Today

- Quiz 4
- Pattern matching
- Guards

Assignments

- Project status updates due
- HW-5, R-5 out
Pattern Matching

Functions are defined as a series of equations

- Each equation has a different “pattern” of input
  
  -- simple myNot definition
  myNot x = if x == True then False else True

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

  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 ... stops at first match
A more involved example with lists ...

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

Q: What do the following return?

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

Q: What are the patterns?

- xs empty ... return ys
- ys empty ... return xs
- neither empty ... return else expression

The \textit{mix} function defined using patterns

\[
mix2 \; [] \; ys = ys \\
mix2 \; xs \; [] = xs \\
mix2 \; xs \; ys = head \; xs : head \; ys : mix2 \; (\text{tail} \; xs) \; (\text{tail} \; ys)
\]

Q: Are these patterns “exhaustive”?

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

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

- We are “deconstructing” the values in the pattern
- Note the parens around \((x:xs)\) are required
- Using “:" is like calling \texttt{head} and \texttt{tail} on the left-hand side ...
- This is much more succinct and way cooler! ... use in your homework

Example evaluation of \texttt{mix3}

\[
\begin{align*}
\text{mix3 } &[1,3] [2,4] \\
\Rightarrow &\text{mix3 } (1: [3]) (2: [4]) \\
\Rightarrow &1 : 2 : \text{mix3 } [3] [4] \\
\Rightarrow &1 : 2 : \text{mix3 } (3: []) (4: []) \\
\Rightarrow &1 : 2 : 3 : 4 : \text{mix3 } [] [] \\
\Rightarrow &1 : 2 : 3 : 4 : []
\end{align*}
\]

matches 3rd pattern

matches 3rd pattern

matches 1st pattern
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:

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

• \_\_ stands for any value

• corresponding value cannot be accessed on RHS

• helps readability ... focuses attention on the important stuff

• again, use wildcards in your homework!
Another Example: the \texttt{init} and \texttt{last} functions

\begin{verbatim}
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]
\end{verbatim}

Q: How can we define these using pattern matching?

\begin{verbatim}
last [] = error "empty list"  -- more on error later
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
\end{verbatim}

Order of patterns matters

- e.g., \texttt{[x]} and \texttt{(x:xs)} both match a one-element list (e.g., \texttt{[1]})
  - the patterns “overlap” (really, \texttt{(x:xs)} subsumes \texttt{[x]})
- if you put \texttt{(x:xs)} pattern first, \texttt{[x]} will never be reached
Exercise: Firsts

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

\[
\text{firsts} :: [(a,b)] \rightarrow [a] \\
\text{firsts} \ ([] ) = [] \\
\text{firsts} \ ((x,\_):ps) = x : \text{firsts} \ ps
\]

Exercise: myTake

\[
\text{myTake} \ _ \ [] = [] \\
\text{myTake} \ n \ (x:xs) = \text{if} \ n > 0 \ \text{then} \ x : \text{myTake} \ (n-1) \ xs \\
\text{else} \ []
\]

Exercise: myDrop

\[
\text{myDrop} \ _ \ [] = [] \\
\text{myDrop} \ n \ (x:xs) = \text{if} \ n \leq 0 \ \text{then} \ x:xs \\
\text{else} \ \text{myDrop} \ (n-1) \ xs
\]

* We’ll see how to get rid of these if-then-else expressions soon ...
Exercise: myCycle

The Haskell cycle function takes a list and repeats it infinitely many times:

```
Prelude> take 5 (cycle [1,2])
[1,2,1,2,1]

Prelude> take 9 (cycle "abc")
"abcabcabc"

Prelude> cycle []
*** Exception: Prelude.cycle: empty list

Prelude> :t cycle
cycle :: [a] -> [a]
```

Define cycle (as myCycle):

```
myCycle [] = error "empty list"
myCycle xs = xs ++ myCycle xs
```

Q: How does this work?

```
head (take 5 (cycle [1,2]))
  ==> head (head (cycle [1,2]) : take 4 (tail (cycle [1,2])))
  ==> head (cycle [1,2])
  ==> head ([1,2] ++ cycle xs)
  ==> head (1 : tail [1,2] ++ cycle [1,2])  -- recall append!
  ==> 1
```
Guards

Patterns specify “coarse” conditions for matching

- Matching on parts of a structure

Guards allow us to define conditions for a pattern

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

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

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

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

\[
\text{letterGrade } p \begin{cases} 
  p \geq 90 & = "A" \\
  p \geq 80 & = "B" \\
  p \geq 70 & = "C" \\
  p \geq 60 & = "D" \\
  \text{otherwise} & = "F"
\end{cases}
\]

- Each guard is an expression of type \text{Bool}
- \text{otherwise} is a special variable bound to \text{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 ...
```

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