Original image by Luc Viatour / www.Lucnix.be

CAN'T SEE THE FOUR-EST FOR THE TREES.

2 June 2017

I was recently reading an excellent book about math education, *What's Math Got to Do with It* by Jo Boaler which included the following puzzle called "The Four 4s".

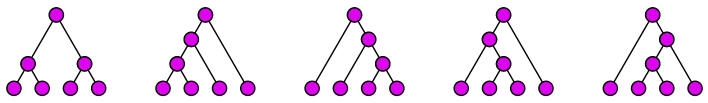
> Try to make every number between 0 and 20 using only four 4s and any mathematical operation (such as multiplication, division, addition, subtraction, raising to a power, or finding a square root), with all four 4s being used each time. For example
>
> $$5 = \sqrt{4} + \sqrt{4} + \frac{4}{4}$$
>
> How many of the numbers between 0 and 20 can be found?

It's a fun puzzle so you may want to take a crack at it yourself before reading on. But this isn't really about that puzzle. This is about how sometimes when the mathematical insights aren't flowing it's good to be a programmer.

When I tried to do the puzzle I pretty quickly worked out expressions for the numbers zero to ten. I got stuck on eleven so I skipped it. The rest were also easy except for thirteen and nineteen. So then I went back and started banging my head against eleven. After many hours (off and on) and endless scraps of paper filled with algabraic manipulations that all seemed to go in circles I started to wonder whether maybe the answer to the last question in the problem ("How many of the numbers between 0 and 20 can be found?" was "Not all of them.") So I took a quick, squinty peek at the answer in the back of the book just to make sure it didn't say something like, "11, 13, and 19" can't be expressed." But it looked like the answer was just a table of all twenty-one numbers.

So back to the scrap paper. More hours. Go to bed hoping I'll wake up with a brilliant insight. No dice. Finally I decide it's time to apply some computer science to the problem, by which I mean, of course, Brute Force Searching.

The expressions I'm looking for can be represented by trees with four leaves, all of which are 4s. How many trees is that? First off there are some differently shaped trees. With four leaves I can build trees with binary operators (such as + and ×) by grouping the leaves in various ways and then grouping those resulting trees until I reach the top. With just four leaves it's easy enough to just draw all the possible trees.



If you're not convinced that these are all the possible shapes of binary trees with four leaves, we can take a quick excursion into Catalan numbers, C_n, which can be defined as:

$$C_n = \frac{1}{n+1}\binom{2n}{k}$$

As it turns out, the Catalan numbers come up a lot in combinatorics. Their entry in the excellent *Online Encyclopedia of Integer Sequences*, where they are sequence A000108, contains fourteen pages of discussion and points out: "This is probably the longest entry in the OEIS, and rightly so." Richard P. Stanley has helpfully posted on his website an excerpt from his book *Enumerative Combinatorics: Volume 2* containing a problem in which the reader is asked to prove that the Catalan numbers count sixty-six different sets which he lists. The number of full binary trees with $n+1$ leaves is item (d) his list. And indeed, $C_3 = 5$ so the five trees shown above must be all the full binary trees with four leaves.

So we have five different shapes and each of those trees there are seven nodes: four leaf nodes and three nodes representing binary operators such as +, -, ×, /, and exponentiation. So there are 5^3 possible ways to fill in those nodes with different combinations of binary operators. Furthermore, each of the seven nodes could be wrapped in an unary operator such as negation or square root or left alone which gives us 3^7 more combinations. Multiply all these combinations by the five shapes of tree and there are $5^3 \times 3^7 \times 5 = 1,366,875$ possible trees.

Luckily, even my ancient laptop will be able to generate 1.4 million trees without breaking a sweat so it's time to write some code.[1]

My basic plan is to generate trees, evaluate them, and see if there were any that evaluated to eleven, thirteen, or nineteen without actually looking at what the trees are. If there are then I can go back to my scrap paper and buckle down to find the solutions, hopefully without having to try 1.4 million trees by hand. Haskell, with it's pleasant notation for defining recursive data types and its lazy lists seemed like as good language as any to solve this problem.[2] So here's what I came up with.

I can start by defining a data type to represent the trees. Haskell shines here as I can concisely define the structures I want to work with. (Lispy languages would also do well as first-class symbol objects and lists are perfect for represenenting abstract trees.)

```
data Tree = Plus Tree Tree
          | Minus Tree Tree
          | Times Tree Tree
          | Divide Tree Tree
          | Expt Tree Tree
          | Negate Tree
          | Sqrt Tree
          | Four
  deriving (Show)
```

I could obviously define a more general tree by replacing `Four` with something that could represent arbitrary values (e.g. `Value Integer`) but this is all I need for this problem so why overgeneralize.

