Assignment 2

Getting Started with Haskell

All Parts Due: 11:59 PM, Monday, February 8, 2016

Ideally, get it done by Sunday night.

Preliminaries

You may work on this assignment with a partner (preferred) or by yourself. If you work with a partner, you must follow pair programming rules (see the PairProgramming page on the wiki for full rules). Both members of the pair must fully understand all jointly submitted work. You are not required to work with your partner from lab.

Files for this assignment are at https://svn.cs.hmc.edu/cs131/spring16/given/hw2. Copy them in the usual way (see CopyingAssignmentFiles on the wiki for a reminder of how).

Required Code Quality

Your final submission must not have syntax or type errors in order to get credit for the assignment. If you know code does not work, explain how or why in a comment. Code that does not even compile must be commented out and replaced with a function that defines the missing function as undefined. Functions must have the exact names and types given in the assignment (although you are free to, and encouraged to, define as many helper functions as you need). The code should be clear and properly commented, just as we would expect for CS 70 or your other courses.

The largest part of the assignment grade is based on writing correct, clear, and well-documented code; see Rubric.txt. Strive for elegance and clarity, rather than “efficiency” (especially since your intuitions about “efficiency” in Haskell are likely to be poor). Excessively convoluted or inefficient (unless specifically allowed in the question) code will not receive full credit, however.

Questions

For all questions, you may define helper functions if it makes your code more readable.

1. Edit lab2.hs so that it contains a completed lab. You may copy over work from lab, but if you are working in a pair different from lab, you may only copy work where both members of the pair created essentially the same code in lab—ideas that only one member of a pair had must be recreated together, as a pair. If you have not done so already, finish the lab.

   • For the rest of the assignment, you will be working on hw2.hs.
2. Define the function

\[ \text{fib} :: \text{Integer} \to \text{Integer} \]

that computes the Fibonacci number \( F_n \) when given \( n \geq 0 \):

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2} & \text{when } n \geq 2.
\end{align*}
\]

Go for a direct implementation of the above specification, even though it is not efficient (i.e., an exponential running-time is okay).

- Copy over your definition of `evenodds` from `lab2.hs`.

3. Write a function `riffle` that is the inverse of `evenodds`, with type

\[ \text{riffle} :: ([a], [a]) \to [a] \]

In other words, it should interleave the elements of the two given lists into a single list.

For example,

\[ \text{riffle (["Pig","Sheep","Goat","Dog"], ["Cow","Duck","Cat"])} \]

should produce `["Pig","Cow","Sheep","Duck","Goat","Cat","Dog"]`.

Make sure to document what happens if the lists are of different lengths.

[We would expect that `riffle (evenodds s) = s` for every list `s`. Is this true even when `s` is an infinite list (like `zeros` or `ones` or `nats`)?]

4. There are two ways to define a two-argument function in Haskell. The arguments can either be supplied simultaneously (as a pair) or they can be supplied separately through successive applications.

For example, `riffle` above is supposed to take the two lists together in a pair, i.e.,

\[ \text{riffle} :: ([a], [a]) \to [a] \]

but we could have asked for a slightly different interface where the two lists are supplied in succession, i.e.,

\[ \text{riffle'} :: [a] \to [a] \to [a] \]

In different circumstances, one or the other interface might be preferable. Fortunately, a function defined in either fashion may be converted to the other.

The Haskell standard prelude defines the functions
curry :: ( (a, b) -> c ) -> a -> b -> c
uncurry :: ( a -> b -> c ) -> (a, b) -> c

that do the conversion. For example, if we define

g = curry f

then g x y and f(x, y) yields the same answer for all x and y. (In particular, riffle’ = curry riffle.) Similarly, if we say

h = uncurry k

then h(x, y) and k x y yields the same answer for all x and y. (In particular, riffle = uncurry riffle’.)

Without peeking at the definitions in the standard prelude and without using the preexisting ones, write your own versions, called curry’ and uncurry’.

Hint: Once you understand how partial function application works, these functions are actually very trivial to write.

5. Use a Haskell data declaration to declare a new type Day which can represent a day of the week; specifically, its range of possible values is Mon, Tue, Wed, Thu, Fri, Sat, or Sun. Then write a function isWeekday :: Day -> Bool which uses pattern matching to determine whether a given day is a weekday (i.e., isWeekday Tue will return True and isWeekday Sun will return False).

6. Use a Haskell data declaration to declare a new type PaperSize to represent paper sizes to use in a print shop. Specifically, there should be four possible PaperSize values: LetterSize, LegalSize, TabloidSize, or CustomSize w h where w and h are numbers (Doubles) representing inches for the width and height of the paper, respectively.

(a) Write a function toPaperSize :: Double -> Double -> PaperSize that uses pattern matching to detect letter, legal or tabloid sized paper and returns the proper PaperSize value (e.g., toPaperSize 8.5 11.0 will return LetterSize but toPaperSize 3.0 5.0 will return CustomSize 3.0 5.0).

(b) Write a function fromPaperSize :: PaperSize -> (Double, Double) that is the inverse of toPaperSize. (i.e., fromPaperSize TabloidSize returns (11.0, 17.0).

