KTH Information and
Communication Technology

ID1019
Johan Montelius

Programming II (ID1019) 2017-03-13 08:00-12:00

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:

Namn:_____

1 Data structures and pattern matching

1.1 what is Y [2 point]

What is the binding of `Y` in the following pattern matching expressions (each by it self) in the case where the matching succeeds:

- `[X,_,Y] = [1,2,3]` **Answer:** `Y = 3`
- `[X,Y,Z] = [1|[2,3]]` **Answer:** `Y = 2`
- `[Y|_] = []` **Answer:** *fails*
- `Z = 42, X = {Z, foo}, {Y, _} = X` **Answer:** `Y = 42`
- `X = 32, Z = [52], Y = [X|Y]` **Answer:** *fails*

1.2 A phone number [2 points]

We can represent a phone number by a number, a string or as list of individual numbers. For each of the alternatives, what is the advantage?

- `737652065` **Answer:** simpe to compare, works as a key
- "`737652063`" **Answer:** simple to print, can use numbers with characters "+46.."
- `[7,3,7,6,5,2,0,6,5]` **Answer:** good if we need to know the individual numbers

Answer: The difference between the last two is not really large, both are lists of integers.

2 Functional programming

2.1 increment a frequency [2 points]

Write a function `freq(Key, Freq)` that takes a key (an atom) and a list of frequencies represented by tuples `{Key, F}` of keys and their frequencies.

Namn:_____

The function should return an updated list where the frequency for the given key has been incremented by one.

Answer:

```
freq(Key, []) -> [{Key, 1}];
freq(Key, [{Key, F}|Rest]) -> [{Key, F+1}|Rest];
freq(Key, [{K, F}|Rest]) -> [{K, F}| freq(Key, Rest)];
```

2.2 create a frequency table [2 points]

Use the function `freq/2` and implement a function `fraq(Keys)` that takes a list of keys and returns a list of frequencies on the form above. You must only traverse the list once and can not use any built-in or library functions.

Answer:

```
fraq([]) -> [];
fraq([K|Rest]) ->
   freq(Key, fraq(Rest)).
```

Answer: We can write it tail-recursive but why.

2.3 an efficient queue [2 points]

Assume we want to implement a queue in Erlang. The queue should be such that we, in most cases, can take the first element from the queue in constant time and always add an element to the end of the queue in constant time. The trick we will use is to represent the queue by a tuple of two lists `{queue, First, Last}`. A queue with the elements 1 to 6 would then look like follows:

```
{queue, [1,2], [6,5,4,3]}
```

We have the first elements in the first list and teh rest of the element - in reveresed order - in the second list. If we now remove the first element from the queue we will get the queue:

```
{queue, [2], [6,5,4,3]}
```

If we then add 7 to the end of the queue, we get:

```
{queue, [2], [7,6,5,4,3]}
```

The problem of course arise when we try to remove an element from the queue but the first lists is empty. We then take the second list, reverse it, and places it first. This means that it sometimes takes longer time to take an element from the queue.

Namn:_____

Implement a function `new()`, `enqueue(Elem, Queue)` and `dequeue(Queue)`. The function `new/0` should return an empty queue, `enqueue/2` returns an updated queue and `dequeue/1` returns either `{ok, Elem, Updated}`, where `Updated` is the rest of the queue, or `fail` if the queue is empty. You may use a function `reverse/1` that reverses a list.

Answer:

```
new() -> {queue, [], []}.

enqueue(Elem, {queue, First, Last}) -> {queue, First, [Elem|Last]};

deque({queue, [], []}) -> fail;
deque({queue, [Elem|Rest], Last}) -> {ok, Elem, {queue, Rest, Last}};
deque({queue, [], Last}) ->
    [Elem|Rest] = reverse(Last),
    {ok, Elem, {queue, Rest, []}}.
```

2.4 append to queues [2 points*]

Some what more complicated is to append two queues. All elements in the first queue should be before any elements in the second queue and all elements in the two queues should maintain their order. Implement the function `app_queue/2` that appends two queues.

You can make use of a function `append/2` that appends two lists and `reverse/1` that revereses a list.

