



KTH Information and  
Communication Technology

**ID1019**

Johan Montelius

# Programmering II (ID1019)

2015-06-11 08:00-11:00

**Namn:** \_\_\_\_\_

## Instruktioner

- Du får inte ha något materiel med dig förutom skrivmateriel. Mobiler etc, skall lämnas till tentamensvakten.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

## Betyg

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *poäng\**, och ger poäng för de högre betygen. Vi delar alltså upp tentamen in grundpoäng och högre poäng. Se först och främst till att klara de normala poängen innan du ger dig i kast med de högre poängen.

Notera att det av de 24 grundpoängen räknas bara som högst 20 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är enligt nedan.

- E: 14 grundpoäng
- D: 18 grundpoäng
- C: 20 grundpoäng

- B: 20 grundpoäng och 8 högre poäng
- A: 20 grundpoäng och 12 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt. Kursens slutbetyg kan bli högre om man ligger nära en gräns och har skrivit mycket bra rapporter.

**Erhållna poäng**

Skriv inte här, detta är för rättningen.

<b>Uppgift</b>	1	2	3	4	5	$\Sigma$
<b>Max G/H</b>	4/-	10/2	2/6	4/2	4/4	24/14
<b>G/H</b>						

**Totalt antal poäng:**

**Betyg:**

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

## 1 Datastrukturer och mönstermatchning

### 1.1 vad är Y [2 poäng]

Vad är bindningen för Y i följande mönstermatchningar (var för sig), i de fall där matchningen lyckas:

- $[Y|_] = [1,2,3]$
- $[X, _ |Y] = [1,2,3]$
- $[X,Y,Z] = [1|[2,3]]$
- $Z = 32, X = \{foo, Z\}, \{Y, _\} = X$
- $X = head, Z = tail, Y = [X,Y]$

### 1.2 tal, lista, sträng eller tuple [2 poäng]

Man kan välja att representera information med tal, listor, strängar eller tupler mm, lite beroende på hur informationen skall användas. Antag att vi vill representera ett schackparti och väljer mellan två olika representationer. Den första är en tupel av åtta tupler där de inre tuplerna representerar rader med 8 rutor. Varje ruta är representerad med en atom som beskriver vad som står på rutan (eller om den är tom):  $\{pjas, Färg, Typ\}$  eller  $tom$ . Färgen representeras med atomerna  $vit$  och  $svart$ , och pjäsernas typ av atomerna  $kung, dam, lopare, springare, torn, bonde$

Det andra sättet som vi har att välja på är två tupler som representerar positionerna av de vita och de svarta pjäser (varje tupel har 16 element). Varje pjäs representeras av en tupel  $\{pos, Rad, Kolumn\}$  om den finns på brädet eller atomen  $dead$  om den är bortplockad. Vilken för och nackdelar finns med de två olika representationerna?

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

## 2 Funktionell programmering

### 2.1 ASCII-svenska [2 poäng]

När jag började studera datavetenskap så hade vi bara 7-bitars ASCII. Terminalerna vi hade kunde inte skriva å, ä eller ö så vi använde }, { och | istället. Man vand sig ganska snabbt vid att lsa men det såg oölsbart ut för vanligt folk.

Skriv en funktion `seven/1` som tar en mening med riktiga å, ä och ö och som returnerar samma mening där vi ersatt dessa med }, { och |. Vi antar för enkelhetens skull att meningen vi har endast innehåller gemener (för den som undrar använde vi |, [ och \ för versaler). Tecknet 'å' skriver vi naturligtvis som `$å` och '}' som `$}` osv.

### 2.2 traversera ett träd [2 poäng]

Ett träd kan antingen vara tomt eller bestå av en nod med ett tal och en höger- respektive vänstergren, där grenarna är träd. Antag att trädet är ordnat så att alla tal som är mindre än nodens tal finns i dess vänstra gren och de större i dess högra gren.

Beskriv hur trädet är representerat och definierar en funktion `traverse/1`, som traverserar hela trädet och returnerar en ordnad lista av alla tal i trädet. Använd enklast möjliga algoritm dvs: traversera först vänstergrenen, sedan

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

högergrenen och slå sen ihop dem med nodens tal i mitten (du kan använda '++' för att göra *append*). Lägg till ett basfall och saken skall vara klar.

### 2.3 lite bättre [4 poäng]

Den första lösningen är inte så effektiv; vi skapar först en lista av alla element i vänstra grenen som vi sedan går igenom en gång till när vi skall göra *append*. Skapa en bättre version genom att definiera funktionen `traverse/2` som tar två argument: grenen som skall traverseras och *de element som skall stå efter elementen i den gren vi traverserar*. Idéen är att först traversera ett trädets högra gren och när vi har alla element i en lista så traverserar vi dess vänstra gren men då har vi ju redan en lista med element som alla är större än elementen i dess vänstar gren och då skall stå efter dessa. Om vi gör detta

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

rätt så slipper vi anropa `append` och får en betydligt mer effektiv algoritm.  
Definiera även `better/1` som anropar `traverse/2` med rätt argument.

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

#### **2.4 lägg till ett element [2 poäng]**

Antag att vi har representerar ett träd som i uppgiften ovan. Definierar en funktion `insert/2` som tar ett träd och ett element och returnerar ett träd där vi lagt till elementet på dess rätta plats så att trädet är ordnat.

