Functional Programming with Haskell, Part 3

Kenneth Lai

Brandeis University

September 21, 2022

Announcements

- Student hours this week
  - Thu 9/22 4-5pm, Ken, remote only
  - Fri 9/23 2:15-3:15pm, Bingyang, hybrid
- Final project presentations 12/14 6-9pm
  - Save the date!
- For next Wednesday:
  - HW1 due

Today's Plan

- Functional Programming with Haskell

Ranges

- "Lists can also be given by not enumerating all their elements but by indicating the range of elements: `[n..m]` is the list bounded below by `n` and above by `m`."

Ranges

- "Lists can also be given by not enumerating all their elements but by indicating the range of elements: `[n..m]` is the list bounded below by `n` and above by `m`."
  - Works for objects that can be enumerated (i.e., converted to and from `Int`)
    - Integers
    - Characters
    - etc.

Ranges

- "Lists can also be given by not enumerating all their elements but by indicating the range of elements: `[n..m]` is the list bounded below by `n` and above by `m`."
  - Works for objects that can be enumerated (i.e., converted to and from `Int`)
    - Integers
    - Characters
    - etc.
  - `[1..423]` is the list of all numbers from 1 to 423
  - `['g'..'s']` is the list of all characters from g to s

Infinite Lists

- `[0..]` denotes the list of all natural numbers

Infinite Lists

- `[0..]` denotes the list of all natural numbers
- "Since Haskell does not evaluate an argument unless it needs it, it can handle infinite lists as long as it has to compute only a finite amount of its elements."
  - `[0..]` does not terminate, but `take 5 [0..]` does

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Mapping

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sections

- In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its righthand side argument.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.
- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

  `[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`

Sections

- In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument.
- `(x op)` is the operation resulting from applying op to its lefthand side argument.
- `(op)` is the prefix version of the operator.
- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

```
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

- Similarly, `(>3)` denotes the property of being greater than 3, and `(3>)` the property of being smaller than 3.

Map

If *p* is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

Map

If p is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

```
map :: (a -> b) -> [a] -> [b]
```

Map

If *p* is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

Filter

A function for filtering out the elements from a list that satisfy a given property.

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

Guarded Equations

```
foo t | condition_1 = body_1
      | condition_2 = body_2
      | otherwise   = body_3
```

- If `condition_1` is true, then `foo t = body_1`
- Else if `condition_2` is true, then `foo t = body_2`
- Else, `foo t = body_3`

Guarded Equations

- Can also be written as:

```
foo t = if condition_1 then body_1
        else if condition_2 then body_2
        else body_3
```

Guarded Equations

- Can also be written as:

```
foo t = if condition_1 then body_1
        else if condition_2 then body_2
        else body_3
```

- Guards are more common, though, especially when you have multiple if conditions

List comprehension

List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Examples

```
someEvens     = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens      = [ x | x <- [1..], even x ]
```

Examples

```
someEvens     = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens      = [ x | x <- [1..], even x ]
```

Equivalently:

```
someEvens     = filter even [1..1000]

evensUntil n = filter even [1..n]

allEvens      = filter even [1..]
```

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".
- Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Function Composition

- The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.
- Standard notation for this: $f \cdot g$.
- This is pronounced as "f after g".
- Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

- Note the types!

Type Classes

- **Exercise 3.7** Check the type of the function `(\ x y -> x /= y)` in Haskell. What do you expect? What do you get? Can you explain what you get?

Type Classes

- **Exercise 3.7** Check the type of the function `(\ x y -> x /= y)` in Haskell. What do you expect? What do you get? Can you explain what you get?

- `(\ x y -> x /= y) :: Eq a => a -> a -> Bool`
  - `(\ x y -> x /= y)` has type `a -> a -> Bool`, where `a` is in type class `Eq`

Type Classes

- Type classes are collections of types that implement certain behaviors
  - Like Java interfaces, not like Java (or Python) classes

Type Classes

- Type classes are collections of types that implement certain behaviors
  - Like Java interfaces, not like Java (or Python) classes
  - `Eq` contains types that can be compared for equality (i.e., that implement `(==)` and `(/=)`)
  - `Ord` contains types that can be ordered (i.e, that implement `(<=)` and `compare`)
  - `Enum` contains types that can be enumerated
  - `Show` contains types that can be printed (i.e., that can be presented as strings)
  - etc.

Type Classes

- **Exercise 3.8** Is there a difference between `(\ x y -> x /= y)` and `(/=)`?

Type Classes

- **Exercise 3.8** Is there a difference between `(\ x y -> x /= y)` and `(/=)`?

- Eta reduction: One can convert between $\lambda x.f(x)$ and f whenever x does not appear free in f

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []      = False
elem x (y:ys) = x == y || elem x ys
```

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y:ys) = x == y || elem x ys

all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys

all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of `.` for function composition.

elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

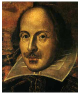
```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of . for function composition.

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Sonnet 73

