# *CSCI 490 problem set 4*

## *Due Tuesday, February 16*

*Revision 1: compiled Tuesday 9th February, 2016 at 15:51*

---

On this problem set, when you are asked to *prove* something, you should give a formal-style proof using a structured proof format and equational reasoning. On the other hand, if you are asked to *show* or *justify* something, an informal (but still convincing) argument will suffice.

The exercises may refer to the following standard definitions:

```
length :: [a] -> Integer
length []     = 0
length (_:xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

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

(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
```

## *List induction*

**Exercise 1** You must complete **at least three** out of the following seven proofs involving list induction. You need not do more than three, although you may do more if you feel that the practice is useful.

(a) Prove that `map id = id`.

(b) Prove that for all functions `f` and `g` of appropriate type,

`map (f . g) = map f . map g.`

(c) Give a function `f` such that

`length = foldr f 0,`

and prove it.

To prove that two functions are equal, it suffices to show they have the same result on all inputs. That is, to prove `f = g`, it suffices to prove that for all `x` of the appropriate type, `f x = g x`.

(d) Prove that for all lists `xs` and `ys`,

```
length (xs ++ ys) =   length xs + length ys.
```

(e) Give an alternate definition of `(++)` via `foldr`, and prove that your implementation gives the same results as the standard definition.

(f) Prove that `(++)` is associative. That is, prove that for all lists `xs`, `ys`, and `zs`,

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs).
```

(g) State and prove a theorem relating `map` and `(++)`.

## *Difference lists and tree induction*

Recall the `isBST` function from Problem Set 2, to test whether a binary tree is a valid binary search tree. One way to implement it is by checking whether an inorder traversal of the tree is sorted, as follows:

```
data Tree a where
  Empty  :: Tree a
  Node   :: a -> Tree a -> Tree a -> Tree a

foldTree :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree e n Empty          = e
foldTree e n (Node a l r)  = n a (foldTree e n l) (foldTree e n r)

inorder :: Tree a -> [a]
inorder = foldTree [] (\a l r -> l ++ [a] ++ r)

isSorted :: [Integer] -> Bool
isSorted xs = and (zipWith (<) xs (tail xs))

isBST :: Tree Integer -> Bool
isBST = isSorted . inorder
```

Optional exercise for the curious: why doesn't `isSorted` crash when given the empty list as input?

However, as noted before, `inorder` takes $O(n^2)$ time in the worst case. The remainder of this problem set first walks you through the process of understanding why `inorder` can take $O(n^2)$ time, and then through the development of an implementation with worst-case $O(n)$ running time.

**Exercise 2** For each of the following functions, state its type, and explain what it does.

(a) `foldr (++) []`

(b) `foldl (++) []`

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

**Exercise 3 (Extra Credit)** Prove that

```
foldr (++) [] = foldl (++) [].
```

*Hint*: Exercise 3 is nontrivial. You will probably need to prove a lemma about `foldl`. Ask me if you want some hints on what lemma to prove. Also, you are welcome to assume without proof that `(++)` is associative, whether or not you chose to do Exercise 1(f).

---

Although they have the same *result*, it turns out that these two folds do *not* have the same *time complexity*!

Analyzing the time complexity of lazy programs turns out to be somewhat tricky in general[1]. For the purposes of this assignment, however, we can get away with just assuming that the functions are strict (that is, that function arguments are fully evaluated before the function body), which simplifies the analysis.

[1] See Okasaki [1999] for a beautiful and comprehensive treatment; come by my office if you want to look at it.

To analyze a function, we first assume that its arguments are completely evaluated, and then we count the *number of reduction steps* needed to reach a result which is itself fully evaluated. A reduction step is simply defined as replacing the left-hand side of a function definition with its right-hand side.[2] For example, if evaluating the function `f` takes constant time, `map f xs` takes time linear in the length of `xs`—it has to do one reduction step for each cons in the list (plus one more for the empty list).

[2] In general, one must be careful when variables on the left-hand side of a function definition are used multiple times on the right-hand side; the values of such variables will be shared and only evaluated once. GHC's runtime system actually maintains a *graph* of expressions rather than just doing simple textual substitution. However, on this assignment the issue will not come up.

**Exercise 4**

(a) Show that `xs ++ ys` has time complexity $O($`length xs`$)$, *i.e.* fully evaluating `xs ++ ys` requires a number of reduction steps linear in the length of `xs`. Note in particular that the time complexity is independent of the length of `ys`.

(b) Let `xss :: [[T]]` be a list of lists (of some type `T` which does not really matter). Assume, for simplicity, that all the elements of `xss` are lists of length 1. That is,

```
xss = [[x], [y], [z], ...]
```

Show that `foldr (++) [] xss` has time complexity $O($`length xss`$)$.

(c) What is the time complexity of `foldl (++) [] xss`? Justify your answer.

*Hint*: it is not the same as for `foldr`!

---

Now consider the type of *binary parenthesizations*,

```
data Parens a where
  Leaf :: a -> Parens a
  Bin  :: Parens a -> Parens a -> Parens a
```

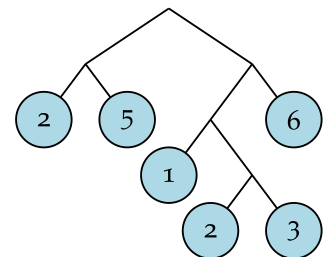


Figure 1: An example value of type `Parens Integer`

A value of type `Parens a` can be thought of as a binary tree with data at the leaves, as illustrated in Figure 1. It can also be thought of as a fully parenthesized sequence of `a` values, where each pair of

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

parentheses contains two expressions (either a value of type `a` or another parenthesized expression). For for example, the tree in Figure 1 corresponds to the parenthesized expression ((25)((1(23))6)).

**Exercise 5** Write a fold for `Parens`, and use it to implement a function

```
flatten :: Parens a -> [a]
```

which "removes the parentheses" by concatenating all the `a` values together.

Note that in various guises, operations such as this are rather common—for example, consider traversing a syntax tree representing a program and collecting a list of errors or warnings.

**Exercise 6** State the induction principle for `Parens`.

**Exercise 7** What is the worst-case big-O time complexity of `flatten`, where $n$ is the number of values in the input tree?

*Hint*: consider Exercise 4.

---

Now consider the type

```
type DList a = [a] -> [a]
```

of *difference lists*[3]. A value of type `DList a` is thus a function which transforms one list of type `[a]` into another. The idea is to use `DList a` as a (slightly[4] strange) way to encode a list. In particular, we will encode the list `xs :: [a]` as the *function* which prepends `xs`, that is, the function

```
\ys -> xs ++ ys
```

which takes any list `ys` and yields `xs ++ ys` as the result.

[3] This terminology originates in Prolog.

[4] OK, very.

**Exercise 8** Implement functions

```
toDList :: [a] -> DList a
```

and

```
fromDList :: DList a -> [a].
```

`toDList` should send a list `xs` to its encoding as described above. `fromDList` should send an encoding of a list back to the original list.

**Exercise 9** If you have defined `toDList` and `fromDList` correctly, exactly one of the following two statements should be true. Prove the one that is true, and give a counterexample for the other.

- `toDList . fromDList = id`
- `fromDList . toDList = id`

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

**Exercise 10** Define a function

```
append :: DList a -> DList a -> DList a
```

such that

```
toDList (xs ++ ys) = toDList xs `append` toDList ys.
```

Prove it.

**Exercise 11** Define `emptyDList :: DList a` (it should be an identity for `append`).

**Exercise 12** What is the time complexity of

```
fromDList . foldr append emptyDList . map toDList?
```

What about

```
fromDList . foldl append emptyDList . map toDList?
```

Briefly justify your answers.

*Hint*: you will probably not get very far on this problem without trying some examples to develop your intuition. Just try both functions on a short example list, and write out the step-by-step reductions.

**Exercise 13** Now, let's bring everything together. Using the functions you have defined previously, define a variant function

```
flatten' :: Parens a -> [a]
```

which works by converting the `a` values in the tree to `DList a` values (representing singleton lists), combining them with `append`, and then, at the end, converting the resulting `DList a` back to `[a]`.

**Exercise 14** Using the induction principle for `Parens`, prove that `flatten = flatten'`. What is the worst-case big-O time complexity of `flatten'`?

## *References*

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.