Why Functional Programming Matters

Roger Costello
January 2013

Below is my summary of the outstanding paper by John Hughes titled, *Why Functional Programming Matters*.

Hughes says that the purpose of his paper is to demonstrate to the "real world" that functional programming is vitally important.

I say that the purpose of his paper is to make you smarter. The paper presents a series of problems. As I proceeded through the problems, I realized that I was on a beautiful journey to the promised land of solving problems by simply combining existing parts. It was a thrilling, mind-expanding journey.

Hughes implemented the solutions to the problems using Miranda. I converted them to Haskell.

Modularity is the Key to Successful Programming

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs.

Modular design brings with it great productivity improvements:

1. Small modules can be coded quickly and easily.
2. General purpose modules can be re-used, leading to faster development of subsequent programs.
3. The modules of a program can be tested independently; helping to reduce the time spent debugging.

It is now generally accepted that modular design is the key to successful programming.

Functional Languages Provide Two New Modularity Mechanisms

Conventional languages place limits on the way problems can be modularized. Functional languages push those limits back. In particular, these two features of functional languages can contribute greatly to modularity:

1. Higher-order functions
2. Function composition, accompanied by lazy evaluation

Why is it called "Functional Programming?"

Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other

functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. These functions are much like ordinary mathematical functions.

More Modularity = More Glue

When writing a modular program to solve a problem, one first divides the problem into sub-problems, then solves the sub-problems, and combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language.

Analogy

One can appreciate the importance of glue by an analogy with carpentry. A chair can be made quite easily by making the parts – seat, legs, back, etc. – and sticking them together in the right way. But this depends on an ability to glue wood together. Lacking wood glue, the only way to make a chair is to carve it in one piece out of a solid block of wood, a much harder task. This example demonstrates both the enormous power of modularization and the importance of having the right glue.

Functional Languages Provide Two New Kinds of Glue

Functional languages provide two new, very important kinds of glue. The glue enables programs to be modularized in new ways and thereby greatly simplified. This is the key to functional programming's power – it allows improved modularization. It is also the goal for which functional programmers must strive – smaller and simpler and more general modules, glued together with the new glue.

What is "Glue" in Programming?

Suppose function (module) `f` is defined as follows:

```
f = g a b
```

Module `f` is defined by gluing together modules `g`, `a`, and `b`. Specifically, function `f` is defined as the application of function `g` to `a` and `b`. So the "glue" is the ability apply function `g` to arguments `a` and `b` (some people call this "binding" `a` and `b` to `g`, which makes the glue analogy very appropriate).

Two New Kinds of Glue

The first kind of glue enables simple functions to be glued together to make more complex ones. Function "`h`" may be an argument to function "`g`". Function "`g`" is called a higher-order function.

The second kind of glue that functional languages provide enables whole programs to be glued together. Recall that a complete functional program is just a function from its input to its output. If `f` and `g` are such programs, then (`g . f`) is a program which, when applied to its input, computes

```
g (f input)
```

The program f computes its output which is used as the input to program g. This might be implemented conventionally by storing the output from f in a temporary file. The problem with this is the temporary file might occupy so much memory that it is impractical to glue the programs together in this way. Functional languages provide a solution to this problem. The two programs f and g are run together in strict synchronization. f is only started once g tries to read. Then f is suspended and g is run until it tries to read another input. As an added bonus, if g terminates without reading all of f's output then f is aborted. f can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies – a powerful modularization.

Since this method of evaluation runs f as little as possible, it is called "lazy evaluation." It makes it practical to modularize a program as a generator which constructs a large number of possible answers, and a selector which chooses the appropriate one. While some other systems allow programs to be run together in this manner, only functional languages use lazy evaluation uniformly for every function call, allowing any part of a program to be modularized in this way. Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.

Gluing Functions Together

The first kind of glue enables simple functions to be glued together to make more complex ones. It will be illustrated with several list processing problems.

