
KTH Information and
Communication Technology

**ID1019**
Johan Montelius

# Programming II (ID1019)
# 2019-03-08

### Instructions

- All answers should be written in these pages, use the space allocated after each question to write down your answer (not on the back side).
- Answers should be written in English.
- You should hand in the whole exam.
- No additional pages should be handed in.

### Grade

The exam consists of two parts. The first five questions are about basic functional programming: pattern matching, recursion, immutable data structures etc. The first part is the basic requirement to pass the course:

- FX: 7 points
- E: 8 points

The second part, questions 6-9, are about: semantics, higher-order functions, complexity, processes etc. The higher grades are based only on these questions but is only given (and only corrected) if the basic part has been answered satisfyingly (8 out of 10 point).

- D: one question correctly answered
- C: two questions correctly answered
- B: three questions correctly answered
- A: all questions correctly answered

Namn:_____ Persnr:_____

# 1 expand a coded sequence [2p]

Implement a function, `decode/1`, that takes a coded sequence and returns the decoded sequence. A coded sequence is represented by a list of tuples `{char, n}` where `char` is an element in the decoded sequence and `n` the number of consecutive occurrences.

Example: `decode([{:a, 2}, {:c, 1}, {:b, 3}, {:a, 1}])` should give the answer `[:a, :a, :c, :b, :b, :b, :a]`.

**Answer:**

```
def decode([]) do [] end
def decode([{elem, 0}|code]) do
  decode(code)
end
def decode([{elem, n}|code]) do
  [elem | decode([{elem, n-1} | code])]
end
```

# 2 zip/2 [2p]

Implement a function `zip/2` that takes two lists, `x` and `y`, of the same length and returns a list where the i'th element is a tuple `{xi, yi}`, of the i'th elements of the two lists.

Example: `zip([:a,:b,:c], [1,2,3])` returns `[{:a,1}, {:b,2}, {:c,3}]`.

**Answer:**

```
def zip([], []) do [] end
def zip([xi|xs], [yi|ys]) do
  [{xi,yi}|zip(xs, ys)]
end
```

Namn:_____ Persnr:_____

## 3 the balance in a tree [2p]

Implement a function, `balance/1`, that takes a tree and returns a tuple, `{depth, imbalance}`, where `depth` is the depth of the tree (the empty tree has depth 0) and `imbalance` is the largest *imbalance* in the tree.

The *imbalance* of a tree is max of:

- the imbalance of the right branch,
- the imbalance of the left branch, and
- the difference in depth of the branches.

The *imbalance* of an empty tree is 0.

Example: a tree with branches of depth 2 and 4 with imbalance 0 and 1 has a depth of 5 and imbalance of 2 since the two branches differ by depth 2.

Example: a tree with branches of depth 3 and 4 with imbalance 1 and 3 has a depth of 5 but a imbalance of 3 since the one branch has a imbalance of 3.

You can use the function `max/2` and `abs/1` to take the maximum of two numbers and the absolute value of a number.

Trees are represented as follows:

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

**Answer:**

```
def balance(:nil) do {0,0} end
def balance({:node, _, left, right}) do
  {dl, il} = balance(left)
  {dr, ir} = balance(right)
  depth = max(dl, dr) + 1
  imbalance = max(max(il, ir), abs(dl-dr))
  {depth, imbalance}
end
```

Namn:_____ Persnr:_____

# 4 Evaluate expressions [2p]

Implement a function `eval/1` that takes an arithmetic expression and returns its value. Arithmetic expressions are represented as follows (with the natural interpretation):

```
@spec expr() :: integer() |
                {:add, expr(), expr()} |
                {:mul, expr(), expr()} |
                {:neg, expr()}
```

Example `eval({:add, {:mul, 2, 3}, {:neg, 2}})` should return `4` since $2 * 3 + -2 = 4$.

**Answer:**

```
def eval({:add, x, y}) do
  eval(x) + eval(y)
end
def eval({:mul, x, y}) do
  eval(x) * eval(y)
end
def eval({:neg, x}) do
   - eval(x)
end
def eval(x) do
  x
end
```

Namn:_____ Persnr:_____

## 5 Gray coding [2p]

Implement a function `gray/1` that takes one argument `n`, a number greater than zero, and generates a list of so called Gray codes for the bit sequences of length `n`.

Example: `gray(3)` should return:

```
[[0, 0, 0],
 [0, 0, 1],
 [0, 1, 1],
 [0, 1, 0],
 [1, 1, 0],
 [1, 1, 1],
 [1, 0, 1],
 [1, 0, 0]]
```

A list of Gray codes have the property that two sequences after each other differ in exactly one position. The regular way of encoding binary numbers does not fulfill the requirement since two numbers after each other can differ in many positions (binary 3 and 4 is "011" and "100"). This sounds complicated but there is a simple solution - we do it recursively.

The list of Gray codes for sequences of length zero is the list that only contains the empty sequence `[[]]`.

