RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2.html

Prof. Dr. Jürgen Giesl
Darius Dlugosz

Exercises *Functional Programming* – Sheet 4

Solutions will be collected until Thursday, May 31, 2001 (in 2 weeks!).

Important:

- On Tuesday, May 22, there is a practical course (Übung) instead of lecture course (Vorlesung).
- On Friday, May 25, there is an extra practical course (Übungsstunde mit zusätzlichen Beispielen) instead of lecture course (Vorlesung).

Exercise 1 (4 points)

(a) Under what conditions on `xs` and `ys` does the following equation hold?

```
[ x | x <- xs, y <- ys ] = [ x | y <- ys, x <- xs ]
```

(b) Define a function `pairs` so that `pairs n` is a list of all distinct pairs of integers $1 \leq x, y \leq n$.

(c) Write a program to find all quadruples (a, b, c, d) in the range $0 < a, b, c, d \leq n$ such that $a^2 + b^2 = c^2 + d^2$

(d) How can the function `member :: [Int] -> Int -> Bool` be defined using a list comprehension and an equality test?

(e) A number is perfect if it is the sum of its divisors. Define a list of perfect numbers.

(f) Make a list of all natural numbers which are not dividable by any square number.

$$\{n \in \mathbb{N} \mid \forall m > 1.\ m^2 \not| n\}$$

(g) List comprehensions provide an alternative to a combinatorial style based on `map`, `filter`, and `concat`. There are also a number of derived rules for reasoning about list comprehensions, including the following:

```
[ f x | x <- xs ] = map f xs
```

```
[ x | x <-xs, p x ] = filter p xs
[ e | Q, P ] = concat [ [ e | P ] | Q ]
```

The function `filter` (in the second equation) is defined by

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x: filter xs
                | otherwise = filter p xs
```

Convert the following list comprehensions to combinatory style

(i) `[ (x, y) | x <- [1 .. n], odd x, y <- [1 .. n] ]`

(ii) `[ (x, y) | x <- [1 .. n], y <- [1 .. n], odd x ]`

Are they equal? Compare the costs of evaluating the two expressions.

Exercise 2 (4 + 3* points)

(a) Define `dither` by

```
dither = yes
         where yes   = "Yes." ++ no
               no    = "No!" ++ maybe
               maybe = "Maybe?" ++ yes
```

What will be printed if `dither` is typed in a session? Draw the graph of the cyclic structure that represents this value.

(b) Given the function

```
mer :: [Integer] -> [Integer] -> [Integer]
mer (x:xs) (y:ys)
  | x < y    = x : mer xs (y:ys)
  | x == y   = x : mer xs ys
  | x > y    = y : mer (x:xs) ys
```

we define `hamming` by

```
hamming :: [Integer]
hamming = 1 : mer (map (2*) hamming)
                  (mer (map (3*) hamming)
                       (map (5*) hamming))
```

Initially, `hamming` will be represented by the following cyclic structure:

```
1 : mer (map (2*) •) (mer (map (3*) •) (map (5*) •))
```

After the first seven elements of `hamming` have been printed, the above structure will have reduced to the following:

```
1:2:3:4:5:6:8: mer (10:map (2*) •) (9:mer (map (3*) •) (10:map (5*) •))
```

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

(c) Define the function `runningSums :: [Int] -> [Int]` which calculates the running sums $[0,\ a_0,\ a_0 + a_1,\ a_0 + a_1 + a_2,\ \ldots]$ of a list $[a_0,\ a_1,\ a_2,\ \ldots]$.

(d)* The data type `Tree` is defined as follows:

```
data Tree a = Tree a [Tree a]
```

Define the function `minTree :: Ord a => Tree a -> Tree a` which replaces all values in all nodes with the minimum of all nodes' values in only *one* pass through the tree.

Exercise 3 (4 points)

(a) Define the *then* operator `(>>)` using the *bind* operator `(>>=)`.

(b) Give a definition of the function `repeat :: IO Bool -> IO () -> IO ()` so that `repeat test oper` has the effect of repeating `oper` until the condition `test` is `True`.

(c) Define a new data type `data STE a = ...` which combines the state monad `ST` and the failure monad `Maybe`. Declare `STE` as an instance of class `Monad` and declare `STE a` as an instance of class `Show`.

Using `STE`, define the function `eval` which evaluates the terms of data type `data Term = Con Float | Div Term Term` .
`eval (Div a b)` returns the value (a/b) and the number of divisions used iff $b \neq 0$ and prints an error message for $b = 0$.

The monad `Maybe` and the monad `ST` are defined as follows:

```
data Maybe a = Nothing | Just a

data ST a = MakeST (Int -> (a, Int))
```

```
instance Monad Maybe where
   return          = Just
   Nothing >>= q   = Nothing
   Just x  >>= q   = q x

instance Monad ST where
     return x        = MakeST (\s -> (x, s))
     MakeST f >>= q  = MakeST (\s -> let (x, s') = f s
                                     in apply (q x) s')

apply :: ST a -> Int -> (a, Int)
apply (MakeST f) s = f s
```