Problem #1: Define a function sum that adds up all the elements of a list.

sum must be defined for two kinds of argument: an empty list and a non-empty list. We define the sum of an empty list to be zero:

```
sum [] = 0
```

The sum of a non-empty list can be calculated by adding the first element of the list to the sum of the others:

```
sum (x:xs) = x  + (sum xs)
```

Here's our definition of sum:

```
sum [] = 0
sum (x:xs) = x  + (sum xs)
```

Note that (:) is replaced with (+) and [] is replaced with 0.

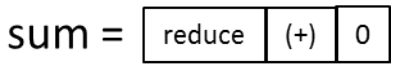
The problem with those replacements is that they are specific to sum. The result is a function that can only be used to solve one problem. We seek to create general modules that can be used in solving many problems.

Terminology: In functional programming, a reduce is a function that splits apart a recursive data structure (such as a list) and recombines it through the use of a given combining operation, building up a return value. The arguments to reduce are a combining function, some default values, and a recursive data structure.

Let's create a general list processing module reduce. It takes two arguments, something to replace (:) with and something to replace [] with. Computing sum is accomplished by gluing together reduce and the two replacements, (+) and 0:

```
sum = reduce (+) 0
```

In graphical form:



reduce is a function and (+) is a function. So we are gluing together a generic function (reduce), a specialized function (+), and a simple value (0) to create a complex function (sum).

Here's an example:

```
sum [1, 2, 3, 4]          -- returns 10
```

Having modularized sum in this way, we can reap benefits by reusing its parts. The most interesting part is reduce, which we will use over and over.

Problem #2: Define a function product that multiplies the elements of a list.

reduce is reused to solve this problem. It is glued together with multiplication (*) and 1:

```
product = reduce (*) 1
```

In graphical form:

Here's an example:

```
product [1, 2, 3, 4]          -- returns 24
```

We solved a new problem with no further programming.

Defining reduce

The definition of reduce can be derived by replacing each occurrence of sum (in its original definition) with reduce f a, where f is any function (such as (+)) and a is the replacement value for an empty list. Each occurrence of 0 is replaced with a, and each occurrence of + is replaced with f. So this:

```
sum [] = 0
```

is replaced with this:

```
reduce f a [] = a
```

And this (I've switched to using prefix notation for addition):

```
sum (x:xs) = (+) x (sum xs)
```

is replaced with this:

```
reduce f a (x:xs) = f x (reduce f a xs)
```

Here's our definition of reduce:

```
reduce f a [] = a
reduce f a (x:xs) = f x (reduce f a xs)
```

The first argument f is a function. f replaces the list constructor operator (:). The second argument a is a value. a replaces the empty list. The third argument is the list to be reduced.

Problem #3: Define a function anytrue that returns True if there are any True values in a list of Boolean values.

reduce is used again to solve this problem. It is glued together with the Boolean 'or' (||) and False:

```
anytrue = reduce (||) False
```

Here's an example:

```
anytrue [False, False, True, False]          -- returns True
```

Problem #4: Define a function alltrue that returns True if all values in the list are True.

reduce is glued together with the Boolean 'and' (&&) and True:

```
alltrue = reduce (&&) True
```

Here's an example:

```
alltrue [False, False, True, False]          -- returns False
```

Problem #5: Define a function identity that returns the list unchanged.

reduce is glued together with the list constructor (:) and the empty list, []:

```
identity = reduce (:) []
```

Here's an example:

```
identity "Hello World"          -- returns "Hello World"
```

Problem #6: Define a function append that appends list k onto list j.

reduce is glued together with the list constructor (:), list k, and list j:

```
append j k = reduce (:) k j
```

Notice that the arguments to reduce are reversed from the arguments to append. So j is "the list" and k is "the replacement part for the empty list."

Here's an example:

```
append [1, 2] [3, 4]            -- returns [1,2,3,4]
```

Recall the definition of reduce:

```
reduce f a [] = a
reduce f a (x:xs) = f x (reduce f a xs)
```

When "the list" j is empty then return "the replacement part for the empty list" k:

```
append [] [3, 4]          -- returns [3,4]
```

When "the list" j is not empty then take its first element and prepend it to the list formed by reducing the rest of j.

