = []`
- `[X,Z,Y] = [1|[2|[3]]]` **Svar:** `Y = 3`
- `Z = 2, X = {foo, Z}, {_, Y} = X` **Svar:** `Y = 2`
- `X = 1, Z = [], Y = [X,Z]` **Svar:** `Y = [1,[]]`

1.2 String concatentation in constant time [2 points]

If we represent strings as lists of ASCII values it is easy to read from a string since the first element can be reached in constant time (it's in the beginning of the list). It is however rather cumbersome to concatenate two strings since this would be done using an append operations. We could represent a string as either 1/ a list of ASCII values or 2/ a structure that holds two strings. The advantage would be that one would then be able to concatenate two strings in constant time. The disadvantage would of course be that it sometimes is a bit harder to find the first character.
Give a description of this form of representing a string. You can decides exactly how it is represented; if possible, use the so called type notation.
Implement the function `concat/2` that takes two strings as arguments and returns the concatenated string.

Svar:

```
-type string() ::  [char()] | {string, string(), string()}.

concat(S1, S2) -> {string, S1, S2}.
```

2 Recursive functions

2.1 remove all sequences of repetiotions [2 points]

Assume that we have a list of elements and want to create a list consisting of the same elements in the same order but were we have removed repetitions of elements that come after each other. If we have the list:
`[1,2,2,3,1,2,4,4,4,2,3,3,1]`

Name:_______________________________________

We should create the list:

```
[1,2,3,1,2,4,2,3,1]
```

Note that we still have duplicates of elements, such as 2 in the example, but that these are not immediately after each other.
How is the function `reduce/1` implemented that has the above given properties?

Svar:

```
reduce([]) ->
    [];
reduce([A, A |Rest]) ->
    reduce([A|Rest]);
reduce([A|Rest]) ->
    [A|reduce(Rest)].
```

2.2 Caesar cipher [2 points]

A Caesar cipher is maybe the simplest form of encryption and is performed by replacing every character in a message by the character that is three positions earlier in the alphabet. The world “hej” is thus encoded as “ebg”. Assume that we only use the characters ’a’ to ’z’ and that the alphabet is a ring; that is, ’a’ is encoded as ’x’, ’b’ as ’y’ and ’c’ as ’z’. Space is encoded as space so the message “encoded you are” is encoded as “bkzlaba vlr xob”. The ASCII value for space is 32 (we can write `$ ` but it is a bit hard to see the space). The value for ’a’ is 97 and ’z’ has the value 122. Write the function `encode/1` that takes a string and returns the encoded versions.

Svar:

```
encode([]) ->
    [];
encode([32|Rest]) ->
    [32|encode(Rest)];
encode([X|Rest]) when X < 100 ->
    [X-3+26|encode(Rest)];
encode([X|Rest])  ->
    [X-3|encode(Rest)].
```

2.3 Three of a kind in poker [2 points]

Assume that we want to implement a program that plays poker. We have chosen to represent a “hand” as a list of five unordered cards.
Write a function `triss/1` that determines if we have a hand that holds three-of-a-kind (three cards with the same rank). The function should return `true`

Name:____________________________________

if we have a three-of-a-kinda otherwise `false`. You decide how cards are represented.
It could return `true` if we have four-of-a-kind of a so called “full house” (a three-of-a-kind and a pair) since both these hands contains a three-of-a-kind. You can use the library function `lists:filter/2` that takes a function and a list of elements and returned a list of those elements for which the function returns `true`.
As an example the call:

```
> lists:filter(fun(X) -> X > 3 end, [8,2,6,3]).
```

will return `[8,6]`.

Svar:

```
triss([]) ->
    false;
triss([{card, _, V}|Hand]) ->
    case lists:filter(fun({card, _, N}) -> N == V end, Hand) of
        [_, _ | _] ->
            true;
        _ ->
            triss(Hand)
    end.
```

2.4 *merge sort* better than *quick sort* [2 points]

The algorithm *merge sort* is based on that you first divide a list into two equal parts, sort the two parts and then concat the two sorted lists. The algorithm can be implemented as follows:

```
msort([]) -> [];
msort(A) ->
   {L1, L2} = split(A),
   merge(msort(L1), msort(L2)).
```

Assume the list contains integers. How do you implement the function `merge/2`?

