KTH Information and
Communication Technology

ID1019
Johan Montelius

Programming II (ID1019) 2018-06-07 08:00-12:00

Name: _____

Instructions

- 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 */p*/*, 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.

- E: 12 basic points
- D: 15 basic points
- C: 18 basic points
- B: 20 basic points and 8 higher points
- A: 20 basic points and 10 higher points

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

Namn:_____ Persnr:_____

1 Lambda calculus [2p]

Evaluate the following lambda expressions:

- $(\lambda x \to x + x)4$ **Answer:** 8
- $(\lambda x \to (\lambda y \to y + 2 * x)2)4$ **Answer:** 10
- $(\lambda x \to (\lambda x \to x + x)(x + 2))4$ **Answer:** 12

2 Operational semantics [2p]

Given the rules for the operational semantics in the appendix, show step by step which rules are used and evaluate the following expressions:

Answer:

1.

$$\frac{:\mathtt{a} \equiv a}{E\{y/b\}(:\mathtt{a}) \to a}$$

2.

$$\frac{y/b \in \{y/b\}}{E\{y/b\}(\mathtt{y}) \to b}$$

3.

$$\frac{\textit{from 1} \qquad \textit{from 2}}{E\{y/b\}(\mathtt{\{:a,\ y\}}) \to \{a, b\}}$$

4.

$$\frac{E\{y/a, x/b\}(\mathtt{y}) \to a \qquad E\{y/a, x/b\}(\mathtt{x}) \to b}{E\{y/a, x/b\}(\mathtt{\{y,\ x\}}) \to \{a, b\}}$$

5.

$$\frac{\textit{from 3} \qquad S(\{y/b\}, \mathtt{\{y,\ x\}}) \to \{\} \qquad P\{\}(\mathtt{\{y,\ x\}}, \{a, b\}) \to \{y/a, x/b\} \qquad \textit{from 4}}{E\{y/b\}(\ \mathtt{\{y,\ x\}\ =\ \{:a,\ y\};\ \{y,x\}}) \to \{a, b\}}$$

6.

$$\frac{:\mathtt{b} \equiv b}{E\{\}(:\mathtt{b}) \to b}$$

$$\frac{\text{from 6} \qquad S(\{\}, \mathtt{y}) \to \{\} \qquad P\{\}(\mathtt{y}, b) \to \{y/b\} \qquad \text{from 5}}{E\{\}(\ \mathtt{y\ =\ :b;\ \{y,\ x\}\ =\ \{:a,\ y\};\ \{y,x\}}) \to \{a, b\}}$$

Namn:_____ Persnr:_____

3 Pattern matching [2 p]

Given the expressions below, what is the resulting environment in the cases where it succeeds?

a: [x, y | z] = [1, 2, 3] **Answer:** x = 1, y = 2, z = [3]

b: [x, y | z] = [1, [2, 3]] **Answer:** x = 1, y = [2,3] z = []

c: [x, y | z] = [1 | [2, 3]] **Answer:** x = 1, y = 2, z = [3]

d: [x, y | z] = [1 | [2, 3 | [4]]] **Answer:** x = 1, y = 2, z = [3,4]

e: [x, y | z] = [1, 2, 3, 4] **Answer:** x = 1, y = 2, z = [3, 4]

Namn:_____ Persnr:_____

4 Recursion

Fizz-Buzz [2 p]

We should implement a function `fizzbuzz/1` that given a number $n \geq 0$ returns a list of the n first elements in the fizz-buzz series. Fizz-buzz is a series from 1 to n where you replace all numbers that are a multiple of 3 by `:fizz`, those multiple by 5 by `:buzz` and those a multiple of both 3 and 5 by `:fizzbuzz`. The first five elements is thus: `[1,2,:fizz,4,:buzz]`.

You should implement the function `fizzbuzz/4` that helps us do this. The first argument is the next element in the list, the second tells us if we are done and the third and fourth keeps track of if the number is a multiple of 3 or 5. You are only allowed to use addition, no other arithmetic operation. You should not make the function tail recursive.

```
def fizzbuzz(n) do fizzbuzz(1, n+1, 1, 1) end
```

Answer:

```
def fizzbuzz(n, n, _, _) do [] end
def fizzbuzz(i, n, 3, 5) do [:fizzbuzz| fizzbuzz(i+1, n,   1,   1)] end
def fizzbuzz(i, n, 3, b) do [:fizz    | fizzbuzz(i+1, n,   1, b+1)] end
def fizzbuzz(i, n, f, 5) do [:buzz    | fizzbuzz(i+1, n, f+1,   1)] end
def fizzbuzz(i, n, f, b) do [i        | fizzbuzz(i+1, n, f+1, b+1)] end
```

Namn:_____ Persnr:_____

fairly balanced [2 p*]

Define a function `fairly/1` that returns either `{:ok, depth}` if a tree is fairly balanced and depth is the depth of the tree or `:no`. A tree is fairly balanced if the two branches both are fairly balanced and the difference in depth is at most one. You decide how the tree is represented and you can apart from `max/2` use a function `abs/1` that returns the absolute value.

