COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 10.

QUESTION 1

Recall the discussion of the Maybe monad in lectures, and the definitions of maybe_head, maybe_sqrt and maybe_sqrt_of_head. In a similar style, write Haskell code for the function

```
    maybe_tail :: [a] -> Maybe [a]
```

which returns the tail of a list if the list is not empty, and

```
    maybe_drop :: Int -> [a] -> Maybe [a]
```

which is like the prelude function drop ("drop n xs" drops the first n elements of the list xs), but returns a Maybe type. If n is greater than the length of xs, it should return Nothing (drop returns [] in this case), otherwise it should return Just the resulting list.

Code two versions of maybe_drop. Both should use maybe_tail. One should explicitly check for Nothing and the other should use >>=.

ANSWER

Let us make this file a proper module so we can import things for some later questions:

```
>module Main where
```

```
>import Data.Char (isDigit, digitToInt)
```

These are needed for QUESTION 5 below, but have to be right after the module declaration.

```
>import System.IO (hFlush, stdout)
```

```
>maybe_tail :: [a] -> Maybe [a]
>maybe_tail [] = Nothing
>maybe_tail (_:xs) = Just xs

>maybe_drop :: Int -> [a] -> Maybe [a]
>maybe_drop 0 xs = Just xs
>maybe_drop n xs | n > 0 = maybe_tail xs >>= maybe_drop (n-1)

>maybe_drop’ :: Int -> [a] -> Maybe [a]
>maybe_drop’ 0 xs = Just xs
>maybe_drop’ n xs | n > 0 =
>        let mt = maybe_tail xs in
>        case mt of
>                Nothing -> Nothing
>                Just xs1 -> maybe_drop’ (n-1) xs1
```

As you can see, the version of maybe_drop that uses the monad sequencing operation (the version without the apostophe) is significantly shorter, and (if you understand what the monad operation does) also significantly simpler.

QUESTION 2
Given the tree data type defined below, write the Haskell function

```
   print_tree :: Show a => Tree a -> IO ()
```

which does an inorder traversal the tree, printing the contents of each node
on a separate line. What are the advantages and disadvantages of this approach
compared to traversing the tree and returning a string, and then printing the string?

```
>data Tree a = Empty | Node (Tree a) a (Tree a)
```

ANSWER
Here is a version using do notation:

```
>print_tree :: Show a => Tree a -> IO ()
>print_tree Empty = return ()
>print_tree (Node l d r) = do
>   print_tree l
>   print d
>   print_tree r
```

Printing things directly has the advantage of avoiding the creation of some intermediate data structures (potentially lots of concatenating of strings). However, it is less flexible. Although strings are not a great data structure, you can do certainly do more with strings than you can do with IO actions. In most cases, it is best to limit input/output to a small section of top level code. This is because in most cases, the extra CPU time spent concatenating strings is unlikely to have a significant impact on the elapsed time of your program, and the extra flexibility is worth the small cost.

QUESTION 3
Write a Haskell function

```
    str_to_num :: String -> Maybe Int
```

that converts a string containing nothing but digits to Just the number they represent, and any other string to Nothing. Hint: the standard library module Data.Char has a function isDigit that tests whether a character is a decimal digit, and another function digitToInt that converts such characters to a number between between 0 and 9.

ANSWER
We first check for the special case of an empty string, then call a helper function with the value of the digits so far. Note: we allow any number of leading zeros.

```
>str_to_num :: String -> Maybe Int
>str_to_num [] = Nothing
>str_to_num (d:ds) = str_to_num_acc 0 (d:ds)
```

Each time we get another digit we multiply the value so far by 10 and add the new digit. If we get to the end we return Just the value and if we get a non-digit we return Nothing.

```
>str_to_num_acc :: Int -> String -> Maybe Int
```

```
>str_to_num_acc val [] = Just val
>str_to_num_acc val (d:ds) =
>   if isDigit d then str_to_num_acc (10*val + digitToInt d) ds
>   else Nothing
```

QUESTION 4
Write two versions of a Haskell function that reads in a list of lines containing numbers, and returns their sum. The function should read in lines until it finds one that contains something other than a number.

