Midlands Graduate School 2012, Birmingham, UK

Exercises Functional Programming (FUN)

Dr. Henrik Nilsson

The following exercises illustrate some of the points covered in the corresponding lecture. However, they are just suggestions for what you might want to do, and there is no particular order among them. So just go ahead and attempt whatever problems seem most interesting to you. And if you feel inspired to try something all-together different, all the better!

Lecture 1: Lazy Functional Programming

23 April 2012

1. Consider any remaining exercises from the lecture notes.

2. 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`.)

3. 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.

4. 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
   ```

5. 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.)

6. 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`

Lecture 2: Purely Functional Data Structures

24 April 2012

1. Consider any remaining exercises from the lecture notes.
2. Write a function `drop :: Int -> RList a -> RList a` that deletes the first n elements for a binary random-access list. Your function should run in $O(logn)$ time. (From *Purely Functional Data Structures* by Chris Okasaki, 1998.)
3. Reimplement binary random-access lists using a sparse representation such as:

   ```
   data Tree a = Leaf a | Node Int (Tree a) (Tree a)
   type RList a = [Tree a]
   ```

   (From *Purely Functional Data Structures* by Chris Okasaki, 1998.)
4. Implement `drop` as specified above for *skew* binary random-access lists.

Lecture 3: Monads

25 April 2012

1. Consider any remaining exercises from the lecture notes.
2. The *Identity Monad* can be understood as representing *effect-free* computations:

   ```
   type I a = a
   ```

   (a) Provide suitable definitions of `return` and `>>=`.

   (b) Verify that the monad laws hold for your definitions.
3. Verify that `Maybe a` indeed is a monad by verifying the monad laws for `mbReturn` and `mbSeq`.
4. It turns out that many familiar data types in fact can be viewed as monads. For example, `[a]` can be understood as representing a computation with zero or more possible results ("nondeterminism"), and thus forms a monad with the appropriate definitions for `return` and `>>=`. Without "cheating" by looking ahead at the next lecture, show that `[a]` is a monad. *Hint:* `return` corresponds to a computation with exactly one result, while `>>=` needs to feed all possible outcomes form the first computation into the second, and then collect all possible results from that.

Lecture 4: More about Monads

26 April 2012

1. Consider any remaining exercises from the lecture notes.

2. Below are the type signatures for a number of monad utility functions from the Haskell prelude and the module `Monad`. Define these utilities in terms of the basic monad operations. (If it is not reasonably clear from the type signatures what the intended meaning of each function is, ask!)

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_     :: Monad m => (a -> m b) -> [a] -> m ()
when      :: Monad m => Bool -> m () -> m ()
foldM     :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
liftM     :: Monad m => (a -> b) -> (m a -> m b)
```

3. The Diagnostics monad `D` mentioned in the lectures represents computations that can emit error messages and, if necessary, give up completely and stop. Here is a variation of some of the operations on this monad:

Operation	Type	Purpose
`emitErrD`	`String -> D ()`	Emit an error message.
`failD`	`String -> D a`	Emit an error message and stop.
`failIfErrorsD`	`String -> D a`	Stop if one or more error messages have been emitted.
`stopD`	`D a`	Stop.
`runD`	`D a -> (Maybe a, [String])`	Run a diagnostic computation, returning any result and a list of all emitted error messages.

   - Think about what effects the diagnostics monad combine. For example, there is a standard notion of a *writer monad*:

     type W a = (a, T)

     for any type `T` that is a *monoid*: has an identity element and an associative binary operation. Such a monad is typically used for logging purposes. For example, `T` could be taken to be lists of error messages (strings), with list concatenation `++` as the binary operation to combine the output from sequentially composed computations and `[]` as the identity element. Would a writer monad be suitable for the logging part of the diagnostics monad, or is a more general notion of state needed?
   - Implement the diagnostics monad from scratch.
   - Reimplement the diagnostics monad by using monad transformers to define the basic monad, and then defining the application-specific interface described above in terms of the standard operations for the monad obtained through the transformations.