Today

- Quiz 5
- Parametric User-Defined Types

Assignments

- HW-6, R-6 out
User defined parametric types

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

The Haskell *Maybe* type ...

```haskell
data Maybe a = Just a
             | Nothing
```

• Here `a` is a type variable (... like a “box” around a values)

• Maybe used to represent values that are optional

```haskell
Prelude> :type Just
Just :: a -> Maybe a

Prelude> :type Nothing
Nothing :: Maybe a
```

• Creating Maybe values

```haskell
Prelude> let m1 = Just True
Prelude> m1
Just True

Prelude> :type m1
m1 :: Maybe Bool

Prelude> let m2 = Just "something"
```
Prelude> m2
Just "something"

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

• A simple (unrealistic) use of the Maybe type

```
myDiv x y
  | y == 0    = Nothing
  | otherwise = Just (x / y)
```

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

*Main> myDiv 1 0
Nothing

*Main> myDiv 1 1
Just 1.0
**Maybe versus Error**

**The standard “error” function**

\[
\text{error} :: \text{String} \rightarrow a
\]

- Given a string produces a value of any type \(a\)

**Why does error return any type?**

- Always returns a value of the “correct” type
- So can be called from anywhere, without causing a type conflict
- \(\text{error}\) never returns though ...
  - aborts execution (exception) without returning an actual value

**Using error to return the second element of a list**

\[
\text{sndL} :: [a] \rightarrow a
\]

\[
\text{sndL} \ (\_ : x : \_) = x
\]

\[
\text{sndL} \ _ = \text{error} \ "\text{List too short!}"
\]

```
*Main> sndL []
*** Exception: List too short!

*Main> sndL [1]
*** Exception: List too short!

*Main> sndL [1,2]
2
```
• But in this one ...

  *Main> head (sndL [1])
  *** Exception: List too short!

• the error exits immediately

• and we may want to “recover” (e.g., in the outer function)

Using **Maybe** instead of **error**

• an **error** is represented by **Nothing**

• a **success** is represented by **Just**

  \[
  \text{sndL’ :: } [a] \rightarrow \text{Maybe } a
  \]

  \[
  \text{sndL’ } (_:x:_:_) = \text{Just } x
  \]

  \[
  \text{sndL’ } _ = \text{Nothing}
  \]

  *Main> sndL’ []
  Nothing

  *Main> sndL’ [1]
  Nothing

  *Main> sndL’ [1, 2]
  Just 2
More on Parameterized Types

Exercise 1: Define a linked list structure using a parameterized type

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

- This is a “recursive” data structure
- Can use with pattern matching
- Uses default implementations of `show` and `(==)

Exercise 2: 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]

:type list2
list2 :: List Integer

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

\[
\text{isEmpty} :: \text{List a} \rightarrow \text{Bool} \\
\text{isEmpty} \text{ Nil} = \text{True} \\
\text{isEmpty} \ _ = \text{False}
\]

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

\[
\text{listLength} :: \text{List a} \rightarrow \text{Int} \\
\text{listLength} \text{ Nil} = 0 \\
\text{listLength} \ (\text{Node} \ _ \ \text{tail}) = 1 + \text{listLength} \ \text{tail}
\]

**Exercise:** Write a function to return the last element of a List

\[
\text{removeLast} :: \text{List a} \rightarrow \text{List a} \\
\text{removeLast} \text{ Nil} = \text{error "empty list"} \\
\text{removeLast} \ (\text{Node} \ _ \ \text{Nil}) = \text{Nil} \\
\text{removeLast} \ (\text{Node} \ x \ \text{tail}) = \text{Node} \ x \ (\text{removeLast} \ \text{tail})
\]
Binary Trees

Exercise 1: Define a binary tree structured using a parameterized type

    data Tree a = Node a (Tree a) (Tree a)
        | Nil
    deriving (Show, Eq)

- Again, this is a recursive data structure

Exercise 2: Create a 3-node int and 5-node string binary search tree

    tree1 = Node 2 (Node 1 Nil) (Node 3 Nil)
    subtree1 = Node "b" (Node "a" Nil Nil) (Node "c" Nil Nil)
    tree2 = Node "d" subtree1 (Node "e" Nil Nil)

Exercise 3: Write an isEmpty function for trees

    isEmpty Nil = True
    isEmpty _  = False

Exercise 4: Write a size function for the tree (# of nodes)

    size Nil       = 0
    size (Node _ l r) = 1 + size l + size r

Exercise 5: Write a height function for the tree (# of levels)

    -- note max is defined in Haskell
    height Nil          = 0
    height (Node _ l r) = 1 + max (height l) (height r)
Example Binary Search Tree Implementation

\[
\begin{aligned}
\text{insert} &: (\text{Ord } a) \Rightarrow a \to \text{Tree } a \to \text{Tree } a \\
\text{insert } x \ \text{Nil} &= \text{Node } x \ \text{Nil} \ \text{Nil} \\
\text{insert } x \ (\text{Node } y \ l \ r) \\
| \ x < y &= \text{Node } y \ (\text{insert } x \ l) \ r \\
| \ \text{otherwise} &= \text{Node } y \ l \ (\text{insert } x \ r)
\end{aligned}
\]

\[
\begin{aligned}
\text{find} &: (\text{Ord } a) \Rightarrow a \to \text{Tree } a \to \text{Bool} \\
\text{find } x \ \text{Nil} &= \text{False} \\
\text{find } x \ (\text{Node } y \ l \ r) \\
| \ x == y &= \text{True} \\
| \ x < y &= \text{find } x \ l \\
| \ \text{otherwise} &= \text{find } x \ r
\end{aligned}
\]

\[
\begin{aligned}
\text{delete} &: (\text{Ord } a) \Rightarrow a \to \text{Tree } a \to \text{Tree } a \\
\text{delete } x \ \text{Nil} &= \text{Nil} \\
\text{delete } x \ (\text{Node } y \ l \ r) \\
| \ x < y &= \text{Node } y \ (\text{delete } x \ l) \ r \\
| \ x > y &= \text{Node } y \ l \ (\text{delete } x \ r) \\
| \ x == y &= \text{delete'} (\text{Node } x \ l \ r)
\end{aligned}
\]

where

\[
\begin{aligned}
\text{delete'} (\text{Node } _ \ Nil \ Nil) &= \text{Nil} \\
\text{delete'} (\text{Node } _ \ l \ Nil) &= l \\
\text{delete'} (\text{Node } _ \ Nil \ r) &= r \\
\text{delete'} (\text{Node } x \ l \ r) &= \\
| & \ \text{let } y = \text{inorderSucc } r \\
| & \ \text{in Node } y \ l \ (\text{delete } y \ r) \\
\text{inorderSucc } (\text{Node } x \ Nil \ _) &= x \\
\text{inorderSucc } (\text{Node } _ \ l \ _) &= \text{inorderSucc } l
\end{aligned}
\]