Let's trace the example:

```
append [1, 2] [3, 4]
        = reduce (:) [3, 4] [1, 2]
        = reduce (:) (3 : 4 : []) (1 : 2 :[])
        = (:) 1 (reduce (:) (3 : 4 : []) (2 : []))
        = (:) 1 ( (:) 2 (reduce (:) (3 : 4 : []) []))
        = (:) 1 ( (:) 2 (3 : 4 : []))
```

```
= (:) 1 (2 : 3 : 4 : [])
= (1 : 2 : 3 : 4 : [])
= [1, 2, 3, 4]
```

Problem #7: Define a function doubleall that doubles each element in a list.

Here's an example:

```
doubleall [1, 2, 3, 4]                    -- returns [2, 4, 6, 8]
```

reduce is glued together with doubleandconstruct and [], where doubleandconstruct doubles a number and then prepends it to the list:

```
doubleall = reduce doubleandconstruct []

doubleandconstruct num list = (:) (2*num) list
```

Recall that:

```
reduce f a (x:xs) = f x (reduce f a xs)
```

So,

```
reduce doubleandconstruct [] (x:xs)
```

results in invoking doubleandconstruct with x and some list:

```
doubleandconstruct x list
```

doubleandconstruct doubles x and prepends it to list:

```
doubleandconstruct x list = (:) (2*x) list
```

The problem with doubleandconstruct is that it is a specific, dedicated function; it can only be used to solve only one problem, doubling a number and prepending it to a list.

Let's modularize doubleandconstruct.

fandconstruct is a generalized version of doubleandconstruct. fandconstruct takes three arguments: a function f, a value x, and a list. It constructs a new list by applying f to x and then prepending the result to the list:

```
fandconstruct f x list = (:) (f x) list
```

Now doubleandconstruct is modularized by gluing together fandconstruct and a function double:

```
doubleandconstruct = fandconstruct double

double n = 2*n

fandconstruct f x list = (:) (f x) list
```

fandconstruct applies f to x and then applies (:) to (f x) and list:

```
(:) (f x) list
```

Any definition of this form:

```
g(h x) y
```

can be simplified to this:

```
(g . h) x y
```

And the arguments x y can be eliminated, to just show the functions:

```
g . h
```

Thus, fandconstruct can be simplified by eliminating the second and third arguments and composing the list constructor operator (:) with f:

```
fandconstruct f = (:) . f
```

Let's assemble all the parts and see what we've got:

```
doubleall = reduce doubleandconstruct []        (1)

doubleandconstruct = fandconstruct double       (2)

double n = 2*n                                  (3)

fandconstruct f = (:) . f                       (4)
```

Now substitute fandconstruct double in (2) with (:) . double:

```
doubleall = reduce doubleandconstruct []        (5)

doubleandconstruct = (:) . double               (6)

double n = 2*n                                  (7)
```

Substitute doubleandconstruct in (5) with (:) . double:

```
doubleall = reduce ((:) . double) []

double n = 2*n
```

doubleall is now highly modularized. It is defined by gluing together general, reusable parts.

Let's trace the evaluation of this example:

```
doubleall [1, 2]
```

First, recall the definition of reduce:

```
reduce f a [] = a
reduce f a (x:xs) = f x (reduce f a xs)
```

Here's the trace:

```
doubleall [1, 2]
    = reduce ((:) . double) [] [1, 2]
    = reduce ((:) . double) [] (1 : 2 : [])
    = ((:) . double) 1 (reduce ((:) . double) [] (2 : []))
    = ((:) . double) 1 (((:) . double) 2 (reduce ((:) . double) [] []))
    = ((:) . double) 1 (((:) . double) 2 [])
    = ((:) . double) 1 ((:) 4 [])
    = ((:) . double) 1 (4 : [])
    = (:) 2 (4 : [])
    = 2 : 4 : []
    = [2, 4]
```

doubleall can be simplified still further:

```
doubleall = map double
```

map is a standard function that applies its argument (e.g., double) to every element of a list.

