Answer Key: Designing Your Own List Type

Functional Programming with Haskell: Answer Key 1

Important

First we define a linked list:

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

Note that `List` is parameteric in the value `a` that it stores. If it's equal to `Empty`, it's empty, otherwise it holds an element and a pointer to the next list (the tail).

We can print one with the following function:

```haskell
-- Convert a list of ints into a string.
showIntList :: List Int -> String
showIntList Empty = "."
showIntList (Cons int next) = intToString int <> " " <> showIntList next
```

Let's operate on these lists:

```haskell
-- Return the first element.
head :: List a -> a
head Empty = error "Empty list"
head (Cons x _) = x

-- Drop the first element, return all others.
tail :: List a -> List a
tail Empty = error "Empty list"
tail (Cons _ xs) = xs

-- Apply a function to all elements.
-- Return a list with the results.
map :: (a -> b) -> List a -> List b
map _ Empty = Empty
map f (Cons x xs) = Cons (f x) (map f xs)
```

Highly Recommended

```haskell
-- Define your own list indexing operator.
(!!) :: List a -> Int -> a
Empty !! _ = error "Empty list"
(Cons x _) !! 0 = x
(Cons _ xs) !! n =
  if n < 0
  then error "Negative intext"
  else xs !! (n - 1)

-- Define a list concatenation operator.
(++) :: List a -> List a -> List a
Empty ++ xs = xs
xs ++ Empty = xs
(Cons a as) ++ bs = Cons a (as ++ bs)

-- 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
take 0 _ = Empty
take n (Cons x xs) = Cons x (take (n - 1) xs)

-- Repeat an element an infinite number of times.
repeat :: a -> List a
repeat x = Cons x (repeat x)

-- Compute the length of a list.
length :: List a -> Int
length Empty = 0
length (Cons _ xs) = 1 + length xs

-- "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
foldl _ start Empty = start
foldl f start (Cons x xs) =
  foldl f (f start x) xs
```

Good Practice

```haskell
-- Check whether an integer exists in a list of integers.
elem :: List Int -> Int -> Bool
elem Empty _ = False
elem (Cons x xs) y =
  x == y || elem xs y

-- 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
foldr _ start Empty = start
foldr f start (Cons x xs) =
  f x (foldr f start xs)

-- Return a list after dropping a few elements from the front.
drop :: Int -> List a -> List a
drop 0 xs = xs
drop n (Cons x xs) = drop (n - 1) xs

-- Repeat a list multiple times. The integer dictates how many copies you want.
-- Return the conjoined list.
replicate :: Int -> List a -> List a
replicate num list =
  foldl (++) Empty $ take num $ repeat list

-- Convert from two lists into a single list of tuples.
zip :: List a -> List b -> List (a, b)
zip Empty _ = Empty
zip _ Empty = Empty
zip (Cons x xs) (Cons y ys) = Cons (x, y) (zip xs ys)
```