Svar:

```
merge([], R) ->
    R;
merge(L, []) ->
    L;
merge([H1|T1]=L, [H2|T2]=R) ->
    if
      H1 < H2 -> [H1|merge(T1,R)];
      true -> [H2|merge(L,T2)]
    end.
```

Name:_______________________________________

2.5 from a heap to a list [2 points]

Assume we have a so called “heap” that is represented as a binary tree. A heap is either empty, that we represent by the atom `nil`, or a node that consist of a one element and two branches that are both heaps, `{node, Element, Left, Right}`.
An important property of a heap is that the smallest element is always found in the root of the tree. The second smallest element is either in the left or right branch but since both these are heaps it will be in the root of either branch.
Implement a function `heap_to_list/2` that take a heap as argument and returns a ordered list of all elements. You can not use and library functions but you can use something that you just have written ;-).

Svar:

```
heap_to_list(nil) -> [];
heap_to_list({heap, E, Left, Right}) ->
    L = heap_to_list(Left),
    R = heap_to_list(Right),
    [E | merge(L, R)].
```

2.6 remove the smallest [2 points*]

The big advantage of a heap is that we always will find the smallest element in the root. It might not be trivial to remove the smallest element and rearrange the remaining elements to form a new heap. The solution is however rather simple if you think recursively.
Implement a function `pop/1`, that takes a heap and returns `{ok, Value, Rest}` where `Value` is the smallest element of the heap and `Rest` a new heap where the value has been removed. If you try to pop from an empty heap the function should return `false`.

Svar:

```
pop(nil) -> false;
pop({heap, V, L, nil}) ->
   {ok, V, L};
pop({heap, V, nil, R}) ->
   {ok, V, R};
pop({heap, V, L, R}) ->
    {heap, VL, _, _} = L,
    {heap, VR, _, _} = R,
    if
       VL < VR ->
          {ok, VL, Rest} = pop(L),
          {ok, V, {heap, VL, Rest, R}};
       true ->
```

Name:______________________________________

```
            {ok, VR, Rest} = pop(R),
            {ok, V, {heap, VR, L, Rest}}
    end.
```

3 Evaluating expressions

We have during the course worked with describing how a language can be defined by formally describing which terms, expressions and data structures we have and how we by rules can describe what should happen when we evaluate expressions. The following questions assume that we have defined a small language given the guidelines we have presented.

3.1 evaluating an expression [2 points]

Evaluate the following expressions, assume that:

$$\sigma = \{X/a, Y/\{a, b\}\}$$

- $E\sigma(a) \rightarrow$ **Svar:** a

- $E\sigma(\{X, X\}) \rightarrow$ **Svar:** $\{a, a\}$

- $E\sigma(Y) \rightarrow$ **Svar:** $\{a, b\}$

3.2 and, or and xor [2 points*]

It would be very nice if we in the language could use built in Boolean operators. To handle this we would extend the syntax of the language and also add rules for how these new constructs should be evaluated.
To make things simple we write all Boolean expressions with parenthesis so that the associations are clear. The operands are '&' for and, '|' for or and 'x' for xor. We of course also want to have the two Boolean values `true` and `false`. We want to be able to write sequences as:
`A = true, B = false, ((A & B) | (A x B))`

Since we want to extend the expressions we can handle we change the descriptions of <*expr*> to also include Boolean expressions.

$\langle expr \rangle$ ::= ... | $\langle bool \rangle$

Now we only need to describe what the Boolean expressions <*bool*> look like using a BNF grammar, what does the description look like?

Svar:

Name:______________________________________

$$\begin{array}{lll}\langle \textit{bool} \rangle & ::= & \text{true} \mid \text{false} \mid \\ & & \text{'('}\ \langle \textit{expr} \rangle\ \text{'\&'}\ \langle \textit{expr} \rangle\ \text{')'} \mid \\ & & \text{'('}\ \langle \textit{expr} \rangle\ \text{'|'}\ \langle \textit{expr} \rangle\ \text{')'} \mid \\ & & \text{'('}\ \langle \textit{expr} \rangle\ \text{'x'}\ \langle \textit{expr} \rangle\ \text{')'}\end{array}$$

We also need rules that describe what to do when evaluating a Boolean expression. How do we describe the new rules for the evaluation function E? In this description you should use the notation $\wedge$, $\vee$ and $\oplus$ to describe what should be done.