These two are equivalent:

```
map f = reduce ((:) . f) []
```

Problem #8: Define a function matrix that adds together all the elements in a matrix (which we represent by a list of lists).

Here's an example:

```
summatrix [[1,2,3,4],[1,2,3,4]]                    -- returns 20
```

Recall the definition of sum from Problem #1:

```
sum = reduce (+) 0
```

sum adds together all the elements in a list.

We just saw in the last problem that map f applies f to each element in the list. So, the solution to this problem is to apply sum to each element in the list and then sum the results:

```
summatrix = sum . map sum
```

Let's step back for a moment and reflect on what we have accomplished. By modularizing a simple function (sum) as a combination of a "higher order function" and some simple arguments, we have arrived at a part (reduce) that can be used to write down many other functions on lists with no more programming effort.

Problem #9: Consider a datatype of ordered labeled trees, defined by:

```
data Treeof a = Node a [Treeof a]
```

The definition says that a tree is a Node with label a followed by a list of subtrees.

For example, the tree:

is represented by:

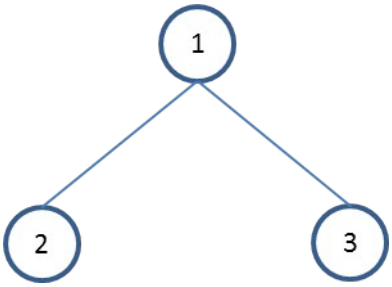
```
Node 1 []
```

The tree:

is represented by:

```
Node 1 [Node 2 []]
```

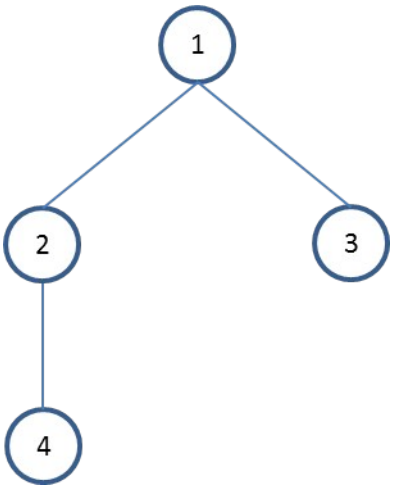
The tree:



is represented by:

```
Node 1 [Node 2 [], Node 3 []]
```

The tree:



is represented by:

```
Node 1 [Node 2 [Node 4 []], Node 3 []]
```

And so forth.

Let's create a function, `redtree` (reduce tree), analogous to `reduce`. Recall that `reduce` took two arguments, something to replace `(:)` with and something to replace `[]` with. Since trees are built using `Node`, `(:)`, and `[]`, `redtree` must take three arguments – something to replace each of these with. And since trees and lists are of different types, we will have to define two functions, one that operates on trees (`redtree`) and one that operates on lists (`redtree'`).

Here's the definition of `redtree` and `redtree'`:

```
redtree f g a (Node label subtrees) = f label (redtree' f g a subtrees)
redtree' f g a ((:) subtree rest) = g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a [] = a
```

f is any function. It replaces the Node constructor function.
g is any function. It replaces the list constructor function (:).
a is any simple value. It replaces the empty list, [].

Epiphany on designing functions

Design functions by first providing a series of generic arguments that are replacements for parts in the thing to be operated on. Some of the replacements are functions, others are simple values. Let f denote a replacement that is a function and let a denote a simple value:

```
myfunction f a
```

a is the replacement for the base case (when the thing being operated on is empty). f is the replacement for the constructor.

