

CSci 360, Fall 2004, Assignment 11

This assignment, worth 40 points, is due Friday, November 19, at 4pm. To submit your solution, send e-mail to cburch@cburch.com with the file containing your function definitions as an attachment.

If we wanted to define a tree where each node can have any number of subtrees, we might define the Tree type as follows.

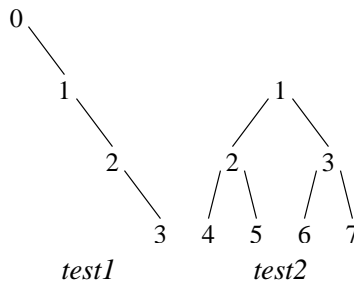
```
data Tree a = Node a [Tree a] deriving Show
```

Given this, we can easily define the following two accessor functions to retrieve data from a tree value.

```
value (Node ret _) = ret
children (Node _ ret) = ret
```

For testing purposes, we also define two sample trees.

```
test1 = Node 0 [Node 1 [Node 2 [Node 3 []]]]
test2 = Node 1 [Node 2 [Node 4 [], Node 5 []], Node 3 [Node 6 [], Node 7 []]]
```



Your assignment is to define the functions described below based on the above definitions.

Important: None of your functions should include the logic to step through any lists. Instead, you should rely on the library of list functions, including *map*, *foldl*, *(++)*, and *sum*. That is, if you were to define the *length* function on a list, you would *not* write either of the following.

```
myLength [] = 0 -- neither of these is allowed
myLength (x:xs) = 1 + myLength xs
```

```
myLength list = if list == [] then 0 else 1 + myLength (tail list)
```

Instead, you might write one of the following.

```
myLength list = sum (map (const 1) list) -- both of these are OK
```

```
myLength list = foldl (\x y->x+1) 0 list
```

Before you start, you might also want to glance at the note following the third problem.

1. The *treeSize* function, which returns the total number of nodes in the tree.

```
treeSize :: Tree a -> Integer
```

For example:

```
treeSize test1 returns 4
treeSize test2 returns 7
```

2. The *level* function, which takes an integer *k* and a tree and returns a list of all the values occurring on level *k* of the tree (where we understand the root node to be 0).

```
level :: Integer -> Tree a -> [a]
```

For example:

```
level 3 test1 returns [3]
level 2 test2 returns [4,5,6,7]
```

3. The *mapTree* function, which takes a function *f* and a tree *T* and returns a new tree whose structure is identical to *T*'s structure, but each value of the new tree is the result of applying *f* to the corresponding value in *T*.

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

For example:

```
mapTree (*2) test1 returns Node 0 [Node 2 [Node 4 [Node 6 []]]]
```

Note: If you found the above problems challenging, stop: These last 10% of the assignment points won't be worth it. But as an additional challenge, to earn the final 10%, write each function so that no identifiers or patterns relating to trees occur in any of the definitions, similarly to how one can define a *length* function on lists without actually naming the list as a parameter.

```
myLength = sum . map (const 1)
```

```
myLength = foldl (\x y->x+1) 0
```

You can continue to have *level* take a named Integer parameter, though.

To do this, the following built-in functions for manipulating functions are useful.

<code>f . g</code>	<code>= \x -> f (g x)</code>	composes two functions
<code>flip f</code>	<code>= \x y -> f y x</code>	flips a function's argument order
<code>const c</code>	<code>= \x -> c</code>	creates a constant function
<code>id</code>	<code>= \x -> x</code>	the identity function

The following is not a built-in function, but you can define and use it for the situation where you want a function to access a parameter multiple times.

```
spread f g = \x -> (f x) (g x)
```

It's a mathematical fact that given the above pre-defined functions, one can always write a function in this way — i.e., using no identifiers whatsoever, including no lambda expressions. (In fact, only *spread*, *const*, and *id* are necessary.) Illustrating this, here's the definition of the factorial function — not that I think puzzling it out will help you much. (First, though, we define *cond* to implement *if* as a genuine function.)

```
cond b x y = if b then x else y
```

```
factorial = spread (flip cond 1 . (==) 0) (spread (*) (factorial . flip (-) 1))
```

Trying to write functions like this isn't easy.