Designing Your Own List Type

Functional Programming with Haskell: Homework 1

Boilerplate Material

Since this is the first homework, read a little bit of boilerplate to understand how this is going to work.

Goals:

The homeworks for this pseudo-course will be designed with a few goals in mind:

- Extend the content of the lectures,
- Give people ample opportunity to practice (because that's how you really get good at any sort of programming),
- and allow plenty of flexibility in how much time you can and want to commit to this course.

Homework Tiers:

With that in mind, the homeworks will be divided into three tiers, as follows:

- **Important:** These parts are *really* important to do. If you don't do these, you might have trouble keeping up with the course.
- **Highly Recommended:** These parts are very helpful for really understanding what's going on, and you really should do them if you want to be good at this material. If you don't do these, you can probably continue along with the course just fine, but you really should try to do these — or at least make sure you feel confident that you could.
- **Good Practice:** These parts are for practice and may force you to come to grips with some slightly more complex ideas. Programming is a craft, and to be able to do this quickly and write good code, you need lots of practice; these sections are very good for additional practice and introducing you to more complex ideas. If you like this subject and have time, I definitely suggest doing these, but you'll be perfectly fine if you don't.

Development Environment

I recommend that you set up GHC and Cabal on your own computer. However, if this is very difficult for you, you should also be able to use the setup on Knuth. If you run Mac or Linux, I highly recommend using IHaskell for interactive development.

I will gladly help you individually to install any of these things, as setup can often be one of the trickiest bits.

You may find the following programs helpful as well. All of these may be used from command-line as well as via editor integration:

- `hdevtools`: Integrates with editors to quickly check your source code and give you warnings and errors.
- `ghc-mod`: Similar to `hdevtools`, integrates with editors to quickly check your source code and give you warnings and errors.
- `hlint`: Searches your Haskell code for common stylistic problems (as well as other issues) and suggests corrections. May not be applicable for the first few homeworks until we begin using the true Haskell Prelude.

Homework Organization

Haskell is a very difficult language to learn and comes with quite a few built-in functions and types. These can take a while to learn, so instead of using all of them at once, the homeworks will build up to the full Haskell built-in library (called the Haskell `Prelude`). The idea is that once you've completed the first few homeworks, you will understand how the `Prelude` works, and feel like you can write almost any function available in it.

With that said, keep in mind that the library you'll be using for the first few assignments is *not* representative of real-world Haskell. However, working through this will give you a very good grasp for later, and will motivate a lot of the choices we make later (and will explain why the `Prelude` and the language itself are designed the way they are).

In this assignment, you can use the functions and types available in the custom `Prelude.Week0` module. You can set this up in your assignments with the following code in a Haskell source file:

```
{-# LANGUAGE NoImplicitPrelude #-}
import Prelude.Week0
```

or the following code in an IHaskell cell at the top of a notebook:

```
:set -XNoImplicitPrelude
import qualified Prelude as P ()
import Prelude.Week0
```

Note that in IHaskell, the type declation (and fixity declaration for operators) must come immediately before the actual value or function declaration. You can install the necessary package `haskell-course-preludes` via the commands

```
# Update available packages
$ cabal update

# Install the preludes for this course
$ cabal install haskell-course-preludes
```

Details

Random other things I figure I should mention:

- Once you're done with whatever you plan on doing, go ahead and send it to me at `andrew.gibiansky@gmail.com`. I'll take a look and maybe give you some feedback on what works, what doesn't, what you did well, and what you could have done more neatly.
- If you have any trouble, harass me. I'm happy to allocate time to help anyone individually — Haskell is hard, and it really helps to have someone to explain bits and pieces to you.
- If you finish part of the homework, I will make and email you a custom-made AWESOME certificate detailing that you are awesome. It will have pretty colors and fonts, I promise!

Exercises

Finally, onwards to the interesting bits! The goal of this assignment is to practice defining data types, writing type signatures, and writing recursive function definitions. What you create as a final product is a small library for dealing with linked lists. In the real Haskell prelude, lists of type `a` are written as `[a]`, and most of these functions are available either in `Prelude` or `Data.List`.

