Lecture 38:

- More higher-order functions
- Pattern matching basics

Announcements:

- HW-8 out
The map and fold higher order functions

The map function applies a function to each element of a list

Prelude> map even [1,2,3,4]
[False,True,False,True]

Prelude> map (+1) [1,2,3,4]
[2,3,4,5]

Defining map (without pattern matching):

map f xs =
  if null xs then []
  else f (head xs) : map f (tail xs)

Q: What is the type of map?
The fold function combines (accumulates) values

\textbf{foldl} accumulates from the left: \(((a \oplus x_1) \oplus x_2) \oplus \cdots\)

\textbf{foldl} in functional form: \((f (f (f a x_1) x_2) x_3) x_4)\) ... for 4 elements

\textbf{foldr} accumulates from the right: \((x_1 \oplus (x_2 \oplus \cdots (x_n \oplus a)))\)

\textbf{foldr} in functional form: \((f x_1 (f x_2 (f x_3 (f x_4 a))))\) ... for 4 elements

Examples:

Prelude> foldl (+) 0 [1,2,3,4]
10

Prelude> foldl (+) 0 []
0

Prelude> foldr (-) 0 [10, 4, 3, 1]
8

Prelude> foldl (-) 0 [10, 4, 3, 1]
-18

Prelude> foldl min 10 [4, 3, 1, 5]
1

Prelude> foldl (+) 0 (map (\x -> 1) [4, 3, 1, 5])
4

Note: Here we are using a lambda function!
Defining \texttt{foldl} and \texttt{foldr} (without pattern matching)

\[
\begin{aligned}
  \texttt{foldl} \ f \ a \ xs &= \\
  &\quad \text{if \ null \ xs \ then \ a} \\
  &\quad \text{else} \ \texttt{foldl} \ f \ (f \ a \ (\text{head} \ xs)) \ (\text{tail} \ xs)
\end{aligned}
\]

\[
\begin{aligned}
  \texttt{foldr} \ f \ a \ xs &= \\
  &\quad \text{if \ null \ xs \ then \ a} \\
  &\quad \text{else} \ f \ (\text{head} \ xs) \ (\texttt{foldr} \ f \ a \ (\text{tail} \ xs))
\end{aligned}
\]

Q: What is the type of \texttt{foldl} (\texttt{foldr})?
Type Inference and the “Occurs Check” Error

Defining a list flatten function in Haskell

- unnests the nested lists
- e.g., flatten $[[1,2],[3,4]]$ should return $[1,2,3,4]$

```
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: $[?] \to ?$
2. head xs is a list since we use ++ on it so: $[[a]] \to ?$
3. result must have same type as head xs so: $[[a]] \equiv [a]$
4. but result must be same type as xs (from then xs) so: $[a] \equiv [[a]]$
5. repeating this out gives $[[[[ \cdots a \cdots ]]]]$ infinite type!

Q: How can we fix this?

```
flatten :: [[a]] \to [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

  -- 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

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

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

Q: What do the following return?

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

Q: What are the patterns?

– \(xs\) empty ... return \(ys\)
– \(ys\) empty ... return \(xs\)
– neither empty ... return else expression

The \text{mix} function defined using patterns

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

Q: Are these patterns “exhaustive”?

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

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

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

**Example evaluation of mix3**

\[
\begin{align*}
mix3 \; [1,3] \; [2,4] \\
&\Rightarrow mix3 \; (1:[3]) \; (2:[4]) \quad \text{matches 3rd pattern} \\
&\Rightarrow 1 : 2 : mix3 \; (3:) \; (4:) \quad \text{matches 3rd pattern} \\
&\Rightarrow 1 : 2 : 3 : 4 : mix3 \; [] \; [] \\
&\Rightarrow 1 : 2 : 3 : 4 : [] \quad \text{matches 1st pattern}
\end{align*}
\]
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:

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

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

Use wildcards in your homework!!!
More examples

The \texttt{firsts} function that takes a list of pairs \texttt{ps} and returns a list containing the first element of each pair.

\begin{verbatim}
firsts :: [(a,b)] -> [a] 
firsts [] = [] 
firsts ((x,_) : ps) = x : firsts ps
\end{verbatim}

Q: Can you define this function using \texttt{map}? How?

\begin{verbatim}
firsts ps = map fst ps
\end{verbatim}

\textbf{Defining} \texttt{take} \textit{using recursion w/ patterns}

\begin{verbatim}
take _ [] = [] 
take n (x:xs) = if n > 0 
    then x : take (n-1) xs 
    else []
\end{verbatim}

\textbf{Defining} \texttt{drop} \textit{using recursion w/ patterns}

\begin{verbatim}
drop _ [] = [] 
drop n (x:xs) = if n <= 0 
    then x : xs 
    else drop (n-1) xs
\end{verbatim}
**Guards**

Patterns specify **structural** conditions for matching

- Matching on the parts of a data structure
- Note: can’t check equality (or inequality) of the parts (e.g., $x:x:xs$)

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

- Checking that the parts of a data structure satisfy Boolean conditions

```haskell
-- drop with just patterns (no guards)
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 -- if n <= 0 then xs
drop n (_:xs) = drop (n-1) xs -- otherwise
```

- The guard gives a Boolean 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 just a binding 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?

\[
\begin{align*}
pairs [] &= [] \\
pairs [_] &= [] \\
pairs (x:y:zs) &= \begin{cases} 
  (x,y) : \text{pairs (y:zs)} & \text{if } x == y \\
  \text{pairs (y:zs)} & \text{otherwise}
\end{cases}
\end{align*}
\]

- Note: patterns and guards can be mixed (as above)
- The otherwise case here is the “default” \( x /= y \) case

Q: What is the result of \( \text{pairs [1,2,2,2,3]} \)?

\[ [(2,2),(2,2)] \]

Q: What is the type of \( \text{pairs} \)?

\[ \text{pairs :: (Eq a) => [a] -> [(a, a)]} \]

Guards add another layer to checking if cases are exhaustive

Note on where with guards vs. let ... i.e., where “spans” guards for pattern

\[
\begin{align*}
f x &= \begin{cases} 
  g_1 = e_1 \\
  g_2 = e_2 \\
  \text{where ...}
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
f x &= \begin{cases} 
  g_1 = \text{let ... in } e_1 \\
  g_2 = \text{let ... in } e_2
\end{cases}
\end{align*}
\]