COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 1 (for workshops in week 2).

QUESTION 1
What are the most annoying limitations of a programming language you
have used? It would be good if you posted something on this topic to
the LMS discussion forum.

ANSWER
This question is intended to provoke general discussion about programming
languages. Often, you don't miss some feature (or recognise a limitation)
until you have programmed in a language which has that feature.
For example, after programming in Haskell, many people see the lack of
e.g. discriminated union types and convenient higher order programming
features in languages like C, Java and Python as annoying limitations. (C
does not support closures in their full generality. It supports function
pointers, which are like closures that contain no hidden arguments. This
limitation makes higher order programming much more tedious.)

Learning different programming languages enriches us as programmers
and also allows us to make better choices of which tools to use to solve
each problem.

QUESTION 2
What are some useful Haskell resources on the web?

ANSWER
The site haskell.org has a good collection, including the classic Gentle
Introduction To Haskell (perhaps not quite gentle enough for the average
computer science undergraduate). The web is a great resource for information
about programming languages, libraries etc. There is never enough time
in lectures (or workshops) to cover every significant feature of any
programming language.

Note that the web is generally not a good resource for solving programming
assignments and other forms of assessment. Apart from the issue that
using other people's code in assignments that are meant to be your own
work is a violation of ethical rules, code from the web rarely solves
the exact problem an assignment asks you to solve, and adapting the web
code to your needs typically requires as much work as writing the code
from scratch. This is because to adapt the code to your needs, you must
first understand it.

QUESTION 3
What will be printed by this C code fragment?

```
#include <stdio.h>
    int f(int x, int y)
    {
           return 10 * x + y;
    }

    int main(void)
    {
           int i, j;

           i = 0;
```

```
            j = f(++i, ++i);
            printf("%d\n", j);
        return 0;
        }
```

What does this show about the impact of side effects in C?

ANSWER
The two preincrements of i in the call to f are side effects. C does not guarantee the order of evaluation of side effects, which means that several possible outputs are possible.

- If the preincrements are executed left to right, the output will be 12.
- If the preincrements are executed right to left, the output will be 21.
- If the additions in preincrements are executed before either preincrement updates i, the output will be 11.
- If both preincrements are executed before i is passed to f, the output will be 22.

If you as the programmer expect the program to generate e.g. 12 as the output, and on your machine, the C compiler happens to execute side effects left to right, you may think that your code is correct, when in fact the program will fail the first time you run it on a platform whose C compiler orders these side effects differently.

This shows that side effects in C can make program testing much harder.

This problem cannot occur in Haskell, since in Haskell, there are no side effects.

QUESTION 4
Fix the formatting errors (due the offside rule) in the

following code,
making the minimal possible changes.

```
zero = len []
one
 = len
     []
two = len [1,2] three = len [1,2,3]
four = len [1,
    2,3,
       4]
 len []    = 0
 len (x:xs) = 1 + len xs
```

ANSWER

```
zero = len []
one
 = len
     []
two = len [1,2]
three = len [1,2,3] -- has to be on new line
four = len [1,
    2,3,
       4]
len []    = 0    -- has to start in column 0
len (x:xs) = 1 + len xs -- has to start in column 0
```

QUESTION 5
Implement a function to perform the logical XOR (exclusive or)
operation. The XOR of two truth values is true if exactly one of them
is true. You may approach this using pattern matching or using other
logical operations.

ANSWER

```
--Using pattern matching

>xor1 False False = False
>xor1 False True  = True
>xor1 True  False = True
>xor1 True  True  = False
```

--Using logical operations

```
>xor2 x y = (x || y) && not (x && y)
```

QUESTION
Implement your own versions of the 'head' and 'tail' functions included in
the Haskell Prelude. Do not use the existing 'head' and 'tail' functions in
your implementation. Note you should call your functions 'myHead' and
'myTail' to avoid shadowing the existing functions.

