CS 115: Functional Programming, Spring 2016

Assignment 5: Monads and Lists, Oh My!

Due: *Monday, May 16, 02:00:00*

Coverage

This assignment covers the material up to lecture 18.

Submitting your assignment

Write your code in a single text file called `lab5.hs`. Submit your assignment to csman as usual.

Writing and testing your code

As usual, all of your Haskell functions, even internal ones, should have explicit type signatures.

All of your code should be compiled with the `-W` (warnings enabled) command-line option (even when using `ghci`). Your code should be commented appropriately and neatly and reasonably formatted.

Part A: The list monad

In this section you will do some exercises involving the list monad.

1. **[10]** A classic problem in number theory is to find positive integers that can be expressed as the sum of two cubes in two different ways. This problem dates back to the mathematicians Hardy and Ramanujan; supposedly Ramanujan could tell at a glance that 1729 was the smallest such integer.

   Ben Bitfiddle has written a very elegant list comprehension to generate all these numbers:

   ```
   hr_solutions :: [((Integer, Integer), (Integer, Integer), Integer)]
   hr_solutions =
     [((i, l), (j, k), i^3 + l^3) |
      i <- [1..],
      j <- [1..i-1],
      k <- [1..j-1],
      l <- [1..k-1],
      i^3 + l^3 == j^3 + k^3]
   ```

   Here's an example of a run in `ghci`:

   ```
   Prelude> take 5 hr_solutions
   [((12,1),(10,9),1729),((16,2),(15,9),4104),((24,2),(20,18),13832),((27,10),(24,19),20683),((32,4),(30,18),32832)]
   ```

   Rewrite Ben's code using the list monad instead of a list comprehension. Use the `do`-notation and the `guard` function (in the module `Control.Monad`) where appropriate.

2. **[15]** Redo problem A.5 of assignment 1 using the list monad instead of a list comprehension. Write two different solutions: one using the `guard` function and one which doesn't use `guard` but which does use `mzero` from the `MonadPlus` instance for lists to filter out undesired values. Don't use the `filter` function.

   The problem is: Write an expression which computes the sum of the natural numbers below one thousand which are multiples of 3 or 5. (Your expression doesn't have to fit on one line this time.)

3. **[20]** Solve Euler problem 4 using the list monad.

   The problem description is as follows. A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 * 99. Find the largest palindrome made from the product of two 3-digit numbers.

   Write a one-line function called `isPalindrome` which takes an `Integer` and returns `True` if the integer's decimal representation is a palindrome. (*Hint:* Convert the integer to a string; note that `Integer` is an instance of the `Show` type class.) Then write a function called

`largestPalindrome` which computes the desired quantity. Do not use a list comprehension or recursion, but do use the list monad. The `maximum` function from the Haskell Prelude will probably be useful to you.

Write the solution to the problem in a comment.

4. **[60]** The list monad is a great tool for solving combinatorial problems. Consider this puzzle:

> Take the digits 1 to 9 in sequence. Put either a "+", a "-", or nothing between each digit to get an arithmetic expression. Print out all such expressions that evaluate to 100.

In this problem we will use the list monad to solve this problem.

We will start by defining some datatypes to represent expressions.

```
type Expr = [Item]

data Item = N Int | O Op
  deriving Show

data Op = Add | Sub | Cat
  deriving Show
```

In words: an expression is a list of items, an item is either a number or an operator, and an operator is either addition, subtraction, or concatenation. Note that these datatypes don't enforce any reasonable invariants; you can have an "expression" which is just operators, for instance. We could design more clever datatypes to enforce such invariants, but we won't for simplicity. However, that means we'll have to detect invalid cases in our functions.

We'll also define a list of all operators, for convenience:

```
ops :: [Item]
ops = [O Add, O Sub, O Cat]
```

Here are the functions/values we want you to define.

1. Define a value called `exprs` which consists of a list of all possible valid expressions from the puzzle description *i.e.* all possible combinations of the digits 1 to 9 (in order) with one of the operators from the `Op` datatype between each digit. Use the list monad. Your function shouldn't be more than about a dozen lines long. There should be exactly `3^8` or `6561` possible expressions in the list. (Please don't write a brute-force expression that is 6561 lines long! That would take way longer than solving the problem would.) *Hint:* the numbers can only be one thing at any position in the expression, but between each pair of adjacent numbers you can have any one of three operators.