To generate a list with Gray codes of length `n` then:

- generate the list of Gray codes of length `n-1`
- reverse the list to obtain a reversed copy
- update the original to a list where we have added `0` to the beginning of all codes
- update the copy to a list where we have added `1` to the beginning of all codes
- create the resulting list by appending the two lists

You can use the builtin function `reverse/1` to reverse a list and `append/2` to append two lists.

You might want to implement a function `update/2` that takes a list of codes and a value (0 or 1) and returns a list of the updated codes.

Write your answer on the next page.

Namn:_____ Persnr:_____

Answer to question 5.

**Answer:**

```
def gray(0) do [[]] end
def gray(n) do
  g1 = gray(n-1)
  r1 = reverse(g1)
  append( update(g1, 0), update(r1, 1))
end

def update([], _) do [] end
def update([h|t], b) do
  [[b|h]| update(t,b)]
end
```

Namn:_____ Persnr:_____

# 6 Formal semantics [P/F]

## lambda calculus

Assume that we have a simple functional language that is very similar to the one that we have been working on in the course. The difference is that the only literals that we have are Roman numerals, we have no compound structures but have builtin operations that can add numbers.

Assume that we have the sequence `x = III; y = x + x; y + II` show:

- how the expression could be written using the lambda calculus (change the Roman numerals to Arabic numbers) and
- how the lambda expression could be reduced, step by step, to a value.

**Answer:**

$$(\lambda x \rightarrow (\lambda y -> y + 2)x + x)3$$

$$(\lambda y \rightarrow y + 2)3 + 3$$

$$(\lambda y \rightarrow y + 2)6$$

$$6 + 2$$

$$8$$

## operational semantics

The rules of the operational semantic of the language is given in the Appendix. Show, step by step, how the given sequence is evaluated to a number.

Write your answer on the next page.

Namn:_____ Persnr:_____

Answer to second part of question 6

**Answer:**

$$\frac{\mathtt{y}/6 \in \{\mathtt{x}/3, \mathtt{y}/6\}}{E\{\mathtt{x}/3, \mathtt{y}/6\}(\mathtt{y}) \to 6} \qquad \frac{\mathtt{II} \equiv 2}{E\{\mathtt{x}/3, \mathtt{y}/6\}(\mathtt{II}) \to 2}$$

$$\frac{E\{\mathtt{x}/3, \mathtt{y}/6\}(y) \to 6 \quad E\{\mathtt{x}/3, \mathtt{y}/6\}(\mathtt{II}) \to 2 \quad 6+2=8}{E\{\mathtt{x}/3, \mathtt{y}/6\}(\mathtt{y\ +\ II}) \to 8}$$

$$\frac{\mathtt{x}/3 \in \{\mathtt{x}/3\}}{E\{\mathtt{x}/3\}(\mathtt{x}) \to 3}$$

$$\frac{E\{\mathtt{x}/3\}(\mathtt{x}) \to 3 \quad E\{\mathtt{x}/3\}(\mathtt{x}) \to 3 \quad 3+3=6}{E\{\mathtt{x}/3\}(\mathtt{x+x}) \to 6} \qquad \frac{\mathtt{y}/n \notin \{\mathtt{x}/3\}}{P\{\mathtt{x}/3\}(\mathtt{y}, 6) \to \{\mathtt{x}/3, \mathtt{y}/6\}}$$

$$\frac{E\{\mathtt{x}/3\}(\mathtt{x+x}) \to 6 \quad S(\{\mathtt{x}/3\}, \mathtt{y}) \to \{\mathtt{x}/3\} \quad P\{\mathtt{x}/3\}(\mathtt{y}, 6) \to \{\mathtt{x}/3, \mathtt{y}/6\} \quad E\{\mathtt{x}/3, \mathtt{y}/6\}(\mathtt{y\ +\ II}) \to 8}{E\{\mathtt{x}/3\}(\mathtt{y\ =\ x\ +\ x;\ y\ +\ II}) \to 8}$$

$$\frac{\mathtt{III} \equiv 3}{E\{\}(\mathtt{III}) \to 3} \qquad \frac{\mathtt{x}/m \notin \{\}}{P\{\}(\mathtt{x}, 3) \to \{\mathtt{x}/3\}}$$

$$\frac{E\{\}(III) \to 3 \quad S(\{\}, \mathtt{x}) \to \{\} \quad P\{\}(\mathtt{x}, 3) \to \{\mathtt{x}/3\} \quad E\{\mathtt{x}/3\}(\mathtt{y\ =\ x\ +\ x;\ y\ +\ II}) \to 8}{E\{\}(\mathtt{x\ =\ III;\ y\ =\ x\ +\ x;\ y\ +\ II}) \to 8}$$

Namn:_____ Persnr:_____

# 7 a strem of fibonacci values [P/F]

Implement a function `fib/0` that returns an infinite sequence of all Fibonacci numbers. The sequence should be represented by a function that takes no argument and returns a tuple `{:ok, next, cont}` where `next` is the next Fibonacci number and `cont` is the continuation of the sequence represented as function with the same properties.

