SI-möte #10, Programkonstruktion och Datastrukturer

Lösningsförslag

Elias Castegren & Kristiina Ausmees
elca7381@student.uu.se ∥ krau6498@student.uu.se

1.

```
(* printList(l)
   TYPE: string list -> unit
   PRE: ()
   POST: ()
   SIDE-EFFECTS: alla strängar i l skrivs ut på varsin rad på terminalen
   EXAMPLES: printList(["Hej","på","dig"]) = ()
             ("Hej\npå\ndig" skrivs ut på terminalen)
*)

(* VARIANT: antalet element i l*)
fun printList([]) = ()
  | printList(f::r) = (print (f ^ "\n"); printList(r));
```

Två varianter med funktioner ur List-biblioteket:

```
fun printList(l) = foldl (fn (s,_) => print(s ^ "\n")) () l;

fun printList(l) = ((map (fn (s) => print(s ^ "\n")) l); ());
```

Notera att map-uttrycket ovan returnerar en unit-lista med lika många element som `l`. För att uppfylla typen i specifikationen returneras därför `()` efter att map-uttrycket beräknats.

Det finns en inbyggd högre ordningens funktion i ML som heter `app`, med typen `('a -> unit) -> 'a list -> unit`. Den applicerar en funktion utan eftervillkor på varje element i en lista och returnerar till sist unit. Med den hade man kunnat skriva funktionen så här:

```
fun printList(l) = app (fn (s) => (print(s ^ "\n"))) l;
```

2.
Med While-loop:

Här använder vi en `let`-sats för att öppna strömmarna och initiera en radräknar-referens `i`, sen loopar vi så länge vi inte stöter på slutet på inströmmen och avslutar med att stänga båda strömmarna.

```
open TextIO;
(* lineNumbers(f)
   TYPE: string->unit
   PRE: f är namnet på en fil som går att läsa
   POST: ()
   SIDE-EFFECTS: En inström öppnas från filen f och en utström öppnas
                 till filen f.numbered. Alla rader i f skrivs till
                 f.numbered men med radnummer före. Alla strömmar stängs.
   EXAMPLES: lineNumbers("fil") = ()
             Om "fil" innhöll raderna
                Inte
                utan
                min
                ML-tolk!
             innehåller nu "fil.numbered"
                1. Inte
                2. utan
                3. min
                4. ML-tolk!
*)

fun lineNumbers(f) =
    let
        val ins = openIn(f)
        val os = openOut(f^".numbered")
        val i = ref 1
    in
        (while(not (endOfStream(ins))) do
            (output(os, Int.toString(!i) ^
                        ". " ^
                        valOf(inputLine(ins)));
             i := (!i + 1));
         closeIn(ins); closeOut(os))
    end;
```

Med rekursion:

Här använder vi en hjälpfunktion `transferLines` som givet en inström och en utström kopierar inströmmen och numrerar raderna. Huvudfunktionen öppnar strömmarna, skickar dem till `transferLines` och stänger sedan strömmarna.

```
open TextIO;
fun lineNumbers(f) =
    let
        (* transferLines(ins, os, i)
           TYPE: instream * outstream * int -> unit
           PRE: ins och os är läs- respektive skrivbara strömmar
           POST: ()
           SIDE-EFFECTS: Alla rader i ins skrivs till os, numrerade
                         från i och uppåt
        *)

        (*VARIANT: Antalet rader i ins innan filslut*)
        fun transferLines(ins, os, i) =
            if endOfStream(ins) then
                ()
            else
                (output(os, Int.toString(i) ^
                            ". " ^
                            valOf(inputLine(ins)));
                 transferLines(ins, os, i+1))

        val ins = openIn(f)
        val os = openOut(f ^ ".numbered")
    in
        (transferLines(ins, os, 1); closeIn(ins); closeOut(os))
    end;
```

3.
Poly/ML har inget bibliotek för slumpade tal så för att testa funktionen kan man antingen skriva en funktion i stil med `fun randomRange(a,b) = (a+b) div 2` eller använda någon annan ML-dialekt som har sluptalsmöjligheter (SML New Jersey och Moscow ML fungerar båda).

