TENTAMEN I DVA 229 FUNKTIONELL PROGRAMMERING MED F#

Torsdagen den 18 augusti 2016, kl 14:10 – 18:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (6 POÄNG)

a) En rättfram rekursion genom listan där vi för varje nytt rekursivt anrop testar om listans huvud är lika med `x` eller ej:

```
let rec remove x l =
  match l with
  | [] -> []
  | y::ys when x = y -> remove x l
  | y::ys            -> y::remove x l
```

b) Med hjälp av `List.filter` sållar vi lätt ut de element som inte är lika med `x`:

```
let remove x = List.filter (fun y -> y <> x)
```

UPPGIFT 2 (2 POÄNG)

Den mest generella typen för `sumfirst` är `(int * 'a) list -> int`. Man ser från match-uttrycket att `l` måste vara en lista och ur mönstret för det andra fallet i match-uttrycket följer att `l` måste vara en lista av par, Vidare följer ut första fallet i match-uttrycket att `sumfirst` måste returnera en `int`. Alltså måste `x` i andra fallet också vara en `int`, däreot spelar det ingen roll vilken typ `y` har. Alltså tar `sumfirst` ett argument af typ `(int * 'a) list` och returnerar en `int`. dvs. den har funktionstypen ovan.

UPPGIFT 3 (4 POÄNG)

Vår lösning är en rättfram rekursion, i den lokalt deklarerade funktionen `s2l`, som rekurserar genom indexen i strängen i stigande ordning, plockar ut motsvarande tecken från strängen och successivt bygger en lista av dessa;

```
let string2list s =
  let n = String.length s in
    let rec s2l i =
      if i = n then []
               else s.[i] :: s2l (i+1)
    in s2l 0
```

UPPGIFT 4 (3 POÄNG)

`y` kommer att få typen `Lazy<int>`. Första gången som `y.Force()` evalueras kommer uttrycket att räknas ut, varvid strängen `"abc\n"` skrivs ut och värdet `57` returneras. Vid andra evalueringen returneras det uträknade värdet `57`. Ingen sidoeffekt äger rum.

UPPGIFT 5 (4 POÄNG)

En rättfram lösning som öppnar filen, skriver raderna i en lokal, rekursiv funktion och slutligen stänger filen:

```
open System.IO

let filewrite f (s:string) n =
  let file = File.CreateText(f)
  let rec writelines i =
    if i = n then () else file.WriteLine(s); writelines (i+1)
  writelines 0
  file.Close()
```

UPPGIFT 6 (7 POÄNG)

a) För att representera en rund ask behövs dess radie och höjd. På liknande sätt behövs sidan och höjden för att representera en kvadratisk ask. Sen kan en ask innehålla en följd av askar, så vi gör datatypen rekursiv. Det finns också en möjlighet att asken inte innehåller några andra askar, så vi behöver ett "tomt" element som fungerar ungefär som den tomma listan. Deklarationen ser ut som följer (med några typaliases för att göra modelleringen tydligare):

```
type radius = float
type height = float
type side = float
type Box = Cyl of (radius * height * Box)
           | Rec of (side * height * Box)
           | Empty
```

b) Vi deklarerar en funktion som rekurserar genom följden av askar och kollar de olika fall som uppstår. Obs. att det blir ett antal olika kombinationer av asktyper att kontrollera. Fallet när en cylindrisk ask innehåller en kvadratisk kräver lite elementär geometrikunskap: kvadraten får plats i cylindern brecis när kvadratens diagonal är mindre än två gånger cylinderns radie. Diagonalen är i sin tur lika med kvadratroten av två gånger sidan.

```
let rec check box =
  match box with
  | Empty -> true
  | Cyl (r,h,Empty) -> true
  | Rec (s,h,Empty) -> true
  | Cyl (r,h,Cyl (r1,h1,nextbox))
                    -> h > h1 && r > r1 && check (Cyl (r1,h1,nextbox))
  | Cyl (r,h,Rec (s,h1,nextbox))
                    -> h > h1 && 2.0 * r > s * sqrt 2.0 && check (Rec (s,h1,nextbox))
  | Rec (s,h,Cyl (r,h1,nextbox))
                    -> h > h1 && s > 2.0 * r && check (Cyl (r,h1,nextbox))
  | Rec (s,h,Rec (s1,h1,nextbox))
                    -> h > h1 && s > s1 && check (Rec (s1,h1,nextbox))
```

UPPGIFT 7 (4 POÄNG)

För att förenkla notationen skriver vi `map` och `tail` för `List.map` och `List.tail`. Vi påminner oss också definitionen av funktionskomposition `(>>)` i F#:

```
(f >> g) x = g (f x)
```

för alla `x`. Slutligen repeterar vi definitionen av likhet mellan funktioner: funktionerna `f` och `g` är lika om och endast om

```
f x = g x
```

för alla `x`. Vi visar nu att

```
(map f >> tail) l = (tail >> map f) l
```

För alla listor `l`.

Fall 1, `l = []`: då har vi för vämsterledet att

```
(map f >> tail) [] = tail (map f [])
                   = tail []
```

En evaluering av `tail []` kommer att leda till felavbrott. För högerledet har vi

```
(tail >> map f) [] = map f (tail [])
```

Även här kommer evalueringen av `tail []` att leda till felavbrott. Så båda leden ger samma resultat.

Fall 2, `l = [x1;...;xn]`: Vi visar att

```
(map f >> tail) [x1;...;xn] = (tail >> map f) [x1;...;xn]
```

för alla icke-tomma listor `[x1;...;xn]`. Vi har

```
(map f >> tail) [x1;...;xn] = tail (map f [x1;...;xn])
                            = tail [f x1;...;f xn])
                            = [f x2;...;f xn]
                            = map f [x2;...;xn]
                            = map f (tail [x1;...;xn])
                            = (tail >> map f) [x1;...;xn]
```

vilket skulle visas.