Now I just need a function that can build up all possible trees starting from four `Four` leaves. I can write this as a recursive function that takes a list of subtrees and returns is a list of all possible trees that can be built on top of them.

```
trees :: [Tree] -> [Tree]
```

The base case is a list of just a single subtree upon which I can build three possible trees: the original subtree as is and also the tree wrapped in each of the unary operators.

```
trees [x] = [ f x | f <- [ id, Negate, Sqrt ] ]
```

Now I need two helper functions before I can get to the meat of `trees`. First `splits` which takes a list and returns all the ways it can be split into two non-empty parts:

```
splits xs = init $ tail $ zip (inits xs) (tails xs)
```

For instance:

```
splits [1..4] => [([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]
```

The other, `pairs`, takes two `Trees` and returns a list of all the ways they can be combined with the binary operators.

```
pairs a b = [ f a b | f <- [ Plus, Minus, Times, Divide, Expt ] ]
```

Now I implement the rest of `trees`:

```
trees xs = do
  (left, right) <- splits xs
  t1            <- trees left
  t2            <- trees right
  p             <- pairs t1 t2
  trees [p]
```

If you're not familiar with Haskell's `do` notation or perhaps only used to it in the context of the `IO` monad, note that here I'm using it with the list monad where it means, basically, for each item in the list returned by the expression on the right of a `<-`, bind the variables in the pattern on the left and evaluate the next line. The final expression, `trees [p]`, yields a list for each `p` and all those lists are concatenated together so the final value of the `do` block is one list of all the trees. Here's a simpler example. Given these two silly functions:

```
foo x = [x, 4 * x]
bar y = replicate y y
```

we can write a `do` expression like this:

```
do x <- [1,2]; y <- foo x; bar y
```

However that is simply syntactic sugar for this:

```
[1, 2] >>= (\x -> foo x >>= (\y -> bar y))
```

That may not tell us much unless we know that the definition of `>>=` in the list monad is:

```
xs >>= f = concat (map f xs)
```

So the desugared `do` expression is equivalent to this:

```
concat $ map (\x -> concat $ map bar $ foo x) [1, 2]
```

If you want, you can check that all these expressions evaluate to:

```
[1,4,4,4,4,2,2,8,8,8,8,8,8,8,8]
```

Alternatively, if you're familiar with Python generators, all of these are basically equivalent to:

```
for x in [1,2]:
    for y in foo(x):
        for z in bar(y):
            yield z
```

Next I need a way to evaluate the trees.

```
eval :: Tree -> Maybe Float
```

The `Float` part of the return type is because I'm going to be taking square roots so I can't use integers or rationals (and rationals are a pain in the butt in Haskell anyway). The `Maybe` is because not every tree is necessarily evaluable. For instance the sub-tree `Divide Four (Minus Four Four)` would result in dividing by zero.

Now I can use pattern matching to handle each of the different kinds of trees and—if I compile with appropriate warnings—the compiler will let me know if I missed any. The four binary operators that never fail all follow the same pattern which I can define in a helper function, `binop`.

```
binop :: (Float -> Float -> Float) -> Tree -> Tree -> Maybe Float
binop op a b = liftM2 op (eval a) (eval b)
```

This function takes a binary operator on two `Floats` and two `Trees` and uses the operator to combine the result of evaluating the two trees. I use `liftM2` from `Control.Monad` to handle unpacking and repacking the `Maybes` I get from evaluating the two trees. With `binop` in hand, I can easily define the evaluation rules for four of the five kinds of binary trees.

```
eval (Plus a b)  = binop (+) a b
eval (Minus a b) = binop (-) a b
eval (Times a b) = binop (*) a b
eval (Expt a b)  = binop (**) a b
```

Division follows the same pattern but I need to make sure that the second tree does not evaluate to zero. The function `mfilter`, also from `Control.Monad`, takes care of unwrapping and rewrapping the result of `eval`'ing the tree, returning `Nothing` if the result is zero.

```
eval (Divide a b) = liftM2 (/) (eval a) (mfilter (/=0) (eval b))
```

For evaluating trees representing unary ops the strategy is similar. A helper function, `unaryop`, uses `liftM` to lift a operator on `Floats` into the Maybe monad.

```
unaryop :: (Float -> Float) -> Tree -> Maybe Float
unaryop op a = liftM op (eval a)

eval (Negate a) = unaryop (0-) a
eval (Sqrt a)   = unaryop sqrt a
```

Finally, the evaluation for the literal tree, `Four`, is trivial:

```
eval Four = Just 4
```