Först skriver vi en hjälpfunktion som jämför två tal och skriver ut ifall det ena är större, mindre eller lika med det andra. Genom att ge den returtypen `bool` kan vi senare använda den i en `if`-sats för att avgöra om användaren har matat in rätt gissning.

```
(* compareNumbers(guess, correct)
   TYPE: int * int -> bool
   PRE: ()
   POST: true om guess = correct, annars false
   SIDE-EFFECTS: skriver ut "Mindre!", "Storre!" eller "Ratt!" beroende på
                 hur correct förhåller sig till guess.
   EXAMPLES: compareNumbers(45, 50) = false
             <"Storre!" skrivs ut på terminalen>
*)
fun compareNumbers(guess, correct) =
    if guess > correct then
        (TextIO.print "Mindre!\n";
         false)
    else if guess < correct then
        (TextIO.print "Storre!\n";
         false)
    else
        (TextIO.print "Ratt!\n";
         true);
```

Sedan skriver vi en funktion som låter användaren gissa ett givet tal (vi väntar alltså med slumpningen).

```
(* guessNumber'(ans)
   TYPE: int -> int
   PRE: ()
   POST: ans
   SIDE-EFFECTS: låter användaren mata in tal i terminalen tills hon
                 matar in ans eller en tom rad. Instruktionerna
                 "Mindre!", "Storre!", "Ratt!" och "Du måste skriva
                 ett heltal!" skrivs till terminalen vid behov.
   EXAMPLES: guessNumber'(50) = 50
          <Användaren får gissa tills hon matar in 50 eller en tom rad>
*)
```

```
(* VARIANT: Antalet gissningar tills användaren matar in ans *)
fun guessNumber'(ans) =
    let
        val input = TextIO.inputLine(TextIO.stdIn)
        val guess = Int.fromString(valOf(input))
    in
        if input = SOME "\n" then
            ans
        else if guess = NONE then
            (print "Du maste skriva ett heltal!\n";
             guessNumber'(ans))
        else
            if compareNumbers(valOf(guess), ans) then
                ans
            else
                guessNumber'(ans)
    end;
```

