Today

- More Recursion
- Pattern Matching

Assignments

- HW8 due
- HW9 out
- Quiz 7 out (due Tues before class)
Exponentiation (Power)

Q: Using recursion, define a function \( \text{pow} \ x \ y \) that computes \( x^y \) for \( y \geq 0 \)

\[
\text{pow} \ x \ y = \begin{cases} 
1 & \text{if } y = 0 \\
 x \cdot \text{pow} \ x \ (y-1) & \text{otherwise}
\end{cases}
\]

Q: What is the type of \( \text{pow} \)?
- Hint 1: \((*) :: (\text{Num} \ a) \Rightarrow a \rightarrow a \rightarrow a\)
- \(\text{pow} :: (\text{Num} \ a, \text{Num} \ b, \text{Eq} \ b) \Rightarrow a \rightarrow b \rightarrow a\)

Append

Q: Using recursion, define a function \( \text{concat} \ xs \ ys \) that computes \( xs ++ ys \)

\[
\text{concat} \ xs \ ys = \begin{cases} 
ys & \text{if null } xs \\
 (\text{head } xs) : \text{concat} \ (\text{tail } xs) \ ys & \text{otherwise}
\end{cases}
\]

The basic idea:
- appending an empty list to \( ys \) is just \( ys \) (base case)
- otherwise, create the new list by:
  - adding head of \( xs \) to the result of a smaller append
  - which simply appends \( ys \) to the tail of \( xs \)
How it works: w/out laziness

\[\text{append } [1,2] [3,4] \]
\[\Rightarrow 1 : \text{append } [2] [3,4] \]
\[\Rightarrow 1 : 2 : \text{append } [] [3,4] \]
\[\Rightarrow 1 : 2 : [3,4] \]
\[\Rightarrow [1,2,3,4] \]

Another example: w/ laziness

\[\text{append } [1,2] [3..] \]
\[\Rightarrow (\text{head } [1,2]) : \text{append } (\text{tail } [1,2]) [3..] \]
\[\Rightarrow (\text{head } [1,2]) : (\text{head } [2]) : \text{append } (\text{tail } [2]) [3..] \]
\[\Rightarrow (\text{head } [1,2]) : (\text{head } [2]) : [3..] \]

- second list never has to (is never asked to) be evaluated
- result is called a “thunk” (“suspension”, “delayed computation”, “future”)

Why does \text{append} have the type \[ [a] \rightarrow [a] \rightarrow [a] \]?

- \text{null} :: [a] \rightarrow \text{Bool} \hspace{1cm} \ldots \text{xs} :: [a]
- \text{(:) :: b} \rightarrow [\text{b}] \rightarrow [\text{b}] \hspace{1cm} \ldots \text{result must be of type} [\text{b}]
- \text{then} \text{ys} \hspace{1cm} \ldots \text{ys} \text{must of type} [\text{b}]
- \text{head} \text{xs} :: \text{a} \hspace{1cm} \ldots \text{so} \text{a} = \text{b}
Type Inference and the “Occurs Check” Error

Q: How can we define a `flatten` function in Haskell?

e.g., `flatten [[1,2],[3,4]]` should return `[1,2,3,4]`

```haskell
flatten xs = if null xs
    then xs -- this doesn't work!
    else head xs ++ flatten (tail xs)
```

Q: What is the type of `flatten`?

1. `xs` must be a list since we call `null xs` so: `?` -> `?`
2. `head xs` is a list since we use `++` on it so: `[[a]]` -> `?`
3. result must have same type as `head xs` so: `[[a]]` -> `[a]`
4. but result must be same type as `xs` (from `then xs`) so: `[a]` ≡ `[[a]]`
5. repeating this out gives `[[[⋯ a ⋯ ]]]` *infinite type!*

Q: How can we fix this?

```haskell
flatten :: [[a]] -> [a]
flatten xs = if null xs
    then [] -- this works!
    else head xs ++ flatten (tail xs)
```

Q: Why does this work?

• ... we broke the connection between the input type and the output type
Pattern Matching

Functions are defined as a **series of equations**

- Each equation has a different “**pattern**” of input

  ```haskell
  -- simple (but verbose) myNot definition
  myNot x = if x == True then False else True
  ```

- In this case, \( x \) has two (value) patterns: **True** and **False**

  ```haskell
  -- myNot definition w/out if-then-else
  myNot True = False
  myNot False = True
  ```

- Here we are defining the function using “**pattern matching**”

How does this work?

Say we call:

```haskell
myNot False
```

The Haskell runtime:

- Checks the value supplied (**False**) against the first pattern
- In this case, it isn’t a match (**False \neq True**)  
- The second pattern is checked, which succeeds
- The right-hand side of the second equation is returned

Haskell tries patterns in order ... **and stops at first match**
A more involved example with lists ...

\[
\text{mix1 } xs \text{ ys } = \begin{cases} 
\text{null } xs \text{ || null } ys & \text{then } xs ++ ys \\
\text{else head } xs \text{ : head } ys : \text{mix1 } (\text{tail } xs) \text{ (tail } ys)
\end{cases}
\]

Q: What do the following return?

\[
\begin{align*}
\text{mix1 } \[] \[] &= \[] \\
\text{mix1 } [1,3,5] \[] &= [1,3,5] \\
\text{mix1 } \[] [2,4,6] &= [2,4,6] \\
\text{mix1 } [1,3,5] [2,4,6] &= [1,2,3,4,5,6]
\end{align*}
\]

Q: What are the patterns?