The last argument is the thing to be operated on. It consists of values and constructors, which I denote by v and c, respectively:

```
myfunction f a (v c v)
```

On the right hand side of the equal sign replace the constructor with a replacement:

```
myfunction f a () = a
myfunction f a (v c v) = f v v
```

Many interesting functions can be defined by gluing together redtree with other functions. For example, the sum of all the labels in a tree can be found using:

```
sumtree = redtree (+) (+) 0
```

The first (+) adds the number in Node to the sum of its subtrees. The second (+) adds the sum of the first subtree to the sum of the rest of the subtrees. The 0 is returned by redtree' when the list is empty.

```
sumtree (Node 1 [Node 2 [Node 3 []], Node 4 []])          -- returns 10
```

The product of all the labels in a tree can be found using:

```
productTree = redtree (*) (*) 1
```

The first (*) multiplies the number in Node to the product of its subtrees. The second (*) multiplies the product of the first subtree to the product of the rest of the subtrees.

```
productTree (Node 1 [Node 2 [Node 3 []], Node 4 []])     -- returns 24
```

What does this function do:

```
redtree (+) (*) 1
```

It adds the number in `Node` to the product of its subtrees. `1` is returned on an empty list. Let's apply it to a few trees.

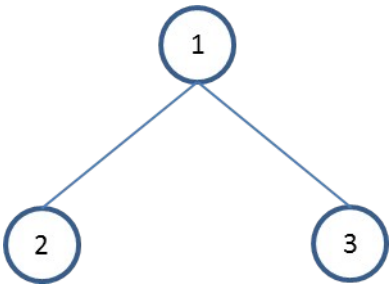
Tree #1


```
1 + 1                    -- returns 2
```

Tree #2

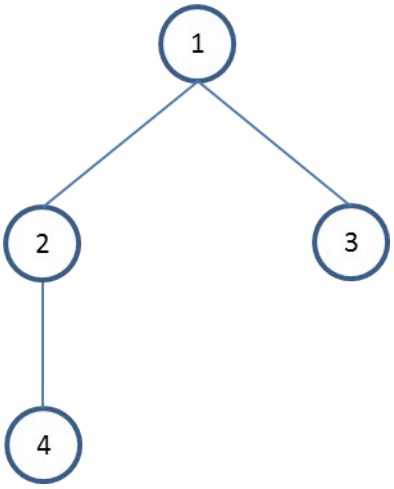

```
1 + (2 + 1)                    -- returns 4
```

Tree #3




```
1 + ((2 + 1) * (3 + 1))                -- returns 13
```

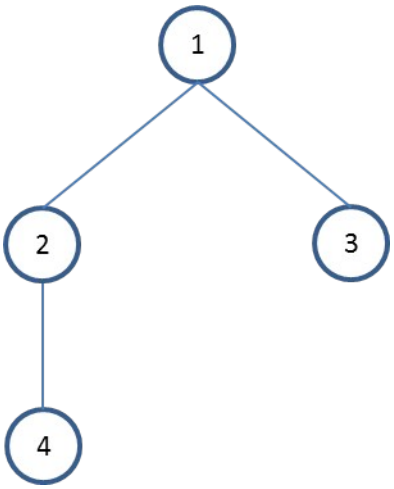
Tree #4




```
1 + ((2 + (4 + 1)) * (3 + 1))                -- returns 29
```

Problem #10: Define a function labels that produces a list of all the labels in a tree (i.e., labels serializes the tree to a list).

For this tree:



labels produces this list:

```
[1,2,4,3]
```

Notice that the tree is serialized to a list in a depth-first fashion.

Recall function append. It appends list k onto list j:

```
append [1, 2] [3, 4]              -- returns [1,2,3,4]
```

And recall the definition of redtree and redtree':

```
redtree f g a (Node label subtrees) = f label (redtree' f g a subtrees)
redtree' f g a ((:) subtree rest) = g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a [] = a
```

To solve the problem we will use (:) as the value of f. (:) will prepend a Node's label onto a list of its subtree's labels.

We will use append as the value of g. append will combine the list of labels from the first subtree with the list of labels from the remaining subtrees.

We will use [] as the value of a.

Here is the definition of labels:

```
labels = redtree (:) append []
```

That is a fantastic definition.

