

LiU-FP2016, Linköping, 23–27 May 2016
Problem Set 2: Lazy Functional Programming
Henrik Nilsson

1. Implement the Sieve of Eratosthenes for computing an (infinite) list of all 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 from the remaining numbers. And so on. (The function `filter` is helpful for the striking out numbers.) (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.(a) Given the following function to merge two ordered lists:

```
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
```

define the infinite list of Hamming numbers. (You may find using the function `map` or list comprehensions helpful.)
 - (b) (Optional) Draw the four cyclic graphs that represent `hamming` after the first 1, 2, 3, and 4 elements have been printed.
3. (a) Extend the spreadsheet evaluator developed during the lecture (code available here: <http://www.cs.nott.ac.uk/~psznhn/LiU-FP2016/Exercises/Sheet.hs>) with two new forms of expressions to compute the sum and average respectively of a range of cells:

```
data Exp = ...
         | Sum CellRef CellRef
         | Avg CellRef CellRef
```

You may find the function `range` and list comprehensions useful.
 - (b) Is there any weaknesses of this evaluator? Suggest a way to fix the problem.

4. Lazy evaluation is very handy for implementing attribute grammar evaluators. The idea 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
$S \rightarrow T$	$T\downarrow itips = []$
	$T\downarrow isorted = sort\ T\uparrow stips$
	$S\uparrow tree = T\uparrow tree$
$T \rightarrow Tip\ x$	$T\uparrow stips = x : T\downarrow itips$
	$T\uparrow ssorted = tail\ T\downarrow isorted$
	$T\uparrow tree = Tip\ (head\ T\downarrow ssorted)$
$T \rightarrow Node\ T_L\ T_R$	$T_R\downarrow itips = T\downarrow itips$
	$T_L\downarrow itips = T_R\uparrow stips$
	$T\uparrow stips = T_L\uparrow stips$
	$T_L\downarrow isorted = T\downarrow isorted$
	$T_R\downarrow isorted = T_L\uparrow ssorted$
	$T\uparrow ssorted = T_R\uparrow ssorted$
	$T\uparrow tree = Node\ T_L\uparrow tree\ T_R\uparrow tree$

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 = Tip 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 problem is inspired by Thomas Johnson's paper Attribute Grammars as a Functional Programming Paradigm, FPCA 1987.)

5. (Optional) 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>