Answer:

```
@type tree() :: nil | {:node, any(), tree(), tree()}

def fairly(nil) do {:ok, 0} end
  def fairly({:node, _, left, right}) do
    case fairly(left) do
      {:ok, l} ->
        case fairly(right) do
          {:ok, r} ->
            if abs(r-l) < 2 do
              {:ok, 1 + max(l,r)}
            else
              :no
            end
          :no -> :no
        end
      :no -> :no
    end
  end
```

Namn:_____ Persnr:_____

5 Time complexity

sorting a list [2 p]

If we have the below definition of a function that sorts a list, what is the asymptotic time complexity of the function?

```
def sort([]) do [] end
def sort([h|t]) do
  insert(h, sort(t))
end

def insert(e, []) do [e] end
def insert(e, [h|t]=l) do
  if e  < h do
    [e|l]
  else
    [h|insert(e, t)]
  end
end
```

Answer: The time complexity is $O(n^2)$ where n is the length of the list.

Namn:_____ Persnr:_____

a graph [2 p*]

Assume that we represent a directed acyclic graph and a function that searches the graph as described below. What is the time complexity to determine if an element is found in the graph? Assume that the nodes of the graph has at most k edges.

```
@type graph :: {:node, any(), [graph()]} |  nil

def search(_, nil) do :fail end
def search(e, {:node, e, _}) do :found end
def search(e, {:node, _, paths}) do
  List.foldl(paths,
    :fail,
    fn(p,a) ->
      case a do
        :found -> :found
        :fail ->
          search(e, p)
      end
    end)
end
```

Answer: The time complexity is $O(k^n)$ where n is the number of vertices in the graph and k the branching factor.

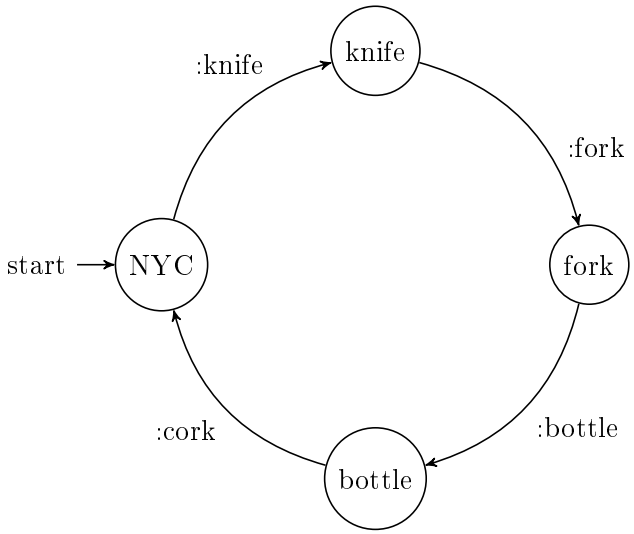
Namn:_____ Persnr:_____

6 Process description

A knnife, a fork ... [2 p]

Below you see a simple state diagram for a process. Each time the process enters the state `nyc` it should print "Hey Jim!" on stdout (using a call to `IO.puts("Hey Jim!")`). Messages that are not shown should remain in the message queue until they can be handled.

Implement the process and a function `dillinger/0` that starts the process.



Answer:

```
def dillinger() do
  spawn(fn() -> nyc() end)
end
```

```
def nyc() do
 IO.puts("Hey Jim!")
 receive do
   :knife -> knife()
 end
end
```

```
def knife() do
 receive do
   :fork -> fork()
 end
end
```

```
def fork() do
 receive do
   :bottle -> bottle()
 end
end
```

```
def bottle() do
 receive do
   :cork -> nyc()
 end
end
```

Namn:_____ Persnr:_____

tic-tac-toe [2 p]

Assume that we have the following definition of `first/1`, `second/1` and `third/1`.

```
def first(p) do
  receive do
     :tic ->
       second(p, [:tic])
     :tac ->
       second(p, [:tac])
  end
end

def second(p,all) do
  receive do
    :tic -> third(p, [:tic|all])
    :toe -> third(p, [:toe|all])
  end
end

def third(p, all) do
  receive do
     x ->  send(p, {:ok, [x|all]})
  end
end
```

What is the result when we evaluate the call `test/0`?

```
def test() do
   self = self()
   p = spawn(fn()-> first(self) end)
   send(p, :tic)
   send(p, :tac)
   send(p, :toe)
   receive do
     {:ok, res} -> res
   end
end
```

Answer: `[:tac, :toe, :tic]`

Namn:_____ Persnr:_____

parallel foldp/2 [2 p*]

We should do "fold" on the elements in a list but instead of doing it from the right or left we should do it in parallel. Given a list we should divide it in two equal parts, do parallel fold on each of the sub-lists and then apply our function on the result of the recursive calls. The challenge is that the two recursive calls should be in done in separate processes to utilize parallelism.

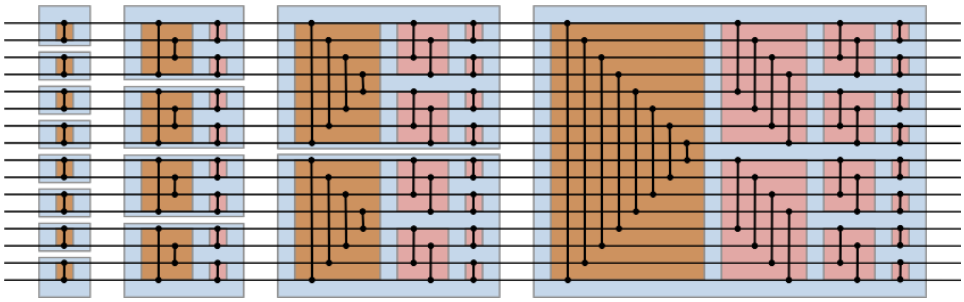
We do not have an initial value for our operation so we require that the list contain at least one element. To do foldp on a list of one element simply returns the element it self.

Answer:

```
def foldp([x], _) do x end
def foldp(list, f) do
  {l1, l2} = split(list, [], [])
  me = self()
  spawn(fn() -> res = foldp(l1, f); send(me, {:res, res}) end)
  spawn(fn() -> res = foldp(l2, f); send(me, {:res, res}) end)
  receive do
    {:res, r1} ->
      receive do
        {:res, r2} ->
          f.(r1,r2)
      end
  end
end

def split([], l1, l2) do {l1, l2} end
def split([h|t], l1, l2) do
  split(t, l2, [h|l1])
end
```

Namn:_____ Persnr:_____



Figur 1: A bitonic sort network of size 16.

7 Programming

In this task we should build a network for sorting. You use similar networks when you want to parallel sorting or have several streams of messages that should be sorted to several outgoing streams. The method we should use is called bitonic sorter and has recursive pattern.

In fig 1 you see a sorting network of size 16, it has 16 in-going streams to the left and 16 outgoing streams to the right. The idea is that 16 numbers go through the network so that the smallest number always exits at the uppermost stream and the largest number on the lowermost stream.

Where two streams are connected by a vertical line a comparison is performed. The smaller of the two values is forwarded on the upper stream and the larger on the lower.

We see that we have a recursive pattern. A network for two streams is of course trivial and only requires one comparison. To sort four streams we 1/ first do a recursive sorting of two streams, then 2/ an operation that in the picture is a brown area and the we will call *cross* followed by 3/ two operations marked in red that we will call *merge*.

The merge operation actually consist of a recursive repetition of a operation that we call *zipc*. If you look at how the network is sorting eight streams you see the two merge operations (one for the upper four streams and one for the lower four streams) internally consist of a zipc operation followed by two zipc operations on two streams each.

When we implement these networks the comparisons will be processes and the network defined by the route messages are sent. A comparison process will wait for two incoming messages, compare the the values and forward the messages in the graph.

Namn:_____ Persnr:_____

a comparing process [2 p]

The first you should do is to define a process that receives two messages, compares the values and forwards the smaller to one process and the larger to another. We must keep track of the messages and make sure that they belong to the same group so we keep track of which *epoch* that should be handled and only accept messages of the current epoch.

The messages to the process will have the following structure:

- `{:epoch, n, x}` : `x` is a value of two in epoch `n`
- `{:done, n}` : the process should terminate

When the process is started it expects two messages from epoch 0, two messages in epoch 1 etc. It will forward the messages in the same format and same epoch number. A "done-message" is forwarded to both processes to terminate the whole network.

Implement a function `comp/2` that takes two arguments, identifiers to the two processes to which the sorted values should be sent, starts a comparing process and returns the process identifier. The process should expect messages starting with epoch 0.

