Higher-Order Functions

Plan for this week

Last week:

- user-defined *data types*
- manipulating data-types with *pattern matching* and *recursion*
- how to make recursive functions more efficient with *tail recursion*

The long arc of history

Pattern matching is a very old PL idea ...

- Variants of LISP from 1970 by Fred McBride
  (https://personal.cis.strath.ac.uk/conor.mcbride/FVMcB-PhD.pdf)

... but will finally be added to Python 3.10

- https://www.python.org/dev/peps/pep-0622/
def make_point_3d(pt):
    match pt:
        case (x, y):
            return Point3d(x, y, 0)
        case (x, y, z):
            return Point3d(x, y, z)
        case Point2d(x, y):
            return Point3d(x, y, 0)
        case Point3d(_, _, _):
            return pt
        case _:
            raise TypeError("not a point we support")

Plan for this week

Last week:

- user-defined data types
- manipulating data-types with pattern matching and recursion
- how to make recursive functions more efficient with tail recursion

This week:

- code reuse with higher-order functions (HOFs)
- some useful HOFs: map, filter, and fold
Recursion is good...

- Recursive code mirrors recursive data
  - Base constructor -> Base case
  - Inductive constructor -> Inductive case (with recursive call)
- But it can get kinda repetitive!

Example: evens

Let’s write a function evens:

```haskell
-- evens [] ==> []
-- evens [1,2,3,4] ==> [2,4]

evens :: [Int] -> [Int]
evens [] = ...
evens (x:xs) = ...
```

Example: four-letter words

Let’s write a function fourChars:

```haskell
```
Yikes! Most Code is the Same!

Lets rename the functions to foo:

```haskell
foo [] = []
foo (x:xs)
  | x mod 2 == 0 = x : foo xs
  | otherwise     = foo xs

foo [] = []
foo (x:xs)
  | length x == 4 = x : foo xs
  | otherwise     = foo xs
```

Only difference is condition

- x mod 2 == 0 vs length x == 4
Moral of the day

D.R.Y. Don’t Repeat Yourself!

Can we

- reuse the general pattern and
- plug-in the custom condition?

Higher-Order Functions

General Pattern

- expressed as a higher-order function
- takes plugin operations as arguments

Specific Operation

- passed in as an argument to the HOF

The “filter” pattern
The filter Pattern

General Pattern

- HOF filter
- Recursively traverse list and pick out elements that satisfy a predicate

Specific Operations

- Predicates `isEven` and `isFour`

```
evens []  = []
evens (x:xs)  | x `mod` 2 == 0 = x : evens xs | otherwise = evens xs
fourChars [] = []
fourChars (x:xs) | length x == 4 = x : fourChars xs | otherwise = fourChars xs

filter f [] = []
filter f (x:xs)
  | f x = x : filter f xs
  | otherwise = filter f xs

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

```
evens   = filter even
  where
    isEven x = x `mod` 2 == 0
fourChars = filter isFour
  where
    isFour x = length x == 4
```

Avoid duplicating code!

**QUIZ**: What is the type of `filter`?
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x = x `mod` 2 == 0

-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x = length x == 4

So what's the type of filter?

{- A -} filter :: (Int -> Bool) -> [Int] -> [Int]

{- B -} filter :: (String -> Bool) -> [String] -> [String]

{- C -} filter :: (a -> Bool) -> [a] -> [a]

{- D -} filter :: (a -> Bool) -> [a] -> [Bool]

{- E -} filter :: (a -> b) -> [a] -> [b]
Type of $\text{filter}$

-- evens $[1,2,3,4] \Rightarrow [2,4]$
evens :: [Int] -> [Int]evens xs = filter isEven xs

where
  isEven :: Int -> Bool
  isEven x = x `mod` 2 == 0

-- fourChars $["i","must","do","work"] \Rightarrow ["must","work"]$
fourChars :: [String] -> [String]fourChars xs = filter isFour xs

where
  isFour :: String -> Bool
  isFour x = length x == 4

For any type $a$

- **Input** a predicate $a \rightarrow \text{Bool}$ and collection $[a]$
- **Output** a (smaller) collection $[a]$

$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

$\text{filter}$ does not care what the list elements are

- as long as the predicate can handle them

$\text{filter}$ is **polymorphic** (generic) in the type of list elements
Example: ALL CAPS!

Lets write a function `shout`:

```
-- shout []                   ==> []
-- shout ['h','e','l','l','o'] ==> ['H','E','L','L','O']
```

```
shout :: [Char] -> [Char]
shout []   = ...
shout (x:xs) = ...
```

Example: squares

Lets write a function `squares`:

```
-- squares []       ==> []
-- squares [1,2,3,4] ==> [1,4,9,16]
```

```
squares :: [Int] -> [Int]
squares []   = ...
squares (x:xs) = ...
```

Yikes, Most Code is the Same
Lets rename the functions to foo:

```haskell
-- shout
foo [] = []
foo (x:xs) = toUpper x : foo xs

-- squares
foo [] = []
foo (x:xs) = (x * x) : foo xs
```

Lets refactor into the common pattern

```haskell
pattern = ... 
```

### The “map” pattern

The map Pattern

General Pattern

- HOF `map`
- Apply a transformation `f` to each element of a list

Specific Operations

- Transformations `toUpper` and `x -> x * x`
map f [] = []
map f (x:xs) = f x : map f xs

Let's refactor shout and squares

shout = map ...

squares = map ...

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

QUIZ
What is the type of map?

map :: (i -> o) -> [i] -> [o]

(A) (Char -> Char) -> [Char] -> [Char]
(B) (Int -> Int) -> [Int] -> [Int]
(C) (a -> a) -> [a] -> [a]
(D) (a -> b) -> [a] -> [b]