CS 115: Functional Programming, Spring 2016

Assignment 1: Basics

Due: *Monday, April 18, 02:00:00*

Coverage

This assignment covers the material up to lecture 5.

Grading policy

You will receive an integral score from 0 to 3 for each section. The meaning of each number is:

- **0**: Completely wrong; unacceptable
- **1**: Some parts may be correct, but some have serious problems
- **2**: All important parts correct; may be some minor flaws
- **3**: No significant flaws

The grade of the assignment as a whole is the *minimum* of the section scores. After your initial lab submission has been graded, you have one week to submit a redo of the lab, after which it will be re-graded in the same way.

Time hints

Each problem comes with a time hint as a number in bold text at the beginning of the description. This is our estimate of the *maximum* time that the problem should take you to solve (in minutes). If you find that a problem takes significantly longer, you should let us know so we can better calibrate the assignments. You should also talk to the lecturer to find out if the problem is that you just don't understand a concept, or if the problem really is unreasonably hard or long.

Background reading

Read the first four chapters of the Gentle Introduction to Haskell. *Warning*: It isn't that gentle!

Writing and testing your code

All of your Haskell functions, even internal ones, should have explicit type signatures. This is not a Haskell requirement, but we consider it to be good style as well as good documentation, and it is likely to save you from some problems that might occur if you let `ghc` infer the types. Also, error messages will be easier to understand.

All of your code should be tested using `ghci` with the `-W` (warnings enabled) command-line option. Any warnings will be considered errors, so make sure your code does not give any warnings. You can invoke `ghci` explicitly with the `-W` option as follows:

```
% ghci -W
```

However, since you always want the `-W` option to be enabled, a better solution is to define this once and for all in a `.ghci` file. The `.ghci` file is just a regular file that contains stuff to be automatically loaded into `ghci` when it's launched. All you need to put in this file is this line:

```
:set -W
```

and you are all set. We recommend that this file be placed in your home directory. To test whether it worked, run `ghci` and type the following line:

```
let f (x:xs) = x
```

If this line gives you a warning about non-exhaustive pattern matches, it's working.

Evaluations and answers to questions should be written as Haskell comments. You'll find the multi-line comment syntax `{- ... -}` useful in this regard.

Submitting your assignment

Write all of your code in a single text file called `lab1.hs`. Evaluations should be written inside Haskell comments (the `{-` and `-}` multi-line comment delimiters work well for this). Submit your assignment to csman.

Part A: Basic exercises

1. **[20]** Haskell lets you define your own operators. Here are a couple of simple examples.

    1. Write a definition for an operator called `+*` that computes the sum of the squares of its arguments. Assume both arguments are `Double`s. Make it left-associative and give it a precedence of 7.

    2. Write a definition for an operator called `^||` that computes the exclusive-OR of its two (boolean) arguments. Make it right-associative with a precedence of 3. Your definition should be only two (very simple) lines (not counting the type declaration) and shouldn't use `if`. *Hint:* Use pattern matching; if the first argument is `False`, what must the answer be?

2. **[10]** Write a recursive function called `rangeProduct` that takes two `Integers` and computes the product of all the integers in the range from one integer to the other (inclusive). For instance:

    ```
    rangeProduct 10 20
    ```

```
--> 10 * 11 * 12 * 13 * 14 * 15 * 16 * 17 * 18 * 19 * 20
--> 6704425728000
```

If the first argument is greater than the second argument, signal an error using the `error` function. Use pattern guards instead of explicit `if` expressions to test the cases.

3. **[10]** Use `foldr` to define a one-line point-free function called `prod` that returns the product of all the numbers in a list of `Integer`s (or `1` if the list is empty). Use this function to define a one-line (not including the type signature) non-recursive definition for `rangeProduct`. (This will be a two-line definition if you also include the line containing `error`.)

4. You learned about the `map` higher-order function in class. Here you will write some variations on `map`.

   1. **[10]** First, write a function `map2` which maps a two-argument function over two lists. Write it as a recursive procedure (there is a trivial definition using `zipWith` which you shouldn't use). It could be used like this:

      ```
      Prelude> map2 (*) [1, 2, 3] [4, 5, 6]
      [4, 10, 18]
      ```

      Your solution should have a completely polymorphic type signature (*i.e.* the most general type signature). Extra elements in either list should be ignored.

