Data Model 04

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Guards

- A **guard** is part of pattern-matching syntax that allows you to refine a pattern using Boolean condition(s)
- Consider the function `posneg` written in mathematical expression:

$$\texttt{posneg}(n) = \begin{cases} \text{"Negative"} & \text{if } n < 0 \\ \text{"Positive"} & \text{if } n \geq 0 \end{cases}$$

- Guards in Haskell can be used in a similar meaning:

```
posneg n | n < 0  = "Negative"
posneg n | n >= 0 = "Positive"
```

```
posneg n | n < 0  = "Negative"
         | n >= 0 = "Positive"
```

- The keyword `otherwise` is supported but can be omitted:

```
posneg n | n < 0     = "Negative"
posneg n | otherwise = "Positive"
```

- Without `otherwise`, the last pattern will be a catch-all pattern

Binomial Coefficient Example

- Consider the definition of the Binomial Coefficient:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

- This function can be defined using pattern matching and guards as:

```
binom _ 0           = 1
binom n k | n == k = 1
binom n k           = (binom (n - 1) (k - 1)) + (binom (n - 1) k)
```

- Otherwise, we need to use `if-then-else` expression for the pattern `binom n k`

Records

- Haskell also support C style of component/filed name in `struct`
- Records are used in data declaration by replacing a type parameter by `name :: type`
- Here is a new style of our data type client

```
data Client = GovOrg     { clientName :: String }
            | Company    { clientName :: String,
                           companyID :: Integer,
                           person :: Person,
                           position :: String }
            | Individual { person :: Person,
                           ads    :: Bool }
            deriving Show

data Person = Person { firstName :: String,
                       lastName :: String }
            deriving Show
```

Records

- Without field names, the order of argument does matter:

```
ghci> Person "John" "Smith"
Person "John" "Smith"
ghci> Person "Smith" "John"
Person "Smith" "John"
```

- With field names, it can be in any order

```
ghci> Person { firstName = "John", lastName = "Smith" }
Person {firstName = "John", lastName = "Smith"}
ghci> Person { lastName = "Smith", firstName = "John" }
Person {firstName = "John", lastName = "Smith"}
```

- Field names are also used to create functions that access fields

```
ghci> firstName (Person "John" "Smith")
"John"
ghci> clientName (GovOrg "First Order")
"First Order"
ghci> :t position
position :: Client -> String
ghci> :t companyID
companyID :: Client -> Integer
```

Records

- Since field names are used to create functions, they must not clash with any other field or function names
- Same field names within the same type are fine:
  - `clientName` in both `GovOrg` and `Company` has type `Client -> String`
  - `person` in both `Company` and `Individual` has type `Client -> String`
- Records also help with pattern matching to eliminate a collection of _:

```
greet :: Client -> String
greet Individual { person = Person {firstName = fName}} = "Hi, " ++ fName
greet Company    { clientName = cName }                  = "Hi, " ++ cName
greet GovOrg     { }                                     = "Welcome"
```

- No need to worry if the structure of a data type is changed