Svar:

- $E\sigma((e_1 \ \&\ e_2) \rightarrow E\sigma(e_1) \wedge E\sigma(e_2)$
- $E\sigma((e_1 \mid e_2) \rightarrow E\sigma(e_1) \vee E\sigma(e_2)$
- $E\sigma((e_1 \text{ x } e_2) \rightarrow E\sigma(e_1) \oplus E\sigma(e_2)$

3.3 lambda [4 points*]

Assume that we have extended our language to also handle lambda expressions. We have a syntax for them and evaluate them using the following rules:

- $E\sigma((\text{fun(vars) -> sequence end}) \rightarrow \textit{closure}(\textit{param}, \textit{sequence}, \theta)$

Here *param* is a sequence of variable identifiers and θ the subset of σ that holds the variable bindings for the free variables in *sequence*.
The question is now what to do when we apply our *closure* on a sequence of arguments.

- $E\sigma(\textit{closure}(\textit{param}, \textit{sequence}, \theta)(\text{args})) - > ...?$

Svar: We shall evaluate a sequence of expressions that we have in our closure but it has to be done in the right environment.

$$E\sigma(\textit{closure}(\textit{param}, \textit{sequence}, \theta)(\text{args})) - > E\theta'(\textit{sequence})$$

Detta gäller om *param* är en sekvens av i parametrar $p_1, ..p_i$, args är en sekvens av uttryck $a_1, ...a_i$ och

$$E\sigma(a_i) - > s_i$$

Omgivningen θ' är unionen av bindningar p_i/s_i och θ,

Svar:

Name:____________________________________

4 Complexity

In the answers to the following questions, make sure that you describe what n is and justify your answer.

4.1 traverser a tree [2 points]

Assume that we have a binary tree where a node is represented as either `{leaf, E}` or `{tree, Left, Right}`. Assume that the tree is balanced - what is the asymptotic time complexity to traverse the tree using the following function:

```
traverse({leaf, E}) -> [E];
traverse({node, Left, Right}) ->
    traverse(Left) ++ traverse(Right).
```

Svar: You need to traverser the tree wich is an $O(n)$, where n is the number of nodes in the tree, operation but also perform an append operation in each node. The cost of the append operation depends on the length of the list and the result is therefore:

$$O(n \lg(n))$$

4.2 maybe better ... or [2 poäng]

Is there any difference if we instead write like this:

```
traverse(Tree) -> traverse(Tree, []).
traverse({leaf, E}, Sofar) -> [E|Sofar];
traverse({node, Left, Right}, Sofar) ->
    traverse(Left, traverse(Right, Sofar)).
```

Svar:
Traversal is $O(n)$, where n is the number of nodes in the tree. The work at each node is constant.

$$O(n)$$

4.3 traverse a heap [2 points*]

In question we turned a heap into a lists. What is the asymptotic time complexity for this operation? Assume that the heap is balanced so we're not interested in the extreme situation where we have all elements in one branch.

Name:____________________________________

Svar:
The depth of the recursion is $O(lg(n))$ where n is the number of elements in the heap. In every recursion we split the heap in two and do a merge operation. The merge operation is $O(l)$ where l is the number of elements in the two lists that we merge. The size of the lists will be cut is half in each recursion but we have two recursive calls and the amount of work at each level then remains $O(n)$. In the end we have:

$$O(n \lg(n))$$

5 Concurrency

5.1 atomic swap [2 points]

Implement a procedure `new/1` that creates a process that can work as a memory cell. The argument to the procedure is the initial value of the cell. The process should handle two messages: `{swap, New, From}` and `{set, New}`. In the first case, the process that requested the update, `From`, receive a message in return `{ok, Old}`, where `Old` is the value the cell had before the update. In the second case, no messages is returned.

Svar:
```
new(Value) ->
   spawn(fun() -> cell(Value) end).

cell(Old) ->
  receive
    {swap, New, From} ->
       From ! {ok, Old},
       cell(New);
    {set, New} ->
       cell(New)
  end.
```

5.2 spin-lock [2 points]

A not so efficient way of implementing a lock is a so called *spin-lock*. The idea is to try to take the lock and keep trying until the lock is taken. Assume that we have implemented the function `new/1` in the previous question and we want to use the cell to implement a spin-lock.
Implement three procedures: `create/0`, `lock/1` and `release/1`. The procedure `create/0` should return a lock that `lock/1` and `release/1` can use. The procedures `lock/1` and `release/1` shall return `ok`.

