Today

• Pattern Matching with Guards
• User-defined types

Assignments

• R-9, HW-9 out

Announcements

• Quiz next Tuesday
• Exam 2 on Thursday
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

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

Exercise: myDrop

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

* We'll see how to get rid of these if-then-else expressions soon ...
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

```
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?

\[
\text{pairs [] } = [] \\
\text{pairs [ ] } = [] \\
\text{pairs (x:y:zs)} \\
\text{\quad | x == y } = (x,y) : \text{pairs (y:zs)} \\
\text{\quad | otherwise } = \text{pairs (y:zs)}
\]

• 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 \(\text{pairs [1,2,2,2,3]}\)?

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

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

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

Note on where with guards vs. let

\[
f x \\
\text{\quad | g1 = e1} \\
\text{\quad | g2 = e2} \\
\text{\quad where ...}
\]

\[
f x \\
\text{\quad | g1 = let ... in e1} \\
\text{\quad | g2 = let ... in e2}
\]
Haskell User-Defined Data Types

We can define new data types in Haskell

- New data types are defined using the data keyword
- For example, a simple book “record” of book ids, titles, and authors

```
data BookInfo = Book Integer String [String]
deriving (Show)
```

The definition

- BookInfo is a type constructor ... types are always capitalized
- Book is a value (data) constructor ... also capitalized
- everything after Book and up to deriving are fields
  - each field here is given as an existing type
- deriving says BookInfo is a member of the Show typeclass
  - Haskell takes care of the Show implementation here
  - Will also take care of Eq implementation (if given)
Once defined, we can use our new type ...

Prelude> :load books

– Our new type is defined in books.hs

Main*> Book 35 "Neuromancer" ["Gibson"]
Book 35 "Neuromancer" ["Gibson"]

– We use the value constructor (Book) to create a value
– Here we see Show at work ... the value is printable!

Main*> let b1 = Book 35 "Neuromancer" ["Gibson"]
Main*> :type b1
b1 :: BookInfo

– Our Book value is of type BookInfo

Main*> :type Book
Book :: Integer -> String -> [String] -> BookInfo

– a value constructor is just another function!
– that happens to create a value of the corresponding type