ANSWER

```
>myHead []    = error "myHead called on an empty list"
>myHead (x:_) = x

>myTail []    = error "myTail called on an empty list"
>myTail (_:xs) = xs
```

Note: The built-in function 'error' is mostly just a convenience
for beginners, and possibly for prototyping. It's rarely used
in production code. Haskell has far better ways of handling error situations, which we'll look at later on in the semester.

QUESTION 6
Implement a function to append two lists in Haskell (this is
the ++ function in the standard prelude). What is the type of this function?

ANSWER

```
>append [] lst = lst
>append (e:es) lst = e:£¨append es lst£©

*Main> :t append
append :: [a] -> [a] -> [a]
```

QUESTION 7
Implement your own version of the 'reverse' function included in the Haskell
Prelude. Do not use the existing 'reverse' function in your implementation.
Note you should call your function 'myReverse' to avoid shadowing the
existing function.

ANSWER

```
>myReverse []    = []
>myReverse (x:xs) = (myReverse xs) ++ [x]
```

Later on, we'll see more efficient, though less obvious, ways of
performing operations like reversing a list.

QUESTION 8
Implement a function 'getNthElem' which takes an integer 'n' and a list, and
returns the nth element of the list.

ANSWER

Here's one possible solution:

```
>getNthElem 1 xs = head xs
>getNthElem n [] =
>   error "n is greater than list length in getNthElem"
>getNthElem n xs
>   | n < 1    = error "n is non-positive in getNthElem"
>   | otherwise = getNthElem (n-1) (tail xs)
```

Note, this assumes one-origin indexing for the list elements. The built-in Haskell operator (!!) assumes zero-origin indexing,
which is more usual, and arguably more natural. It's straightforward to modify the code for zero-origin indexing.

You might like to critique the above solution in terms of how
error situations are handled, and of how much work is done at

each recursive step.

Here are two other solutions for comparison (with shorter, different names, for convenience). The first re-arranges the
tests:

```
>getn n [] = error "n was too big in getn"
>getn n (x:xs)
>   | n > 1    = getn (n-1) xs
>   | n == 1   = x
>   | otherwise = error "n was too small in getn"
```

The second (thanks to Rob Holt) makes use of an auxiliary function in a where clause (and uses zero-origin indexing):

```
>getNth :: Int -> [a] -> a
>getNth n xs
>   | n < 0    = error "Invalid index: negative index value in getNth"
>   | otherwise = go n xs
>   where
>     go _ []    = error "Invalid index: value exceeds list length in getNth"
>     go 0 (x:xs) = x
>     go k (x:xs) = go (k-1) xs
```

Two things to keep in mind:

1. Even though Haskell is a strongly statically typed language,
   in almost all cases, Haskell can automatically infer types.
   So rarely is it necessary to write explicit type declarations
   (as you need to do in say C or Java). Even so, it's
   considered good practice to write type declarations for
   everything defined at top-level in your program (usually
   functions, but possibly non-function data). There are two
   main reasons for this: First, it provides a check that the
   type you intend something to have matches up with the type
   inferred by the Haskell compiler. Second, it provides type

documentation for programmers reading your code.

When you're first getting started, you can bootstrap by writing your code without type annotations, loading it into GHCI, and using the :type directive to query the type, which you can paste into your source file. However, that's just for beginners. Really, when you go to write a function, the first thing you should think about is its type, and you need to work towards developing skill in thinking about and expressing types.

2. Writing a function like 'getNthElem' is a worthwhile exercise. However, despite the superficial similarity, Haskell lists are very different from arrays in other programming languages. Almost always, if you write code that indexes into a list (treating it like an array), then you're taking a wrong approach. In most situations, you operate on lists by considering the natural two cases, empty or non-empty, usually by pattern matching. There are situations in which you might need to index into a list, but these are rare. (Haskell does actually have arrays, but they're an advanced topic in Haskell, which we won't get to in an introductory subject like this.) In Haskell, as in most declarative languages, lists are ubiquitous workhorse data structures.