# TENTAMEN I DVA 201 FUNKTIONELL PROGRAMMERING MED F#

*Fredagen den 5 juni 2015, kl 14:10 – 18:30*

LÖSNINGSFÖRSLAG (reviderat 2015-06-07)

---

**UPPGIFT 1** (7 POÄNG)

*a)* Vi gör en rättfram lösning där vi rekurserar genom listan, returnerar `true` om vi stöter på ett element för vilket predikatet blir sant samt returnerar `false` om vi når den tomma listan:

```
let rec exists p l =
    match l with
    | []             -> false
    | x::xs when p x -> true
    | _::xs          -> exists p xs
```

*b)* Först "mappar" vi predikatet `p` över listan, vilket ger en lista av booleans. Sen tar vi "or" över alla elementen i den listan med hjälp av `List.fold`. För att göra det hela riktigt elegant uttrycker vi allting som funktioner över listor, med hjälp av partiell applikation, och kopplar ihop funktionerna med funktionskomposition:

```
let exists p = List.map p >> List.fold (||) false
```

**UPPGIFT 2** (2 POÄNG)

Det beror på om man skriver in deklarationerna efter varandra, i F# interactive, eller om man lägger båda deklarationerna i en modul i en fil och läser in.

I det första fallet kommer man att få ett typfel. I den första deklarationen kommer typen på `x + x` att defaulta till `int`, vilket medför att `f` får typen `int -> int`. I deklarationen av `g` får `f` sedan en `float` som argument, vilket inte går ihop sig.

I det andra fallet kommer däremot typinformationen från deklarationen av `g` att användas tillsammans med typinformationen för `f`, för att lösa upp ev. tvetydigheter. Typinferensen kommer då att komma fram till att både `f` och `g` kan ha typen `float -> float`, vilket blir den erhållna typningen.

Jag kommer att godkänna endera av svaren ovan, om den givna motiveringen är tillräcklig.

**UPPGIFT 3** (4 POÄNG)

Återigen en enkel rekursion över listorna, där vi går igenom element per element, matchar ut de olika fallen, och beräknar motsvarande element i resultatlistan på det sätt som anges i varje enskilt fall:

```
let rec optadd l1 l2 =
    match (l1,l2) with
    | (Some x::xs,Some y::ys) -> (x + y) :: optadd xs ys
    | (Some x::xs,None::ys)   -> x :: optadd xs ys
    | (None::xs,Some y::ys)   -> y :: optadd xs ys
    | (None::xs,None::ys)     -> 0 :: optadd xs ys
    | _                       -> []
```

**UPPGIFT 4** (3 POÄNG)

*a), b)* Båda evalueringsordningarna kommer att ge samma resultat. När `g 0` räknas ut kommer villkoret att bli sant, varvid 1 returneras.

*c)* Ivrig evaluering (call-by-value).

**UPPGIFT 5** (4 POÄNG)

En övning i hur man kombinerar sekventiell programmering med rekursion i F#. Lösningen består av två delar: en yttre "wrapper" `writearray` som är den som anropas: den öppnar och stänger filen och däremellan lämnar den kontrollen till den lokala funktionen `write_local`. Den lokala funktionen fungerar som en loop: den skriver ut en rad till filen, räknar upp sitt lokala numeriska argument med ett, och anropar rekursivt sig själv.

```
open System.IO
let writearray name (a : int []) =
  let file = File.CreateText(name)
  let n = a.Length
  let rec write_local i =
    if i = n then () else file.WriteLine(string a.[i]) ; write_local (i+1)
  in write_local 0 ; file.Close()
```

(Notera den explicita typningen av array-argumentet `a`. Denna är nödvändig för att deklarationen ska gå genom typinferensen. Jag drar inte några poäng om man har missat den.)

**UPPGIFT 6** (6 POÄNG)

*a)* Det är lämpligt med en träd-datatyp som moellerar hur lådor kan ligga inuti varandra. Vi använder konstruktorerna `Black` och `White` för att markera en lådas färg. Den "tomma lådan" `Empty` markerar avsaknad av låda:

```
type BoxTree = White of BoxTree * BoxTree * BoxTree * BoxTree
             | Black of BoxTree * BoxTree * BoxTree * BoxTree | Empty
```

*b)* En enkel trädrekursion:

```
let rec count_white t =
  match t with
  | Empty               -> 0
  | White (b1,b2,b3,b4) -> 1 + count_white b1 + count_white b2
                             + count_white b3 + count_white b4
  | Black (b1,b2,b3,b4) -> count_white b1 + count_white b2
                           + count_white b3 + count_white b4
```

**UPPGIFT 7** (4 POÄNG)

Vi definierar $P(\mathtt{xs}) \iff \mathtt{xs} \mathbin{@} [\,] = \mathtt{xs}$. Sen visar vi $\forall \mathtt{xs}.P(\mathtt{xs})$ med induktion över listor. Detta bevisar påståendet. Vi kan använda funktionsdeklarationen av "@" (`List.append`). Den ger upphov till följande två ekvationer:

$$[\,] \mathbin{@} \mathtt{xs} = \mathtt{xs} \tag{1}$$

$$(\mathtt{x}::\mathtt{xs}) \mathbin{@} \mathtt{ys} = \mathtt{x}::(\mathtt{xs} \mathbin{@} \mathtt{ys}) \tag{2}$$

Vi genomför nu induktionsbeviset.

**Basfall**, visa att $P([\,])$ gäller, dvs. $[\,] \mathbin{@} [\,] = [\,]$. Vi har $[\,] \mathbin{@} [\,] =$ (pga. (1)) $= [\,]$ vilket skulle visas.

**Induktionssteg**: Antag att $P(\mathtt{xs})$ ("induktionshypotesen") gäller, visa att $P(\mathtt{x}::\mathtt{xs})$ gäller för godtyckligt $\mathtt{x}$. $P(\mathtt{xs})$ är samma som

$$\mathtt{xs} \mathbin{@} [\,] = \mathtt{xs} \tag{3}$$

Vi vill visa $P(\texttt{x::xs})$, dvs.

$$(\texttt{x::xs}) \mathbin{@} [\,] = \texttt{x::xs}$$

Låt os visa att vänsterledet (VL) är lika med högerledet (HL). Vi har, för godtyckligt $\texttt{x}$, att

$$\begin{array}{rcl}
\text{VL} & = & (\texttt{x::xs}) \mathbin{@} [\,] \\
 & = & \text{(pga. (2))} \\
 & = & \texttt{x::}(\texttt{xs} \mathbin{@} [\,]) \\
 & = & \text{(pga. induktionshyopotesen (3))} \\
 & = & \texttt{x::xs} \\
 & = & \text{HL}
\end{array}$$

Detta visar induktionssteget.

Eftersom vi visat både basfall och induktionssteg har vi visat $\forall \texttt{xs}.P(\texttt{xs})$, vilket ger det sökta resultatet.