Today

- User-Defined Types (cont)

Assignments

- HW9 out (due Tues)
- Exam 3 out (due by class next Thurs)
Pattern matching with algebraic data types

Can use data constructors and fields with pattern matching

- values must be enclosed in parentheses

- Simple example

  bookID (Book id title authors) = id
  bookTitle (Book id title authors) = title
  bookAuthors (Book id title authors) = authors

- Can simplify with wildcards

  bookID (Book id _) = id
  bookTitle (Book _ title _) = title
  bookAuthors (Book _ title _) = authors
User defined parametric types

Q: What is a parametric type?
— A type containing a type parameter (e.g., [a])

The Haskell \texttt{Maybe} type ...

\begin{verbatim}
data Maybe a = Just a | Nothing
\end{verbatim}

- Here \texttt{a} is a type variable (... like a “box” around a values)
- \texttt{Maybe} used to represent values that are optional

\begin{verbatim}
Prelude> :type Just
Just :: a -> Maybe a

Prelude> :type Nothing
Nothing :: Maybe a
\end{verbatim}

- Creating \texttt{Maybe} values

\begin{verbatim}
Prelude> let m1 = Just True

Prelude> m1
Just True

Prelude> :type m1
m1 :: Maybe Bool

Prelude> let m2 = Just "something"

Prelude> m2
\end{verbatim}
Just "something"

Prelude> :type m2
m2 :: Maybe [Char]

• A simple (unrealistic) use of the Maybe type

    myDiv x 0 = Nothing
    myDiv x y = Just (x/y)

*Main> :type myDiv
(Eq a, Fractional a) => a -> a -> Maybe a

*Main> myDiv 1 0
Nothing

*Main> myDiv 1 1
Just 1.0

• Q: How would we define a maybeHead and maybeTail function?

    maybeHead :: [a] -> Maybe a
    maybeHead [] = Nothing
    maybeHead xs = Just (head xs)

    maybeTail :: [a] -> Maybe [a]
    maybeTail [] = Nothing
    maybeTail xs = Just (tail xs)
More on Parameterized Types

A linked list can be defined using a (recursive) parameterized type:

```haskell
data List a = Node a (List a)  
             | Nil  
             deriving (Show, Eq)
```

- A node value consists of an `a`-value followed by an `a`-list value
- `Nil` is a list “terminator” value
- Uses default implementations of `show` and `(==)`

**Exercise**: Create a 3-element list of strings and a 4-element list of ints

```haskell
list1 = Node "foo" (Node "bar" (Node "baz" Nil))
list2 = Node 1 (Node 2 (Node 3 (Node 4 Nil)))
```

**Q**: What are the types of the two lists?

```haskell
:type list1
list1 :: List [Char]
```

```haskell
:type list2
list2 :: Num a => List a
```

```haskell
:type Node 1 (Node 2 Nil)
Node 1 (Node 2 Nil) :: Num a => List a
```
We can use pattern matching to define List functions:

```haskell
isEmpty :: List a -> Bool
isEmpty Nil = True
isEmpty _ = False
```

**Exercise:** Write a function to return the length of a List

```haskell
listLength :: List a -> Int
listLength Nil = 0
listLength (Node _ tail) = 1 + listLength tail
```