Note: Letter sized paper is 8.5 × 11, legal sized paper is 8.5 × 14, and tabloid sized paper is 11 × 17.
7. For this problem, don’t worry about inefficiency caused by repeated appends (++)

Consider the following datatype for representing arithmetic expressions, similar to the one seen in class on Wednesday:

```haskell
    deriving (Show, Eq)

data Exp = Num Double
         | BinOp Exp Op Exp
deriving (Show)
```

For example, the mathematical expression \((2 + 3) \times 7\) is represented with the Haskell code (shown diagrammatically in Figure 1):

```
BinOp (BinOp (Num 2) PlusOp (Num 3)) TimesOp (Num 7)
```

In contrast, certain HP calculators and programming languages evaluate expressions using a stack. An arithmetic computation is a sequence of stack instructions, represented using:

```haskell
data StackInstr = Push Double
    | DoOp Op
    | Swap
deriving (Show)
```

The instruction `Push r` means push the number `r` onto the stack; the instructions `DoOp PlusOp`, `DoOp MinusOp`, `DoOp TimesOp`, and `DoOp DivOp` mean “replace the top two numbers on the stack with their sum”, “... their difference”, and so forth. The `Swap` instruction swaps the top two numbers on the stack. In summary,
If the stack looks like and the operation is afterwards the stack should be

<table>
<thead>
<tr>
<th>If the stack looks like</th>
<th>Operation is</th>
<th>... afterwards the stack should be</th>
</tr>
</thead>
<tbody>
<tr>
<td>... (a\ b\ ...) (a\ b\ ...) (a\ b\ ...) (a\ b\ ...) (a\ b\ ...)</td>
<td>Push (r) DoOp PlusOp DoOp MinusOp DoOp TimesOp DoOp DivOp Swap</td>
<td>(r) ((b + a)) ((b - a)) ((b \times a)) ((b / a)) (b) (a)</td>
</tr>
</tbody>
</table>

Note: The top of the stack is shown on the left

(a) We will represent the stack as a list of numbers,

```haskell
type StackValue = Double

type Stack = [StackValue]
```

where the head of the list corresponds to the top of the stack.

Write a recursive function

```
 evalRPN :: [StackInstr] -> Stack -> StackValue
```

that returns the number at the top of the stack after performing the given operations in order, starting with the given stack. Be careful to get the order right for MinusOp and DivOp. For example,

```
 evalRPN [Push 2.0, Push 1.0, DoOp MinusOp] []
```

should return 1.0, not -1.0.

For this assignment, you don’t need to worry about stack underflow (stack operations executed without enough values on the stack) or other run-time errors.

(b) Write a function

```
 toRPN :: Exp -> [StackInstr]
```
that converts an arithmetic expression to a list of stack operation instructions to compute the same expression. There should be an DoOp PlusOp stack operation for every PlusOp in the input, and so on; evaluating the input expression to a number \( r \) and then returning \([\text{Push } r]\) is not acceptable.

Hint: this function corresponds exactly to a postfix traversal of the input expression viewed as a tree.

(c) The same arithmetic expression can be computed in several ways using the stack machine. For each subexpression, you can choose to evaluate the left side first or the right side first. Because subtraction and division are not commutative, evaluating the right side before the left requires a Swap to fix things up.¹

For example, \(1.0 - (2.0 + 3.0)\) can be computed either by the sequence

\[
[\text{Push } 1.0, \text{Push } 2.0, \text{Push } 3.0, \text{DoOp PlusOp}, \text{DoOp MinusOp}]
\]

or by

\[
[\text{Push } 2.0, \text{Push } 3.0, \text{DoOp PlusOp}, \text{Push } 1.0, \text{Swap}, \text{DoOp MinusOp}]
\]

The first list of instructions requires a stack that can hold at least three numbers simultaneously, whereas the second list never requires more than two numbers on the stack at any one time.

Define (any) two arithmetic expressions named

\[
\begin{align*}
\text{depth3} & : \text{ Exp} \\
\text{depth4} & : \text{ Exp}
\end{align*}
\]

that, even when taking maximal advantage of commutativity, force the stack to depth 3 or 4, respectively.

In your comments, explain why there can be no clever evaluation order for these expressions that requires a smaller stack depth.

(d) **WARNING: THIS PART REQUIRES SOME THOUGHT AHEAD OF TIME.**

Define the function

\[
\text{toRPNopt} :: \text{ Exp} \to ([\text{StackInstr}], \text{ Integer})
\]

that returns a pair containing (1) an optimal sequence of operations to evaluate the given arithmetic expression, and (2) the maximum number of values simultaneously on the stack during the execution of this sequence. Optimal here is defined to mean "requiring the smallest amount of stack space", which means having the smallest possible maximum number of values on the stack at once.

¹ There would be even more possibilities if we permit reassociation, e.g., computing \((a + b) + (c + d)\) as \(a + (b + (c + d))\). Unfortunately, floating-point arithmetic is not associative in general (ask if you don’t know why!) so this might change the answer. Thus, for this problem you should only consider the choice of evaluating each operator by doing the left-subexpression first or the one starting on the right.
• This function can be computed inductively, using the optimal instruction sequence for the first operand, the optimal sequence for the second operand, and the stack sizes they each require.
• You can decide whether to evaluate the left side first or the right side first just by looking at the stack depth each requires (and without looking at the particular operations!). Do not write a separate helper function that does nothing but compute stack depths.
• You can use pattern-matching and a where clause (or let...in) to get the two results from each recursive call.

8. Run `svn commit` to commit your answers, thereby submitting your homework.

Extra Credit [5%]

• Write the function

```
fromRPN :: [StackInstr] -> Exp
```

that converts a list of stack operations to the corresponding arithmetic expression. You may assume that evaluation of these stack operations starting with an empty stack would yield a stack containing only a single number, the answer. (That is, you may assume that the stack operations really do correspond to an arithmetic expression.)