Lesson Learned
Before I read this section of Hugh's paper I had written my own function to print (serialize) all the labels in a tree as a list:

```
printTree :: Treeof a -> [a]
printTree (Node a []) = [a]
printTree (Node a [Node b []]) = a : b : []
printTree (Node a [Node b [], c]) = a : (printTree (Node b [c]))
printTree (Node a [Node b [c]]) = a : b : (printTree (c))
printTree (Node a [Node b [c], d]) = a : (printTree (c)) ++ (printTree (Node b [d]))
```

Contrast with Hugh's definition:

```
labels = redtree (:) append []
```

I am embarrassed at my implementation. Rather than gluing together modules, I brute-forced solved the problem.

This is a valuable lesson for me: Write code by gluing together modules. Don't brute-force implementations.

Recall that map is a standard function which applies its argument (a function) to every element of a list. Our final problem creates an analogous function for the tree data structure. We will call the function maptree.

Problem #11: Define a function maptree that applies a function h to each label in a tree.

Again, here's the definition of redtree and redtree':

```
redtree f g a (Node label subtrees) = f label (redtree' f g a subtrees)
redtree' f g a ((:) subtree rest) = g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a [] = a
```

We want maptree to apply h to the Node's label and then reconstruct Node with the updated label. So use (Node . h) as the value of f.

Use (:) to combine the mapped list of labels from the first subtree with mapped list of labels from the remaining subtrees. So use (:) as the value of g.

For an empty list return []. So use [] as the value of a.

Here is the definition maptree:

```
maptree f = redtree (Node . f) (:) []
```

That is another fantastic definition.

Let's take an example which uses maptree. Recall the function double:

```
double n = 2*n
```

Let's use it to double the value of each label in a tree. Then let's list the tree's values:

```
sampleTree = Node 1 [Node 2 [Node 4 []], Node 3 []]

sampleTreeDoubled = maptree double sampleTree

labels sampleTreeDoubled                    -- returns [2,4,8,6]
```

Final Comments

We are done with our discussion of the first kind of glue. We have accomplished much.

All this was achieved because functional languages allow functions to be expressed as combinations of parts – a general function and some particular specializing functions. Once defined, such higher-order functions allow many operations to be programmed very easily.

Whenever a new datatype is defined, higher-order functions should be written for processing it. This makes manipulating the datatype easy, and also localizes knowledge about the details of its representation.

Challenge Myself

Now it's time to challenge myself and see if I have learned the valuable lessons presented in Hughes' marvelous paper. I will attempt to create a function `treecontains x` that returns `True` if the tree contains `x` and `False` otherwise. Success means that I implement `treecontains x` by simply combining existing parts.

Here's the definition of `redtree` and `redtree'`:

```
redtree f g a (Node label subtrees) = f label (redtree' f g a subtrees)
redtree' f g a ((:) subtree rest) = g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a [] = a
```

I want `redtree` to apply `(== x)` to a `Node`'s `label` and then "or" its result with the result of checking for the presence of `x` in the subtrees. So I will use `((||) . (== x))` as the value of `f`.

I want to "or" the results of checking the labels from the first subtree with the results of checking the labels from the remaining subtrees. So I will use `(||)` as the value of `g`.

For an empty list return `False`. So I will use `False` as the value of `a`.

Here's my definition of `treecontains x`:

```
treecontains x = redtree ((||) . (== x)) (||) False
```

I did a couple tests and it works beautifully:

```
treecontains 4 (Node 1 [Node 2 [Node 4 []], Node 3 []])          -- returns True

treecontains 9 (Node 1 [Node 2 [Node 4 []], Node 3 []])          -- returns False
```

Without any coding I was able to implement a new function, simply by gluing together a general function (`redtree`) with some specialized functions and values (`==`,`||`, and `False`). This is so cool.

