CA320 Computability and Complexity

Haskell Lab 6: User-Defined Data Types

Aim

The aim of this lab is to help you to learn about defining your own types in Haskell and to define functions over these types.

1. The Day of the Week

You are to write a function that, given a date, tells you the day of the week for that date. There is actually a rather complicated formula called *Zeller's Congruence* which works out the number of the day of the week given any date on the modern calendar, but we will try and solve it ourselves by breaking it down into smaller more manageable parts.

(a) To begin, you should give suitable type declarations for weekdays, months and dates. Weekdays should be defined using an enumerated type with possible values:
`Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday`
and should belong to the type classes `Enum` and `Show`. Months should also be defined using an enumerated type with possible values:
`Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec`
and should belong to the type classes `Enum` and `Read`. Monthdays and years should be represented using `Int`, and dates should be represented as a tuple giving `(monthday,month,year)` e.g. 24th October 2017 should be represented as `(24,Oct,2017)`.

(b) Then, you should write a function `leap :: Int -> Bool` that determines if a given year is a leap year:

```
> leap 2000
True
> leap 1900
False
> leap 2010
False
> leap 2012
True
```

A year is a leap year if it is divisible by 4, unless it is also divisible by 100. A year which is divisible by 100 is only a leap year if it is also divisible by 400. Thus, the following years are **not** leap years:

1300, 1400, 1500, 1700, 1800, 1900

because they are divisible by 100 but **not** by 400, but the following years **are** leap years:

1200, 1600, 2000

because they **are** divisible by both 100 and 400.

To check your answer is correct (and to register your progress), you should save your program to a file named `leap.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

(c) Now we need to deal with the numbers of days in each month. Define a function `mLengths :: Int -> [Int]` that takes a year and returns a list containing the number of days in each month of that year:

```
> mLengths 2015
[31,28,31,30,31,30,31,31,30,31,30,31]
> mLengths 2016
[31,29,31,30,31,30,31,31,30,31,30,31]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `mLengths.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

(d) The Gregorian calendar is only correct for years 1753 onwards (to see why, try the unix command `cal 9 1753` - there were actually riots over these 'lost' 11 days). Define a function `numDays` that, when given a date, returns the number of days since 31st December 1752. You should be able to use the `leap`, `mLengths` and `fromEnum` functions for this purpose. For example:

```
> numDays (7,Jan,1752)
7
> numDays (13,Mar,2054)
110010
```

To check your answer is correct (and to register your progress), you should save your program to a file named `numDays.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

(e) Define the function `dayOfWeek` that, when given a date, returns the day of the week on that date. You should be able to use `numDays`, `toEnum`, `` `mod` `` and the fact that 31st December 1752 was a Sunday for this purpose. For example:

```
> dayOfWeek (1,Jan,2000)
Saturday
> dayOfWeek (2,Jan,2000)
Sunday
> dayOfWeek (7,Jan,2000)
Friday
> dayOfWeek (13,Mar,2054)
Friday
```

Make sure that you test your function against a calendar (try the unix `cal` command e.g. `cal 1 2000`). Use your `dayOfWeek` function to find out what day of the week you were born on.

To check your answer is correct (and to register your progress), you should save your program to a file named `dayOfWeek.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

2. Monkey Puzzle Sort

A binary search tree has a value at each node. The left subtree of each node holds only values less than at the node, and the right subtree holds only values greater than or equal to that at the node. For example, the following is a binary search tree:

```
    5
   / \
  1   7
   \
    3
```

(a) Define a parametric data type `Tree a` to represent a binary tree with values of type `a` as given in lectures. This should belong to the type classes `Read` and `Show`.

(b) Write a recursive function `addNode :: Ord a => a -> Tree a -> Tree a` which, when given a value and a binary search tree, will add the value at the correct position in the tree. In order to insert the value at the correct position, if it is less than the value at the current node, then it is placed in the left subtree, otherwise it is placed in the right subtree. The value is placed at the first unoccupied node (null tree). For example, if the integer 4 were added to the above tree, the following tree would result:

```
    5
   / \
  1   7
   \
    3
     \
      4
```

To check your answer is correct (and to register your progress), you should save your program to a file named `addNode.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

(c) Write a recursive function `makeTree :: Ord a => [a] -> Tree a` which, when given a list of values, will create a binary search tree by inserting the head of the list into the correct position in the tree created from the tail of the list. For example, applying `makeTree` to the list `[4,3,1,7,5]` would create the tree given above. This function should make use of the `addNode` function.

To check your answer is correct (and to register your progress), you should save your program to a file named `makeTree.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

(d) The values in a tree can be converted into a list by traversing the tree in a specified order. For example, an inorder traversal traverses the left subtree first, then places the root in the result, and then traverses the right subtree. Write a recursive function `inOrder :: Tree a -> [a]` which, when given a tree, will return the list giving the result of an inorder traversal of the tree. For example, applying `inOrder` to the tree given above would give the list `[1,3,4,5,7]`.

To check your answer is correct (and to register your progress), you should save your program to a file named `inOrder.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein

(e) Monkey puzzle sort works by creating a binary search tree from a list, and then traversing the list in inorder. Write a function `mpSort :: Ord a => [a] -> [a]` which, when given a list of values, will return the list in ascending order using monkey puzzle sort. For example, applying `mpSort` to the list `[4,3,1,7,5]` would give the list `[1,3,4,5,7]`. This function should make use of the `makeTree` and `inOrder` functions.

To check your answer is correct (and to register your progress), you should save your program to a file named `mpSort.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein