Data Model 01

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Characters

- A character in Haskell has type `Char`
- Can be created in a couple ways:
  - A character itself inbetween single quotes:

```
ghci> 'a'
'a'
ghci> :t 'a'
'a' :: Char
```

  - Use the numeric value (Unicode):

```
ghci> '\97'
'a'
ghci> :t '\97'
'\97' :: Char
```

  - For a hexadecimal value, put the value in between `'\x` and `'`

```
ghci> '\x61'
'a'
```

Module `Data.Char`

- The standard `Prelude` library only provides a few functions related to the type `Char`

```
ghci> :t toUpper

<interactive>:1:1: error: Variable not in scope: toUpper
```

- We need to import the modeul `Data.Char`

```
ghci> import Data.Char
ghci> :t toUpper
toUpper :: Char -> Char
ghci> toUpper 'a'
'A'
```

- A reference can be found at https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Char.html

Module `Data.Char`

- Some useful functions:

```
ghci> chr 97
'a'
ghci> ord 'a'
97
ghci> isLower 'a'
True
ghci> isUpper 'a'
False
ghci> isDigit 'a'
False
ghci> isDigit '0'
True
ghci> isHexDigit 'F'
True
ghci> isLetter '0'
False
ghci> isLetter 'a'
True
```

Numbers

- `Int` is the bounded integer type
  - The size generally depends on the architecture
  - Guaranteed 32 bits
- `Integer` is an unbounded integral type
- `Float` and `Double` are sigle and double precisions, respectively
- A ratio type is supported using the module `Data.Ratio`

```
ghci> import Data.Ratio
ghci> 1 % 2 + 1 % 3
5 % 6
ghci> toRational 2.3
2589569785738035 % 1125899906842624
ghci> fromRational (2 % 3)
0.6666666666666666
```

- Watch out for the precision problem

```
ghci> toRational (fromRational (2 % 3))
6004799503160661 % 9007199254740992
```

Types

- Suppose we try to check the type of `12` and `4.9`:

```
ghci> :t 12
12 :: Num p => p
ghci> :t 4.9
4.9 :: Fractional p => p
```

- `Num` and `Fractional` are type classes
  - `Num` typeclass includes `Int`, `Integer`, `Float`, and `Double`
  - `Fractional` typeclass includes only `Float`, and `Double`
- `12` can behave like either `Int`, `Integer`, `Float`, or `Double`
- `4.9` can behave like either `Float`, or `Double`
- For now, think about a type class in Haskell as an Interface in Java
- We will discuss about type classes in detail later

Strings

- A string is a list of character
- Basic list functions like cons (`:`) and concat (++) work with strings

```
ghci> :t "Hello"
"Hello" :: [Char]
ghci> :t ['H', 'e', 'l', 'l', 'o']
['H', 'e', 'l', 'l', 'o'] :: [Char]
ghci> "He" ++ "llo"
"Hello"
ghci> 'H' : "ello"
"Hello"
ghci> 'H' : ['e', 'l', 'l', 'o']
"Hello"
ghci> ['H', 'e'] ++ "llo"
"Hello"
```

Lists

- Formally, a list is defined using the following inductive definition:
  - `[]` is a list
  - Given a list `lst`, append an element `e` to the front of the list `lst`, result it a list `e:lst`
- The basic operations to construct lists are `[]` and cons (`:`)

```
ghci> []
[]
ghci> 5 : []
[5]
ghci> 'a' : []
"a"
ghci> 1 : 2 : 3 : []
[1,2,3]
ghci> 3 : [2, 1]
[3,2,1]
ghci> 'H' : 'i' : []
"Hi"
```

More List Functions

- The function `reverse` returns a list in the reverse order:

```
ghci> :t reverse
reverse :: [a] -> [a]
ghci> reverse "Hello"
"olleH"
ghci> reverse [1, 5, 2, 4, 3]
[3,4,2,5,1]
ghci> reverse 1:[2,3]

<interactive>:6:1: error...
ghci> reverse (1:[2,3])
[3,2,1]
```

- The function `null` checks whether a list is empty

```
ghci> null [1,2,3]
False
ghci> null []
True
ghci> null "Hello"
False
ghci> null ""
True
ghci> null (1 : [])
False
```

More List Functions

- The function `head` returns the first element on the given list:

```
ghci> head [1, 2, 3]
1
ghci> head "Hello"
'H'
```

- The function `tail` returns the list with out the first element:

```
ghci> tail [1, 2, 3]
[2,3]
ghci> tail "Hello"
"ello"
```

- Be careful when using `head` or `tail` on the empty list:

```
ghci> head []
*** Exception: Prelude.head: empty list
ghci> tail []
*** Exception: Prelude.tail: empty list
```

List of Booleans

- A list of booleans is possible:

```
ghci> :t [True, False, False, True]
[True, False, False, True] :: [Bool]
ghci> [3 < 5, 5 == 9, 2 /= 6]
[True,False,True]
```

- /= is the Haskell's **not equal** operator
- The function `or`, logically `or` every element on the list:

```
ghci> or [True, True]
True
ghci> or [True, False]
True
ghci> or [False, True]
True
ghci> or [False, False]
False
ghci> or [True, False, False, True]
True
ghci> or [False, False, 3 == 5, 2 < 10]
True
```

List of Booleans

- The function `and`, logically `and` every element on the list:

```
ghci> and [True, True]
True
ghci> and [True, False]
False
ghci> and [False, True]
False
ghci> and [False, False]
False
ghci> and [True, False, False, True]
False
ghci> and [True, 3 /= 5, 2 < 10, False]
False
```

- Let's look at types of `or` and `and`

```
ghci> :t or
or :: Foldable t => t Bool -> Bool
ghci> :t and
and :: Foldable t => t Bool -> Bool
```

- `Foldable` is a type class and `List` is one of them

Simple `if-then-else` Expression

- An expression `if b then t else f` evaluates to the expression `t` if the value of `b` is `True` and it evaluates to the expression `f` otherwise

```
ghci> if 2 < 5 then "2 is less than 5" else "2 is not less than 5"
"2 is less than 5"
```

- Both `then` and `else` must be present along with the `if`
- Without the `else` keyword, the expression has no value when the condition is false
- The entire `if-then-else` expression must have a defined type

```
ghci> if True then 1 else "Zero"

<interactive>:103:14: error:
    • No instance for (Num [Char]) arising from the literal ‘1’
    • In the expression: 1
      In the expression: if True then 1 else "Zero"
      In an equation for ‘it’: it = if True then 1 else "Zero"
ghci> :t if True then "one" else "zero"
if True then "one" else "zero" :: [Char]
```

Multi-line Expression

- Multi-line expression in GHCi:

```
ghci> :{
Prelude| if 2 < 5
Prelude| then "2 is less than 5"
Prelude| else "2 is not less than 5"
Prelude| :}
"2 is less than 5"
```

- The first line must contain only `:{`
- Use `:}` to end the multi-line expression
- Only in GHCi

A Haskell Source File

- We use the extension `.hs` for a Haskell source code
- Generally, each source code is associated with a Haskell module
  - A Haskell module name always starts with an uppercase letter
    - `Data.Char`
    - `Data.Ratio`
  - A Hasekll filename should start with an uppercase
  - Not necessary if not associate with a module
- To test a program/module, we simply load it into GHCi using the command `:load` (`:l` for short) followed by a file name

A Haskell Source File

- Consider the following program in the file `Simple.hs`

```
-- Simple.hs
firstOrEmpty lst = if not (null lst) then head lst else "Empty"
```

- Load `Simple.hs` into GHCi and do some test

```
ghci> :l Simple.hs
[1 of 1] Compiling Main             ( Simple.hs, interpreted )
Ok, one module loaded.
ghci> firstOrEmpty ["Hello", "World", "!!!"]
"Hello"
ghci> firstOrEmpty []
"Empty"
ghci> firstOrEmpty "Hello"

<interactive>:5:14: error:...
ghci> firstOrEmpty [[1,2], [3,4,5]]

<interactive>:6:16: error:...
```

- Why the last two expressions result in errors?

Type Inference

- Haskell provides type inference
- We do not need to specify the type of an expression

```
-- Simple.hs
firstOrEmpty lst = if not (null lst) then head lst else "Empty"
```

```
ghci> :t firstOrEmpty
firstOrEmpty :: [[Char]] -> [Char]
```

- It is a good practice to explicitly specify the type of a function:
  - There may be more than one possibility
  - Clearly specify our intention about the function
- Use `::` to specify the type of an expression before defining it

```
-- Simple.hs
firstOrEmpty :: [[Char]] -> [Char]
firstOrEmpty lst = if not (null lst) then head lst else "Empty"
```

- Use the command `:r` to reload the current file (module)

Simple Functions

- There are multiple way to define a function in Haskell
- For now, we will stick with `if-then-else` expression
- `List` is the most used ADT in Haskell
- Let's practice defining functions on lists
- Recall the inductive definition of a list:
    1. `[]` is a list (empty list)
    2. An element appended to the front of a list is a list
- Thus, a function on a list must be able to handle two cases:
    1. What to do when the list is empty
    2. What to do when the list has the first element and a tail

Concatenation

- We are going to create a concatenation function as an **infix** function
- The name of our concatenation function will be +++
- This will allow a user to create an expression

  ```
  lst1 +++ lst2
  ```

  where `lst1` and `lst2` are lists with the same type
- Recall that we can turn an infix operator to a prefix operator by putting parentheses around it:

  ```
  (+++) lst1 lst2
  ```

- Note that there are no loops in functional programmings
- **Think recursively**

Concatenation

- Traditionally, we represent a non-empty list as `x:xs`
  - `x` is the first item on the list (head)
  - `xs` is the rest of the list (tail) which can be an empty list
- If `x:xs` is `[3,5,1,2,4]`, `x` is 3 and `xs` is `[5,1,2,4]`
- Try to handle all possible situation about list
  - `[] +++ [] = []`
  - `[] +++ (y:ys) = (y:ys)`
  - `(x:xs) +++ [] = (x:xs)`
  - `(x:xs) +++ (y:ys) = ???`
- Try to handle the last case using just cons `(:)` and our concatenation operator (+++) **recursively**

```
(x:xs) +++ (y:ys) = x:(xs +++ (y:ys))
```

- Instead of handling four conditions, can we do it in two?

Concatenation

- An example of concatenation:

```
(+++) :: [a] -> [a] -> [a]
(+++) lst1 lst2 = if null lst1
                  then lst2
                  else head lst1 : ((tail lst1) +++ lst2)
```

- Load it into GHCi and do some tests

```
ghci> :t (+++)
(+++) :: [a] -> [a] -> [a]
ghci> [1,2] +++ []
[1,2]
ghci> [1,2] +++ [3,4,5]
[1,2,3,4,5]
ghci> "Hello" +++ " World" +++ "!!!"
"Hello World!!!"
ghci> [] +++ []
[]
ghci> :t it
it :: [a]
```

- Let's trace `[1,2] +++ [3,4]`:

$$
\begin{aligned}
\texttt{[1,2] +++ [3,4]} &\leadsto \texttt{1:([2] +++ [3,4])} \\
&\leadsto \texttt{1:(2:([] +++ [3,4]))} \\
&\leadsto \texttt{1:(2:[3,4])} \\
&\leadsto \texttt{1:[2,3,4]} \\
&\leadsto \texttt{[1,2,3,4]}
\end{aligned}
$$