Almost there. However, in addition to the many trees that evaluate to `Nothing`, a lot of trees will evaluate to values outside the 0-20 range I care about or to non-integer values. I can write some functions to help check whether a value is one I care about:

```
good :: Float -> Bool
good v = 0 <= v && v <= 20 && isInt v

isInt :: Float -> Bool
isInt n = fromIntegral (round n :: Int) == n
```

With those two functions I can write a wrapper around `eval` that returns `Maybe Int` when the value of the tree is an integer in the range 0-20.

```
evalToInt :: Tree -> Maybe Int
evalToInt t = round <$> mfilter good (eval t)
```

Again, I use `mfilter`, this time to make sure the value (if any) from `eval` satisfies my predicate `good`. If so, then `<$>` applies the function `round` to the value inside the `Maybe` and wraps the result back up in the same `Monad`[3]. So if the result of `mfilter` is a `Just` the result of `evalToInt` will be `Just` the rounded value while if `mfilter` returned `Nothing` then the result of `evalToInt` will also be `Nothing`.

Now I have all the pieces I need to compute the set of numbers that can be represented with these trees:

```
solutions :: Set Int
solutions = fromList $ mapMaybe evalToInt $ trees $ replicate 4 Four
```

The expression `replicate 4 Four` gives me the four starting leaves from which `trees` builds all the possible trees. The `mapMaybe` function is like `map` except that it discards all the `Nothing`s and unwraps all the `Just`s returned by the mapped function. Finally `fromList`, imported from `Data.Set`, will reduce the list to a set of the unique numbers produced. In `main` I'll print the difference between the set I'm looking for and the set of solutions:

```
main :: IO ()
main = print $ elems $ difference (fromList [0..20]) solutions
```

The moment of truth:

```
$ ./trees
[11,13,19]
```

Wait. Exactly the three values I couldn't find by hand! Maybe my peek at the answer was too squinty. Do I dare look at the answer? Finally I decide that I'm confident enough in this program to just go ahead and risk the spoiler by looking. Let's see. Oh @#%☠☆: they use factorial!

Factorial was not explicitly mentioned in the original problem statement but it did say "any mathematical operation" so of course that's legit. And it's actually quite easy to produce 11, 13, and 19 using factorial. For instance: $11 = \frac{\sqrt{4}\cdot(4!-\sqrt{4})}{4}$.

Just for grins, let me make sure my program can find the solutions too. Just need to make a few small changes starting with adding a new kind of `Tree`, `Factorial`:

```
data Tree = Plus Tree Tree
          | Minus Tree Tree
          | Times Tree Tree
          | Divide Tree Tree
```

```
            | Expt Tree Tree
            | Negate Tree
            | Sqrt Tree
            | Factorial Tree
            | Four
  deriving (Show)
```

Then add it to the set of unary wrappers around a single tree:

```
trees [x] = [ f x | f <- [ id, Negate, Sqrt, Factorial ] ]
```

Evaluating `Factorial` trees would not be hard to code. However the arbitrary trees I'm producing can contain fairly large values such as 4^{4^4}. The number of bits required to represent $n!$ is on the order of $n \log_2 n$, the number of bits required to represent $4^{4^4}!$ is greater than 10^{156} which well exceeds the number of atoms in the universe, estimated to be between 10^{78} to 10^{82}. Since my computer is definitely smaller than the universe and thus trying to take the factorial of such a large number isn't going to work, I'll cheat a bit and evaluate `Factorial Four` to the appropriate value and return `Nothing` for all other `Factorial` subtrees.

```
eval (Factorial Four) = Just (4 * 3 * 2)
eval (Factorial _)    = Nothing
```

Recompile and let's see if that did the trick.q

```
$ ./trees
[]
```

Ta da!

There are a few more things that one could do with this code. After I figured out eleven, thirteen, and nineteen myself, I hacked up some code to transate `Trees` into TeX output so I could look at the solutions. I've posted two sets of answers, the simplest and most complicated my program found here. And all the code from this essay, plus some extra, is here.

---

[1] To be completely honest, I didn't actually do all this math before I started writing code. It didn't feel like that big a problem though obviously it would grow quickly if we cared about larger trees where the number of shapes would grow and the number of ways to fill in the growing number of operator nodes would grow exponentially.

[2] I'm far from an expert Haskell programmer so please forgive me if I've done anything silly here.

[3] `<$>` is actually a function on `Functors` but all `Monads` are `Applicatives` and all `Applicatives` are `Functors`. Historically that wasn't actually true in Haskell but it was

a wart that was removed in GHC 7.10, released in March 2015.