```
sonnet73 =
 "That time of year thou mayst in me behold\n"
 ++ "When yellow leaves, or none, or few, do hang\n"
 ++ "Upon those boughs which shake against the cold,\n"
 ++ "Bare ruin'd choirs, where late the sweet birds sang.\n"
 ++ "In me thou seest the twilight of such day\n"
 ++ "As after sunset fadeth in the west,\n"
 ++ "Which by and by black night doth take away,\n"
 ++ "Death's second self, that seals up all in rest.\n"
 ++ "In me thou see'st the glowing of such fire\n"
 ++ "That on the ashes of his youth doth lie,\n"
 ++ "As the death-bed whereon it must expire\n"
 ++ "Consumed with that which it was nourish'd by.\n"
 ++ "This thou perceivest, which makes thy love more strong,\n
 ++ "To love that well which thou must leave ere long."
```



Counting

```
count :: Eq a => a -> [a] -> Int
count x []                  = 0
count x (y:ys) | x == y     = succ (count x ys)
               | otherwise = count x ys
```

Counting

```
count :: Eq a => a -> [a] -> Int
count x []                    = 0
count x (y:ys) | x == y       = succ (count x ys)
               | otherwise    = count x ys
```

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

Nub

nub removes duplicates, as follows:

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- ``filter (`elem` "aeiou") sonnet73``
- `count 't' sonnet73`
- `count 't' (map toLower sonnet73)`
- `count "thou" (words sonnet73)`
- `count "thou" (words (map toLower sonnet73))`

Hello World!

```
main :: IO ()
main = putStrLn "Hello World!"
```

Hello World!

```
main :: IO ()
main = putStrLn "Hello World!"
```

```
ghc --make helloworld
[1 of 1] Compiling Main ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

```
./helloworld
Hello World!
```

Hello World!

```
main :: IO ()
main = putStrLn "Hello World!"
```

```
ghc --make helloworld
[1 of 1] Compiling Main ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

```
./helloworld
Hello World!
```

- `main` is the entry point to a compiled program

Hello World!

```
main :: IO ()
main = putStrLn "Hello World!"
```

```
ghc --make helloworld
[1 of 1] Compiling Main ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

```
./helloworld
Hello World!
```

- `main` is the entry point to a compiled program
- `IO a` is the type of a function that performs an I/O action and returns an object of type a in a box
  - Printing a string doesn't really have a return value, so we return the empty tuple (i.e., unit) `()`

Computing with Boxes

- Why boxes?
  - Haskell functions are supposed to be pure
    - Do not change state
    - If you call a function twice with the same arguments, you should get the same results each time

Computing with Boxes

- Why boxes?
  - Haskell functions are supposed to be pure
    - Do not change state
    - If you call a function twice with the same arguments, you should get the same results each time
  - But I/O actions have side effects
    - Communicate with and change the state of the outside world

Computing with Boxes

- Why boxes?
  - Haskell functions are supposed to be pure
    - Do not change state
    - If you call a function twice with the same arguments, you should get the same results each time
  - But I/O actions have side effects
    - Communicate with and change the state of the outside world
  - Boxes separate the pure and impure parts of our programs

Computing with Boxes

```
f :: IO ()
f = do
    s <- getLine
    putStrLn ("Hello " ++ s ++ "!")
```

Computing with Boxes

```
f :: IO ()
f = do
    s <- getLine
    putStrLn ("Hello " ++ s ++ "!")
```

- Do syntax
  - "Glues" I/O actions together

Computing with Boxes

```
f :: IO ()
f = do
    s <- getLine
    putStrLn ("Hello " ++ s ++ "!")
```

- Do syntax
  - "Glues" I/O actions together
- `<-` (pronounced bind) gets stuff out of boxes
  - `getLine :: IO String` waits for the user to input a string, and then puts it in a box
  - We then open the box and bind the contents to `s`

String Processing

- What are the differences between the following functions?
  - `show`
  - `putStr`
  - `putStrLn`
  - `print`

String Processing

- What are the differences between the following functions?
  - `show` takes an object of type `a`, where `a` is in type class `Show`, and presents it as a string
    - Quotes its argument, by putting double quotes around it
  - `putStr` takes an object of type `String`, and prints it (without quotes)
  - `putStrLn` is like `putStr`, except it also prints a newline character
  - `print` takes an object of type `a`, where `a` is in type class `Show`, and prints it as a string
    - Equivalent to `(putStrLn . show)`
    - Expressions input to the Haskell interpreter are implicitly `printed`

File Processing

```
Prelude> :t readFile
readFile :: FilePath -> IO String
Prelude> :t writeFile
writeFile :: FilePath -> String -> IO ()
Prelude> :t appendFile
appendFile :: FilePath -> String -> IO ()
```

File Processing

```
Prelude> :t readFile
readFile :: FilePath -> IO String
Prelude> :t writeFile
writeFile :: FilePath -> String -> IO ()
Prelude> :t appendFile
appendFile :: FilePath -> String -> IO ()
```

- `readFile` takes a `FilePath` (i.e., `String`) and outputs an `IO` action that reads the file and puts its contents in a box

File Processing

```
Prelude> :t readFile
readFile :: FilePath -> IO String
Prelude> :t writeFile
writeFile :: FilePath -> String -> IO ()
Prelude> :t appendFile
appendFile :: FilePath -> String -> IO ()
```

- `readFile` takes a `FilePath` (i.e., `String`) and outputs an `IO` action that reads the file and puts its contents in a box
- `writeFile` and `appendFile` take a `FilePath` and a `String` and return an `IO` action that writes the string to the file
  - `writeFile` overwrites the file, while `appendFile` concatenates the string to the end