



KTH Information and  
Communication Technology

**ID1019**

Johan Montelius

# Programmering II (ID1019)

## 2020-03-09

### 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 togglar [2p]

Implementera en funktion, `toggle/1` som tar en lista och returnerar en lista där element har bytt plats två-och-två. Om antalet element är udda så lämnas det sista elementet på sin plats.

Exempel: `toggle([:a, :b, :c, :d, :e, :f, :g])` skall ge svaret `[:b, :a, :d, :c, :f, :e, :g]`.

## 2 push och pop [2p]

Implementera en stack: föreslå en lämplig datastruktur och implementera sedan funktionerna `push/2` och `pop/1`. Funktionen `push/2` skall returnera den uppdaterade stacken och `pop/1` skall returnera antingen `{:ok, any(), stack()}` om det finns ett element på stacken eller `:no` om stacken är tom.

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

### 3 **flatten** [2p]

Implementera en funktion `flatten/1` som tar en lista av listor, som in sin tur kan vara listor av listor osv, och returnera en platt lista av alla element av de som finns in den ursprungliga listan i samma ordning. Du kan använda `++` för att slå ihop två listor.

Exempel, anropet: `flatten([1, [2], [[3, [4, 5]], 6]])` reurnerar: `[1, 2, 3, 4, 5, 6]`

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

## 4 h-värde [2p]

Ett h-värde kan används för att beskriva hur mycket och långt man springer. Ditt h-värde är det högsta tal  $h$  där du sprungit åtminstone  $h$  km åtminstone  $h$  gånger. Ditt h-värde är till exempel 12 om du sprungit 12 km eller längre 16 gånger men inte sprungit 13 km eller längre åtminstone 13 gånger. Det är relativt enkelt att komma upp i ett h-värde på tio, men lite tuffare att komma upp i 30.

Implementera en funktion `index/1` som givet en lista av tal som anger löpsträckor, räknar ut en löpares h-värde. Listan av löpsträckor kommer att vara ordnad med de längsta sträckorna först. Du kommer kunna räkna ut resultatet utan att behöva gå igenom listan flera gånger. Algoritmen är enkel:

- Den initiala uppskattningen av det sökta h-värdet är 0.
- Gå igenom listan och om nästa värde är högre än din nuvarade uppskattning,
  - så räkna upp uppskattningen med 1 och fortsätt annars,
  - så har du hittat det sökta h-värdet.

Exempel: `index([12,10,8,8,6,4,4,4,2])`

skall returnera: 5

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

## 5 kompaktare [2p]

Implementera en funktion `compact/1` som tar ett träd på formen nedan och returnerar ett träd där en nod med samma nycklar i sina två löv (eller om det bara finns ett löv), ersätts med ett löv. Funktionen skall fungera rekursivt så att ändringar propagerar mot roten.

Träden är representerade enligt följande, notera att vi endast har värden i löven:

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

Exempel:

```
compact({:node, {:leaf, 4}, {:leaf, 4}})
```

skall returnera: `{:leaf, 4}`

```
compact({:node, {:leaf, 5}, {:node, :nil, {:leaf, 4}}})
```

skall returnera: `{:node, {:leaf, 5}, {:leaf, 4}}`

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

## 6 Formell semantik [P/F]

I den formella semantiken har vi använt oss av en S-regel eller *scoping rule*. Regeln ger oss möjlighet att beskriva vad som skall hända när vi evaluerar en sekvens av mönstermatchningar.

Vi har i Elixir möjligheten att definiera en funktion och använda oss av mönster-matchning i huvudet av definitionen (se exemplet nedan). Hur skall vi beskriva semantiken för ett anrop om vi tillåter mönster i huvudet?

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

Exempel på att använda mönstermatchning i huvudet av en funktionsdefinition:

```
def member(_, []) do :no end
def member(x, [x|_]) do :yes end
def member(x, [_|h]) do member(x, h) end
```

Detta är hur vi definierade ett funktionsanrop om definitionen endast tilläts ha unika variabler i huvudet:

$$\frac{E\sigma(e) \rightarrow \langle v_1, \dots : \text{seq} : \theta \rangle \quad E\sigma(e_i) \rightarrow s_i \quad E\{v_1/s_1, \dots\} \cup \theta(\text{seq}) \rightarrow s}{E\sigma(e.(e_1, \dots)) \rightarrow s}$$

Hur förändras detta om vi tillåter funktionen att ha mönster i huvudet (vi antar att det bara finns ett huvud)?

$$\frac{E\sigma(e) \rightarrow \langle p_1, \dots : \text{seq} : \theta \rangle}{E\sigma(e.(e_1, \dots)) \rightarrow s}$$

Svara på nästa sida.

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

Svar på formell semantik:

## 7 Nästa primtal [P/F]

Du skall implementera en function `primes/0` som returnerar en funktion som representerar en oändlig sekvens av primtal. Funktionen som returneras skall, när den appliceras utan argument, returnera en tuple `{:ok, prime, next}` där `prime` är nästa primtal och `next` en funktion som representerar resten.

```
@type next() :: {:ok, integer(), ( -> next())}
```

```
@spec primes() :: ( -> next())
```

Ett sätt att lösa problemet är att implementera Eratosthenes såll. När man har hittat ett primtal (det första är 2) så ser man till att inte returnera något tal  $n$  som är jämt delbart med det primtalet ( $rem(n, p) == 0$ ). Nedan följer en bra start för denna algoritm, det är nu din uppgift att implementera `sieve/2`.

```
def primes() do
  fn() -> {:ok, 2, fn() -> sieve(2, fn() -> next(3) end) end} end
end
```

```
def next(n) do
  {:ok, n, fn() -> next(n+1) end}
end
```



Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

## 8 en bättre flatten [P/F]

Om man implementerar `flatten/1` på enklast möjliga sätt så får man en algorithm som inte har speciellt bra tidskomplexitet. Man kommer att slå ihop listor som efter hand blir längre och längre och komplexiteten blir kvadratisk med avseende på antalet element. Du skall nu implementera `flatten/1` men se till så att komplexiteten blir linjär med avseenden på antalet element i listorna.

Namn: \_\_\_\_\_ Persnr: \_\_\_\_\_

## 9 parallell map [P/F]

Du skall implementera en funktion `pmap/2` som fungerar som vanliga `map/2` men där varje element evalueras i en separat process och därmed kan köras parallellt.

Du skall använda dig av den vanliga `map/2` när du startar processer och när du samlar ihop resultatet.

Exempel: uttrycket nedan

```
pmap([1,2,3,4], fn(x) -> x + 2 end)
```

skall returnera: [3,4,5,6]