Svar:
```
create() ->  new(open).
```

Name:____________________________________

```
lock(Cell) ->
   Cell ! {swap, taken, self()},
   receive
     {ok, open} ->
         ok;
     {ok, taken} ->
         lock(Cell)
   end.

release(Cell) ->
   Cell ! {set, open}, ok.
```

5.3 a semaphor [4 points*]

A spin-lock might do, but it more convenient to have a lock in form of a so called *semaphore*. A semaphore is a construction where you can request a lock and the lock will be given to you when it is available. If the lock is taken, you have to wait but you don't have to do anything or even be aware of that you're waiting. A semaphore is also more general in that it can allow more than one process to hold the lock but it has a maximum value on how many processes. If this value is 1 we call it a binary semaphore.
How shall we implement a semaphore in Erlang? We could implement it as a process that can take two messages: `{request, From}` and `release`. A process that that sends a request message will if/when there are locks available, receive a messages `granted` in return. When the process is done in the critical section it should send a release message to the semaphore that then can hand the resource to the next process in line.
Implement the function `new/1` that creates a semaphore with the given functionality. The argument to the procedure is the number of resources that the semaphore controls.

Svar:

```
new(N) -> spawn(fun() -> semphore(N) end).

semphore(0) ->
  receive
    release ->
       semphore(1)
  end;
semphore(N) when N > 0 ->
  receive
    {request, From} ->
       From ! granted,
       semphore(N-1);
```

Name:_______________________________________

```
    release ->
       semphore(N+1)
  end.
```

6 Programming

6.1 a small calcylator [total 6 + 2 points]

You shall implement a small calcylator that should handle arithmetic expressions with addition and subtraction over integers.

6.1.1 represent and evaluate [6 points]

You should first give a representation of arithmetic expressions and then implement the function `eval/1`, that takes an expressions and returns the result of evaluating the expression.

Svar:
Assume an integer is represented as `{int, I}`, where `I` is an integer and the arithmetic operations are represented as: `{add, A, B}` och `{sub, A, B}`.

```
-type expr() :: {int, int()}  | {add, expr(), expr()} | {sub, expr(), expr()}.
```

The function `eval/1` can now be defines as follows:

```
eval({int, I}) -> I;
eval({add, A, B}) -> eval(A) + eval(B);
eval({sub, A, B}) -> eval(A) - eval(B).
```

6.1.2 add an environment [2 points*]

Assume that we also want to handle variables in our arithmetic expressions. We must extend our representation to also describe variables and then implement a function `eval/2` that takes an expression and an environment and returns an answer.
To make things simpler we assume that we already have implemented a module `env`, that exports a function `lookup/2` that takes an identifier and an environment and returns either `{Id, Value}` of there is a binding for the variable or `false` if there is no binding.
The function `eval/2` should return either `{ok, Res}` if the evaluation succeeds or `error` if we for example try to evaluate an expression with unbound variables. Implement the function `eval/2`.

Svar:
Vi extend the representation of expressions by adding a representation of variables.

```
-type expr() :: ... | {var, atom()}.
```

Name:______________________________________

Funktionen eval/2 kan nu definieras som följer:

```
eval({int, I}, _) -> {ok, I};
eval({var, Id}, Env) ->
    case env:lookup(Id, Env) of
       {Id, Val} -> {ok, Val};
       false -> error
    end;
eval({add, A, B}, Env) ->
    case eval(A, Env) of
       {ok, Aval} ->
           case eval(B, Env) of
             {ok, Bval} ->
                 {ok, Aval + Bval};
             error ->
                 error;
       error ->
           error
    end;
  :
  same for {sub, A, B}
```

6.2 a barber [total 6 + 2 points]

A classical example on a synchronization problem is how to handle customers in a barber shop. Assume that we have one barber that of course only can shave one customer at a time. We do however have three chairs in a waiting room so customers that enter the shop can take a seat and wait. A customer who enters the door and finds that all chairs are taken will turn in the door and take a walk before trying again. If you do turn in the door you will not be guaranteed to be shaved but if you sit down to wait you will be shaved when it's your turn and no one will be shaved before you if the enter the shop after you.

6.2.1 how to implment a waiting room [6 points]

When we implement this in Erlang it is natural to divide the problem in two processes: one that handles the waiting room and one that acts as the barber. The barber can ask the waiting room for the next customer and will then receive a *pid* of the next man to be shaved. The customer is asked to sit down in the barbers chair and after a while he will receive a message that the swing is complete.

The procedure that implements the barber could be implemented as follows:

```
barber(Waiting) ->
   Waiting ! {next, self()},
```

Name:________________________________

```
   receive
     {ok, Customer} ->
          Customer ! have_a_seat,
          cut_cut_cut_shave_hot_towel_talk(),
          Customer ! thatz_it,
          barber(Waiting)
   end.
```

The waiting room will answer greetings from curomers that enter the shop, and will either ask them to sit down and wait or tell them that it is full. How can the recursive function that describes the waiting room be implemented.

