# Tentamen i DD1361 Programmeringsparadigm, 19 November 2007

Kursansvarig: Lars Arvestad

**Skriv tydligt! Skriv bara på en sida av pappret och behandla bara en uppgift per pappersblad. Ge dina svar tydliga motiveringar. Förklara dina funktioner och predikat! Lämna plats för kommentarer vid rättning.**

För varje baslabb du har redovisat ges 1 bonuspoäng. För godkänt, betyg E, krävs 20 poäng. Möjlighet till komplettering garanteras vid 18 poäng. Gränserna för högre betyg är 25, 30, 35, och 40 poäng.

**Ni som gick kursen innan avsnittet om syntaxanalys stoppades in behöver inte göra sista avsnittet på tentan.** Er tenta är alltså på 42 poäng. För er är gränserna godkänt vid 17 poäng, 25 poäng ger betyg 4, och vid 34 poäng ges betyg 5.

**Lösningsförslag kommer att hittas på kursens hemsida.**

**Godkända hjälpmedel:** Skrivmedel. "Prolog programming" av Brna. "Programming in Haskell" av Hutton, eller det tidigare särtrycket av samma bok.

**Lycka till!**

## Allmänna frågor

Samla gärna svaren på detta avsnitt på ett papper.

1. Vad innebär *strikt evaluering*? (2p)
2. Vad innebär *svansrekursion*? (2p)
3. Vad menar man med "first class functions"? (2p)
4. Vad innebär *Turingfullständighet*? (2p)
5. Varför rekommendarar man användning av *abstrakta datatyper*? (2p)

## Funktionell programmering i Haskell

6. (a) Vad menas med

```
\x -> x^2
```

   i Haskell? (1p)

   (b) Vad betyder följande uttryck?

```
f = foldl analyze 0
```

   Funktionen `analyze` definierar vi på annat håll, men din förklaring ska vara oberoende av vad den gör. (1p)

7. Implementera funktionen `intersperse :: a -> [a] -> [a]` som lägger in sitt första argument mellan varje grannpar i det andra elementet. Denna funktion finns i modulen `List`, men du måste implementera med de funktioner du hittar i `Prelude`.

   *Exempelkörning:*

   ```
   Hugs> intersperse '-' "KTH"
   "K-T-H"
   Hugs> intersperse (-1) [1,2..5]
   [1,-1,2,-1,3,-1,4,-1,5]
   Hugs> intersperse 17 []
   []
   Hugs> intersperse 17 [0]
   [0]
   ```

   Var nogrann med att det inte ska sättas in nåt först eller sist i listan! (3p)

8. Du vill arbeta med både ändliga och oändliga mängder av heltal i Haskell och har svårt att skilja på dem. Ändliga mängder kan du representera med listor, men för oändliga listor behöver du en generator, till exempel en funktion som

   ```
   ones = 1 : ones
   ```

   eller

   ```
   [1,2..]
   ```

   Problemet är bara att deras signatur inte skiljer sig från en listas typ, så du vill använda en egen datastruktur för att skilja på de två mängdklasserna.

   (a) Föreslå en enkel datastruktur `IntSet` i Haskell som hjälper dig göra skillnad på ändliga och oändliga listor. (1p)

   (b) Skriv ett predikat `finite :: MySet -> Bool` som avgör om om en mängd är ändlig eller ej. (1p) *Exempelkörning:*

   ```
   Hugs> finite mittTestData
   True
   ```

   (c) Till datastrukturen ska du definiera en funktion `cardinality` som hanterar både ändliga och oändliga mängder. Den ska svara på två sätt, beroende på vilken mängd det är, och detta kräver ytterligare en `data`-definition. Kardinalitet för ändliga mängder är antalet element i mängden, men oändliga mängder delas upp i olika storleksklasser. Eftersom de mängder vi representerar har en generator så är *uppräkningsbara*. Din funktion ska uppföra sig så här (på tänkta testdata):

   ```
   Hugs> cardinality ettTillTre
   SizeIs 3
   Hugs> cardinality ettTillTio
   SizeIs 10
   Hugs> cardinality allIntegers
   Countable
   ```

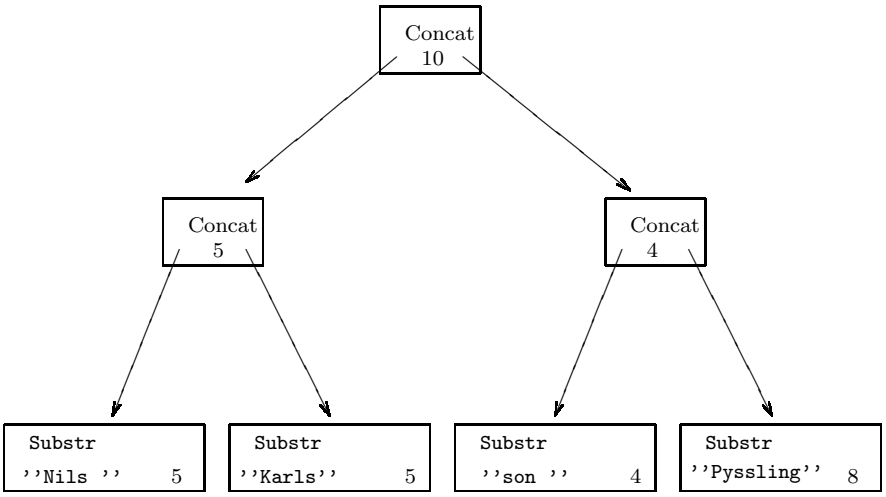
   Du behöver alltså inte skriva kod för att detektera om en mängd är ändlig eller oändlig. (2p)

