Lösningsförslag för tentamen i 2D1361 Programmeringsparadigm, 21 november 2006

Kursansvarig: Lars Arvestad

1. (a) Syntaktiskt socker är enkelt uttryckssätt för något som kan uttryckas på ett annat (ofta mer generellt) sätt. Till exempel är funktionsdefinitioner i Haskell, tex

   ```
   square x = x * x
   ```

   syntaktiskt socker för

   ```
   square = \x -> x * x
   ```

   där ett λ-uttryck binds till variabeln `square`.

   (b) Ett idiom är ett sätt att lösa en återkommande programmeringsuppgift. Ett idiom kan lanseras därför att det har vissa fördelar som att minimera risken för buggar eller snabbhet. Att använda en ackumuleringsparameter i rekursioner tycker jag är ett typiskt idiom.

2. (a) Statisk typning innebär att kompilatorn känner typen på en variabel redan vid kompilering. Det kan antingen vara genom programmerarens försorg (som i Java) eller genom typhärledning (Haskell).

   (b) Stark typning innebär att språket inte kompromissar om tolkningen av en variabel. I Haskell måste du explicit tala om när ett variabelvärde ska tolkas om från en typ till en annan (tex Int till Float), men i C sker många typkonverteringar automatiskt.

3. Lat evaluering innebär att en funktion inte evalueras förrän dess värde verkligen behövs.

4. Imperativa programmeringsspråk är en klass av språk som har gemensamt att de använder sig av tilldelning och serier av satser som ändrar tillståndet i ett program.

5. I `f = map length` tilldelas `f` funktionen som är resultatet av `map` med `length` som dess första argument. Vi får alltså en funktion som applicerar `length` på varje element i en lista och returnerar längden (ett heltal). Det betyder att `f` tar en lista av listor och returnerar en lista med längder.

6. (a) `tentakul l1 l2 = [(x, y) | x <- l1, y <- l2, x /= y]`

   (b) `Eq a =>` är en begränsning på vilka typer som `a` kan vara. Här måste de vara jämförbara, dvs operatorn `==` ska kunna tillämpas på den.

7. Bowling.

```
module Bowling where

-- Basfall: Ingen serie
bowling [] = 0

-- Strajk i sista kastet.
bowling [10] = 20
```

```
-- Strajk med ett flertal kast kvar
bowling (10 : x1 : x2 : xs) =
    10 + x1 + x2 + bowling (x1 : x2 : xs)

-- Strajk med bara ett kast kvar
bowling (10 : x1 : []) =
    10 + x1 + bowling [x1]

-- Poäng för icke-strajk
bowling (x : xs) = x + bowling xs
```

8. Representation av och funktioner för polynom i Haskell.

   (a)
   ```
   module Polynomial where

   data Polynomial = Polynomial
                     Int   -- Polynomets grad
                     [Float] -- Dess koefficienter

   -- Exempel:
   p1 = Polynomial 3 [1, 2, 3, 4]
   p2 = Polynomial 3 [1, 1, 1, 1]
   ```

   (b)
   ```
   coefficient :: Polynomial -> Int -> Float
   coefficient (Polynomial degree coeffs) i =
       coeffs !! i   -- Element i of the list
   ```

   (c)
   ```
   -- Lösning som använder att  1+ 2x + 3x^2 + 4x^3 =1 + x(2 + x(3 + x(4+ 0)))
   evaluate1 :: Polynomial -> Float -> Float
   evaluate1 (Polynomial _ coeffs) x =
      helper coeffs
      where revCoeff = reverse coeffs
            helper [] = 0
            helper (c:cs) = c + x * helper cs

   -- Ackumulering av x-potens
   evaluate2 :: Polynomial -> Float -> Float
   evaluate2 (Polynomial _ coeffs) x =
      helper coeffs 1
      where helper [] _ = 0
            helper (c:cs) xpower = c * xpower
                                   + helper cs (x * xpower)

   -- Ackumulering av x-potens och summa
   evaluate3 :: Polynomial -> Float -> Float
   evaluate3 (Polynomial _ coeffs) x =
      helper coeffs 1 0
      where helper [] _ sum = sum
            helper (c:cs) xpower sum = helper cs (x * xpower) (sum + c * xpower)
   ```