Important

Complete the following definition of a linked list data type in Haskell:

```haskell
-- Linked list with a head and a tail.
-- Potential can be empty (nil).
data List a = ...
```

Note that `List` is parameteric in the value `a` that it stores; make sure your lists could store any value, such as `Int`s or `String`s.

In order to debug the functions you will need to define below, you'll probably want something that converts a list into a `String`. You may want to start by defining a function with the following type signature:

```haskell
-- Convert a list of ints into a string.
showIntList :: List Int -> String
```

For the actual assignment, start by implementing the following functions that operate on these linked lists:

```haskell
-- Return the first element.
head :: List a -> a

-- Drop the first element, return all others.
tail :: List a -> List a

-- Apply a function to all elements.
```

```haskell
-- Return a list with the results.
map :: (a -> b) -> List a -> List b
```

In some cases, your functions might not really make sense — for instance, what do you do if the user tries to take the `head` of an empty list? For that, you should use the `error` function:

```haskell
error :: String -> a
```

This function will crash the program with an informative error message (the `String`) if the result is ever evaluated.

Highly Recommended

In Haskell, in addition to defining your own functions, you can define your own binary operators. This functionality is used very often by library authors, and you'll often see unusual operators being used in Haskell code. These operators definitions look like the following example:

```haskell
-- Strange demo operator!
-- 11 <<!>> 10 == 21
(<<!>>) :: Int -> Int -> Int

-- Set the fixity of this operator.
-- 'r' stands for 'right-associative'
-- There's also 'infixl' for left associative operators and
-- just 'infix' for non-associative operators.
-- The digit is the precedence, and must be between 0 and 9 inclusive,
-- where 9 binds most tightly and 0 least tightly. (Function application has precedence 10).
infixr 3 <<!>>

-- In IHaskell, this declaration should be immediately
-- preceeded by the fixity declaration and the type signature above.
x <<!>> y =
  if x > y
  then x + y
  else x - y
```

To keep with the spirit of Haskell, let's start off by defining two of our own operators:

```haskell
-- Define your own list indexing operator.
(!!) :: List a -> Int -> a

-- Define a list concatenation operator.
(++) :: List a -> List a -> List a
```

Next, let's define a few common functions:

```haskell
-- Return the first few elements of a list.
-- The integer says how many elements to return. It must be non-negative.
take :: Int -> List a -> List a

-- Repeat an element an infinite number of times.
-- Remember that since Haskell is lazy, as long as you don't evaluate the entire list,
-- it won't take forever! Infinite data structures are common and very useful.
repeat :: a -> List a

-- Compute the length of a list.
length :: List a -> Int

-- "Fold" over a list. This function is a bit complicated; you may have seen it
-- called 'reduce'. It takes three arguments:
--   1. A function which aggregates results.
--   2. A starting value (before aggregation).
--   3. A list over which to aggregate the results.
-- For instance, to sum all the elements of a list, you can write:
--     add x y = x + y
--     foldl add 0 myList -- Sum all elements in myList
-- To multiply all the elements of a list, you can write:
--     mul x y = x * y
--     foldl mul 1 myList -- Multiply all elements in myList
foldl :: (a -> b -> a) -> a -> List b -> a
```

Good Practice

These last few are very good practice, so do them if you have time!

```haskell
-- Check whether an integer exists in a list of integers.
elem :: List Int -> Int -> Bool

-- Perform a fold from the right.
-- This is like foldl, except it goes from the right instead of the left.
foldr :: (a -> b -> b) -> b -> List a -> b

-- Return a list after dropping a few elements from the front.
drop :: List a -> Int -> List a

-- Repeat a list multiple times. The integer dictates how many copies you want.
-- Return the conjoined list.
replicate :: Int -> List a -> List a

-- Convert from two lists into a single list of tuples.
zip :: List a -> List b -> List (a, b)
```