
KTH Information and
Communication Technology

**ID1019**
Johan Montelius

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

### Instruktioner

- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar (inte på baksidan).
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

### Betyg

Tentamen består av två delar. De första fem frågorna handlar om grundläggande funktionell programmering: mönstermatchning, rekursion, icke-modifierbara datastrukturer mm. Den första delen är grundkravet för att klara kursen:

- FX: 7 poäng
- E: 8 poäng

Den andra delen, frågorna 6-9, handlar om: semantik, högre-ordningens funktioner, komplexitet, processer mm. Betygsgränserna för de högre betygen baseras enbart på dessa frågor men ges endast (och rättas enbart) om den grundläggande delen har besvarats tillfredsställande (8 av 10 poäng):

- D: en fråga korrekt besvarad
- C: två frågor korrekt besvarade
- B: tre frågor korrekt besvarade
- A: alla frågor korrekt besvarade

Namn:_____ Persnr:_____

# 1 expandera en kodad sekvens [2p]

Implementera en funktion, `decode/1` som tar en kodad sekvens och returnerar den avkodade sekvensen. En kodad sekvens är representerad som en lista av tupler `{char, n}` där `char` är ett element i den avkodade sekvensen och `n` antalet förekomster av elementet i följd.

Exempel: `decode([{:a, 2}, {:c, 1}, {:b, 3}, {:a, 1}])` skall ge svaret `[:a, :a, :c, :b, :b, :b, :a]`.

**Svar:**

```
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]

Implementera en funktion `zip/2` som tar två listor, `x` och `y`, av samma längd och returnerar en lista där det i:te elementet är en tuple, `{xi, yi}`, av de i:te elementen i de båda listorna.

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

**Svar:**

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

Namn:_____ Persnr:_____

# 3 balansen i ett träd [2p]

Implementera en funktion, `balance/1`, som tar ett träd och returnerar en tuple `{depth, imbalance}`, där `depth` är djupet på trädet (det tomma trädet har djup 0) och `imbalance` är den största *obalans* som finns i trädet.

Ett träds obalans är max av:

- obalansen i dess högra gren,
- obalansen i dess vänstra gren, och
- skillnaden i djupet av dess två grenar.

Obalansen i ett tomt träd är 0.

Exempel: ett träd vars två grenar har djup 2 och 4 med obalans 0 och 1 har ett djup på 5 och en obalans på 2 eftersom dess två grenar skiljer sig i djup med 2.

Exempel: ett träd vars två grenar har djup 3 och 4 med obalans 1 och 3 har ett djup på 5 men en obalans på 3 eftersom en gren har en obalans på 3.

Du kan använda funktionerna `max/2` och `abs/1` för att räkna ut max av två värden och absolutbeloppet av ett värde.

Träden representeras som följer:

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

**Svar:**

```
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 Evaluera utttryck [2p]

Implementera funktionen `eval/1` som tar ett aritmetiskt uttryck och returnerar dess värde. Aritmetiska uttryck representeras som följer (med sin naturliga tolkning):

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

Exempel: anropet `eval({:add, {:mul, 2, 3}, {:neg, 2}})` skall returnera `4` eftersom $2 * 3 + -2 = 4$.

**Svar:**

```
  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-kodning [2p]

Implementera en funktion `gray/1` som tar ett argument `n`, ett tal större eller lika med noll, och genererar en lista av s.k. Gray-koder för bitsekvenser av längd `n`.

Exempel: `gray(3)` skall returnera:

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

En lista av Gray-koder har den egenskapen att två sekvenser efter varandra skiljer sig på exakt en position. Det vanliga sättet att koda binära tal uppfyller inte det kravet eftersom två tal efter varandra kan förändras på flera olika positioner (binärt 3 och 4 är "011" och "100"). Detta verkar komplicerat men det finns ett enkelt sätt att generera en lista av Gray-koder av längd $n$ - vi gör det rekursivt.

Listan av Gray-koder för sekvenser av längd noll är listan som innehåller endast den tomma sekvensen `[[]]`.

För att generera en lista med Gray-koder av längd `n` så:

- generera en lista av Gray-koder av längd $n - 1$,
- vänd på listan så att vi har en omvänd kopia,
- skapa en uppdatering av den första listan där vi lagt till 0 i början på alla koder,
- skapa en uppdatering av den omvända listan där vi lagt till 1 i början på alla koder och
- skapa den slutgiltiga listan genom att slå ihop de två nya listorna.

Du kan använda dig av en inbyggd funktion `reverse/1`, som returnerar en omvänd lista, och `append/2` för att slå ihop två listor.

Du vill med stor säkerhet implementera en funktion `update/2` som tar en lista av koder och ett tal (0 eller 1) och returnerar den uppdaterade listan av koder.

Svara på nästa sida.

Namn:_____ Persnr:_____

Svar på fråga 5.

**Svar:**

```
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 Formell semantik [P/F]

## lambdakalkyl

Antag att vi har ett enkelt funktionellt språk som är väldigt likt det som vi har titta på i kursen. Skillnaden är att de enda literaler vi har är romerska siffror, vi har inga sammansatta termer men vi har en inbyggd funktion som kan addera heltal.

Antag att vi har sekvenen `x = III; y = x + x; y + II` visa:

- hur uttrycket skulle kunna skrivas i lambdakalkylen (byt ut de romerska talen mot arabiska siffror) och
- hur uttrycket i lambdakalkylen reduceras, steg för steg, till ett värde.

**Svar:**

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

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

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

$$6 + 2$$

$$8$$

## operationell semantik

Reglerna för språkets operationella semantik ges i Appendix. Visa, steg för steg hur den ovan givna sekvensen evalueras till ett naturligt tal.

Svara på nästa sida.

Namn:_____ Persnr:_____

Svar på andra del av fråga 6

**Svar:**

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

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

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

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

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

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

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

Namn:_____ Persnr:_____

# 7 en ström av fibonacci-tal [P/F]

Implementera en funktion `fib/0` som returnerar en oändlig ström av alla fibonaccital. Strömmen skall representeras som en funktion som inte tar några argument och returnerar en tuple `{:ok, next, cont}`, där `next` är nästa fibonaccital och `cont` är de nästkommande talen representerade av en funktion med samma egenskaper.

Exempel:

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

`test()` skall returnera `[1,1,2]`

Implementera sedan en funktion `take/2` som tar en funktion med ovan nämnda egenskaper och ett tal $n$, och returnerar `{:ok, first, cont}` där `first` är en lista av de $n$ första talen och `cont` är resten av den oändliga sekvensen representerad som ovan. Funktionen behöver inte vara svansrekursiv.

Exempel:

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

returerar

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

Svara på nästa sida.

Namn:_____ Persnr:_____

Svara på fråga 7.

**Svar:**

```
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 lista av fakultet [P/F]

Funktionen `fac/1`, som returnerar fakulteten $n!$ av ett tal $n$, är given. Implementera en funktion `facl/1` som tar ett tal, `n` större än noll, och returnerar en lista av tal `[n!, (n-1)!, ... 1]`. Funktionen skall ha linjär tidskomplexitet med avseende på `n`.

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

**Svar:**

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

Namn:_____ Persnr:_____

## 9 parallell hantering [P/F]

Antag att vi en process som är definierad enligt nedan. Skriv om den så att vi kan utnyttja en parallell hårdvara. Övriga processer som finns i systemet skall inte märka någon skillnad, den enda effekten skall vara att saker (förhoppningsvis) går snabbare.

Funktionen `doit/1` är helt deterministisk och har inga sidoeffekter. Det är dock en tung beräkning som kan ta olika tid beroende på uppgiften. Tänk på ordningen av meddelanden.

```
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
```

Svara på nästa sida.

Namn:_____ Persnr:_____

Svar uppgift 9.

**Svar:**

```
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 -operationell semantik

## Romerska tal och addition

Vårt språks enda literaler är tal skrivna med romerska siffror: `I, II, III, IV...`, för att beskriva de naturliga talen : $1, 2, 3, 4....$ Vi har en relation, $\equiv$, som gäller mellan alla romerska tal och deras motsvarande naturliga tal. I reglerna nedan används bokstaven $r$ för att beteckna ett romerska tal och $n$ och $m$ för att beteckna de naturliga talen.

Vi har även en "inbyggd" addition och kan alltid addera två naturliga tal för att få deras summa.

## mönstermatchning

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

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

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

## scoping

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

## uttryck

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

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

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

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