- \text{xs empty ... return ys}
- \text{ys empty ... return xs}
- \text{neither empty ... return else expression}

The \text{mix} function defined using patterns

\[
\begin{align*}
\text{mix2 } \[] \text{ ys } &= \text{ys} \\
\text{mix2 } xs \[] &= xs \\
\text{mix2 } xs \text{ ys } &= \text{head } xs : \text{head } ys : \text{mix2 } (\text{tail } xs) \text{ (tail } ys)
\end{align*}
\]

Q: Are these patterns “exhaustive”?

- Yes!
  - e.g., calling \text{mix2 } \[] \[] matches the first case
Even fancier patterns ...

\[
\begin{align*}
    \text{mix3 } []\ & y s = y s \\
    \text{mix3 } x s \ & [] = x s \\
    \text{mix3 } (x:x s) \ (y:y s) = x : y : \text{mix3 } x s \ y s \\
\end{align*}
\]

• We are “deconstructing” the values in the pattern
• Note the parens around \((x:x s)\) are required
• Using “:\)” is like calling head and tail on the left-hand side ...

Example evaluation of mix3

\[
\begin{align*}
    \text{mix3 } [1,3] \ [2,4] \\
    \implies \text{mix3 } (1:[3]) \ (2:[4]) & \quad \text{matches 3rd pattern} \\
    \implies 1 : 2 : \text{mix3 } [3] \ [4] \\
    \implies 1 : 2 : \text{mix3 } (3:[]) \ (4:[]) & \quad \text{matches 3rd pattern} \\
    \implies 1 : 2 : 3 : 4 : \text{mix3 } [] \ [] \\
    \implies 1 : 2 : 3 : 4 : [] & \quad \text{matches 1st pattern}
\end{align*}
\]
Another (simpler) example ...

\[ f \; xs = \text{head} \; xs \]

is the same as

\[ f \; (x:xs) = x \]

And:

\[ g \; xs = \text{tail} \; xs \]

is the same as

\[ g \; (x:xs) = xs \]
Wildcards

Use the "wildcard" symbol (_,) for "don't care"

For example:

\[
\begin{align*}
\text{f} (x:_) &= x \quad \text{-- don't care about tail of the list} \\
\text{g} (_:xs) &= xs \quad \text{-- don't care about head of the list} \\
\text{fst} (x, _) &= x \quad \text{-- don't care about second elem of pair} \\
\text{snd} (_, y) &= y \quad \text{-- don't care about first elem of pair}
\end{align*}
\]

• _ stands for any value
• corresponding value cannot be accessed on RHS
• helps readability ... focuses attention on the important stuff

Use wildcards in your homework!!!
Another Example: the init and last functions

Prelude> :type last
last :: [a] -> a

Prelude> last [1,2,3]
3

Prelude> :type init
init :: [a] -> [a]

Prelude> init [1,2,3]
[1,2]

Q: How can we define these using pattern matching?

last [] = error "empty list" -- undefined for []
last [x] = x -- one element list
last (_:xs) = last xs -- note pattern order

init [] = error "empty list"
init [] = [] -- one element list
init (x:xs) = x : init xs -- build up list

Order of patterns matters

• e.g., [x] and (x:xs) both match a one-element list (e.g., [1])
  – the patterns “overlap” (really, (x:xs) subsumes [x])

• if you put (x:xs) pattern first, [x] will never be reached
Exercise: Firsts

Q: Use recursion to define a `firsts` function that takes a list of pairs `ps` and returns a list containing the first element of each pair. Give the type of `firsts`.

```hs
firsts :: [(a,b)] -> [a]
firsts [] = []
firsts ((x,_) : ps) = x : firsts ps
```

Q: Can you define this function using `map`? How?

```hs
firsts ps = map fst ps
```

Exercise: Define `take` using recursion w/ patterns

```hs
take _ [] = []
take n (x:xs) = if n > 0
    then x : take (n-1) xs
    else []
```

Exercise: Define `drop` using recursion w/ patterns

```hs
drop _ [] = []
drop n (x:xs) = if n <= 0
    then x : xs
    else drop (n-1) xs
```
Guards

Patterns specify “structural” conditions for matching

- Matching on the parts of a data structure

Guards allow us to define **conditions** for a pattern

```haskell
-- previous myDrop function with just patterns
drop _ [] = []
drop n (x:xs) = if n <= 0
               then x : xs
               else drop (n-1) xs
```

- We can rewrite this using guards to remove the if-then-else:

```haskell
drop _ [] = []
drop n xs | n <= 0 = xs
drop n (_:xs) = drop (n-1) xs
```

- The guard gives a condition for applying the pattern
There can be *multiple* guards per pattern

```haskell
letterGrade p
    | p >= 90   = "A"
    | p >= 80   = "B"
    | p >= 70   = "C"
    | p >= 60   = "D"
    | otherwise = "F"
```

- Each guard is an expression of type `Bool`
- `otherwise` is a special variable bound to `True`

How a guard works

- For each pattern, check if first guard succeeds
- If so, RHS is result
- Otherwise, check next guard
- If no guards succeed, go to the next pattern

When calling a function, if no patterns match ...

- Haskell gives a runtime exception (non-exhaustive pattern)
Another (contrived) example

Q: What does this function do?

```haskell
pairs [] = []
pairs [_] = []
pairs (x:y:zs)
  | x == y = (x,y) : pairs (y:zs)
  | otherwise = pairs (y:zs)
```

- Note: patterns and guards can be mixed (as above)
- Also: names in patterns can only appear once!
  - e.g., `x:x:zs` wouldn’t work in last pattern

Q: What is the result of `pairs [1,2,2,2,3]`?

```
[(2,2),(2,2)]
```

Q: What is the type of `pairs`?

```
pairs :: (Eq a) => [a] -> [(a, a)]
```

Note on where with guards vs. let

```haskell
f x
  | g1 = e1
  | g2 = e2
  where ... 

f x
  | g1 = let ... in e1
  | g2 = let ... in e2
```