This is the end of half of Hugh's paper. The next half discusses how to achieve better modularity using:

> Function composition, accompanied by lazy evaluation

I am starting work on that now. Stay tuned…

Haskell Code

Here is all the Haskell code in one place. Place this in a file reduce.hs and you can immediately start using the functions.

```
import Prelude hiding (sum, product)
```

```
reduce f a [] = a
reduce f a (x:xs) = f x (reduce f a xs)

sum = reduce (+) 0

t1 = sum [1, 2, 3, 4]                    -- returns 10

product = reduce (*) 1

t2 = product [1, 2, 3, 4]                -- returns 24

anytrue = reduce (||) False

t3 = anytrue [False, False, True, False]  -- returns True

alltrue = reduce (&&) True

t4 = alltrue [False, False, True, False]  -- returns False

identity = reduce (:) []

t5 = identity "Hello World"               -- returns "Hello World"

append a b = reduce (:) b a

t6 = append [1, 2] [3, 4]                 -- returns [1, 2, 3, 4]

doubleall = reduce doubleandconstruct []

doubleandconstruct num list = (:) (2*num) list

t7 = doubleall [1, 2, 3, 4]               -- returns [2, 4, 6, 8]

doubleall_v2 = reduce doubleandconstruct_v2 []

doubleandconstruct_v2 = fandconstruct double

double n = 2*n

fandconstruct f x list = (:) (f x) list

t8 = doubleall_v2 [1, 2, 3, 4]            -- returns [2, 4, 6, 8]

doubleall_v3 = reduce doubleandconstruct_v3 []

doubleandconstruct_v3 = fandconstruct_v2 double

fandconstruct_v2 f = (:) . f

t9 = doubleall_v3 [1, 2, 3, 4]            -- returns [2, 4, 6, 8]

doubleall_v4 = reduce ((:) . double) []
```

```
t10 = doubleall_v4 [1, 2, 3, 4]                -- returns [2, 4, 6, 8]

doubleall_v5 = map double

t11 = doubleall_v5 [1, 2, 3, 4]                -- returns [2, 4, 6, 8]

summatrix = sum . map sum

t12 = summatrix [[1,2,3,4],[1,2,3,4]]          -- returns 20

data Treeof a = Node a [Treeof a]

tree1 :: Treeof Int
tree1 = Node 1 []

tree2 :: Treeof Int
tree2 = Node 1 [Node 2 []]

tree3 :: Treeof Int
tree3 = Node 1 [Node 2 [], Node 3 []]

tree4 :: Treeof Int
tree4 = Node 1 [Node 2 [Node 3 []]]

tree5 :: Treeof Int
tree5 = Node 1 [Node 2 [Node 4 []], Node 3 []]

-- See function "labels" below for a much more
-- elegant way of accomplishing the same thing
-- as this printTree function. This is an extremely
-- valuable lesson. In my printTree function I
-- am not gluing together modules. I am brute-force
-- solving the problem.

printTree :: Treeof a -> [a]
printTree (Node a []) = [a]
printTree (Node a [Node b []]) = a : b : []
printTree (Node a [Node b [], c]) = a : (printTree (Node b [c]))
printTree (Node a [Node b [c]]) = a : b : (printTree (c))
printTree (Node a [Node b [c], d]) = a : (printTree (c)) ++ (printTree (Node b [d]))


redtree f g a (Node label subtrees) = f label (redtree' f g a subtrees)

redtree' f g a ((:) subtree rest) = g (redtree f g a subtree) (redtree' f g a rest)

redtree' f g a [] = a

sumtree = redtree (+) (+) 0

t13 = sumtree tree5                            -- returns 10

productTree = redtree (*) (*) 1
```

```
t14 = productTree tree5                    -- returns 24

h = redtree (+) (*) 1

t15 = h tree5                              -- returns 29

labels = redtree (:) append []

t16 = labels tree5                         -- returns [1,2,3,4]

maptree f = redtree (Node . f) (:) []

t17 = maptree double tree5

t18 = labels t17                           -- returns [2,4,8,6]

sampleTree = Node 1 [Node 2 [Node 4 []], Node 3 []]
sampleTreeDoubled = maptree double sampleTree
sampleTreeList = labels sampleTreeDoubled

treecontains x = redtree ((||) . (== x)) (||) False

t19 = treecontains 4 tree5                 -- returns True

t20 = treecontains 9 tree5                 -- returns False
```