# CS 115
# Functional Programming

*Lecture 5*:

# Higher-order functions, part 2

# Today

- More Haskell basics
  - `@`-patterns, **`case`** expressions
- More higher-order functions on lists
  - **`foldr`**
  - **`foldl`**
  - **`scanl`**
- List comprehensions

*Functional Programming*

# More Haskell Basics

# @-patterns

- Sometimes, when pattern matching an argument, want to destructure the argument but also have a name for the entire argument
  - so can use both in the definition

# @-patterns

- Example problem from assignment:

```haskell
insert :: Integer -> [Integer] -> [Integer]
insert n [] = [n]
insert n m@(m1:_) | n < m1 = n : m
insert n (m1:ms) = m1 : insert n ms
```

- In second equation, need both the parts of the list (here, just the head **m1**) and the entire list **m**
- The **@**-pattern says that **m** is the name of the entire list and **m1** is the name of the head

# @-patterns

- Without **@**-pattern we would have to write:

```
insert :: Integer -> [Integer] -> [Integer]
insert n [] = [n]
insert n (m1:ms) | n < m1 = n : m1 : ms
insert n (m1:ms) = m1 : insert n ms
```

- Problem: we are re-creating a list (**m1:ms**) which already exists as **m**!

  - Inefficient, poor style

- Use **@**-patterns for more elegant code

# `case` expression

- Most often, we use pattern matching in equations that define a function

- We can also "manually" invoke pattern matching using a `case` expression

- All functions can be written using `case` expressions instead of equations

- Haskell compilers internally convert equations into `case` expressions

*Functional Programming*

# **case** expression

- Familiar example:

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n – 1)
```

- With **case** expression:

```haskell
factorial n =
  case n of
    0 -> 1
    m -> m * factorial (m – 1)
```

*Functional Programming*

# `case` expression

- When should we use **case** expressions?

- Usually not good style to use when we can use normal equational style instead (at top level)

- Need to use a **case** expression if you have to pattern-match against a value which has been computed in the course of the execution of a function

- Also useful if you need a **where** clause that spans multiple cases

# More higher-order functions

# So far

- We have seen the `map` and `filter` higher-order functions

- These functions take in and return lists

- Sometimes want to take in an entire list and return a single value that represents something interesting about the list

- We say that we want to "reduce" or "fold" the list into a single value

- Various Haskell functions exist to do this

*Functional Programming*

# Problem

- Let's write a function to add up all the numbers in a list of **Integer**s

- Writing a recursive definition is straightforward:

```haskell
sum :: [Integer] -> Integer
sum [] = 0
sum (n:ns) = n + sum ns
```

- Let's try to extract the relevant parts of this into a higher-order function on lists

- What aspects of this are likely to change?

# Generalizing `sum`

- The operation to be done on the list elements doesn't have to be **+**

- The value returned from the empty list doesn't have to be **0**

- Let's write a more generic version of **sum** that can take **+** and **0** as arguments

- We'll call it **accumulate**

# Generalizing `sum`

- First try:

```
accumulate :: (Integer -> Integer -> Integer)
 -> Integer -> [Integer] -> Integer
accumulate _ init [] = init
accumulate f init (n:ns) =
  f n (accumulate f init ns)
```

- Note that the structure of this function is identical to the `sum` function, with `sum` replaced by `accumulate f init`, `+` replaced by `f`, and `0` replaced by `init`

# Generalizing **sum** even more

- Arguments don't have to use **Integer**; let's make this polymorphic!