Answer:

one alternative

```
app_queue({queue, F1, L1}, {queue, F2, L2}) ->
   {queue, append(F1, reverse(L1)), append(L2, reverse(F2))}.
```

2.5 a better string [2 points]

Since Erlang represents strings as lists of characters it is quite costly to append two strings. We could make it easier if we represented strings as a tuple in the following way:

```
-type str() :: {str, list(char())} | {str, str(), str()}.
```

A `str()` is either a tuple with a regular string or a tuple containing two `str()`. We can now define a function that appends two `str()` in constant time; define the function.

Namn:_____

Answer:

```
str_append(A, B) -> {str, A, B}.
```

Namn:_____

2.6 straighten me out [2 points]

If we have our own representation of strings, `str()`, we could make use of a function that turns a `str()` into a regular string, that is a list of characters. Write a function `str_to_list/1`, you can use the function `append/2` that appends two lists.

Answer:

```
str_to_list({str, L}) -> L;
str_to_list({str, S1, S2}) ->
   append(str_to_lst(S1), str_to_lst(S2)).
```

3 Evaluating expressions

We have during the course worked with describing a language by formally describing the terms, expressions and data structures we have and how we can defines rules that describe what should happen when we evaluate expressions. The following questions assume that we have defined a small functional language along these guidlines.

3.1 pattern matching [2 points]

Perform the pattern matching below, give that: $\sigma = \{X/a, Y/\{a, b\}\}$.

- $P\sigma(\{Z, b\}, Y) \rightarrow$ **Answer:** $\{Z/a\} \cup \sigma$
- $P\sigma(Z, \{a, X\}) \rightarrow$ **Answer:** $\{Z/\{a, a\}\} \cup \sigma$
- $P\sigma(X, Y) \rightarrow$ **Answer:** fail

3.2 plus and minus [2 points*]

It would be very nice if we in the language could use built in arithmetic 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 arithmetic expressions with parenthesis so that the associations are clear. We want to be able to write sequences as:

Namn:_____

```
((10 - Y) + (8 + 3))
```

Assume that the argument to a arithmetic expression either is: an arithmetic expression, a number or a variable. How can we describe this using a BNF grammar? Assume that we have defined the expressions "<interger>" that describes the numbers and "<var>" that describes variables.

Answer:

$\langle arithm \rangle$ ::= $\langle integer \rangle$ | $\langle var \rangle$ | '(' $\langle arithm \rangle$ '+' $\langle arithm \rangle$ ')' | '(' $\langle arithm \rangle$ '-' $\langle arithm \rangle$ ')'

$\langle expression \rangle$ ::= ... | $\langle arithm \rangle$

We also need rules that describe what to do when evaluating an arithmetic expression. How do we describe the new rules for the evaluation function E?

Answer:

- $E\sigma((e_1 + e_2) \rightarrow E\sigma(e_1) + E\sigma(e_2)$
- $E\sigma((e_1 - e_2) \rightarrow E\sigma(e_1) - E\sigma(e_2)$

3.3 destructive assignment [4 points*]

The language that we have been working with is a functional language where we should not be able to assign new values to variables. Assume that we as an exercise should be able to assign new values to variables and allow constructs such as `X := 4`, that is we give the variable `X` the value 4 regardless if it has a value before. Which additions to the language do we need to make and how should we define the evaluation rules for the new construct?

Answer:

We need a syntax for assignment and we only allow these in sequences.

$\langle assignment \rangle$ ::= $\langle var \rangle$ ':=' $\langle expression \rangle$

$\langle sequence \rangle$ ::= ... | $\langle assignment \rangle$ ',' $\langle sequence \rangle$

Then we need to modify our evaluation of sequences so that it can handle assignment. We can do this by describing that the environment that we should continue with, θ, is an environment where we have removed any previous bindings for the variable, $\{v/u\}$, and then added the new binding $\{v/t\}$.

$$\frac{E\sigma(e) \rightarrow t \qquad \theta = \{v/t\} \cup (\sigma \setminus \{v/u\}) \qquad E\theta(\text{sequence}) \rightarrow s}{E\sigma(v := e, \text{sequence}) \rightarrow s}$$

Namn:_____

4 Complexity

In the answers to the questions below make sure to describe what n is and motivate why your answer is correct.

