

Week 4 Practical Class Exercises

Developing Programs in Haskell

Objectives

The aim of this class is to help you learn more about defining Haskell functions and take you through the process of *developing* definitions by breaking the problem down into more manageable parts, then combining them to give a complete solution.

Exercise 1 (The Day of the Week)

You are to write a function that, given a 20th or 21st century date in the format (day, month, year) tells you the day of the week for that date. For example:

```
*Main> dayOfWeek (1,1,1901)
"Tuesday"
*Main> dayOfWeek (2,1,1901)
"Wednesday"
*Main> dayOfWeek (15,3,2006)
"Wednesday"
```

How should we approach such a problem? There is in fact 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. I presume that you don't know what the formula is and I can't remember it either. We could go searching for a reference, but since we are only interested in dates from the beginning of the 20th century, we might be able to figure something out for ourselves.

If we knew what day the first day of the 20th century was (a Tuesday, I recall) and we could work out the total number of days up to the date we are asking about, that would seem to be a good start. There are quite a few things that need to be considered; the problem of leap years, for example.

To begin, here are some type synonym declarations that will make your script more readable:

```
type Day    = Int
type Month  = Int
type Year   = Int
type Date   = (Day,Month,Year)
```

First, you should write a function called `leap` that determines if a given year is a leap year.

```
*Main> leap 1996
True
*Main> leap 2006
False
```

A year is a leap year if the remainder on dividing it by 4 is zero. The mod or rem function will be useful. (Look in the Prelude if you aren't sure what these functions do.)

Before proceeding, thoroughly test your leap function to make sure that it works properly.

Whenever you design a function, ALWAYS test it to make sure that it does what you expect BEFORE you use in another function.

Now we need to deal with the numbers of days in each month. I have written the following two function definitions for you.

mLengths takes a year and returns the number of days in each month of that year:

```
*Main> mLengths 2006
[31,28,31,30,31,30,31,31,30,31,30,31]
*Main> mLengths 2004
```

Here is the definition:

```
mLengths :: Year -> [Int]
mLengths year
  = [31,feb,31,30,31,30,31,31,30,31,30,31]
  where feb
        | leap year = 29
        | otherwise = 28
```

The following function numDays is given a date in the form described above and returns the number of days since 31st December, 1900:

```
numDays :: Date -> Int
numDays (day,month,year)
  = day                -- days this month
  + sum (take (month-1) (mLengths year)) -- days this year
  + (year-1901) * 365 + (year-1901) `div` 4 -- days this century
```

Let's explain how this function works. Consider numDays(18,7,2011).

- The last line works out the number of days in the whole years 1901–2010 at 365 days per year, and the 'div' 4 part takes account of the number of leap years.
- The second line uses mLengths to give us the days of the months of the year 2011. Then it takes the completed months of the year, then sums them.
- Finally, the first line adds on 18, the number of days so far in the month we are asking about.

For example:

```
*Main> numDays (18,7,2011)
40376
```

You should type these definition into your script. (I ask you to type them rather than cut and paste, to force you to read them carefully.)

Notice how `numDays` uses the function `mLengths` and `mLengths` uses the function `leap` that you wrote. When you write programs, this is the way you usually work, building up more complicated functions by combining simpler ones. The rest of the development of your script should follow the same approach, by building on `numDays`.

The next obvious thing we need is a function, say:

```
dayName :: Int -> String
```

which, given the number of a day in the range 0–6, returns a string containing the name of the day. For example:

```
*Main> dayName 0
"Monday"
*Main> dayName 5
"Saturday"
```

(Note that I chose Monday to be day 0 for a good reason that will soon become clear: the last day *before* the 20th century was a Monday.) I suggest that you use simple *constant patterns* in your definition of `dayName`.

Once you have `dayName` working (and thoroughly tested), you should be able to use `numDays`, `dayName`, `mod` and the fact that the 31st December 1900 was a *Monday*, to construct a function:

```
dayOfWeek :: Date -> String
```

Here are some hints and suggestions. Try to write a function that works correctly for dates between (1, 1, 1901) and (6, 1, 1901) first, then see if you can modify it to work in general. Remember that your function does not have to work for dates before (1, 1, 1901). Make sure that you test your function against a calendar (Remember the unix `cal` command?)

Once you have succeeded, and thoroughly tested your script, show it to your tutor. Of course your script should be well laid out and include plenty of informative comments. It should also include a header comment giving details of the author, date and purpose of the script.

Exercise 2 (Product of the elements of a list of numbers)

There is a standard prelude Haskell function `product` that multiplies together all the elements of a list of numbers, but I want you to define your own, say `myProduct`. It should be fairly easy — look at the definition of `sum` (another standard prelude function) in Chapter 7.3 of the textbook for clues. The sample scripts from the textbook are available in `/dept/dcs/comp1100/public/Thompson/` or through the COMP1100 web site. Don't be tempted to copy the definition from the standard prelude, unless you think you can clearly explain the definition to your tutor and to me!

Exercise 3 (Converting a string to upper case)

The Haskell library `Data.Char` has a pre-defined function `toUpper :: Char -> Char` which converts single lower case letters to upper case. Experiment with this function to find out how it behaves (try it on a variety of characters, not just letters). To load a library like `Data.Char` into GHCi, use the command:

```
:module +Data.Char
```

(or `:m +Data.Char`) The `+` sign tells GHCi to add this module to the others that are already loaded. If you leave it out, any previously loaded modules will be replaced.

You should also look at the documentation for `Data.Char` and the other Haskell hierarchical libraries. The documentation is reachable from the Haskell page of the COMP1100 website.

Define a function that converts all the lower case letters in a *string* to upper case, and leaves the others unchanged. For example, `stringToUpper :: String -> String`. (Remember that type `String` is the same as type `[Char]`.) Your script containing this function must include an `import` statement:

```
import Data.Char
```

to make the functions defined in that library available within that script. When you load a script with an `import` statement into GHCi, that library is also loaded, so there is no need for the `:module Data.Char` command.

Show your tutor your completed work.

Make sure you log out to terminate your session.