Today

- Quiz 8
- Exam 2 overview
- User-defined types

Assignments

- R-9, HW-9 due
- R-10, HW-10 out (due next Thurs)

Announcements

- Exam 2 on Thursday
- Proj update on Thursday
Exam 2 Overview

Basics ...

• Closed book, notes, computer, etc.
• 5 multipart questions
• Worth 15% of final grade

Reading topics to study ...

• union types, garbage collection, strongly typed, name-vs-structure type equivalence (reading 6)
• mixed-mode expressions, type conversions (reading 7)

Exam topics ...

• Emphasis on Haskell
• Haskell basics, types, defining functions (incl. pattern matching and guards)
• Review of grammars, lexical analysis, recursive descent parsing
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
Haskell data types are **nominal**

- that is, types with different names are different types
- in fact ...
  - if two types have the *same structure*
  - but have *different names*
  - they are *different types*

- For example:

```hs
data MagazineInfo = Magazine Integer String [String]
deriving (Show)
```

- while this type has the same structure as BookInfo
- it defines a completely different type
Type and value constructors are **independent**

- so far we’ve used different names (BookInfo vs. Book)
- convention is to use the same name for both (when appropriate)

```haskell
data Book = Book Integer String [String]
    deriving (Eq, Show)

data Magazine = Magazine Integer String [String]
    deriving (Eq, Show)
```

Note that here we also derive Eq for equality checking

- Alternatively, you can make your data type an instance of a typeclass:

```haskell
instance Eq Book where
    (==) (Book id1 _ _) (Book id2 _ _) = id1 == id2
```

- we’ll talk more later about functions and data types
Type Synonyms

The `type` keyword creates **type synonyms**

- a type synonym creates a *new name* for an existing type

```haskell
  type ID = Integer
  type Title = String
  type Authors = [String]
```

- can help give “meaning” to fields

```haskell
  data Book = Book ID Title Authors
              deriving (Show)

  data Magazine = Magazine ID Title Authors
                 deriving (Show)
```

Type synonyms are **structural** (as opposed to **nominal**)

- Authors and `[String]` are the *same type*
- whereas Book and Magazine are *different types*

Q: What type synonym have we already used in Haskell?

```haskell
  type String = [Char]
```

Type synonyms can name complex structures

```haskell
  type BookRecord = (Book, Review, Retailer)
```

- a triple of type Book, Review, and Retailer
Some other features of data types ...

Constructors can have 0 fields ...

```haskell
data RedColor = Red
    deriving (Show, Eq)
```

- A named value of a type
- Another example: True

Can have multiple value constructors ...

```haskell
data RGBColors = Red
    | Green
    | Blue
    deriving (Show, Eq)
```

- Three different constructors for RGBColors type
- Each can have different fields
- Another example: data Bool = True | False
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 Maybe type ...

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

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

Prelude> :type Nothing
Nothing :: Maybe a

• Creating Maybe values

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

\[
\text{myDiv } \text{x y} \\
\mid y == 0 \quad = \text{Nothing} \\
\mid \text{otherwise} = \text{Just } (x / y)
\]

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

*Main> myDiv 1 0
Nothing

*Main> myDiv 1 1
Just 1.0
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:

```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
```

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

```haskell
removeLast :: List a -> List a
removeLast Nil = error "empty list"
removeLast (Node _ Nil) = Nil
removeLast (Node x tail) = Node x (removeLast tail)
```