#### **2.5 plocka bort ett element [2 poäng\*]**

Det blir lite svårare när vi skall plocka bort ett element ur trädet. Definiera en funktion `delete/2` som tar ett träd och ett element och returnerar ett träd där elementet har blivit bortplockat om det fanns. Till din hjälp får du en funktion `rightmost/1` som tar ett träd och returnerar en tuple `{Rest,`

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

$\text{Max}$ }, där  $\text{Max}$  är trädets största element (det som finns längst till höger) och  $\text{Rest}$  är det träd som blir kvar om det största elementet plockas bort.

### 3 Evaluering av uttryck

Vi har under kursen arbetat med att beskriva hur ett språk kan definieras genom att formellt beskriva vilka termer, uttryck och datastrukturer vi har och hur vi med hjälp av regler kan beskriva vad som skall hända när vi evaluerar uttryck. De följande frågorna antar att vi har definierat ett litet funktionellt språk enligt de riktlinjer vi gått igenom.

#### 3.1 evaluera ett uttryck [2 poäng]

Evaluera följande uttryck, antag att  $\sigma = \{X/a, Y/\{a, b\}\}$  .

- $E\sigma(a) \rightarrow$
- $E\sigma(\{X, X\}) \rightarrow$
- $E\sigma(Y) \rightarrow$



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

### 3.2 plus och minus [2 poäng\*]

Det vore väl rätt så bra om vi i språket kunde ha aritmetiska operationer så som addition och subtraktion av heltal. För att hantera detta skall vi först utöka språket och sen även definiera vilka regler som skall gälla vid evaluering.

För enkelhetens skull så skriver vi alla aritmetiska uttryck med parenteser så att vi har associationen helt klar. Vi vill kunna skriva uttryck som:

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

Antag att argumenten till ett aritmetiskt uttryck antingen är: ett aritmetiska uttryck, ett heltal eller en variabel. Hur kan detta beskrivas med hjälp av ett BNF-uttryck? Antag att vi har definierat uttrycken " $\langle \text{integer} \rangle$ " som beskriver heltalen och " $\langle \text{var} \rangle$ " som beskriver variabler.

Vi måste även ha en regel som beskriver vad som skall göras när vi skall evaluera ett aritmetiskt uttryck. Hur skall vi skriva en regel för evalueringsfunktionen  $E$ ?

### 3.3 [if-then-else som icke-strikta funktion 4 poäng\*]

Om man tillhandahåller en if-then-else konstruktion i sitt språk så görs de oftast som *icke-strikt* funktion. En icke-strikt funktion, har egenskapen att den inte nödvändigtvis returnerar  $\perp$  när evalueringen av ett av dess argument returnerar  $\perp$ .

Om alla funktioner i språket är *strikt* så har man fördelen att evalueringsordningen är fri; oavsett i vilken ordning som vi evaluerar argumenten så

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

kommer vi att komma fram till samma resultat. Vad skulle vi vinna på att definiera if-then-else som en icke-strikt funktion?

## 4 Komplexitet

I svaren till nedanstående frågor så var noga med att ange vad till exempel  $n$  är och motivera varför du anser att ditt svar är rätt.

### 4.1 traversera ett ordnat träd [2 poäng]

Vad är den asymptotiska tidskomplexiteten för funktionen `better/1` (dvs den som inte använde sig av `append`) som vi definierade förut?

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

#### 4.2 uppdatera ett träd [2 poäng]

Vad är den asymptotiska tidskomplexiteten för funktionen `insert/2` som vi definierade förut (antag att trädet är balanserat)?

#### 4.3 varianten med appen [2 poäng\*]

Vad är den asymptotiska tidskomplexiteten för `traverse/1` som använde sig av `append/2`? Detta är lite svårare att se och för att få poäng här skall du kunna motivera ditt svar väl.

## 5 Concurrency

### 5.1 ett enkelt lås [2 poäng]

Definiera en procedur `create/0` som skapar en process, och returnerar dess process identifierare (*pid*), som skall fungera som ett *lås* och bara låta en process åt gången hålla låset. Processen skall kunna ta emot och agera på följande två meddelanden:

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

- `{take, Pid}` : Låter en process, `Pid`, ta låset om det inte redan är taget och returnerar då ett meddelande `granted` till processen.
- `release` : Släpper låset så att en annan process kan ta låset.

## 5.2 använd låset [2 poäng]

För att processen som vi skapar med `create/0` skall fungera som ett lås, så förutsätter vi att alla processer verkligen *tar låset* innan de utför en kritisk operation.

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

Definiera en procedur `critical/1` som tar en pid av ett lås och som tar låset, utför en operation genom att anropa `do_it/0` och sedan släpper låset.

### 5.3 parallell merge-sort [4 poäng\*]

Antag att vi har en funktion `split/1` som delar en lista i två lika långa listor och att vi har en funktion `merge/2` som slår ihop två sorterade listor till en sorterad lista. Då kan vi implementerat merge-sort som följer:

```
sort([]) -> [];  
sort([_]=L0) -> L0;  
sort(L0) ->  
    {L1, L2} = split(L0),  
    S1 = sort(L1),  
    S2 = sort(L2),  
    merge(S1, S2).
```

Hur skulle vi kunna parallellisera denna algoritm? Vi vill kunna göra de två sorteringsoperationerna i separata processer. Antag att vi definierar en procedur `psort/1` som följer:

```
psort(L) ->  
    Self = self(),  
    psort(L, Self),  
    receive  
        {ok, S} ->  
            S  
    end.
```

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

Definiera proceduren `psort/2` så att sorteringen kan exekveras parallellt.