Svar:

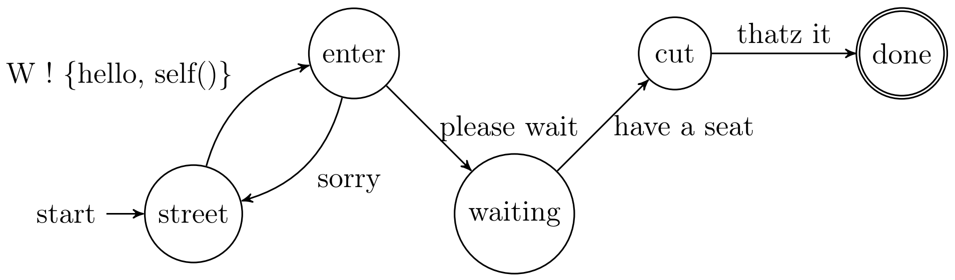
```
waiting([]) ->
   receive
     {hello, Customer} ->
        Customer ! please_wait,
        waiting([Customer])
   end;
waiting([Next|Rest]=Waiting) ->
   receive
     {hello, Customer} ->
        if length(Waiting) >= 3 -> Customer ! sorry;
           true -> waiting(Waiting++[Customer])
        end;
     {next, Barber} ->
        Barber ! {next, Next},
        waiting(Rest)
   end.
```

6.2.2 a customer as a state diagram [2 poäng*]

A customer is, when it is created, given access to the *pid* of a waiting room. He should announce his arrival as he enters the door. If he is asked to sit down he will wait his turn and will when the shaving is completed he can do what ever he likes, but we will let him terminate. Id all chairs are taken the customer should take a walk around the block and then return to the barber shop.
How would you draw the customer as a state diagram. Show all states that the customer can be in and what messages that will take him from one state to the other. Also describe what messages are sent as the customer enters new state.

Svar:

Name:____________________________________



6.3 a LDAP server [total 4 + 6 points]

LDAP, Lightweight Directory Access Protocol, is a protocol over which and can do queries towards a directory service. The protocol is implemented over TCP/IP and has a number of different messages to do queries, add, modify or delete information in a directory. LDAP is used by for example mail clients when they fetch addresses from s shared address register.

6.3.1 a simple solution [4 points]

Assuming we have a module, `ldap`, that can handle a request from a client, a LDAP server could have the following structure:

```
-define(Port, 67).

start() ->
    spawn(fun() -> init(?Port) end).

init(Port) ->
    case gen_tcp:listen(Port, [binary, {active, true}]) of
        {ok, Listen} ->
            handler(Listen),
            gen_tcp:close(Listen);
        {error, Error} ->
            io:format("ldap server: initialization failed: ~w~n", [Error]),
            error
    end.

handler(Listen) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            ldap:request(Client),
            handler(Listen);
        {error, Error} ->
            error
     end.
```

This structure has a limitation in how quickly it can handle requests. Describe this limitation and propose a simple solution that will allow us to handle more request per time unit.

Svar:
The structure will handle requests sequentially. Since each request probably requiers accesing databeses etc, there will be much idle time. To make better use of the server it is possible to handle multiple requests in parallel. This is acheived by spawning a process in each request.

```
handler(Listen) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            spawn(fun() -> ldap:request(Client) end),
            handler(Listen);
        {error, Error} ->
            error
     end.
```

6.3.2 two birds with one stone [3 poäng*]

We assume that you have solved the previous question in a satisfactory way. This would mean that the server would be able to handle many more request per time unit but there is an additional advantage. The solution most certainly gives us an advantage that has to do with the fact that `ldap` is a rather complex and maybe not totally reliable. What is the problem and why is the solution that you provide in the previous question also a solution to this problem?

Svar:
In the solution above the server will survive even of a single request craches. The client that sent teh request will never receive an answer but the other clients will be unaffected.

6.3.3 a lesser problem, maybe [3 points*]

The solution that you have proposed in the previous question is probably fine in most situations. It might however not have ideal properties if our server receives huge amount of requests in a very short time. What is the problem and how could you control the situation.
You don't have to implement a solution but describe what a solution might look like and what if the solution is adding any problems.

Svar:
If we have a solution where we create new processes for each request we could end up with too many processes. The server will be over loaded and the response times will decrease.

Name:______________________________________

To handle this we would set a limit on the number of request that we spawn. This could be implemented by a counter but then each process would have to report back that it has terminated.
We would also need to handle craches and end up with a more complex solution to protect us from something that might not be a problem.