Computer Science 312

Laziness and Its Consequences

Eager Evaluation

Python:

```
def hypotenuse(a, b):
    return math.sqrt(a ** 2 + b ** 2)
```

Haskell:

```
hypotenuse a b =
    sqrt (a ^ 2 + b ^ 2)
```

Eager Evaluation

Python:

```
def hypotenuse(a, b):
    return math.sqrt(a ** 2 + b ** 2)

>>> x = 1, y = 2
>>> hypotenuse(x + 2, y + 2)
5.0
```

The arguments are evaluated ***before*** control is transferred to a function's code

Thus, **`a`** = 3 and **`b`** = 4 before the **`return`** statement is evaluated

Likewise for the arguments to **`sqrt`** and the operands of **`+`**

Lazy Evaluation

Haskell:

```
hypotenuse a b =
    sqrt (a ^ 2 + b ^ 2)

Prelude> (x, y) = (1, 2)
Prelude> hypotenuse (x + 2) (y + 2)
5.0
```

The arguments are evaluated ***after*** control is transferred to a function's code, and ***all the way down, until they are needed***

Thus, **`a`** = 1 + 2 and **`b`** = 2 + 2 ***after*** the **`sqrt`** function is called, and ***even after*** the **`+`** operator is called

Their values are computed only when **`^`** is reached!

Lazy Evaluation

Haskell:

```
hypotenuse a b =
    sqrt (a ^ 2 + b ^ 2)

Prelude> (x, y) = (1, 2)
Prelude> hypotenuse (x + 2) (y + 2)
5.0
```

Extra space must be reserved to track the expressions that are passed down from function to function

The memory reserved for each one of these is called a ***thunk***

Can be costly; Thunks a lot!

Lazy Costs: Folding from the Left

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl _ baseValue [] = baseValue
myFoldl f baseValue (x:xs) = myFoldl f (f x baseValue) xs
```

```
myFoldl (+) 0 [3, 5, 7] ->
    myFoldl (+) 3 [5, 7] ->
        myFoldl (+) 8 [7] ->
            myFoldl(+) 15 [] ->
            <- 15
        <- 15
    <- 15
```

This the trace reflects ***eager evaluation***, where `f` is applied ***before*** each recursive call (but really not the case in Haskell!)

Lazy Costs: Folding from the Left

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl _ baseValue [] = baseValue
myFoldl f baseValue (x:xs) = myFoldl f (f x baseValue) xs
```

```
myFoldl (+) 0 [3, 5, 7] ->
    myFoldl (+) (3 + 0) [5, 7] ->
        myFoldl (+) (5 + (3 + 0)) [7] ->
            myFoldl (+) (7 + (5 + (3 + 0))) [] ->
            <- 15
        <- 15
    <- 15
```

This the trace reflects ***lazy evaluation***, where the applications of `f` are ***delayed until the base case is reached***

Lazy Benefits: Infinite Structures

```
Prelude> take 2 [1,2,3,4]
[1,2]

Prelude> take 2 [1..4]
[1,2]

Prelude> take 2 [1..]      -- An infinite list!
[1,2]

Prelude> [1..]  -- Builds a list until you hit control-c
```

Lazy Benefits: Infinite Structures

```
Prelude> take 2 [1..]        -- An infinite list!
[1,2]
```

Evaluation of the list is delayed here, so items can be stripped off its head indefinitely

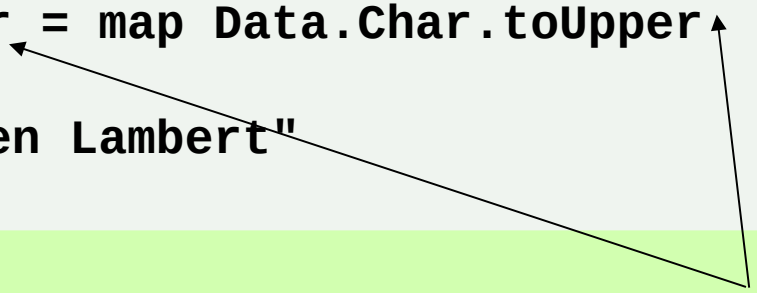
```
take :: Int -> [a] -> [a]
take 0 _ = []
take n (x:xs) = x : take (n - 1) xs
```

