LiU-FP2010 Part II, Linkping, 17–19 May 2010

Exercises, Lecture 2: Lazy Functional Programming

Henrik Nilsson

Problems 1, 3, and 4 are part of the examination. 2 and 5 are optional extras.

1. Implement the Sieve of Eratosthenes for computing prime numbers in Haskell. Recall that the sieve works as follows. Starting from the natural numbers from 2 and up, keep 2 as it is a prime, then strike out all multiples of 2 from the rest of the numbers. The smallest remaining number is also a prime, so keep it, then strike out all multiples of it form the the remaining numbers. And so on. (David Turner was the first to suggest that this algorithm could be implemented very elegantly and concisely in the purely-functional, non-strict language SASL in 1975: `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.)

2. A problem, due to the mathematician W. R. Hamming, is to write a program that produces an infinite list of numbers with the following properties:

   i The list is in ascending order, without duplicates.

   ii The list begins with the number 1.

   iii If the list contains the number x, then it also contains the numbers $2x$, $3x$, and $5x$.

   iv The list contains no other numbers.

   The following Haskell code solves the problem:

   ```
   merge xxs@(x:xs) yys@(y:ys) | x == y = x : merge xs ys
                               | x < y  = x : merge xs yys
                               | x > y  = y : merge xxs ys

   hamming = 1 : merge (map (2*) hamming)
                       (merge (map (3*) hamming)
                              (map (5*) hamming))
   ```

   Draw the four cyclic graphs that represent `hamming` after the first 1, 2, 3, and 4 elements have been printed.

3. Implement a simple spreadsheet evaluator as suggested in the lecture notes. For example, start with the following abstract syntax for the expressions in the spreadsheet cells:

   ```
   type CellRef = (Int,Int)

   data BinOp = Add | Sub | Mul | Div

   data Exp = LitInt Integer         -- Integer constants
            | Ref CellRef            -- Reference to the value of a cell
            | BinOpApp BinOp Exp Exp -- Binary operator application
            | Sum CellRef CellRef    -- Summing a range of cells
   ```

   Is there any weaknesses of this evaluator? How could you improve on that?

4. Lazy evaluation is very handy for implementing attribute grammar evaluators. The idea is is to write a recursive function traversing (derivation) trees with inherited attributes as extra arguments and synthesised attributes as results (or attributed tree nodes, depending on the application). Lazy evaluation will take care of evaluating the attributes in a suitable order (assuming the system of attribute equations has a solution).

   Here is a simple attributed grammar. It's overall purpose is to compute a version of the tree with all leaves sorted numerically as a synthesised attribute. Inherited attributes are indicated by $\downarrow$, synthesised attributes by $\uparrow$.

Productions			**Attribute Equations**		
S	$\rightarrow$	T	$T{\downarrow}itips$	$=$	$[]$
			$T{\downarrow}isorted$	$=$	$sort\ T{\uparrow}stips$
			$S{\uparrow}tree$	$=$	$T{\uparrow}tree$
T	$\rightarrow$	$Tip\ x$	$T{\uparrow}stips$	$=$	$x\ :\ T{\downarrow}itips$
			$T{\uparrow}ssorted$	$=$	$tail\ T{\downarrow}isorted$
			$T{\uparrow}tree$	$=$	$Tip\ (head\ T{\downarrow}sorted)$
T	$\rightarrow$	$Node\ T_L\ T_R$	$T_R{\downarrow}itips$	$=$	$T{\downarrow}itips$
			$T_L{\downarrow}itips$	$=$	$T_R{\uparrow}stips$
			$T{\uparrow}stips$	$=$	$T_L{\uparrow}stips$
			$T_L{\downarrow}isorted$	$=$	$T{\downarrow}isorted$
			$T_R{\downarrow}isorted$	$=$	$T_L{\uparrow}ssorted$
			$T{\uparrow}ssorted$	$=$	$T_R{\uparrow}ssorted$
			$T{\uparrow}tree$	$=$	$Node\ T_L{\uparrow}tree\ T_R{\uparrow}tree$

   Implement an evaluator for this grammar in Haskell. Note that the result is a one-pass algorithm for sorting a tree. Assuming the following definition of the tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

   the type signature for the main tree traversal function could be:

```
sortTreeAux :: Tree -> [Int] -> [Int] -> ([Int], [Int], Tree)
```

   where the second and third arguments correspond to, respectively, the inherited attributes *itips* and *isorted*, and the result tuple to the synthesised attributes *stips*, *ssorted*, *tree* in that order.

   (This exercise is inspired by Thomas Johnsson's paper Attribute Grammars as a Functional Programming Paradigm, FPCA 1987.)

5. Solve some problem(s) using *dynamic programming* in Haskell. If you have Internet access, you can find a number of suitable problems for example at the following addresses:

   - `http://en.wikipedia.org/wiki/Dynamic_programming`
   - `http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html`