```
accumulate :: (a -> a -> a) -> a -> [a] -> a
accumulate _ init [] = init
accumulate f init (n:ns) =
  f n (accumulate f init ns)
```

- Now we can accumulate any data type
- But wait: why does the type of the initial value have to be the same as the type of the contents of the list?

*Functional Programming*

# Generalizing `sum` even more

- We can use (possibly) different types for initial value and list contents (more general):

```
accumulate :: (a -> b -> b) -> b -> [a] -> b
accumulate _ init [] = init
accumulate f init (n:ns) =
  f n (accumulate f init ns)
```

- This very general function has a name in Haskell: **foldr** (for "**fold r**ight")

- Let's see how to define **sum** in terms of **foldr**

# Generalizing **sum** even more

- **sum** in terms of **foldr**:

```
sum :: [Integer] -> Integer
sum lst = foldr (+) 0 lst
```

- We can also write this as follows (eta contraction, more point-free style):

```
sum :: [Integer] -> Integer
sum = foldr (+) 0      -- no lst on either side
```

*Functional Programming*

# Generalizing **sum** even more

- Let's expand this definition using the equations for **accumulate** (**foldr**)

**sum [] = foldr (+) 0 [] = 0**

**sum (n:ns) = foldr (+) 0 (n:ns)**

  **= n + (foldr (+) 0) ns**

- Substituting **sum** for **foldr (+) 0**, we get:

**sum [] = 0**

**sum (n:ns) = n + sum ns**

- We've derived our original function!

# **foldr**

- What does **foldr** actually do?
- Consider a list: **[1, 2, 3, 4, 5]**
- Can write it as: **1 : 2 : 3 : 4 : 5 : []**
- Which really means:
  - **(1 : (2 : (3 : (4 : (5 : []))))))**
- **foldr** takes two arguments besides the list:
  - a function of two arguments
  - an initial value
- How does **foldr** use these to compute its value?

*Functional Programming*

# **foldr**

- What does **foldr** actually do?

- **foldr** takes
  - an operator (**op**)
  - an initial value (**init**)
  - a list

- and returns the result of exchanging **op** for the **:** operator (used to construct the list), and **init** for the empty list

# foldr

```
foldr (+) 0 [1, 2, 3, 4, 5]
foldr (+) 0 (1 : (2 : (3 : (4 : (5 : [])))))
```

- Substitute **+** for **:** , **0** for **[]** to get:

```
1 + (2 + (3 + (4 + (5 + 0)))))
1 + (2 + (3 + (4 + 5))))
1 + (2 + (3 + 9)))
1 + (2 + 12)
1 + 14
15
```

*Functional Programming*

# **foldr**

- **foldr** is a very flexible function

- Many other Haskell functions can be defined in terms of **foldr**

- Let's see some examples

# concat

- **concat** is a function which takes a list of lists and concatenates it into a single list ("flattening" the list of lists)
- We can define **concat** recursively as follows:

```haskell
concat :: [[a]] -> [a]
concat [] = ?
concat (xs:xss) = ?
```

# concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

- In terms of **foldr**:

```
concat xss = foldr (++) [] xss
```

- or just:

```
concat = foldr (++) []
```

# concat

- Why this works:

```
concat [lst1, lst2, ...]
  = lst1 ++ lst2 ++ ... ++ []
```

# ++

- Can also define **++** (list append) in terms of **foldr**
- Let's try to derive it:

```
[1, 2, 3] ++ [4, 5, 6]
(1 : (2 : (3 : []))) ++ [4, 5, 6]
foldr (:) [4, 5, 6] (1 : (2 : (3 : [])
```

- Replace **:** with **:**, **[]** with **[4, 5, 6]** to get:

```
(1 : (2 : (3 : [4, 5, 6])))
[1, 2, 3, 4, 5, 6]
```

*Functional Programming*

# ++

- So we have this definition:

```haskell
(++) :: [a] -> [a] -> [a]
lst1 ++ lst2 = foldr (:) lst2 lst1
```

- Let's look at an even more elegant (point-free) definition of **++**

- Rewrite definition as:

```haskell
(++) lst1 lst2 = foldr (:) lst2 lst1
```

# ++

```
(++) lst1 lst2 = foldr (:) lst2 lst1
```

- We would like to define **++** as:

```
(++) = foldr (:)
```

- But this isn't correct (arguments **lst1** and **lst2** are in wrong order)

- Introduce the **flip** function:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

# ++

- This leads to this definition:

```
(++) = flip (foldr (:))
```

and **concat** can be defined as:

```
concat = foldr (flip (foldr (:))) []
```

- **foldr** is powerful!

- Warning: excessive use of point-free style can lead to impossible-to-understand code!

  - but it'll be really elegant ☺

# Tip on using `foldr`

- The function argument of `foldr` takes two arguments:
  - the current element of the list
  - the result of applying `foldr` to the rest of the list

- I sometimes write the function as `(\x r -> ...)` where `x` is the current element, `r` the rest of the list (after processing by `foldr`) to keep this straight

# map in terms of foldr

- Let's try to define **map** in terms of **foldr**
- The value we're accumulating is the mapped list
- We will end up with

```
map f lst = foldr (\x r -> ...) [] lst
```

- Just need to fill in **...**
- Assume that **r** is the rest of the list, with **f** mapped over it
  - *i.e.* **r** is **map f (tail lst)**
- Then, how to define **(\x r -> ...)**?

*Functional Programming*

# **map** in terms of **foldr**

```
map f lst = foldr (\x r -> ...) [] lst
```

- Assume that **r** is the rest of the list, with **f** mapped over it
  - *i.e.* **r** is **map f (tail lst)**
- **(\x r -> ...)** must be **(\x r -> f x : r)** to get the entire mapped list
- Definition:

```
map f lst = foldr (\x r -> f x : r) [] lst
map f = foldr (\x r -> f x : r) []    -- better
```

*Functional Programming*

# **map** in terms of **foldr**

- Definition:

```haskell
map f = foldr (\x r -> f x : r) []
```

- Even more concise (almost) point-free definition:

```haskell
map f = foldr ((:) . f) []
```

- (Work out how this works.)
- Totally pointfree definition:

```haskell
map = flip foldr [] . ((:) .)
```

- Awesome! ☺

*Functional Programming*

# `foldl`

- We've seen that **`foldr`** can be used to represent computations of this form:

**x1 op (x2 op (x3 op (x4 op init)))**

- This is natural for operators that associate to the right (like **`(:)`**)

- It would be nice to have a fold that can represent computations of this form:

**((((init op x1) op x2) op x3) op x4)**

- This is called a "left fold" or **`foldl`**

*Functional Programming*

# foldl

- **foldl** combines the **init** value with the first list element, and keeps combining with each successive list value until the end of the list is reached
- Definition:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ init [] = init
foldl f init (x:xs) = foldl f (f init x) xs
```

*Functional Programming*

# foldl

- Definition:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ init [] = init
foldl f init (x:xs) = foldl f (f init x) xs
```

- Thoughts on this?  Advantages vs **foldr**? Disadvantages?

- Consider:

```
sum = foldl (+) 0
```

# **foldl**

- Evaluate

```
sum = foldl (+) 0 [1, 2, 3]
    = foldl (+) (0 + 1) [2, 3]
    = foldl (+) ((0 + 1) + 2) [3]
    = foldl (+) (((0 + 1) + 2) + 3) []
    = (((0 + 1) + 2) + 3)
    = (1 + 2) + 3 = 3 + 3 = 6
```

*Functional Programming*

# **foldl**

- This is a *space leak*:

```
sum = foldl (+) 0 [1, 2, 3]
    = foldl (+) (0 + 1) [2, 3]
    = foldl (+) ((0 + 1) + 2) [3]
    = foldl (+) (((0 + 1) + 2) + 3) []
    = (((0 + 1) + 2) + 3)
```

- The definition doesn't look like it would yield an expression O(N) long (N = length of list), but it does

*Functional Programming*

# `foldl`

- Space leaks are very tricky to detect!

  - Special tools ("heap profilers") exist to help

- Once detected, there are ways to make functions evaluate more strictly

- In the case of `foldl`, there is a strict version called `foldl'` which doesn't cause space leaks

  - rarely a good idea to use `foldl` instead of `foldl'`

*Functional Programming*

# Space leaks

- Space leaks are an unfortunate consequence of Haskell's lazy evaluation model

- Haskell tries to be "high-level" (don't have to think about anything not related to the computation) but sometimes "low-level" concerns (like space usage) unavoidably come up

# Scans

- In addition to folds, Haskell also has "scan" functions
- These are like folds except that they save a list of all the partial evaluations in a fold
- Easiest to show by example

# scanl

- The most intuitive one is `scanl` ("scan left")
- Example:

```
ghci> scanl (+) 0 [1,2,3,4,5]
[0,1,3,6,10,15]
```

- See how the partial sums are collected in a list
  - `[0, 0+1, (0+1)+2, ((0+1)+2)+3, (((0+1)+2)+3)+4, ((((0+1)+2)+3)+4)+5]`
- Last element in list is the full sum
- There is also a `scanl'` if you want to avoid space leaks (recommended)

# scanr

- Much less used is **scanr** ("scan right")
- Example:

```
ghci> scanr (+) 0 [1,2,3,4,5]
[15, 14, 12, 9, 5, 0]
```

- This is equal to:
  - `[1+(2+(3+(4+(5+0)))), 2+(3+(4+(5+0))), 3+(4+(5+0)), 4+(5+0), 5+0, 0]`

# Summing up

- It's remarkable how many functions can be written using just simple higher-order functions like `map`, `filter`, `foldl`/`foldr`, `scanl`/`scanr`

- If necessary, it's easy to write new HOFs
  - but it's fun to see how far you can go with just the pre-defined ones!

# List comprehensions

# List comprehensions

- Often the case that you want to build a list with certain properties

- Elements of list are the result of evaluating an expression for certain values of the variables in the expression

- May want to impose some other criteria on the list elements as well

- In Haskell, we can do this with a list comprehension

# List comprehensions

- A list comprehension is a description of
  - the elements of a list (expression)
  - where the variables come from (generators)
  - what other criteria must be met (filters)
- Syntax:

`[<expression> | <generators>, <filters>]`

- Generators have the form

`x <- [1..1000]`

(`x` is taken elementwise from the list `[1..1000]`)

*Functional Programming*

# List comprehensions

- Simple list comprehension:

```
Prelude> [x * 2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

- Two generators:

```
Prelude> [(x, y) | x <- [1..3], y <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),
(3,2),(3,3)]
```

- Note: rightmost generator "changes fastest"

# Filters

- To filter out elements from a list comprehension, add a boolean expression (which must evaluate to **True** for the generator values to be accepted)

```
Prelude> [(x, y) | x <- [1..6], y <- [1..6],
x + y == 7]
[(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

- Can have multiple filters, separated by commas
- Can interleave generators and filters
  - but filters must not refer to generators that follow them

# Patterns in generators

- Generators can bind patterns

```
Prelude> [x + y | (x, y) <- [(1,2),(3,4),(5,6)]]
[3,7,11]
```

- Here, **(1, 2)** unpacked into **(x, y)**, binding **1** to **x** and **2** to **y** in the expression **x + y**

*Functional Programming*

# **map, filter**

- List comprehensions can take the place of **map**:

```
map f [x1, x2, x3, ...]
== [f x | x <- [x1, x2, x3, ...]]
```

- List comprehensions can take the place of **filter**:

```
filter p [x1, x2, x3, ...]
== [x | x <- [x1, x2, x3, ...], p x]
```

*Functional Programming*

# Examples

- List comprehensions can be used to write very concise definitions

```haskell
quicksort :: [Integer] -> [Integer]
quicksort [] = []
quicksort (x:xs) =
  quicksort lt ++ [x] ++ quicksort ge
  where
    lt = [y | y <- xs, y < x]
    ge = [y | y <- xs, y >= x]
```

# Next time

- Defining new data types (algebraic data types)
- Coming up: Type classes

*Functional Programming*