9. Om man i ett datorprogram jobbar med längre texter kan det vara olämpligt att lagra texten i vanliga strängar. Att till exempel sätta in nya bokstäver mitt inne i en sträng kan kräva omfattande kopiering, och upprepade manipulering av textdata kan därför lätt bli ineffektivt. En föreslagen lösning är datastrukturen `Rope` ("ropes are stronger than strings", se Boehm, Atkinson och Plass, *Software–Practice and Experience*, 1995) som använder en trädstruktur där delsträngar lagras i löven. En konkatering av löven ger den egentliga strängen. I den här uppgiften betraktar vi en enkel implementation av `Rope` i Haskell.

```
data Rope = EmptyString          -- Längd = 0
          | Substr String Int    -- En delsträng
          | Concat Rope Int Rope -- Inre nod
          deriving Show
```

Första raden används för att lagra den tomma strängen. `Substr` används till löven i vår trädrepresentation och där lagras en sträng och dess längd. `Concat` sammanfogar två `Rope` och lagrar dessutom längden av det vänstra.

I bilden härunder finner du "Nils Karlsson Pyssling" lagrad i en `Rope`.




(a) Funktionen `words` har typsignaturen `String -> [String]`. Antag att jag nu skapar en typklass `Text` som har `String` och `Rope` som instanser. Hur bör typsignaturen på `words` ändras för att kunna arbeta på båda instanserna? (1p)

(b) Skriv funktionen `ropeLength :: Rope -> Int`. Du ska använda den längdinformation som finns lagrad i datatstrukturen: det är för ineffektivt att konvertera till en sträng/lista och ta längden av den. (2p)

*Exempelkörning:*

```
Hugs> ropeLength EmptyString
0
Hugs> ropeLength (Substr "KTH" 3)
3
Hugs> nils  -- Testdata från bilden!
Concat (Concat (Substr "Nils " 5) 5 (Substr "Karls" 5)) 10
(Concat (Substr "son " 4) 4 (Substr "Pyssling" 8))
Hugs> ropeLength nils
22
```

(c) Skriv funktionen `subRope :: Rope -> Int -> Int -> Rope` som returnerar en delsträng givet en `Rope`. Du får inte konvertera till `String`, ta fram delsträngen, och därefter konvertera tillbaka till `Rope`, utan delsträngen ska plockas fram direkt från datastrukturen. (4p)

*Exempelkörning:*

```
Hugs> subRope nils 0 4
Substr "Nils" 4
Hugs> subRope nils 5 8
Concat (Substr "Karls" 5) 5 (Substr "son" 3)
Hugs> rope2string (subRope nils 5 8)
"Karlsson"
```

Du kan strunta i felhantering. Använd `take :: Int -> [a] -> [a]` och `drop :: Int -> [a] -> [a]` för att plocka ut delsträngar.

# Logikprogrammering med Prolog

10. Vad skiljer Prolog och Constraint Logic Programming? (2p)

11. (a) Skriv ett Prolog-predikat `neg3` som hittar förekomster av tre negativa tal i en lista. Predikatet ska ange vilket index i listan som de tre talen start på. (4p)

    *Exempelkörning:*

    ```
    | ?- L=[-1, 2, -3, -1, -4, 6, -1, -1, -2, -3], neg3(L, I).
    I = 3 ? n
    I = 7 ? n
    I = 8
    yes
    ```

    (b) Modifiera predikatet, med hjälp av snitt, så att endast första förekomsten hittas. (2p)

    *Exempelkörning:*

    ```
    | ?- L=[-1, 2, -3, -1, -4, 6, -1, -1, -2, -3], neg3(L, I).
    I = 3 ? n
    no
    ```

12. Antag att vi lagrar en riktad graf genom att spara dess kanter som sammansatta termer. Ett fakta `edge(x,y)` betyder att det finns en riktad kant från $x$ till $y$ i grafen.