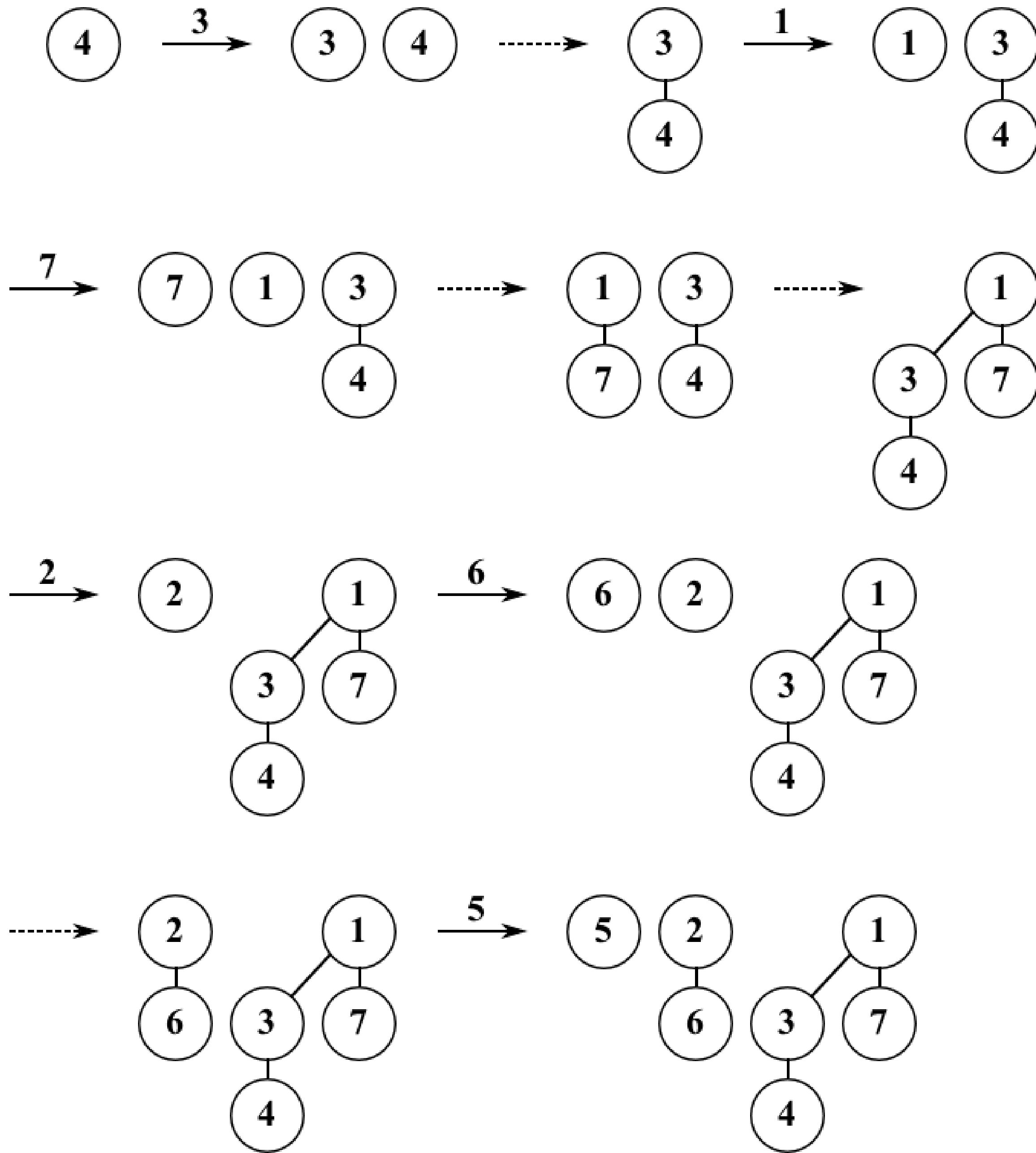
Slutligen skriver vi en funktion som slumpar fram ett tal och sätter igång spelet (genom att anropa hjälpfunktionen `guessNumber'`).

```
(* guessNumber(n)
   TYPE: int -> int
   PRE: n>0
   POST: ett slumpat tal mellan 0 och n
   SIDE-EFFECTS: "Gissa talet!" skrivs ut till terminalen. Sedan får
                 användaren spela ett parti "gissa talet", där svaret
                 är mellan 0 och n, i terminalen.
   EXAMPLES: guessNumber(100) = 50
         <Användaren får gissa tal tills hon gissar rätt eller ger upp>
*)
fun guessNumber(max) =
    let
        val rand = randomRange(0, max)
    in
        (TextIO.print("Gissa talet!\n");guessNumber'(rand))
    end;
```