Answer:

```
def comp(low, high) do spawn(fn() -> comp(0, low, high) end)  end

def comp(n, low, high) do
  receive do
    {:done, ^n} ->
      send(low, {:done, n})
      send(high, {:done, n})
    {:epoc, ^n, x1} ->
      receive do
        {:epoc, ^n, x2} ->
          if x1 < x2 do
            send(low,  {:epoc, n, x1})
            send(high, {:epoc, n, x2})
          else
            send(low, {:epoc, n, x2})
            send(high,{:epoc, n, x1})
          end
          comp(n+1, low, high)
      end
  end
end
```

Namn:_____ Persnr:_____

sorter/1 [2 p]

Assume that we have a function `setup/1` that takes a list of process identifiers and starts all processes needed for the sorting network. The list of process identifiers are the processes, in order, that the network should deliver the sorted values to. The function returns a list of process identifiers that the values in each epoch should be sent to.

If we build a sorting network of size 4, the function will start six processes and return a list `[i1,i1,i2,i2]` where `i1` and `i2` are the identifiers of the two first processes in the network. You should not implement `setup/1` now, we assume that it works.

Implement a function `sorter/1` that takes a list of process identifiers, the processes to which we shall send the sorted epochs,. The function should accept two messages: a request to sort a number of values and one to terminate the execution.

- `{:sort, epoch}` : Where `epoch` is a list of values to be sorted. We assume that there are the same number of values as we have a sorting network for i.e. the values are sent to the in-going streams in the network (order does not matter). Don't forget to set the epoch number and increment a counter so that the next request is marked as the next epoch.
- `:done` : terminate the network, forward to both processes

To your help you can use the following functions:

- `each(list, fun)` Apply a function to each element in the list.
- `zip(list1, list2)` Return a list of tuples that consist of elements from the two lists. A call to Ett anrop till `zip([1,2],[:a,:b])` would return `[{1,:a},{2,::b}]`.

Use next page for answer.

Answer:

```
def start(sinks) do
  spawn(fn() -> init(sinks) end)
end

def init(sinks) do
  netw = setup(sinks)
  sorter(0, netw)
end

def sorter(n, netw) do
  receive do
    {:sort, this} ->
      each(zip(netw, this),
           fn({cmp, x}) -> send(cmp,{:epoc, n, x}) end)
      sorter(n+1, netw)
    :done ->
      each(netw, fn(cmp) -> send(cmp, {:done, n}) end)
  end
end
```

Namn:_____ Persnr:_____

setup for $n = 2$ [2 p]

Time for the tougher part but we start with something simple. You should now implement the function `setup/2` that takes two arguments, a number n and a list of n process identifiers. The process identifiers are the outgoing streams of the network, the function should start and connect all required processes in the network and return a list of entry points to the network. The function is then used to implement `setup/1` that is given below. We assume that n is an even multiple of 2 i.e. $2, 4, 8, ...$

```
def setup(sinks) do
  n = length(sinks)
  setup(n, sinks)
end
```

Since we're starting simple, you should in this question only handle the case when n is equal to 2 (which means that the list only has two elements). You should of course use a function that you have defined in a previous question to start a comparison process.

Answer:

```
def setup(2, [s1, s2]) do
  cmp = comp(s1, s2)
  [cmp, cmp]
end
```

Namn:_____ Persnr:_____

merge for $n = 2$ [2 p]

You now have all you need to start a network of size 2 but this is of course close to ridiculous so we should extend it to size of 4. You start by implementing a function that will set up a network to do a merge operation i.e. the red part in the Fig 1. The function `merge/2` takes a vale n and a list of n process identifiers. The function returns a list of process identifiers that make up the entry points of the merging network.

We start with something simple, you should implement `merge/2` so that it works for $n = 2$.

Answer:

```
def merge(2, [s1,s2]) do
  cmp1 = comp(s1,s2)
  [cmp1, cmp1]
end
```

Namn:_____ Persnr:_____

cross/2 [2 p]

Now for something slightly more complicated, the operation we call *cross* i.e. the brown operation in Fig 1. You should implement the function `cross/2` that takes two lists of process identifiers. The lists represent the $n/2$ upper and $n/2$ lower outgoing streams. The function should return a tuple consisting of two lists, the $n/2$ upper and $n/2$ lower entry points. The function should work for any n.

You can use the function `reverse/1` that reverses a list.