4.1 A regular queueu [2 points]

Assume that we implement a queue as a list where the first element in the list is the first element in the queue. We can then implement `enqueue/2` as follows:

```
enqueue(Elem, []) -> [Elem];
enqueue(Elem, [H|T]) -> [H|enqueue(Elem, T)].
```

Which is the asymptotic time complexity of the function?

Answer: The function has the asymptotic time complexity $O(n)$ where n is the length of the queue. We need to traverse the whole queue when adding an element.

4.2 a bit better [2 points]

The function in the previous question is not tail recursive but we can fix this using an accumulating parameter and the function `reverse/1`.

```
enqueue(Elem, Queue) -> enqueue(Elem, Queue, []).

enqueue(Elem, [], Sofar) -> reverse([Elem|Sofar]).
enqueue(Elem, [H|T], Sofar) -> enqueue(Elem, T, [H|Sofar]).
```

Given that `reverse/1` has the time complexity $O(n)$ then what is the time complexity of `enqueue/1`?

Answer: No difference, the complexity is still $O(n)$. We do traverse the lists twice but the complexity is the same.

4.3 flatten this and that [2 points*]

Assume that we have a list of lists that also can contain lists and we want to create a lists of all of the elements. We want to have the following behavior:

```
> flatten([[1,2], [[3], [4,5,6]]])
[1,2,3,4,5,6]
```

We can easily do this by first call `flatten/1` recursively and then use `append/2` to append the results.

```
flatten([]) -> [];
flatten([H|T]) ->
   append(flatten(H), flatten(T));
flatten(E) -> [E].
```

This is not tail recursive so we might want to implement it in the following way.

```
flatten(L) -> flatten(L, []).

flatten([], Done) -> Done;
flatten([H|T], Sofar) ->
   flatten(T, append(Sofar, flatten(H, [])));
flatten(E, Sofar) -> [E|Sofar].
```

A bit more complicated but now it is tail recursive. What is the down-side of doing like this?

Answer: The first solution has time complexity $O(n)$, if we don't have a pathological list. If we have n elements and they are evenly distributed among l list in the outermost list then each inner list will have n/l elements. We will then do l calls to append and do $O(n/l)$ work in each call. The total amount of work is then $O(n)$. If we have a pathological list of lists where the first list holds $n-1$ elements, then the append will take $n-1$ steps. If the first list now has $n-2$ element in its first list then that append required $n-2$ steps etc. Then we will have a complexity of $O(n^2)$.

The second solution has the complexity $O(n^2)$ since we there even in the general case grow `Sofar` to contain more and more elements. Every time we do a recursive step `Sofar` grows longer and the length is $O(n)$. We will do $O(n)$ number of calls to append so the total complexity is $O(n^2)$

5 Concurrency

5.1 Sesame open [2 points]

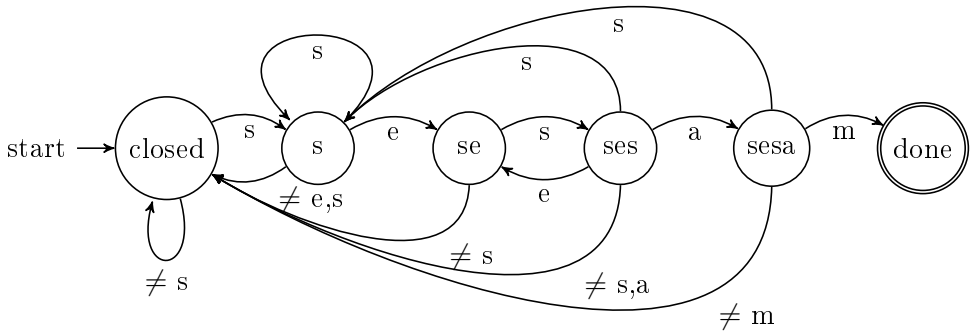
Implement a process that works as a combination lock. The process should start in a closed state and then receive messages one by one. If it receives a sequence "s e s a m" it should die (which does not make much sense but this is an exam).