   2. **[10]** Now, write a `map3` function that will work for functions of 3 arguments. For instance:

      ```
      Prelude> map3 (\x y z -> x + y + z) [1, 2, 3] [4, 5, 6] [7, 8, 9]
      [12,15,18]
      ```

   3. **[20]** It's easy to use `map2` along with the `sum` function to define a function to compute the dot product of two lists of integers, as follows:

      ```
      dot :: [Integer] -> [Integer] -> Integer
      dot lst1 lst2 = sum (map2 (*) lst1 lst2)
      ```

      or we can save some parentheses by writing it as:

      ```
      dot :: [Integer] -> [Integer] -> Integer
      dot lst1 lst2 = sum $ map2 (*) lst1 lst2
      ```

      You might think that you can write this as a point-free version using the `(.)` function composition operator as follows:

      ```
      dot :: [Integer] -> [Integer] -> Integer
      dot = sum . map2 (*)
      ```

      However, this does not type check! The correct point-free version is:

      ```
      dot :: [Integer] -> [Integer] -> Integer
      dot = (sum .) . map2 (*)
      ```

      Write out a short (less than 10 lines) evaluation showing that `dot lst1 lst2` using the point-free definition is equivalent to the explicit (point-wise) definition given above. Note that `(sum .)` is an operator section equivalent to `\x -> sum . x`.

Hint: Don't forget about currying!

5. **[10]** Using a list comprehension, write a one-line expression which computes the sum of the natural numbers below one thousand which are multiples of 3 or 5. Write the result in a comment. You may use the Haskell `sum` function to do the actual sum.

6. **[30]** Calculate the sum of all the prime numbers below 10000.

   Do this as follows:

   - First, generate an infinite list of prime numbers using the "Sieve of Eratosthenes" algorithm. This consists of generating all positive integers and removing all multiples of successive prime numbers. This can be done with a function called `sieve` which takes a list of integers, retains the first one, removes all multiples of the first one from the rest, and then sieves the rest. Name the infinite list of primes `primes`. Note that `sieve` is a two-line definition (not including type declaration) and `primes` is trivial given `sieve`. Make sure your code doesn't generate any compiler warnings.
   - Then, use the `takeWhile` function on the primes list to take the appropriate prime numbers and compute their sum.
   - Write the answer as a Haskell comment.

Part B: Pitfalls

In this section we'll see some examples of bad code and ask you to fix them. Write both the corrected code and (in a Haskell comment) the reason why the original code is bad.

1. **[5]** The following recursive definition has poor Haskell style. How would you improve the style by making a simple change (*i.e.* keep it a recursive definition, but with better style)?

```
sumList :: [Integer] -> Integer
sumList [] = 0
sumList lst = head lst + sumList (tail lst)
```

2. **[10]** What is wrong with this recursive definition? How would you fix it, while keeping it a recursive definition?

```
-- Return the largest value in a list of integers.
largest :: [Integer] -> Integer
largest xs | length xs == 0 = error "empty list"
largest xs | length xs == 1 = head xs
largest xs = max (head xs) (largest (tail xs))
```

Part C: Evaluation

In this section, you'll have to write out the results of evaluating some Haskell expressions step-by-step. These evaluations should look like this:

```
double (3 + 4)
--> (3 + 4) + (3 + 4)
--> 7 + (3 + 4)
--> 7 + 7
--> 14
```

with one reduction step per line. Even more explicit would be the following:

```
double (3 + 4)
--> (3 + 4) + (3 + 4)  [expand from definition]
[outermost redex is + operator]
[+ is strict, needs both operands]
--> 7 + (3 + 4)        [evaluate leftmost branch of + operator]
--> 7 + 7              [evaluate rightmost branch of + operator]
[outermost redex is + operator]
--> 14                 [evaluate + application]
```

though we won't require you to do this (but you may find it helpful if you do).

Pay particular attention to the fact that Haskell is a lazy language. You may assume that the result of the expression being evaluated is going to be printed in its entirety after evaluation. You may also assume that primitive arithmetic expressions evaluate strictly. If the function evaluation does not terminate, compute enough of the evaluation to make it clear that non-termination is the outcome.

Note: Since Haskell uses lazy evaluation, it will evaluate copies of one subexpression at the same time, so the above example should really be:

```
double (3 + 4)
--> (3 + 4) + (3 + 4)
--> 7 + 7    -- evaluate both (3 + 4)'s together
--> 14
```

However, for the evaluations here, assume that all subexpressions are distinct *i.e.* each one should be evaluated separately.