Answer:

```
def cross(low, high) do
  cross(low,  reverse(high), [])
end

def cross([], [], crossed) do
  {reverse(crossed), crossed}
end
def cross([l|low], [h|high], crossed) do
  cmp = comp(l, h)
  cross(low, high, [cmp | crossed])
end
```

Namn:_____ Persnr:_____

setup for $n = 4$ [2*]

We now assume that the functions `setup/2`, `merge/2` and `cross/2` works when $n = 2$, you should use these and extend `setup/2` to work if $n = 4$.

Answer:

```
def setup(4, [s1,s2,s3,s4]) do
  [m1, m2] = merge(2, [s1,s2])
  [m3, m4] = merge(2, [s3,s4])

  {[c1, c2], [c3, c4]} = cross([m1, m2], [m3, m4])

  [i1, i2] = setup(2, [c1, c2])
  [i3, i4] = setup(2, [c3, c4])

  [i1, i2, i3, i4]
end
```

Namn:_____ Persnr:_____

merge/2 [2*]

Time to extend `merge/2` to be able to handle any value of n $(2, 4, 8, ...)$. The function now becomes a recursive function where you in each recursion perform what we called a *zipc operation*. The zipc operation is the transformation marked with red in Fig 1 so start by defining a function `zipc/2` that performs this operation on two set of lists of length $n/2$ and returns the entry points that you need.

You can use the arithmetic operation `div/2` that performs integer division and the library function `split/2` that takes list and a number n and returns a tuple of two sub-lists, the n first elements and the rest.

Answer:

```
def merge(n, sinks) do
  n = div(n,2)
  {sink_low, sink_high} = split(sinks, n)
  merged_low =  merge(n, sink_low)
  merged_high =  merge(n, sink_high)
  zipced = zipc(merged_low, merged_high)
  zipced ++ zipced
end

def zipc([], []) do [] end
def zipc([l|low], [h|high]) do
  cmp = comp(l,h)
  [cmp | zipc(low, high)]
end
```

Namn:_____ Persnr:_____

a network for 2^k [4*]

Now it is time to extend `setup/2` to handle arbitrary values of n $(2, 4, 8, ...)$. Extend the function with the general case, use `div/2`, `split/2` and of course the functions `merge/2` and `cross/2` that we now assume works for all n.

Answer:

```
def setup(n, sinks) do
  n = div(n,2)
  {sink_low, sink_high} = split(sinks, n)
  merge_low = merge(n, sink_low)
  merge_high = merge(n, sink_high)
  {cross_low, cross_high} = cross(merge_low, merge_high)
  in_low = setup(n, cross_low)
  in_high = setup(n, cross_high)
  in_low ++ in_high
```

Namn:_____ Persnr:_____

Appendix - operational semantics

pattern matching

$$\frac{a \equiv s}{P\sigma(a, s) \to \sigma} \qquad \frac{a \not\equiv s}{P\sigma(a, s) \to \text{fail}}$$

$$\frac{v/t \notin \sigma}{P\sigma(v, s) \to \{v/s\} \cup \sigma} \qquad \frac{v/t \in \sigma \qquad t \not\equiv s}{P\sigma(v, s) \to \text{fail}}$$

$$\frac{v/s \in \sigma}{P\sigma(v, s) \to \sigma} \qquad \frac{}{P\sigma(\_, s) \to \sigma}$$

$$\frac{P\sigma(p_1, s_1) \to \sigma' \wedge P\sigma'(p_2, s_2) \to \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \theta}$$

$$\frac{P\sigma(p_1, s_1) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}} \qquad \frac{P\sigma(p_1, s_1) \to \sigma' \wedge P\sigma'(p_2, s_2) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}}$$

scoping

$$\frac{\sigma' = \sigma \setminus \{v/t \quad | \quad v/t \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \to \sigma'}$$

expressions

$$\frac{a \equiv s}{E\sigma(a) \to s} \qquad \frac{v/s \in \sigma}{E\sigma(v) \to s} \qquad \frac{v/s \notin \sigma}{E\sigma(v) \to \bot}$$

$$\frac{E\sigma(e_1) \to s_1 \qquad E\sigma(e_2) \to s_2}{E\sigma(\{e_1, e_2\}) \to \{s_1, s_2\}} \qquad \frac{E\sigma(e_i) \to \bot}{E\sigma(\{e_1, e_2\}) \to \bot}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \theta \qquad E\theta(\text{sequence}) \to s}{E\sigma(p = e; \text{sequence}) \to s}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \text{fail}}{E\sigma(p = e; \text{sequence}) \to \bot}$$

$$\frac{E\sigma(e) \to \bot}{E\sigma(p = e; \text{sequence}) \to \bot}$$