TENTAMEN I DVA 229 FUNKTIONELL PROGRAMMERING MED F#

Torsdagen den 24 mars 2022, kl 14:30 – 18:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (6 POÄNG)

a) en enkel rekursion genom listan där det hittills största talet ackumuleras med `max`-funktionen. Eftersom vi returnerar `0.0` för den tomma listan säkerställer vi att resultatet blir `0.0` om listan bara innehåller icke-positiva tal.

```
let rec maxF l =
  match l with
  | [] -> 0.0
  | x::xs -> max x (maxF xs)
```

b) Vi använder `List.fold` tillsammans med `max` för att erhålla det största värdet. Eftersom vi foldar med konstanten `0.0` garanteras att vi inte returnerar något negativt tal ens om listan bara innehåller sådana tal.

```
let maxF = List.fold max 0.0
```

UPPGIFT 2 (4 POÄNG)

Vår lösning använder en inre rekursiv funktion `findLocal` som rekurserar genom indexen i arrayen, från `0` och uppåt, för att hitta det första elementet där predikatet blir sant:

```
let tryFind p a =
  let n = Array.length a
  let rec findLocal i =
    if i = n then None
    elif p(a.[i]) then Some (a.[i])
    else findLocal (i+1)
  in findLocal 0
```

UPPGIFT 3 (2 POÄNG)

Först binds `x` till det oevaluerade högerledet. I nästa steg gör `x.Force()` att uttrycket räknas ut, varvid `printf` gör en utskrift, och `y` binds till värdet av uttrycket (`3.0`). I det sista steget binds `z` till det redan uträknade värdet för `x.Force()`. Ingen ny uträkning sker och därför blir det heller ingen ny utskrift.

UPPGIFT 4 (4 POÄNG)

Lösningen initierar först den muterbara referenscellen `acc` till `1` och exekverar sen en inre, rekursiv funktion `localProd` som successivt multiplicerar in listelementen i `acc`. När slutet på listan nås (tomma listan) returneras värdet i `acc`:

```
let prod l =
  let acc = ref 1
  let rec localProd l =
    match l with
    | [] -> !acc
    | x::xs -> acc := !acc * x ; localProd xs
  in localProd l
```

UPPGIFT 5 (2 POÄNG)

Problemet är att funktionens exekveringstid och minnesbehov båda är kvadratiska i längden n på strängen, vilket blir mycket dyrt när n är stort. Anledningen är att det rekursiva anropet till `asciiSum` görs på delsträngen `s.[1..]` som har längden $n-1$. Varje gång man tar en delsträng kopieras den, vilket innebär att minnes- och tidsåtgång båda blir proportionella mot $n + (n-1) + \cdots + 1 = O(n^2)$.

UPPGIFT 6 (2 POÄNG)

I F# är en deklarerad storhet synlig bara från den punkt i modulen där den är deklarerad. Det betyder att `g` inte är i scope i deklarationen av `f`. Att byta plats på deklarationerna hjälper inte, då blir det `f` som inte blir i scope i `g`:s deklaration. Lösningen är att envända den speciella konstruktionen i F# för att göra ömsesidigt rekursiva funktioner som använder nyckelordet “`and`”.

UPPGIFT 7 (6 POÄNG)

a) Vi deklarerar en rekursiv, polymorf datatyp med ett fall för varje typ av nod i trädet:

```
type BinTree<'a> = Leaf of 'a | Node1 of 'a * BinTree<'a>
                 | Node2 of 'a * BinTree<'a> * BinTree<'a>
```

b) En lösning med rättfram rekursion genom trädets olika grenar:

```
let rec mapTree f t =
  match t with
  | Leaf x -> Leaf (f x)
  | Node1 (x,t1) -> Node1 (f x, mapTree f t1)
  | Node2 (x,t1,t2) -> Node2 (f x, mapTree f t1, mapTree f t2)
```

UPPGIFT 8 (4 POÄNG)

vi visar att `(map f >> tail) l` = `(tail >> map f) l` för alla listor `l`. Låt $\mathtt{l} = [\mathtt{x}_1, \ldots, \mathtt{x}_n]$. Då gäller, för vänsterledet, att

$$\begin{aligned}(\mathtt{map\ f} >> \mathtt{tail})\,[\mathtt{x}_1, \ldots, \mathtt{x}_n] &= \mathtt{tail}\,((\mathtt{map\ f})\,[\mathtt{x}_1, \ldots, \mathtt{x}_n]) \\ &= \mathtt{tail}\,[\mathtt{f\ x}_1, \ldots, \mathtt{f\ x}_n] \\ &= [\mathtt{f\ x}_2, \ldots, \mathtt{f\ x}_n]\end{aligned}$$

För högerledet gäller

$$\begin{aligned}(\mathtt{tail} >> \mathtt{map\ f})\,[\mathtt{x}_1, \ldots, \mathtt{x}_n] &= (\mathtt{map\ f})\,(\mathtt{tail}\,[\mathtt{x}_1, \ldots, \mathtt{x}_n]) \\ &= (\mathtt{map\ f})\,[\mathtt{x}_2, \ldots, \mathtt{x}_n] \\ &= [\mathtt{f\ x}_2, \ldots, \mathtt{f\ x}_n]\end{aligned}$$

Sålunda är VL = HL för alla listor `l`, vilket bevisar lagens giltighet.