The first version of the function should sum up the numbers as it read them in.
The second should collect the entire list of numbers before it starts summing them up.

ANSWER
We have to do some IO actions and return an Int, so the type will be IO Int.

This version uses do notation:

```
>sum_lines :: IO Int
>sum_lines = do
>   line <- getLine
>   case str_to_num line of
>       Nothing -> return 0
>       Just num -> do
>           sum <- sum_lines
>           return (num+sum)
```

Here is an equivalent version which uses >>= instead of do notation:

```
>sum_lines_no_do :: IO Int
>sum_lines_no_do =
>   getLine >>=
>   \line -> case str_to_num line of
>       Nothing -> return 0
>       Just num ->
>           sum_lines_no_do >>=
>           \sum -> return (num+sum)
```

For the second version asked for in the question, the type at the top level is the same, but we have a helper function which does IO and returns a list of Ints, on which we then invoke the sum function from the prelude:

```
>sum_lines’ :: IO Int
>sum_lines’ = do
>   nums <- list_num_lines
>   return (sum nums)
```

The helper function has the same structure as sum_lines but uses [] and : instead of 0 and +.

```
>list_num_lines :: IO [Int]
>list_num_lines = do
>   line <- getLine
>   case str_to_num line of
>       Nothing -> return []
>       Just num -> do
>           nums <- list_num_lines
>           return (num:nums)
```

QUESTION 5

Write a Haskell main function that repeatedly reads in and executes commands to implement a trivial phonebook program. The commands it should support are:

print	prints the entire phone book
add name num	adds num as the phone number for name
delete name	delete the entry for name
lookup name	print the entries that match name
quit	exit the program

To keep things simple, only check the first letter of commands (so people can abbreviate commands to a single letter). You may assume that a name is a single word, and that it must match exactly. You can use the Haskell prelude function words to split a single string into a list of words. If you print a prompt and expect to read the command on the same line, you need to do hFlush stdout to ensure the prompt is written before reading the user command. To use this, you will need to import System.IO.

ANSWER

Here's a simple solution. For a more sophisticated approach, see workshop10_alt.hs.

A Phonebook is a list of entries, each a pair of a name and a phone number

```
>type Phonebook = [(String,String)]
```

```
>main :: IO ()
>main = phonebook []
```

```
>phonebook :: Phonebook -> IO ()
>phonebook pbook = do
>    putStr ″phonebook> ″
>    hFlush stdout
>    command <- getLine
>    case words command of
>        [] -> phonebook pbook     -- empty command; just prompt again
>        ((commandLetter:_):args) -> executeCommand pbook commandLetter
args
```

```
>executeCommand :: Phonebook -> Char -> [String] -> IO ()
>executeCommand pbook ’p’ [] =
>    printPhonebook pbook >> phonebook pbook
>executeCommand pbook ’a’ [name,num] =
>    phonebook $ pbook ++ [(name,num)] -- add to the end
>executeCommand pbook ’d’ [name] =
>    phonebook $ filter ((/= name) . fst) pbook
>executeCommand pbook ’l’ [name] =
>    printPhonebook (filter ((== name) . fst) pbook) >> phonebook pbook
>executeCommand _ ’q’ [] = return ()
>executeCommand pbook ’h’ [] = usage >> phonebook pbook
>executeCommand pbook ’?’ [] = usage >> phonebook pbook
>executeCommand pbook cmd _ = do
>    putStrLn (″Unknown command letter ’″ ++ [cmd] ++ ″’″)
>    usage
>    phonebook pbook
```

```
>printPhonebook :: Phonebook -> IO ()
>printPhonebook = mapM_ (\(name,num) -> putStrLn $ name ++ ″ ″ ++ num)
```

Not asked for in the spec, but this prints out a usage message:

```
>usage :: IO ()
>usage = putStrLn $ unlines $ [
>        "Usage:",
>        "    print           prints the entire phone book",
>        "    add name num    adds num as the phone number for name",
>        "    delete name     delete the entry for name",
>        "    lookup name     print the entries that match name",
>        "    quit            exit the program",
>        "    help            display this usage message"]
```