Today

- User-Defined Types

Assignments

- HW-5, R-5 due
- HW-6, R-6 out

Announcements

- Quiz 5 on Thursday
Note on where with guards vs. let

\[
f x
  \begin{array}{l}
    \mid g_1 = e_1 \\
    \mid g_2 = e_2 \\
    \text{where} \ldots
  \end{array}
\]

\[
f x
  \begin{array}{l}
    \mid g_1 = \text{let \ldots in } e_1 \\
    \mid g_2 = \text{let \ldots in } e_2
  \end{array}
\]
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 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:

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

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 ...

\[
data \text{RedColor} = \text{Red} \\
deriving (\text{Show, Eq})
\]

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

Can have multiple value constructors ...

\[
data \text{RGBColors} = \text{Red} \\
| \text{Green} \\
| \text{Blue} \\
deriving (\text{Show, Eq})
\]

- Three different constructors for RGBColors type
- Each can have different fields
- Another example: data \text{Bool} = \text{True} | \text{False}
Pattern matching with algebraic data types

Can use data constructors and fields with pattern matching

- values must be enclosed in parentheses

- Simple example

  \[
  \begin{align*}
  \text{bookID (Book id title authors)} &= \text{id} \\
  \text{bookTitle (Book id title authors)} &= \text{title} \\
  \text{bookAuthors (Book id title authors)} &= \text{authors}
  \end{align*}
  \]

- Can simplify with wildcards

  \[
  \begin{align*}
  \text{bookID (Book id \_ \_)} &= \text{id} \\
  \text{bookTitle (Book \_ title \_)} &= \text{title} \\
  \text{bookAuthors (Book \_ title \_)} &= \text{authors}
  \end{align*}
  \]
**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]