Laziness and Partial Functions

```
Prelude> import qualified Data.Char (toUpper)

Prelude Data.Char> Data.Char.toUpper 'a'
'A'

Prelude Data.Char> let toUpper str = map Data.Char.toUpper str

Prelude Data.Char> toUpper "Ken Lambert"
"KEN LAMBERT"
```

Laziness and Partial Functions

```
Prelude> import qualified Data.Char (toUpper)

Prelude Data.Char> Data.Char.toUpper 'a'
'A'

Prelude Data.Char> let toUpper str = map Data.Char.toUpper str

Prelude Data.Char> toUpper "Ken Lambert"
"KEN LAMBERT"

Prelude Data.Char> let toUpper = map Data.Char.toUpper

Prelude Data.Char> toUpper "Ken Lambert"
"KEN LAMBERT"
```



Builds a function that expects a string as an an arg

Omit argument

Laziness and Partial Functions

```
Prelude> let mySum listOfNums = foldr (+) 0 listOfNums

Prelude> mySum [1..4]
10

Prelude> let mySum = foldr (+) 0
```

Laziness and Partial Functions

```
Prelude> let incBy2 x = x + 2

Prelude> incBy2 3
5

Prelude> let incBy2 x = (+) x 2

Prelude> let incBy2 = (+) 2
```

All binary operators can be applied in prefix notation, using **(<operator>) <operand> <operand>**

Currying

- In Haskell Curry's theory of functions, all functions have exactly one argument
- A function of two arguments in Haskell translates to a function of one argument, that builds and returns a second function of one argument
- This process, called ***currying***, is generalized for a function of N arguments

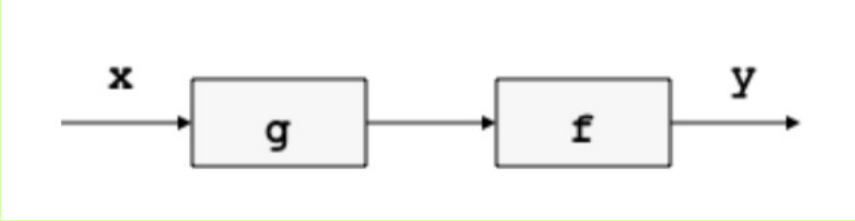
Laziness and Partial Functions

```
Prelude> import Data.Char

Prelude Data.Char> :type mapmap :: (a -> b) -> [a] -> [b]

Prelude Data.Char> :type (map toUpper)

(map toUpper) :: [Char] -> [Char]
```

Each partial application in the curry strips off a type from the type signature

Function Application

```
Prelude> let combination x = f (g x)
```



We often pipe a stream of data through several functions

The value of one function becomes the argument of another funcrion

Function Application

```
Prelude> sqrt (abs (-3))
1.7320508075688772

Prelude> let bigCombo x = f (g (h (i x)))
```

The nesting of calls to guarantee right-to left evaluation gets unwieldy

Function Application with `($)`

```
Prelude> sqrt (abs (-3))
1.7320508075688772

Prelude> let bigCombo x = f (g (h (i x)))

Prelude> sqrt $ abs $ -3
1.7320508075688772

Prelude> let bigCombo x = f $ g $ h $ i $ x
```

Use the application operator `($)` to force left-to-right evaluation

Function Composition with **(.)**

```
Prelude> let bigCombo x = f (g (h (i x)))

Prelude> let bigCombo x = f $ g $ h $ i $ x

Prelude> let bigCombo = f . g . h . i

Prelude> let squareAbs = square . abs

Prelude> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Use the composition operator **(.)** to glue functions together to work like an assembly line

For next time

Defining new data types