CS256 Coursework 2016-17 Part 1

Steve Luci Matthews

Date issued: October 27th 2016 (Thursday, Term 1, Week 4)
Last update: 11/11/15

Holy heart failure, Batman! FP is truly awesome!

This is the first of two parts which compose the coursework for CS256. In turn each part is weighted 15% and 25% of the total module credit, followed by a two-hour June exam counting for 60% credit. *Part 1* covers the following themes of FP.

1. Programming in FP using first order mathematics.
2. `where` clauses as used in Haskell.
3. Non-recursive *function* definitions.
4. *Tail recursive* function definitions
5. First-order iteration in FP (i.e. modelling a *state* using tail recursion).
6. Good programming practice in (first order) FP.
7. Nested tail reursion (i.e. one tail recursion within another).

Daphne, Velma, Shaggy, Fred, & Scooby-Doo

For the Exercises below prepare an answer in the form of a well commented, well structured, well typed Haskell program, placed together in a **single file** having the name `ass16-1.hs` , and submitted online using *Tabula* (see the URL on the CS256 home page for a link to the CS256 Tabula page). Your submission will be individually visually inspected and assessed by the module organiser for the highest standards of functional programming practice, tested upon the Hugs implementation as installed in the DCS Linux systems, and individual feedback returned. Remember to help **me** the reader to help appreciate your good work by providing excellent supporting comments of the form,

```
{- Explanation on something
   ========================
   Explanation goes here ...
-}
Haskell code for something goes here ...
```

In the discipline of *symbolic logic* such hand coded evaluations are accepted as *proofs*, that is, a finite sequence of logically correct inferences. For us as functional programmers starting out at the very beginning becoming familiar with such (tiny) hand coded evaluations is a key skill in appreciating an essential difference between FP and (say) Java.

In many programming paradigms the notion of *function* is carried over from mathematics. However, FP is one of the precious few paradigms to insist that functions be *pure*, that is, they are to be no more and no less correct than would be found in pure mathematics or logic. The whole ethos of FP is to discover the joy of using of simple mathematical functions in FP. Essential points to bear in mind as you undertake CS256 are:

(1) Each function must have a declared type as part of good programming practice;

(2) A thorough case analysis in a function definition will save a lot of debugging pain and time later;

(3) If there is a looped computation then ensure that it is one that will make *progress* in each recursive call;

(4) In Haskell the type `String` is not built-into the language, nor does it miraculously appear from some library. Rather, `String` is a synonym (i.e. just another name) for the type `[Char]` (list of characters).

The coursework

As you will recall I went to great lengths to ensure that all students taking CS256 would be properly briefed upon a specific medical situation that could arise in our lectures or seminars. For the context of this coursework let us term this an **event**. I have found an interesting use of FP in helping me keep track of *events*. Admitedly FP is more of a logic inference engine suitable for small scale totally correct reasoning than it is a large scale number cruncher. Nonetheless I have had great fun in programming some simple functions, which have now developed into interesting examples upon which to base *CS256 Coursework Part 1 2016-17*. For Part 1 of the CS256 Coursework this year I want you to design your own set of functions in Haskell to facilitate the following tracking of a person's events.

1 A function `days :: Month -> Int` to assign the correct number of days to each *month* in a *year*. Include here handling for a **leap year**. According to Google, a leap year is *a year, occurring once every four years, which has 366 days including 29 February as an intercalary day.* Also, we need to note, a leap year is divisble by 4.

2 A type `Date`, consisting of a *day*, a *month*, and a *year*.

3 A *show function* `showd :: Date -> String` for type `Date`.

4 A function `before :: (Date,Date) -> Bool` , where `before(x,y)` evaluates to `True` if and only if `x` is earlier or equal to `y` in the usual ascending temporal ordering. For example, 03/10/2016 is *before* 04/10/2016, and 05/10/2016 is not *before* 04/10/2016.

5 A type `Event`, consisting of a *date*, a member F_i of a set of *forms* $F_1, F_2, \ldots$, and a *comment*. Below is the raw data of all tonic-clonic seizures I have have had since November 27th 2015. Please use this data to test your own work, as then I will be able to compare your results against my own.

Date	Form	Comment
27/11/2015	F_1	In my DCS office. Security and Paramedics called.
10/12/2015	F_1	After a sleepless night, in my kitchen.
17/12/2015	F_1	On the phone, on the couch.
01/01/2016	F_1	Napping in bed this morning.
05/02/2016	F_1	In my DCS office.
02/04/2016	F_1	Earlsdon, Beechwood Avenue. Paramedics called.
13/04/2016	F_1	In the Gents toilets on the 3rd floor of DCS.
17/04/2016	F_1	At the Friends Meeting House (Coventry).
08/05/2016	F_1	In Providence St on my way to the Meeting House.
13/05/2016	F_1	In the Arts Centre Café.
09/06/2016	F_1	In my DCS office.
23/06/2016	F_1	In my DCS office.
15/07/2016	F_1	In my DCS office.
19/07/2016	F_1	In my DCS office.
09/08/2016	F_1	In Jin's Café, Cannon Park Shops.
24/08/2016	F_1	In the Arts Centre Café.
10/09/2016	F_1	At home in my bedroom.
25/09/2016	F_1	At home in my armchair.
25/10/2016	F_1	At home, having just got up.
31/10/2016	F_1	At home, an hour after getting up

6 A variable `history :: History` of events is defined to be a finite sequence of the following tracking information updated after each event. (1) the date of the event, (2) the (integer) number of days (termed the *gap*) since the last most recent event, and (3) the average gap between events from 27/11/2015 (the date of my first event in the above data since and including tracking began) and the present event (named `av_gaps`).

Assessment Criteria

Your work will be examined for its good use and application of functional programming principles and their expression in Haskell. Your final mark for this coursework (of 100%) will be broken down as follows. 30% for your correct **analysis** of the problem. 20% for your appropriate **algorithm** used to implement the problem. 30% for your choice of **language** constructs (coded in Haskell) used to represent the algorithm. 20% for your demonstrative **testing** of key language constructs in your program.

Submission procedure

Submit your solution to this assignment on-line as a single Haskell source file named `ass16-1.hs` on Tabula. Your submission must be made by **12:00 noon** of **Thursday November 17th 2016 (Term 1, week 7)**. This assignment counts for **15%** of your total credit for the module CS256. In accordance with University guidelines, 5% per day (or part thereof) of the maximum possible mark for this assignment will be subtracted from your mark for a late submission.

Plagiarism Detection

Each student is politely reminded that the CS256 coursework is an individual assignment whose contents is presumed, unless otherwise acknowledged, to be only the work of that person. Each submission received will be automatically compared with each and every other such submission using plagiarism detection software. The University's disciplinary procedures **will** be applied to each student whose submission is suspected, within all reasonable doubt, to be the unacknowledged result of either copying from or collusion with another person or persons within or without of Warwick University.