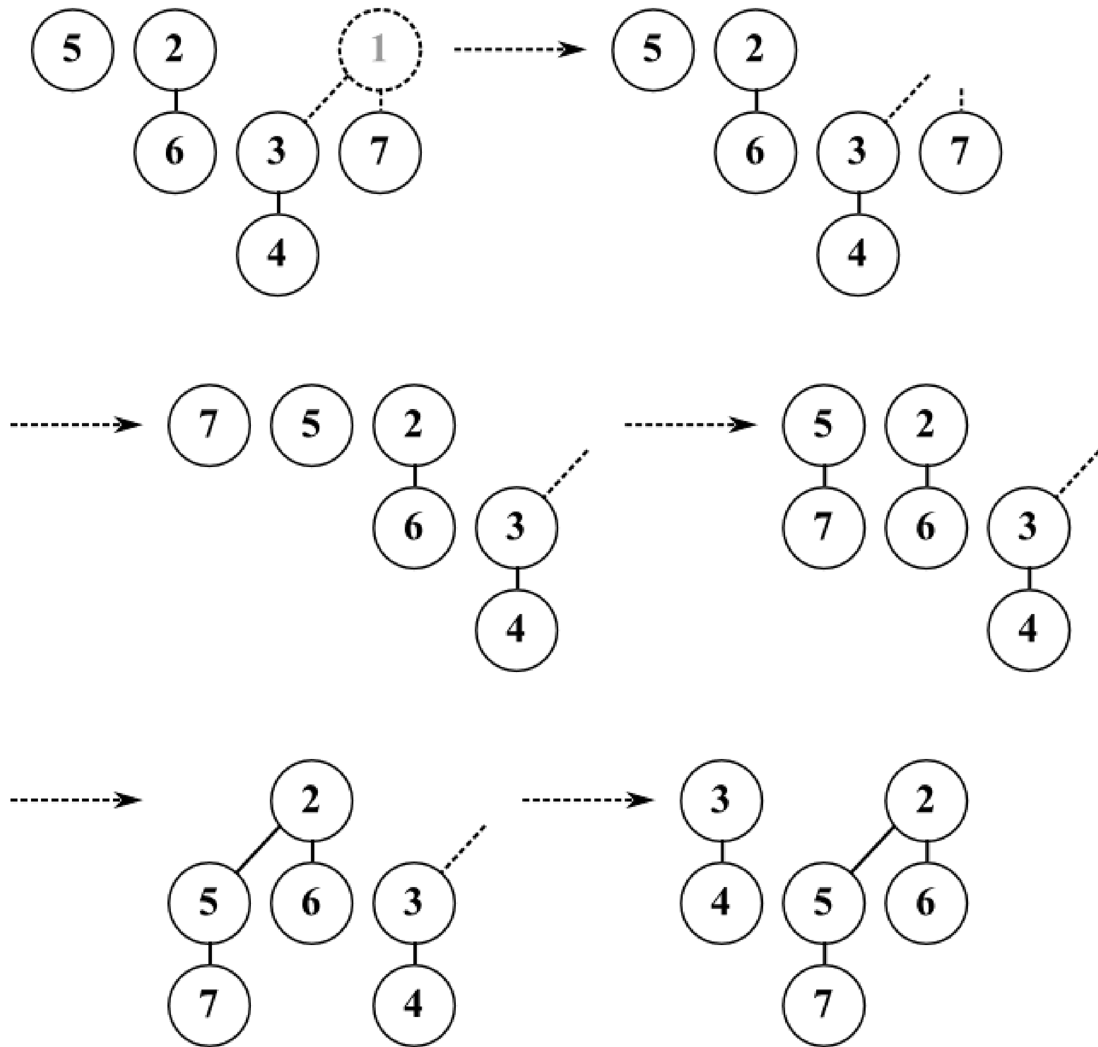
Här följer en helt imperativ version av `guessNumber`, som blir några rader kortare mest för att det är färre `fun`- och `let`-satser.

```
(* guessNumber(n)
   TYPE: int -> int
   PRE: n>0
   POST: ett slumpat tal mellan 0 och n
   SIDE-EFFECTS: Skriver ut "Gissa talet!" och låter sedan användaren
                 mata in tal i terminalen tills hon gissar rätt eller
                 matar in en tom rad. Instruktionerna "Mindre!",
                 "Storre!", "Ratt!" och "Du måste skriva ett heltal!"
                 skrivs till terminalen vid behov.
   EXAMPLES: guessNumber(100) = 50
          <Användaren får gissa tills hon matar in 50 eller en tom rad>
*)
fun guessNumber(n) =
    let
        val ans = randomRange(0,n)
        val input = ref NONE
        val guess = ref NONE
        val continue = ref true
    in
        (print "Gissa talet!\n";
         while (!continue) do
             (input := TextIO.inputLine(TextIO.stdIn);
              guess := Int.fromString(valOf(!input));
              if !input = SOME "\n" then
                  continue := false
              else if !guess = NONE then
                  print "Du maste skriva ett heltal!\n"
              else
                  if valOf(!guess) > ans then
                      print "Mindre!\n"
                  else if valOf(!guess) < ans then
                      print "Storre!\n"
                  else
                      (print "Ratt!\n";
                       continue := false)
             );
         ans)
    end;
```

4.



5.



6.
Ett binomialträd med rang n har 2^n noder. En binomial heap innehåller alltså
$t_0 \cdot 2^0 + t_1 \cdot 2^1 + t_2 \cdot 2^2 + \cdots + t_m \cdot 2^m$ noder, där t_i är ett eller noll beroende på om heapen har ett träd med rang i eller inte.

Ett binärt tal $d_m d_{m-1} d_{m-2} \dots d_0$ blir decimalt $d_0 \cdot 2^0 + d_1 \cdot 2^1 + d_2 \cdot 2^2 + \cdots + d_m \cdot 2^m$ där d_i är ett eller noll. Antalet noder i ett binomialträd är alltså detsamma som ett binärt tal enligt ovan där en etta betyder att heapen har ett träd med motsvarande rang.

1. $(52)_{10} = (110100)_2$ - rangerna hos träden i heapen är alltså 2, 4 och 5
   (index från höger för ettorna i det binära talet)

2. $(53)_{10} = (110101)_2$ - rangerna är 0, 2, 4 och 5

3. $(127)_{10} = (1111111)_2$ - heapen innehåller träd av alla ranger mellan 0 och 7

4. $(128)_{10} = (10000000)_2$ - heaper innehåller ett träd av rang 8

När antalet noder är $2^k - 1$ måste man slå ihop k stycken träd om man lägger in ett nytt element (se övergången från 127 till 128 noder ovan). Det tar alltså längre tid att lägga in nya element när man har många ettor i följd i den binära representationen av antalet noder.

7.
$heapsort(l) : int\ list \rightarrow int\ list$

1. Skapa en tom max-heap H

2. Sätt in alla element i l i H, ett och ett

3. Ta ut det största elementet ur H och lägg det först i en lista l'

4. Är H tom? Gå tillbaka till steg 3, annars returnera l'

Antag att listan som ska sorteras har $|l|$ element. Vi gör först $|l|$ stycken insättningar och sen $|l|$ stycken extraheringar (båda operationerna med komplexitet $\Theta(\log n)$, där n är antalet noder i heapen). Den totala komplexiteten blir alltså

$$|l|\Theta(\log |l|) + |l|\Theta(\log |l|) = \Theta(|l| \log |l|)$$

Med andra ord är den jämförbar med mergesort och quicksort!

En ML-implementation med en min-heap följer nedan. Heapimplementationen kommer från föreläsningarna och finns att hämta från kurshemsidan.

```
(* sort(l)
   TYPE: int list -> int list
   PRE: ()
   POST: l sorterad
   EXAMPLES: sort([5,3,4,2,1]) = [1,2,3,4,5]
*)
fun sort(l) =
    let
        (* extractAll(h)
           TYPE: binoHeap -> int list
           PRE: h uppfyller invarianten för binomiala min-heapar
           POST: en sorterad lista med alla element i h
           EXAMPLES: sort({En heap med elementen <3,4,5>}) = [3,4,5]
        *)
        (* VARIANT: Antalet element i heapen h *)
        fun extractAll(h) =
            if null(h) then
                []
            else
                let
                    val (e, h') = extractMin(h)
                in
                    e::extractAll(h')
                end
        val H = foldr (fn(e, h) => insert(h, e)) [] l
    in
        extractAll(H)
    end;
```

Tänk på att heapen är definierad som:

```
type binoHeap = binoTree list
```

En tom heap är alltså detsamma som en tom lista (se `foldr`-anropet). Därför kan man också använda listfunktioner som `null` för att se om en heap är tom.