Example:

```
def test() do
   cont = fib()
   {:ok, f1, cont} = cont.()
   {:ok, f2, cont} = cont.()
   {:ok, f3, cont} = cont.()
   [f1,f2,f3]
end
```

`test()` should return `[1,1,2]`

Then implement a function `take/2` that takes a function, with the above described behaviour, and a number $n$, and returns `{:ok, first, cont}` where `first` is a list of the $n$ first element from the sequence and the `cont` is the rest of the infinite sequence represented as above. The function need not be tail recursive.

Example:

```
{:ok, _, cont} = take(fib(), 4); take(cont,5)
```

returns

```
{:ok, [5, 8, 13, 21, 34], ...}
```

Write your answer on the next page.

Namn:_____ Persnr:_____

Answer to question 7.

**Answer:**

```
def fib() do
  fn() -> fib(1,0) end
end

def fib(f1, f2) do
  {:ok, f1, fn() -> fib(f1+f2, f1) end}
end

def take(inf, 0) do
  {:ok, [], inf}
end
def take(inf, n) do
  {:ok, next, cont} = inf.()
  {:ok, rest, cont} = take(cont, n-1)
  {:ok, [next|rest], cont}
end
```

Namn:_____ Persnr:_____

# 8 list of factorial [P/F]

The function `fac/1`, that returns the factorial $n!$ of a number $n$, is given. Implement a function `facl/1`, that takes a number `n` greater than zero, and returns a list of numbers `[n!,(n-1)!, ... 1]`. The function should have a linear time complexity given `n`.

```
def fac(1) do 1 end
def fac(n) do
  n * fac(n-1)
end
```

**Answer:**

```
def facl(1) do [1] end
def facl(n) do
  rest = facl(n-1)
  [f| _] = rest
  [n*f| rest]
end
```

Namn:_____ Persnr:_____

## 9 parallel processing [P/F]

Assume we have a process that is defined as given below. Rewrite the code so that we can utilize a parallel hardware. Other processes that are using this process should see no difference, the only difference should be that it (hopefully) runs faster.

The function `doit/1` is deterministic and does not have any side effects. It is a heave computation that will take different time depending on the task. Think about the order of messages.

```
defmodule Proc do

  def start(user) do
    {:ok, spawn(fn() -> proc(user) end)}
  end

  def proc(user) do
    receive do
      {:process, task} ->
         done = doit(task)
         send(user, done)
         proc(user)
      :quit ->
         :ok
    end
  end

end
```

Write your answer on the next page.

Namn:_____ Persnr:_____

Answer to question 9.

**Answer:**

```
defmodule Proc do

  def start(user) do
    collector = spawn(fn() -> collector(user, 0) end)
    {:ok, spawn(fn() ->  para(collector, 0) end)}
  end

  def para(collector, n) do
    receive do
      {:process, task} ->
        spawn(fn() ->
          done = doit(task)
          send(collector, {:done, n, done})
        end)
        para(collector, n+1)
      :quit ->
        send(collector, :quit)
        :ok
    end
  end

  def collector(user, n) do
    receive do
      {:done, ^n, done} ->
        send(user, done)
        collector(user, n+1)
      :quit ->
        :ok
    end
  end
end
```

Namn:_____ Persnr:_____

# Appendix -operational semantics

## Roman numerals and addition

Our languages only literals are numbers written in Roman numerals: `I`, `II`, `III`, `IV`, ..., to describe the natural numbers: $1, 2, 3, 4....$ We have a relation, $\equiv$, that describes the mapping from Roman numerals to natural numbers. In the rules below we use the letter $r$ for a Roman numeral and $n$ and $m$ for a natural numbers

We also have a "built-in" addition and can add any two natural numbers to obtain their sum.

## pattern matching

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

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

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

## scoping

$$\frac{\sigma' = \sigma \setminus \{v/n \quad | \quad v/n \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \to \sigma'}$$

## expressions

$$\frac{r \equiv n}{E\sigma(r) \to n} \qquad \frac{v/n \in \sigma}{E\sigma(v) \to n} \qquad \frac{v/n \notin \sigma}{E\sigma(v) \to \perp}$$

$$\frac{E\sigma(e_1) \to n_1 \qquad E\sigma(e_2) \to n_2 \quad n_1 + n_2 = n}{E\sigma(e_1 + e_2) \to n} \qquad \frac{E\sigma(e_i) \to \perp}{E\sigma(e_1 + e_2) \to \perp}$$

$$\frac{E\sigma(e) \to n \quad S(\sigma, p) \to \sigma' \quad P\sigma'(p, n) \to \theta \quad E\theta(\text{sequence}) \to m}{E\sigma(p = e; \text{sequence}) \to m}$$

$$\frac{E\sigma(e) \to n \quad S(\sigma, p) \to \sigma' \quad P\sigma'(p, n) \to \text{fail}}{E\sigma(p = e; \text{sequence}) \to \perp} \qquad \frac{E\sigma(e) \to \perp}{E\sigma(p = e; \text{sequence}) \to \perp}$$