Evaluation Order

- Most languages use **call by value** for evaluation order.
  I.e. To evaluate f(x,y), evaluate x and y first (which one first depends on the language), then plug into f's body, and then evaluate the body.
- E.g. If there is a function defined as f(x, y) = x:
  f (3+4, div(4, 2)) eval a parameter, arithmetic
  → f (7, div(4, 2)) eval the other parameter, arithmetic
  → f (7, 2) ready to plug in at last
  → 7
- However, a problematic parameter can cause an error/exception even if it would be unused:
  f (3+4, div(1, 0)) eval a parameter, arithmetic
  → f (7, div(1, 0)) eval the other parameter, arithmetic
  → Error caused by div(1,0)
- Haskell uses "**lazy evaluation**." **Lazy evaluation** is also known as **call by need**.
- Lazy evaluation in Haskell (sketch):
  - To evaluate "f x y": don't evaluate x and y first. Just plug x and y into f's right hand side (RHS) and evaluate that.
    If the RHS refers to the same parameter multiple times: same shared copy, no duplication.
  - If that runs you into pattern matching: evaluate parameter(s) just enough to decide whether it's a match or non-match. If match, plug into RHS and evaluate. If it's a non-match, try the next pattern. (If it runs out of patterns, declare "undefined" aka "error".)
  - To evaluate arithmetic operations, use call-by-value.
- E.g.

```
doITerminate = take 2 (from 0)
  where
    from n = n : from (n + 1)
```

```
*Main> doITerminate
[0,1]
```

- E.g.

```
doIEvenMakeSense = take 2 zs
  where
    zs = 0 : zs
```

```
*Main> doIEvenMakeSense
[0,0]
```

Take Function in Haskell:

- The take function takes a number and a list and returns the first n elements of the list, where n is the number inputted.
- E.g. **take 3 [a,b,c,d,e] = [a,b,c]**
- E.g. **take 3 [a,b] = [a,b]**
- The implementation goes like this:
  take 0 _ = []

take _ [] = []
take n (x:xs) = x : take (n-1) xs

Single Linked List:

- Recall that lists in Haskell are linked lists.
- Singly-linked list is a very space-consuming data structure (all languages). And if you ask for “the ith item” you're doing it wrong.
- E.g.

Newton's method with lazy lists. Like in Hughes's «why FP matters». Approximately solve $x^3 - b = 0$, i.e., cube root of b.

So $f(x) = x^3 - b$, $f(x) = 3x^2$

```
x1 = x - f(x)/f'(x)
   = x - (x^3 - b)/(3x^2)
   = x - (x - b/x^2)/3
   = (2x + b/x^2)/3
```

The local function “next” below is responsible for computing x1 from x.

```
cubeRoot b = within 0.001 (iterate next b)
    -- From the standard library:
    -- iterate f z = z : iterate f (f z)
    --               = [z, f z, f (f z), f (f (f z)), ...]
  where
    next x = (2*x + b/x^2) / 3
    within eps (x : x1 : rest)
        | abs (x - x1) <= eps = x1
        | otherwise = within eps (x1 : rest)
```

Equivalently, using the function composition operator “**.**”, we get:
cubeRoot = within 0.001 . iterate next

- With this, you really have a pipeline like Unix pipelines.
- If you use lists lazily in Haskell, it is an excellent control structure—a better for-loop than for-loops. Then list-processing functions become pipeline stages. If you do it carefully, it is even O(1)-space. If furthermore you're lucky (if the compiler can optimize your code), it can even fit entirely in registers without node allocation and GC overhead.
- Thinking in high-level pipeline stages is both more sane and more efficient—with the right languages.
- Some very notable list functions when you use lists lazily as for-loops, or when you think in terms of pipeline stages:
    - **Producers:** repeat, cycle, replicate, iterate, unfoldr, the [x..], [x..y] notation (backed by enumFrom, enumFromTo)
    - **Transducers:** map, filter, scanl, scanr, (foldr too, sometimes) take, drop, splitAt, takeWhile, dropWhile, span, break, partition, zip, zipWith, unzip
    - **Consumers:** foldr, foldl, foldl', length, sum, product, maximum, minimum, and, all, or, any
- A **producer** is some monadic action that can yield values for downstream consumption.
- A **consumer** can only await values from upstream.
- A **transducer** is like a combination of both producers and consumers.
- E.g. of iterate:

```
*Main> take 10 (iterate (\x -> x+1) 4)
[4,5,6,7,8,9,10,11,12,13]
```

When lazy evaluation hurts:

- E.g. Consider the code below:

```
mySumV2 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = g (accum + x) xs
```

It takes a number, 0, and a list of numbers and computes the sum of the numbers in the list.

```
*Main> mySumV2 [1,2,3]
6
*Main> mySumV2 [1,2,3,4,5,6]
21
*Main> mySumV2 [1..10]
55
```

Evaluation of mySumV2 [1,2,3]:
mySumV2 (1 : 2 : 3 : []) plug in
→ g 0 (1 : 2 : 3 : []) match, plug in
→ g (0 + 1) (2 : 3 : []) match, plug in
→ g ((0 + 1) + 2) (3 : []) match, plug in
→ g (((0 + 1) + 2) + 3) [] match, plug in
→ ((0 + 1) + 2) + 3 arithmetic at last
→ (1 + 2) + 3 ditto
→ 3 + 3 ditto
→ 6
This takes Ω(n) space for the postponed arithmetic.

- **Note:** If there is recursion, you bracket right to left. If there is no recursion, you bracket left to right. If you look at the example of mySumV2 [1,2,3], you'll see that it's bracketed left to right. I.e. **(((0 + 1) + 2) + 3)**
Consider the below example:
mySum [] = 0
mySum (x:xt) = x + mySum xt

mySum [1,2,3]
→ 1 + (mySum (2 : 3 : []))
→ 1 + (2 + (mySum (3 : [])))
→ 1 + (2 + (3 + (mySum ([]))))
→ 1 + (2 + (3 + (0)))
→ 1 + (2 + (3 + 0))
→ 1 + (2 + 3)
→ 1 + 5
→ 6

Notice how because there's recursion, the brackets are right heavy.