Evaluating Haskell expressions by hand should use the following guidelines:

a. Find the outermost redex (reducible expression) first. If there is more than one choice (which is very common), choose the leftmost of the outermost redexes.

b. If the redex is inside a lambda expression, leave it alone. Otherwise, evaluate it.

c. Continue until you get the result you need. Don't evaluate beyond that point.

d. When expanding a function application using its definition, keep the result in parentheses until it is fully evaluated so as to make clear what the subexpression is.

e. Note that an expression in parentheses can still be a redex. "Outermost" does not refer to parentheses but to the presence of other language constructs. For instance, an expression inside a data constructor or inside a lambda expression is not at the outermost level.

1. **[10]** Consider this function to compute fibonacci numbers:

   ```
   fib :: Integer -> Integer
   fib 0 = 0
   fib 1 = 1
   fib n = fib (n - 1) + fib (n - 2)
   ```

   Evaluate `fib 3`.

2. **[15]** Consider this definition of the factorial function:

   ```
   fact :: Integer -> Integer
   fact n = n * fact (n - 1)
   fact 0 = 1
   ```

   Evaluate `fact 3`. What is wrong with this definition, why is it wrong, and how would you fix it? Pay special attention to the consequences of lazy evaluation here. Assume that the `*` operator applied to `Integer`s evaluates its arguments strictly from left to right.

3. **[15]** We talked about the `reverse` function in lecture 3. One definition was as follows:

   ```
   reverse :: [a] -> [a]
   reverse xs = iter xs []
     where
       iter :: [a] -> [a] -> [a]
       iter [] ys = ys
       iter (x:xs) ys = iter xs (x:ys)
   ```

   Evaluate the expression `reverse [1,2,3]` with this definition. What is the asymptotic time complexity of this function as a function of the length of the input list? Write your answer in a comment. Don't just give the answer, explain why it's correct.

4. **[30]** Another definition of `reverse` was as follows:

   ```
   reverse :: [a] -> [a]
   reverse [] = []
   reverse (x:xs) = reverse xs ++ [x]
   ```

   For reference: The definition of the `(++)` operator is:

   ```
   (++) :: [a] -> [a] -> [a]
   (++) []     ys = ys
   (++) (x:xs) ys = x : (xs ++ ys)
   ```

   and the `++` operator is also right-associative. Ben Bitdiddle claims that this definition of `reverse` has an asymptotic time complexity which is linear in the length of the input list, giving this argument: "Evaluating `reverse [1, 2, 3]` eventually results in `[] ++ [3] ++ [2] ++ [1]` after a linear number of steps, and since appending a singleton list to another list is an `O(1)` operation, constructing the result list from this point is also linear." What is wrong with this argument? Write out a full evaluation of `reverse [1, 2, 3]` and explain where Ben made his mistake and what the mistake was. What is the actual asymptotic time complexity of this version of `reverse`?

5. **[20]** An "insertion sort" is a particular way to sort lists. The first item in the list is inserted at the right place in the result of insertion sorting the rest of the list. For lists of integers, the code might look like this:

```
isort :: [Integer] -> [Integer]
isort [] = []
isort (n:ns) = insert n (isort ns)
  where
    insert :: Integer -> [Integer] -> [Integer]
    insert n [] = [n]
    insert n m@(m1:_) | n < m1 = n : m
    insert n (m1:ms) = m1 : insert n ms
```

For reference, `head` is defined as:

```
head :: [a] -> a
head [] = error "empty list"
head (x:_) = x
```

Evaluate `head (isort [3, 1, 2, 5, 4])`.

6. **[30]** We discussed the `foldr` (fold right) and `foldl` (fold left) higher-order functions in class. Assume that their definitions are as follows:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ init [] = init
foldr f init (x:xs) = f x (foldr f init xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ init [] = init
foldl f init (x:xs) = foldl f (f init x) xs
```

[In fact, the actual definitions of these functions in `ghc` are slightly different.]

Evaluate the following expressions:

1. `foldr max 0 [1, 5, 3, -2, 4]`
2. `foldl max 0 [1, 5, 3, -2, 4]`

where `max` gives the maximum of two values (which you can assume to be `Integers` for this problem). What can you say about the space complexity of `foldr` compared to `foldl`? *Hint:* Don't forget about lazy evaluation!