    Ett Prolog-program innehåller nu följande fakta och satser:

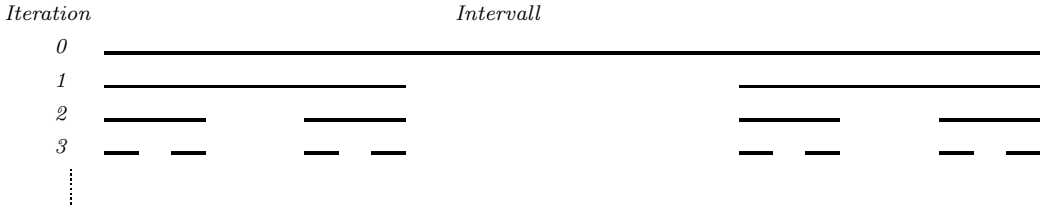
    ```
    edge(a, b).
    edge(b, a).
    edge(b, c).

    eight(X, Y) :- cycle(X, Z), cycle(Z, Y).

    cycle(X, Y) :- edge(X, Y), edge(Y, X).
    ```

    Illustrera med hjälp av lådmodellen exekveringen av målet `eight(X, Y).` (4p)

13. Cantormängden är en enkel fraktal på enhetsintervallet $[0, 1]$ som skapas genom att plocka bort den mittre tredjedelen av intervallet och sedan iterera på de kvarvarande intervallen. Grafiskt sett skapas Cantormängden så här:




    Algoritmen som skapar Cantormängden ser ut så här:

    (a) Starta intervallet $[0, 1]$.

    (b) För varje intervall $[x, y]$ du har, ersätt det med intervallen $\left[x, x + (y - x)/3\right]$ och $\left[x + 2(y - x)/3, y\right]$

    (c) Gå till (b).

    För att skapa Cantormängden itererar man alltså oändligt många gånger. I den här uppgiften ska du approximera Cantormängden genom att iterera ett givet antal gånger. Skriv predikatet `cantor(I, C)` som till `C` unifierar den I:e iteration av Cantormängden, som representeras genom en lista av par (start, slut). (Exempel på nästa sida.) (4p)

*Exempelkörning:*

```
| ?- cantor(0, C).
C = [(0,1)] ? n
no
| ?- cantor(1, C).
C = [(0,0.333),(0.666,1)] ? n
no
| ?- cantor(2, C).
C = [(0,0.111),(0.222,0.333),(0.666,0.777),(0.888,1)] ?
yes
```

(Antalet decimaler är reducerade för tydlighetens skull, du behöver inte bekymra dig över precision.)

## Syntaxanalys

Du som följde kursen innan 2005 struntar i detta avsnitt!

14. Man säger att reguljära uttryck beskriver *språk*, dvs en mängd strängar, över ett visst alfabet. Antag att $R_1$ och $R_2$ beskriver språken $L_1$ och $L_2$ över $\{a, b, c\}$. Vilket reguljärt uttryck beskriver språket $L_1 \cap L_2$ i fallen nedan? Du måste motivera för att få poäng.

    (a) $R_1 = aba^*b^*$ och $R_2 = bab^*a^*$. (1p)

    (b) $R_1 = a^*|b^*|c^*$ och $R_2 = (a|b|c)^*$. (1p)

    (c) $R_1 = ((ab)|c)^*$ och $R_2 = ((ca)|b)^*$. (2p)

15. Du har fått jobb på Bert Karlssons nya forskningsavdelning SARS, *Skara Advanced Research and Schlager*, där ett viktigt projekt är att utveckla ett automatiskt musikverktyg som gör det möjligt att kringå onödiga mellanhänder som till exempel låtskrivare och musiker.

    Genom att detaljstudera Thomas G:sons samlade produktion har ni kommit fram till ett utkast till grammatik för musik, se nedan. Ett par icke önskvärda egenskaper har identifierats hos grammatiken och det blir din uppgift att justera den.

    (a) Bert vill att grammatiken görs prediktiv med 1 "look-ahead", dvs den ska tillhöra $LL(1)$.(2p)

    (b) Låtar genererade från grammatiken visar sig ofta bryta mot den viktiga maxtidsregeln: god schlager är max 3,5 minuter lång. Det visar sig bero på att låtar som följer grammatiken faktiskt kan vara godtyckligt långa. Vad måste du åtgärda för att begränsa deras längd (räknat i antalet produktioner, inte minuter)? (2p)

Startproduktionen är "schlager". Terminaler är skrivna med versaler, produktioner använder gemena bokstäver.

| | | |
|---|---|---|
| schlager | → | intro versref versref mellanspel sistavers final |
| intro | → | DRAMA \| PIANO \| LÅNGSAMT |
| versref | → | VERS REFRÄNG |
| mellanspel | → | JANNESCHAFFERSOLO \| HIPHOPRAP \| STRÅKAR |
| sistavers | → | VERS TONARTUPP REFRÄNG xref TONARTNED |
| | \| | versref xref |
| xref | → | REFRÄNG |
| | \| | $\epsilon$ |
| final | → | CRESCENDO |
| | \| | tonaut |
| tonaut | → | TYSTARE tonaut |
| | \| | $\epsilon$ |