CHALMERS TEKNISKA HÖGSKOLA
Datavetenskap
Jan-Willem Roorda

Monday, October 18th, 2004
Functional Programming
INN040/TDA450

Exam Functional Programming

Monday, October 18th, 2004, 8.30-12.30.
Examiner: John Hughes.
Questions during the exam will be answered by Jan-Willem Roorda, tel 031-7721023.

Permitted aids:
English-Swedish or English-other language dictionary.

- Begin each question on a new sheet. Write your personal number on every sheet.
- You may lose marks for unnecessarily long, complicated, or unstructured solutions.
- Full marks are awarded for solutions which are elegant, efficient, and correct.
- You are free to use any Haskell standard functions, including those whose definitions are attached, unless the question specifically forbids you to do so.
- You may use the solution of an earlier part of a question to help solve a later part, even if you did not succeed in solving the earlier part.
- The exam consists of 4 questions. Chalmers students need 23 points to pass the exam. GU students need 27 points to pass.

1. (a) Define the concept of lazy evaluation. (2 p)

   (b) Give two advantages and one disadvantage of lazy evaluation. (3 p)

   (c) Give an example of an expression that terminates under lazy evaluation, but would not terminate under eager evaluation. (2 p)

   (d) The *Fibonacci* numbers are given by:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_{n+2} &= F_n + F_{n+1}
\end{aligned}
$$

   Give an *efficient* definition of the infinite list of Fibonacci numbers. (4 p)

2. (a) A list `xs` is a *prefix* of a list `ys` if `ys == xs ++ zs` for some (possibly empty) list `zs`. For example: `"add"` is a prefix of `"address"`. But `"dress"` is *not* a prefix of `"address"`. Define a function

```
prefix :: Eq a => [a] -> [a] -> Bool
```

   such that `prefix xs ys` returns `True` if `xs` is a prefix of `ys`, and returns `False` otherwise.

   (You are *not* allowed to use the function `isPrefixOf` in your definition of `prefix`.) (4 p)

   (b) A list `xs` is a *subsequence* of a list `ys` if `ys == as ++ xs ++ bs` for some (possibly empty) lists `as` and `bs`. For example, `"dre"` is a subsequence of `"address"`. But `"adds"` is *not* a subsequence of `"address"`. Define a function

```
subsequence :: Eq a => [a] -> [a] -> Bool
```

   such that `subsequence xs ys` returns `True` if `xs` is a subsequence of `ys`, and returns `False` otherwise. (5 p)

   (c) Define a function `grep`

```
grep :: String -> String -> String
```

   such that `grep xs ys` returns every line from `ys` that contains `xs` as a subsequence.

   For example:

```
grep "ell" "hello\nclouds\nhello world\nthe sky is blue"
==
"hello\nhello world\n"
```

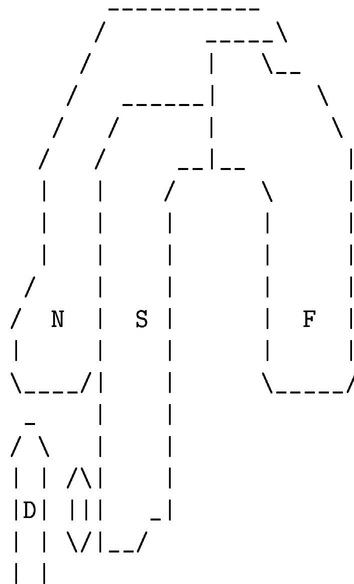
   (3 p)

3. This question concerns *map colouring*: the problem of choosing colours for each country on a map, so that no two neighbouring countries are assigned the same colour. It is known that four colours are enough to colour any planar map.

   We shall represent countries and colours by strings, and define

```
type Colour = String
type Country = String
colours = ["red","yellow","blue","green"]
countries = ["Norway","Sweden","Finland","Denmark"]
```

   We shall consider a map of Scandinavia as an example:



   We shall consider that Sweden and Denmark are neighbours (because of the bridge between Sweden and Denmark), and so must be assigned different colours.

   (a) Define a function

   ```
   neighbours :: Country -> [Country]
   ```

   which given a country name, returns a list of all the neighbouring countries. Your definition need only work for Scandinavia: it is in this function definition that you encode the information in the map above. (2 p)

   In the rest of the question you should not refer to particular countries except via the function `neighbours` and the variable `countries`. The

rest of your code should continue to work even if you change these definitions to represent a different map.

(b) We will represent an assignment of colours to countries as a *colouring*:

```
type Colouring = [(Country, Colour)]
```

For example, `[("Norway","blue"),("Denmark","red")]` records the facts that Norway has been coloured blue, and Denmark has been coloured red; other countries have no colour as yet.
Define a function

```
safeColour :: Colouring -> Country -> Colour -> Bool
```

which returns `True` if the given colouring can be extended by colouring the given country with the given colour, without assigning two neighbours the same colour. For example,

```
safeColour [("Norway","blue"),("Denmark","red")]
           "Sweden"
           "green"
 == True
```

because none of Sweden's neighbours is already coloured green. (4 p)

(c) Define a function

```
colour :: [Country] -> [Colouring]
```

which returns a list of *all* the possible ways to colour the given countries, so that no neighbours are assigned the same colour. (6 p)

4. In this question we will look at different kinds of trees.

(a) Consider the following type for binary trees.

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf  a
```

i. Write the tree

```
        1
      /   \
     4     5
   /   \
  9     3
```

as a value of the data type above. (1 p)

ii. Define a function `elemTree` that checks whether an element appears in a tree. (Warning: do *not* assume the tree is ordered.) Also give the type of this function. For example:

```
elemTree 3 (Node 2 (Leaf 1) (Leaf 3))
  == True

elemTree 'z' (Node 'd' (Leaf 'a') (Leaf 'c'))
  == False
```

(3 p)

iii. Define a function

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

such that `mapTree f t` gives the tree resulting from applying the function `f` to each element in `t`. For instance:

```
mapTree (+1) (Node 1 (Leaf 1) (Leaf 3))
  == Node 2 (Leaf 2) (Leaf 4)
```

(3 p)

(b) Now, consider this type of trees.

```
data GTree a = GNode [GTree a]
             | GLeaf  a
```

i. Write the tree

```
      /\
     /  \
    /|\  5
   9 2 3
```

as a value of the data type above. (1 p)

ii. Define a function `gSum :: GTree Int -> Int` that sums all the elements at the leafs of a `GTree` of integers. For instance:

```
gSum (GNode [GLeaf 2, GLeaf 3, GNode [GLeaf 1,GLeaf 2]])
 == 8
```

(4 p)

iii. Define a function `flatten :: GTree a -> [a]` that flattens a `Gtree` to a list. That is, `flatten t` returns a list containing all elements appearing at the leafs of `t`. For instance:

```
flatten (GNode [GLeaf 2, GLeaf 3, GNode [GLeaf 1,GLeaf 2]])
 == [2,3,1,2]
```

(4 p)

iv. Give an instance definition that makes the type `Gtree a` an instance of the `Eq` class for every type `a` that is in the `Eq` class. (Equality should be defined in the straight-forward way.) (6 p)