9. Strömmar av pseudoslumptal

   (a)
   ```
   psg r0 a b m = (r1 : r2etc)
      where  r1 = (a * r0 + b) `mod` m
             r2etc = psg r1 a b m
   ```
   En generell typsignatur för `psg` är

```
psg :: Integral a => a -> a -> a -> a -> [a]
```

eftersom den följer av de funktioner som `psg` ovan använder, men jag tycker det är helt OK att kräva att argumenten är 32-bitars heltal:

```
psg :: Int -> Int -> Int -> Int -> [Int]
```

(b)
```
metapsg :: (Int -> Int) -> Int ->  [Int]
metapsg rekursion r0 = (r1 : resten)
   where r1 = rekursion r0
         resten = metapsg rekursion r1
```

10. I en logisk läsning av ett Prologprogram betrakar man satserna som logiska utsagor och bryr sig inte om hur Prologsystemet härleder unifieringarna. Det gör man i den procedurella läsningen, där man även tar hänsyn till snitt, exceptions, etc.

11. (a) Det är ett rött snitt eftersom det ändrar hur predikatet uppträder i den procedurella läsningen. Om man tar bort snittet kommer både andra och tredje satsen av `remove` att provas vare sig E=X eller inte.

    (b) Anropet kommer att gå in i en loop, eftersom

    ```
    remove([E|Xs], E, Ys) :- !, remove(Xs, E, Ys).
    ```

    kan användas och unifiera `E` med 1, `[1|Xs]` med `L`, och `Ys` med `[2,3,4]`. Därefter blir `remove([1|Xs], 1, [2,3,4])` ett nytt mål, och vi kommer till samma regel igen. Här uppstår en loop eftersom vi inte kommer att tillämpa någon annan regel, och helt klart inte nåt basfall.

12. Bild för lådmodellen kommer senare...

13. Pascals triangel i Prolog.

```
pascal(1, [1]).
pascal(M, L) :- M > 1, N is M-1, pascal(N, L1), pascalLine(L1,L2), L=[1|L2].

pascalLine([1], [1]).
pascalLine([A,B|Cs], [D|L]) :- D is A + B, pascalLine([B|Cs], L).
```

14. En kö i Prolog.

```
makeQueue(X-X).
removeFirst([H|X]-Y, H, X-Y).
enqueue(Q, Elem, Qny) :-
   dlAppend(Q, [Elem|NewHole] - NewHole, Qny).

% Från föreläsning och/eller boken:
dlAppend(H1 - X, H2 - Y, H1 - Y) :-
     X = H2.
```

15. Reguljära uttryck.

    (a) Här är en lösning: `(((a|b|c|..|ö)*(b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w))?i)|I(c|k)a`. Den säger att strängen ska sluta med "ica" eller "ika", innan dess *kanske* (mha `?`) det kommer ett prefix som har noll eller många godtyckliga bokstäver (`(a|b|c|..|ö)*`), och följs av en konsonant.

    (b) Här låter jag (`^x`) betyda "alla bokstäver utom 'x'", dvs (`^b`) kan skrivas som `(a|c|d|...)`:
    `(Las(^s)e)|(La(^s)se)|(Lass(^e))|(L(^a)sse)|((^L)asse)|(Lase)`

16. En möjlig grammatik:

```
personallista --> STAFFOPEN stafflist STAFFCLOSE

stafflist -->
            | person stafflist
            ;

person --> personopen firstname lastname ssid PERSONCLOSE

personopen --> PERSONOPENHALF STRING LARGERTHAN

firstname --> FNAMEOPEN STRING FNAMECLOSE
lastname  --> LNAMEOPEN STRING LNAMECLOSE
ssid      --> SSIDOPEN  actualssid SSIDCLOSE
actualssid --> DIGIT DIGIT DIGIT DIGIT DIGIT DIGIT
               DASH  DIGIT DIGIT DIGIT DIGIT
```

Jag har här använt terminalerna

```
STAFFOPEN       "<staff>"
STAFFCLOSE      "</staff>"
FNAMEOPEN       "<firstname>"
FNAMECLOSE      "</firstname>"
LNAMEOPEN       "<lastname>"
LNAMECLOSE      "</lastname>"
SSIDOPEN       "<ssid>"
SSIDCLOSE      "</ssid>"
LARGERTHAN      ">"
PERSONOPENHALF "<person position="
PERSONCLOSE     "</person>"

STRING är alla alfabetiska strängar
DIGIT  är siffrorna 0 till 9
DASH   är tecknet "-"
```

som vi antar kommer från en lexikal analysator.