Lösningsförslag för tentamen i 2D1361 Programmeringsparadigm, 24 maj 2007

Kursansvarig: Lars Arvestad

1. Med ortogonalitet avses att programspråkets konstruktioner för tex loopar, if-satser, , osv, ska vara så oberoende av varandra och icke-överlappande som möjligt. `while` och `for`-satser är tex inte alls ortogonal eftersom de används till väsentligen samma sak och kan dessutom ersätta varandra.

2. I programspråk (exv Lisp och Perl) som har dynamisk typning behöver man inte ange vilka typer som variabler/parametrar har. Istället följer typen värdet. Man kan se det som att ett värde lagras som ett par ⟨typ, värde⟩, exv ⟨Int, 17⟩.

3. En högre ordningens funktion tar andra funktioner som argument. I Haskell och andra funktionella språk är tex `map` och `foldr` typexempel på högre ordningens funktioner.

4. Ackumuleringsparametrar används för att man ska kunna slippa "gå tillbaka" i en rekursion. Om man till exempel implementerar en summa över listor som

   ```
   sum [] = 0
   sum (x:xs) = x + sum xs
   ```

   så "sparas" additionen tills efter det rekursiva anropet är klart. Det gör att x samt lite mer data om vad ska göra läggs på stacken. Med en lång lista kan alltså både mycket plats och tid slösas bort. En ackumulerande version av `sum` kan se ut så här:

   ```
   sum x = acksum x 0
   acksum [] s = s
   acksum (x:xs) s = acksum (x + s) xs
   ```

   Additionen görs alltså omedelbart och inget utrymme på stacken behöver användas.

5. Eftersom Haskell saknar sidoeffekter (det är rent funktionellt), kan man utgå ifrån att olika uttryck i koden är oberoende av varandra och kompilatorn kan då trivialt parallellisera deras exekvering.

   Om man tex utför `map fkn lista`, där `fkn` appliceras på varje element i `lista`, så kan en smart kompilator utföra `fkn`-beräkningarna parallellt. I ett imperativt språk, tex C och Java, måste kompilatorn först försäkra sig om att `fkn` inte påverkar andra element eller variabler. Att automatiskt bevisa att skilda användningar av `fkn` är oberoende av varandra kan vara mycket svårt.

6. Deklarationen säger att `fkn` är en funktionen som tar två argument av samma typ a och returnerar ett värde av typ a. Genom instruktionen `Ord a =>` begränsar vi de möjliga typerna som a kan vara till de som implementerar jämförelserna $<, \leq, =, \geq, >$ och `min` och `max`.

7. Min version av `countelems`:

   ```
   countelems :: Eq a => [a] -> [(a, Int)]
   countelems xs = iter xs []
       where iter [] l = l
             iter (x:xs) l = iter xs (update x l)

             update x [] = [(x,1)]
             update x ((y, i) : resten) =
   ```

```
            if (x == y)
            then ((x, i+1) : resten)
            else ((y, i) : update x resten)
```

Det finns många andra tänkbara lösningar som säkert dessutom är snyggare!

8. (a) Funktionen `f` genererar en oändlig lista av ett värde av okänd typ med hjälp av lat evaluering. Anropet `f a` ger ett element `a` samt löftet att beräkna sig självt (`f a`) igen om så behövs.

   (b) Här används funktionskomposition för att ge en funktion som genererar 10 exemplar av ett element. Det partiella uttrycket `take 10` är en funktion som tar de tio första element av en lista och det komponeras ihop med vår funktion `f` med hjälp av punktoperatorn.

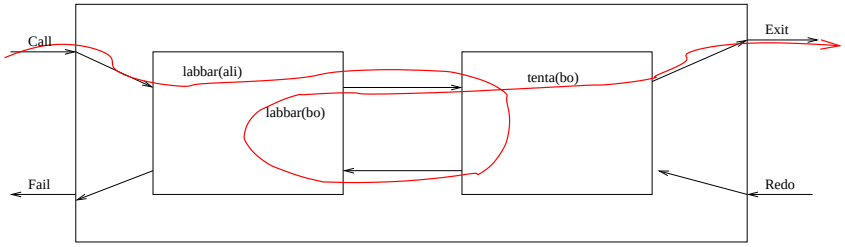
9. En exempellösning:

```
module Matrix where
data Matrix a = M
                Int -- Antal rader
                Int -- Antal kolumner
                [a] -- Matriselementen
                deriving (Show, Eq)

m1 = M 2 2 [1, 2, 3, 4]
m2 = M 2 2 [4, 3, 2, 1]
m3 = M 4 1 ['a', 'b', 'c', 'd']
m4 = M 2 3 [1, 2, 3, 4, 5, 6]

minit :: (Int, Int) -> a -> Matrix a
minit (r, c) x = (M r c [x | i <- [1..r], j <- [1..c]])

msize :: Matrix a -> (Int, Int)
msize (M rows cols _) = (rows, cols)

elemlist :: Matrix a -> [a]
elemlist (M _ _ l) = l

mplus :: Num a => Matrix a -> Matrix a -> Matrix a
mplus m1 m2 = if (msize m1 == msize m2)
              then M r c e
              else error "Wrong dimensions!"
                  where (r, c) = msize m1
                        l1 = elemlist m1
                        l2 = elemlist m2
                        e = map (\(a, b) -> a + b) (zip l1 l2)

msum [] = error "Non-empty matrix list required"
msum (m : ml) = foldr mplus m ml
```

10. Med unifiering avses att två termer mönstermatchas mot varandra och om det är möjligt binds värden (tillfälligt) till oinitierade variabler i båda termerna.

11. Det är ett grönt snitt (programmet skulle fungera likadant utan det) som förhindrar att unifiering provas med den andra regeln när man redan vet att $X \leq Y$.

12. Programflödet går så här:



13. Anropet resulterar i en oändlig loop. Inget faktum passar in på predikatet och det gör att Prolog kommer att prova att unifiera den sista regeln om och om igen.

14. Jag föreslår:

```
edge(v1, v3).
edge(v2, v3).
edge(v2, v4).
edge(v4, v5).
edge(v3, v5).

path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).

xpath(X, Y, [X, Y]) :- edge(X, Y).
xpath(X, Y, [X | L]) :- edge(X, Z), xpath(Z, Y, L).

sink(X) :- edge(_, X), \+ edge(X, _).
```

15.
- Konkatenering: Om a och b är RE då är ab ett RE.
- Alternering: $a|b$ är ett RE som betyder att "antingen a eller b".
- Gruppering: (ab) används så att vi kan säga $(ab)|(cd)$, dvs "ab eller cd".
- Upprepning: a^* betyder "inga eller flera instanser av a".

16. En möjlig grammatik:

```
ref --> authors whatandwhere

authors      --> singleauthor
                 | authorlist AND singleauthor

singleauthor --> LNAME initials

authorlist   --> singleauthor
                 | authors singleauthor

initials --> INI
           | INI INI

paper --> YEAR TITLE JOURNAL optdetails
        | JOURNAL VOL PAGES YEAR
        | TITLE JOURNAL YEAR VOL PAGES

optdetails -->  /* empty! */
              | VOL PAGES
```