Today

- Quiz 9
- Exam 2 Overview
- User-Defined Types

Assignments

- HW 9 due, Proj Status Update due
- Exam 2 Thurs
- HW 10 and Extra Credit coming soon

Note on where with guards vs. let

\[
f \ x \ \\
| \ g_1 = e_1 \\
| \ g_2 = e_2 \\
\text{where} \ldots
\]

\[
f \ x \ \\
| \ g_1 = \text{let} \ldots \text{in} \ e_1 \\
| \ g_2 = \text{let} \ldots \text{in} \ e_2
\]
Exam 2 Overview

Basics ...

- Closed book, notes, computer, etc.
- 4 multipart questions
- Worth 15% of final grade

Exam topics ...

- Pre-Haskell (one question w/ smaller parts):
  - scope, scoping rules, environments, type rules, operational semantics
  - type checking and interpretation
- Emphasis on Functional Programming and Haskell (two questions):
  - $\lambda$-calculus (e.g., evaluation a function application)
  - Haskell basics (properties/characteristics)
  - Haskell types (basic data types, type classes, function types)
  - Defining functions (including pattern matching and guards)
- Review of grammars, lexical analysis, recursive descent parsing
  - one review question (multiple parts)
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

```haskell
data BookType = Book Int 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 `BookType` 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 :: BookType

– Our Book value is of type BookInfo

Main*> :type Book
Book :: Int -> String -> [String] -> BookType

– 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:

```haskell
data MagazineType = Magazine Int String [String]  
                   deriving (Show)
```

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

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

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

data Magazine = Magazine Int 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 pattern matching on data types
Type Synonyms

The type keyword creates **type synonyms**

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

  ```haskell
  type ID = Int
  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