2. Define a function called `normalize` that takes an expression and removes all instances of the `Cat` operator by applying this transformation: `N i, Cat, N j --> N (ij)` anywhere in the list. So, for instance, `N 1, Cat, N 2` would become `N 12` (the digits get concatenated). (Note that digits are represented as `Int`s, not `String`s, so you can't use string concatenation.) This should work for longer stretches of `Cat`s; for instance: `N 1, Cat, N 2, Cat, N 3` becomes `N 123`. Anything that doesn't match the patterns described is just passed through unchanged, except that illegitimate patterns (*e.g.* multiple operators/numbers in a row or expressions that begin or end with operators) give rise to errors. This function has the type `Expr -> Expr` and our implementation is five lines long. *Hint:* Pattern match all valid subexpressions, and anything else becomes an error.

3. Define a function called `evaluate` that will take a normalized expression (*i.e.* one with no `Cat` operators) and evaluate it to give an `Int`. Again, this function should only be a few lines long. One thing to be careful about is that subtraction associates to the left, so `a - b - c` is `(a - b) - c`, not `a - (b - c)`. Therefore, start evaluating from the beginning of the expression towards the end, not the other way around.

We are also giving you the functions `find`, `pprint` and `run` which are defined as follows:

```
-- Pick out the expressions that evaluate to a particular number.
find :: Int -> [Expr] -> [Expr]
find n = filter (\e -> evaluate (normalize e) == n)

-- Pretty-print an expression.
pprint :: Expr -> String
pprint [N i] = show i
pprint (N i : O Add : es) = show i ++ " + " ++ pprint es
pprint (N i : O Sub : es) = show i ++ " - " ++ pprint es
pprint (N i : O Cat : es) = show i ++ pprint es
pprint _ = error "pprint: invalid argument"

-- Run the computation and print out the answers.
run :: IO ()
run = mapM_ putStrLn $ map pprint $ find 100 exprs
```

You should include these in your submitted code for testing purposes.

You can run your code for this problem by loading it into `ghci` and typing `run` at the prompt. Your output should look something like this:

```
1 + 2 + 3 - 4 + 5 + 6 + 78 + 9
1 + 2 + 34 - 5 + 67 - 8 + 9
1 + 23 - 4 + 5 + 6 + 78 - 9
1 + 23 - 4 + 56 + 7 + 8 + 9
12 + 3 + 4 + 5 - 6 - 7 + 89
12 + 3 - 4 + 5 + 67 + 8 + 9
12 - 3 - 4 + 5 - 6 + 7 + 89
123 + 4 - 5 + 67 - 89
123 + 45 - 67 + 8 - 9
123 - 4 - 5 - 6 - 7 + 8 - 9
123 - 45 - 67 + 89
```

You can check that all of these expressions evaluate to 100.

Part B: Puzzles and derivations

Solve the first three of the following puzzles by desugaring the monadic code from the `do` notation to straight uses of the `return` method and the `>>=` and `>>` operators. Then substitute the definitions of `return`, `>>=` and `>>` and derive the result.

NOTE: For the first two problems, you do not need to desugar a `do` expression using the complete desugaring to a `case` expression to handle possible pattern-match failures. Simply rewrite it in terms of the "simple" desugaring involving the `return` and `>>=` operators. For the third problem you will need to desugar the `do` expression using the full `case` expression desugaring (see lecture 11 for more on this).

One way to shorten your derivations a little is to use `concatMap` instead of `concat (map ...)`. For these exercises, you may consider the definition of `concatMap` to be:

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f lst = concat (map f lst)
```

Note that `concatMap <anything> []` is the empty list. You can then write the definition of the `>>=` operator for the list monad as:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
mv >>= f = concatMap f mv
```

Write your solutions as Haskell comments.