Namn:_____

We should be able to send it any sequence of messages but should only open (die) if it receives the secret sequence. Below is the beginning of a state diagram that describes the process. Finish this diagram and also implement the process. Also implement a procedure `sesame/0` that starts the process.

Note that the process should reach its open state after a sequence "f o o s e s e s a m".

Answer:



Answer:

```
semsam() -> spawn(fun() -> closed() end).

closed() ->
  receive
    s -> s();
    _ -> closed();
  end.

s() ->
  receive
    e -> se();
    s -> s();
    _ -> closed();
  end.

se() ->
  receive
    s -> ses();
    _ -> closed();
  end.

ses() ->
  receive
    a -> sesa();
    e -> se();
    s -> s();
    _ -> closed();
  end.

sesa() ->
  receive
    m -> open();
    s -> s();
    _ -> closed();
  end.

open() -> done.
```

Namn:_____

5.2 Baba [2 points]

Assume that our safe, that is protected by the mechanism in the previous question, should enter a state *open* when one has sent the right sequence. In the open state it should be possible to send a message and receive a message in a reply. One should also be able to send a message that makes the process return to its locked state. How would the open state be implemented, what could messages look like and how should replies be delivered?

Answer: One suggestion:

```
open() ->
  receive
     {ali, From} ->
        From ! baba,
        open();
      close ->
       closed()
      _ ->
       open()
   end
```

5.3 parallell matrix multiplication [4 points*]

Assume that we have implemented matrix multiplications by the function `mult/2` below. How can we rewrite the function so that it will perform the multiplication in parallel. You may use built-in and library functions such for example `lists:map/2`, `make_ref/0` etc.

```
mult(A,B) ->
  rows(A, transpose(B)).

rows([], _) -> [];
rows([Row|Rows], Cols) ->
  [cols(Row, Cols) | rows(Rows, Cols)].

cols(R, Cols) -> lists:map(fun(C) -> dot(R,C) end, Cols).

dot([], []) -> 0;
dot([A|R], [B|C]) -> A*B + dot(R,C).

transpose([]) -> [];
transpose([[]|_]) -> [];
transpose(M) ->
```

Namn:_____

```
[[H || [H |_] <- M ] |  transpose([ T || [_| T] <- M ])].
```

Answer:

```
rows(Rows, Cols) ->
    Self = self(),
    Calc = fun(R) ->
             Ref = make_ref(),
             spawn(fun() -> C = cols(R, Cols), Self ! {Ref, C}, Ref end),
             Ref
            end,
    Refs = lists:map(Calc, Rows),
    Collect = fun(Ref) ->
                receive
                   {Ref, C} -> C
                end
               end,
    lists:map(Collect, Refs).
```

6 Programming

6.1 Huffman

6.1.1 Some what better than a list [6 points]

If we implement a decoder for a Huffman coded sequence, we can of course have a list with the mapping from sequences to characters. This is not very efficient and there is a much better way of representing the mapping. How can we represent the table to make the decoding more efficient. Explain in words and draw a picture, you do not have to write any code.

Answer: One can represent the table as a tree where leafs are the characters and the paths are the code sequences of the characters.

6.1.2 Is there a difference? [2 points*]

Assume that we should implement a program that decodes a Huffman coded file. The decoding is not a problem but we have two alternatives of how to implement the file handling. One solution looks like follows:

```
one(Encoded, Result, Table) ->
    {ok, Seq} = file:read_file(Encoded),
    Decoded = decode(Seq, Table),
```

Namn:_____

```
    {ok, Out} = file:open(Result, [write]),
    file:write(Out, Decoded).

decode([], _Table) ->
     [];
decode(Seq, Table) ->
     {Char, Rest} = decode_char(Seq, 1, Table),
     [Char|decode(Rest, Table)].
```

The second solutions looks similar:

```
two(Encoded, Result, Table) ->
    {ok, Seq} = file:read_file(Encoded),
    {ok, Out} = file:open(Result, [write]),
    decode(Sequence, Out, Table).

decode([], _Out, _Table) ->
     ok;
decode(Seq, Out, Table) ->
     {Char, Rest} = decode_char(Seq, 1, Table),
     file:write(Out, [Char]),
     decode(Rest, Table).
```

IS there an advantage to do it the first or second way?

