FUNCTIONAL PROGRAMMING MT2018

Sheet 6

11.1 What are the natural folds on *Bool* and

```
> data Day = Sunday | Monday   | Tuesday | Wednesday |
>                     Thursday | Friday  | Saturday
```

11.2 Given that the ordering on *Bool* is the one that would be obtained by **deriving**(*Ord*), to what logical function of two variables does (<=) correspond?

11.3 Write out the fold function for the data type

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
```

and use it to define a function

```
> isIn :: Eq a => a -> Set a -> Bool
```

which tests whether an element appears as a value in the tree. Hence define a function

```
> subset :: Eq a => Set a -> Set a -> Bool
```

which tests whether all the elements of the first set are elements of the second. Use this to implement the test

```
> instance Eq a => Eq (Set a) where
>   xs == ys = (xs `subset` ys) && (ys `subset` xs)
```

for equality of the sets represented by two trees from *Set*.

11.4 Define a function

```
> find :: Eq a => a -> BTree a -> Maybe Path
```

which searches for a value in the leaves of a *BTree*,

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

returning a path, a sequence of *go left* and *go right* instructions, from the root to the leftmost occurrence of the value, if there is one, where

```
> data Direction = L | R
> type Path = [ Direction ]
```

You should aim to make use of folds and maps where possible.

12.1 A *queue* is a data type with (at least) four operations

```
> empty   :: Queue a
> isEmpty :: Queue a -> Bool
> add     :: a -> Queue a -> Queue a
> get     :: Queue a -> (a, Queue a)
```

The value of *empty* is a queue with nothing in it; a queue satisfies *isEmpty* if all of the values that have been added to it have already been removed; *add* puts a value into a queue (ensuring that it is non-empty); and *get* returns the oldest value still waiting in the queue, along with a queue from which just that value has been removed.

Implement a queue type using a list of the elements in the queue in the order in which they joined. That is, give a declaration of the *Queue* type, and implement each of these four functions.

Estimate roughly how expensive your operations are. Would your answer be any different if the queue were represented by a list of its remaining elements in the reverse of the order in which they join the queue?

Reimplement the *Queue* using two lists of elements, *front* and *back* so that the elements in the queue are those in the list $front + reverse\ back$. What effect does this have on the cost of the operations?

12.2 The Fibonacci sequence

```
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

grows very quickly (each value is about 1.6 times bigger than its predecessor).

Use this definition in a GHCi script and try evaluating *fib* 10, *fib* 20 and *fib* 30. Give a brief explanation of why the later calls are so slow.

Let $two\ n = (fib\ n, fib\ (n+1))$, and synthesize a definition of *two* by direct recursion. Use this to give a more efficient definition of *fib*. How does the time it takes to calculate $fib\ n$ in this way depend on n?

Roughly how big is the 10 000th Fibonacci number? You might want to use

```
> roughly :: Integer -> String
> roughly n = [head xs] ++ "e" ++ show (length xs - 1)
>             where xs = show n
```

to produce a readable estimate.

Let F be the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, and F^n be its nth power, the product of n copies of it.

Explain why $F^n = \begin{pmatrix} \mathit{fib}\ (n-1) & \mathit{fib}\ n \\ \mathit{fib}\ n & \mathit{fib}\ (n+1) \end{pmatrix}$ for $n \geqslant 1$. Use the function *power* from the lecture notes to calculate F^n in no more than about $2 \log n$ matrix multiplications, and use this to give another more efficient definition of *fib*.

Roughly how big is the 1 000 000th Fibonacci number?

Geraint Jones, 2018