1. **[20]** Why does this expression:

   ```
   do n1 <- [1..6]
      n2 <- [1..6]
      []
      return (n1, n2)
   ```

   evaluate to `[]`? Show all the steps in your derivation.

2. **[20]** Why does this expression:

   ```
   do n1 <- [1..6]
      n2 <- [1..6]
      return <anything>
      return (n1, n2)
   ```

   return the same thing as this expression:

   ```
   do n1 <- [1..6]
      n2 <- [1..6]
      return (n1, n2)
   ```

   ? Answer this by reducing both expressions to the same expression. Show all the steps in your derivations.

3. **[30]** You can use the list monad to perform simple pattern-matching tasks. Consider this code:

   ```
   let s = ["aaxybb", "aazwbb", "foobar", "aaccbb", "baz"] in
     do ['a', 'a', c1, c2, 'b', 'b'] <- s
        return [c1, c2]
   ```

   This returns this result:

   ```
   ["xy", "zw", "cc"]
   ```

Explain why by deriving the result, using the full `case`-style desugaring of `do` expressions we covered in lecture 11. Note that the `fail` method of the `Monad` type class has this definition in the list monad:

```
fail _ = []
```

Explain what would happen if instead `fail` for the list monad used the default definition given in the `Monad` type class, which is:

```
fail s = error s
```

Don't forget that the `String` datatype in Haskell is just a list of `Char`s *i.e.* `[Char]`.

4. **[30]** Lecture 15 we mentioned that in GHC the real definition of `>>=` for lists is:

```
m >>= k = foldr ((++) . k) [] m
```

and we claimed that this is the same as the definition we used:

```
m >>= k = concat (map k m)
```

(except that the first version may run faster, which you don't need to concern yourself with here). Show that the two expressions `foldr ((++) . k) [] m` and `concat (map k m)` do indeed compute the same thing. *Hint:* Show that given `m = [x1, x2, ...]` both expressions evaluate to the same thing. Also show this for `m = []`. Write your answer in a comment, as usual. *Hint:* Expand `(++) . k` into an explicit lambda expression.

5. **[20]** We've seen that existential types can be used to create objects which can contain values of any type which implements a particular type class. Ben Bitfiddle learns about this and gets very excited. "I can create a list of items which are all instances of class `Num` but otherwise may be completely different types", he says, "and then add them all up using the `(+)` operator defined for each type. By doing this I can create a generic addition method." He starts to write out his code in a module called `Sum.hs`:

```
{-# LANGUAGE ExistentialQuantification #-}

module Sum where

data AnyNum = forall a . Num a => AnyNum a

anySum :: [AnyNum] -> AnyNum
anySum [] = AnyNum 0
anySum ((AnyNum n) : ns) =
  case anySum ns of
    AnyNum s -> AnyNum (n + s)
```

However, GHC is not happy with this file, and pinpoints the error with this very illuminating error message:

```
Sum.hs:11:29:
    Could not deduce (a1 ~ a)
    from the context (Num a)
      bound by a pattern with constructor
                 AnyNum :: forall a. Num a => a -> AnyNum,
               in an equation for `anySum'
      at Sum.hs:9:10-17
    or from (Num a1)
      bound by a pattern with constructor
                 AnyNum :: forall a. Num a => a -> AnyNum,
               in a case alternative
      at Sum.hs:11:5-12
      `a1' is a rigid type variable bound by
           a pattern with constructor
             AnyNum :: forall a. Num a => a -> AnyNum,
           in a case alternative
           at Sum.hs:11:5
      `a' is a rigid type variable bound by
          a pattern with constructor
            AnyNum :: forall a. Num a => a -> AnyNum,
          in an equation for `anySum'
          at Sum.hs:9:10
    In the second argument of `(+)', namely `s'
    In the first argument of `AnyNum', namely `(n + s)'
    In the expression: AnyNum (n + s)
```

The `~` symbol in `(a1 ~ a)` means "type equality".

What is the problem here? Is there any way to fix it without changing the `AnyNum` datatype?

Note: We don't want you to explain every line in the error message; just tell us what the problem is with the code *i.e.* why the code can never work.

Copyright (c) 2016, California Institute of Technology. All rights reserved.