Answer: In the second solution we have a tail recursive solution that will print the found character. If the file is very large we do not have to store the whole decoded file in memory before printing it. This will save space and possibly time. There might be an overhead of writing one character at a time so for smaller files the first solution might be quicker.

Namn:_____

6.2 Meta interpretor

6.2.1 eval_match/3 [6 points]

The code below is copied from the meta interpreter that we implemented. Explain the different parameters to the function and why we return the values we do.

```
eval_match({var, Id}, Str, Env) ->
    case env:lookup(Id, Env) of
        false ->
            {ok, env:add(Id, Str, Env)};
        {Id, Str} ->
            {ok, Env};
        {Id, _} ->
            fail
    end;
```

6.2.2 apply [2 points*]

Assume that we extend our interpreter to also handle functions. We can create so called *closures* that is represented by a sequence of variable identifiers, a *sequence* and an environment that provide the bindings of the free variables. We can then take this construct and apply it to a sequence of arguments. The code below is the part of the interpreter that would handle that operation.

```
eval_expr({apply, Expr, Args}, Env) ->
    case eval_expr(Expr, Env) of
        {ok, {closure, Par, Seq, Theta}} ->
            case eval_args(Args, Env) of
                error ->
                    error;
                EvaluatedArgs ->
```

Namn:_____

```
                    < What goes here? >

            end;
        error ->
            error
    end;
```

If we should evaluate the expression `F(X,42)` we should first evaluate `F` and hopefully obtain a *closure*. Then we evaluate the arguments `X` and `42` that hopefully goes well. Then we should do something, describe what we should do.

Answer: We should create a new environment given the environment `Theta` and the bindings of the variable identifiers in `Par` and the result of the evaluated arguments `EvaluatedArgs`.

```
                EvaluatedArgs ->
                    New = env:args(Par, EvaluatedArgs, Theta),
                    eval_seq(Seq, New)
```

6.3 A server

6.3.1 a server and a client [4 points]

When we are building a service that uses TCP, the server and client will use different API to create a connection. Who uses what, and how are the following functions used:

- `gen_tcp:connect(IP,Port,Opt)` **Answer:** Used by the client when connecting to a server. Will return a *communication channel*.
- `gen_tcp:accept(Socket)` **Answer:** Used by the server when it has created a *listening Socket*. Will return a *communication channel* when a client connects.
- `gen_tcp:listen(Port, Opt)` **Answer:** Used by the server to create a *listening socket*. The server has registered as owner of a port.

Namn:_____

6.3.2 quick but maybe too quick [4 points*]

If we implement a server we have below a solution of how to parallelize the implementation. We create an Erlang process for each incoming request and handles in parallel while receiving the next request.

```
handler(Listen) ->
    case gen_tcp:accept(Listen) of
  {ok, Client} ->
      spawn(fun() -> request(Client) end),
    handler(Listen);
  {error, Error} ->
    io:format("rudy: error ~w~n", [Error])
     end.
```

What is the down-side with this solution?

Answer: We risk creating more Erlang processes than we could handle. We do not have any control over how many are created.

6.3.3 binary [2 points*]

Assume that we read a message from a datagram socket that was opened in `binary` mode. We read a datagram with the following structure:

```
                                1  1  1  1  1  1  1
    0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                      ID                       |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
   :                                                :
```

Write a function `head/1`, that takes a binary and returns a structure that holds the identifier, the flags coded as integers and the rest of the datagram as a binary. We should be able to do the following:

```
> head(<<"aabbfoobar">>).
{24929,0,12,0,1,0,0,6,2,<<"foobar">>}
```

Answer:

```
head(<<Id:16, QR:1, Op:4, AA:1, TC:1, RD:1, RA:1, Z:3, RC:4, Rest/binary>>) ->
   {Id, QR, Op, AA, TC, RD, RA, Z, RC, Rest}.
```