TENTAMEN I DVA 201 FUNKTIONELL PROGRAMMERING MED F#

Torsdagen den 20 augusti 2015, kl 8:10 – 12:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (7 POÄNG)

a) Vi gör en rättfram lösning där vi rekurserar genom listan, adderar elementen, men skippar de element som inte är positiva:

```
let rec sumpos l =
    match l with
    | []               -> 0
    | x::xs when x > 0 -> x + sumpos xs
    | _::xs            -> sumpos xs
```

b) En lösning sammansatt av två högre ordningens funktioner: `List.filter` filtrerar bort alla element ur listan som inte är positiva och den resulterande listan summeras med `List.sum`:

```
let sumpos = List.filter (fun x -> x > 0) >> List.sum
```

UPPGIFT 2 (2 POÄNG)

Det blir ingen skillnad, `f` och `g` kommer att bli precis samma funktion (som tar ett argument `x` och returnerar `x + x`).

UPPGIFT 3 (4 POÄNG)

Ett enkelt sätt att lösa uppgiften är att använda `List.filter` för att skapa de två listorna. För den första listan filtreras de element `x` ut där `p x` blir sant, för den andra de element där `p x` blir falskt:

```
let partition p l = (List.filter p l, List.filter (fun x -> not (p x)) l)
```

UPPGIFT 4 (2 POÄNG)

a) Eftersiom argumenten räknas ut innan anropet vid call-by-value kommer `f 1` att försöka räknas ut, vilket ger en oändlig rekursion.

b) Vid lat evaluering skjuts uträkningen av ett argument upp tills argumentets värde verkligen behövs. I vårt fall kommer `g` att anropas innan `f 1` räknas ut. Anropet till `g` kommer att returnera `1` utan att `f 1` någonsin räknas ut.

UPPGIFT 5 (4 POÄNG)

Vi gör en lösning som använder en lokalt deklarerad muterbar referenscell `s`, initierad till `0.0`. En lokalt definierad, rekursiv funktion `add_local` utför själva summeringen. Den använder explicit sekevensering (med "`;`") för att summera det aktuella array-elementet innan det rekursiva anropet sker:

```
let addarray (a : float []) =
    let s = ref 0.0
    let rec add_local n i =
        if i = n then !s
                 else s := !s + a.[i]; add_local n (i+1)
    in add_local (Array.length a) 0
```

(Den explicita typannoteringen på argumentet `a` är nödvändig för att få deklarationen att gå igenom typinferensen. Jag har inte dragit några poäng pm den fattas.)

UPPGIFT 6 (6 POÄNG)

a) Tre fall: ingen son (ett löv), en son, eller två söner:

```
type BinTree<'a> = Leaf of 'a | One of BinTree<'a> | Two of (BinTree<'a> * BinTree<'a>)
```

b) En rättfram traversering av trädet, där delträdens summor successivt läggs ihop:

```
let rec sumTree (t : BinTree<float>) =
    match t with
    | Leaf x      -> x
    | One t1      -> sumTree t1
    | Two (t1,t2) -> sumTree t1 + sumTree t2
```

Notera typannoteringen på argumentet till `sumTree`. Uden denna kommmer typen för + att defaulta till `int -> int -> int`, vilket då medför att `sumTree` kommer att få typen `BinTree<int> -> int`.

UPPGIFT 7 (5 POÄNG)

a) `list_and` tar en lista av booleans och tar "and" av alla dessa.

b) Vi vet:

```
[] : 'a list
= : 'b -> 'b -> bool
true : bool
(&&) : bool -> bool -> bool
List.head : 'c list -> 'c
List.tail : 'd list -> 'd list
```

Vi antar:

```
list_and : 'e
a : 'f
```

I vänsterledet (VL) appliceras `list_and` på `a`. Detta är typkorrekt endast om

```
'e = 'f -> 'g
```

för någon typvariabel `'g`. Vidare har VL typen `'g`. Låt oss nu gå igenom högerledet (HL), som är ett villkorsuttryck. För sådana gäller att villkoret måste ha typ `bool` samt att de båda grenarna måste ha samma typ. Från villkoret "`a = []`" erhåller vi således:

```
'f = 'b
'a list = 'b
```

och hela villkoret får då typ `bool`, vilket är korrekt. Ovanstående medför att

```
'e = 'a list -> 'g
```

Låt oss nu titta på grenarna. Den första (`true`) har typ `bool`. För att den andra grenen ska vara typkorrekt måste följande gälla:

```
'c list = 'f = 'a list (List.head a typkorrekt)
'd list = 'f = 'a list (List.tail a typkorrekt)
```

Detta kan bara gälla om `'c` = `'a` och `'d` = `'a`. Vidare måste gälla att

```
'f = 'a list = 'a list (list_and (List.tail a) typkorrekt)
'c = 'a = bool, 'g = bool (List.head a && list_and (List.tail a) typkorrekt)
```

Den första ekvationen ger inget nytt, men den andra medför att

```
'e = bool list -> bool
```

Med ovanstående typning får `List.head a && list_and (List.tail a)` typen `bool`, vilket stämmer med typen för den första grenen. `bool` blir också typen för hela HL. Typen på VL är `'g` = `bool` vilket stämmer. Vi har nu gått igenom båda leden, allt kunde typas och båda leden fick samma typ. Eftersom vi bara gjort minimala antaganden om typningen i varje steg erhåller vi en mest generell typning för `list_all`, som är

```
list_all : bool list -> bool
```