# Lösningsförslag för tentamen i DD1361 Programmeringsparadigm, 19 nov 2007

Kursansvarig: Lars Arvestad

1. Strikt evaluering innebär att värdet av ett uttryck beräknas innan det används. Haskells lata evaluering gör att språket är icke-strikt. I exv C "kortsluter" man boolska beräkningar, vid ett logiskt eller evalueras det andra argumentet enbart om det första är falskt, vilket gör att sådan operationer inte är strikta.

2. Svansrekursion innebär att det sista som utförs i en funktion är den anropar sig själv.

3. "First class functions" innebär att man betraktar funktioner som vilka värden som helst. De kan skickas omkring som funktionsargument, returneras som värden, samt manipuleras med olika operatorer (tex komposition, Currying, etc).

4. Ett programmeringsspråk är Turingfullständigt om det kan beräkna samma klass av funktioner som en Turingmaskin, en abstrakt modell för beräkningar.

5. En abstrakt datatyp (ADT) exponerar inte sin implementation. Det enda sättet att arbeta med en ADT är att använda funktioner som definierats i samband med datatypen. På så sätt är det lätt att byta underliggande implementation: Alla beroende på datatypens implementation finns på ett och samma ställe.

6. (a) Det är en anonym funktion som returnerar kvadraten av sitt argument.

   (b) Vi har skapat funktionen `f` som tar en lista $l$ som argument och applicerar `foldl analyze 0` på $l$. Foldl arbetar iterativt och applicerar `analyze` på första elementet i listan och ett ackumulerat värde som initieras till 0.

7. 
```
intersperse :: a -> [a] -> [a]
intersperse mellan [x] = [x]
intersperse mellan [] = []
intersperse mellan (h : t) = (h : mellan : intersperse mellan t)
```

8. (a)
```
data IntSet = Finite [Integer]
            | Infinite [Integer]
              deriving Show
```

   (b)
```
finite :: IntSet -> Bool
finite (Finite _) = True
finite (Infinite _ ) = False
```

   (c)
```
data Cardinality = Countable
                 | LengthIs Int
                   deriving Show

cardinality :: IntSet -> Cardinality
cardinality (Finite l) = LengthIs (length l)
cardinality (Infinite _) = Countable
```

9. (a) Antingen `words :: Text a => a -> [String]` eller `words :: Text a => a -> [a]`.

   (b)
```
ropeLength :: Rope -> Int
ropeLength EmptyString = 0
ropeLength (Substr _ x) = x
ropeLength (Concat r1 l r2) = l + (ropeLength r2)
```

   (c)
```
subRope :: Rope -> Int -> Int -> Rope
subRope EmptyString _ _ = EmptyString
subRope (Substr s l) start len = (Substr sub len)
    where suffix = drop start s
          sub    = take len suffix
subRope (Concat r1 l1 r2) start len =
    if (start < l1) then
        if (start + len <= l1) then
            subRope r1 start len
        else
            (Concat left leftlen right)
    else
        subRope r2 (start - l1) len
    where
      leftlen = l1 - start
      left = subRope r1 start leftlen
      right = subRope r2 0 (len - leftlen)
```

10. Så här bra skrev en student på tentan:

> Prolog är ett programmeringsspråk där man styr programmet genom att skriva predikat som består av delmål och sedan låter prologmotorn hitta lösningar.
>
> CLP är ett mer generellt sätt att beskriva en viss sorts problem. Givet en viss [datadomän] och ett antal bivillkor så ska en [lösning] genereras som uppfyller detta.
>
> En annan viktig skillnad är att Prolog använder sig av djupet-förstsökning för att hitta lösningar och CLP-lösare använder sig ofta av något smartare sökalgoritmer för att snabbare kunna leverera en lösning.

11. (a)
```
neg3([X,Y,Z|Rest],1) :-
        X<0, Y<0, Z<0.
neg3([_|Rest], I) :-
    neg3(Rest, J),
     I is J + 1.
```

   (b) Vi lägger till ett snitt efter testet för negativa tal:
```
neg3([X,Y,Z|Rest],1) :-
        X<0, Y<0, Z<0, !.
neg3([_|Rest], I) :-
    neg3(Rest, J),
     I is J + 1.
```

12. Jag har inte hunnit fixa en schysst bild för detta.

13.
```
cantor(0, [(0, 1)]).
cantor(I, C3) :-
        I > 0,
        J is I - 1,
        cantor(J, C2),
        cantorsplit(C2, C3).

cantorsplit([], []).
cantorsplit([(S, E) | Rest], [(S, E2), (S2, E) | NewRest]) :-
```

```
E2 is S + (E-S) / 3,
S2 is S + 2 * (E - S) / 3,
cantorsplit(Rest, NewRest).
```

14. (a) Strängarna som beskrivs av $R_1$ börjar alla på $a$, och de som beskrivs av $R_2$ börjar alltid på $b$. Därför är $L_1 \cap L_2 = \emptyset$. Det reguljära uttrycket för det språket är också $\emptyset$.

(b) $R_1$ beskriver strängar som består helt av antingen $a$, $b$, eller $c$, men $R_2$ beskriver alla strängar över $a$, $b$ och $c$. Därför är $L_1 \cap L_2 = L_1$ och $R_1$ beskriver $L_1 \cap L_2$.

(c) En sträng från $L_1$ börjar antingen på $ab$ eller $c$. Strängar från $L_2$ börjar på $ca$ eller $b$. Alltså börjar alla strängar i $L_3 = L_1 \cap L_2$ på $c$. Eftersom det i $L_2$ då innebär att strängen börjar med $ca$, måste detta matchas av $ab$ i $L_1$, vilket i sin tur kräver att $L_2$-strängen fortsätter med $b$:

| | |
|---|---|
| c | ab |

| | |
|---|---|
| ca | b |

Så en sträng i $L_3$ börjar alltid med $cab$. Vi kan nu induktivt använda samma argument, eftersom resten också måste ligga i $L_3$.

Då den tomma strängen tillhör båda språken, är $(cab)^*$ det reguljära uttrycket för $L_3$.

15. (a) Det är bara i "sistavers" som vi inte kan avgöra vilken produktion som ska användas genom att titta på nästkommande terminal. Vi får därför skriva om den produktionen till:

| | | |
|---|---|---|
| sistavers | $\rightarrow$ | VERS sistaref |
| sistaref | $\rightarrow$ | TONARTUPP REFRÄNG xref TONARTNED |
| | $\mid$ | REFRÄNG xref |
| xref | $\rightarrow$ | REFRÄNG |
| | $\mid$ | $\epsilon$ |

(b) Det är den rekursiva produktionen "tonaut" som ställer till det. En enkel lösning är att tillåta ett begränsat antal TONAUT:

| | | |
|---|---|---|
| tonaut | $\rightarrow$ | TYSTARE TYSTARE TYSTARE |
| | $\mid$ | $\epsilon$ |