Lists

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Lists

- Lists in Haskell are **homogeneous** data structures
- Constructor are `[]` and `:`
- Some functions on lists are covered:

```
ghci> 1 : []
[1]
ghci> null []
True
ghci> [1,2] ++ [3,4]
[1,2,3,4]
ghci> head [1,2,3,4]
1
ghci> tail [1,2,3,4]
[2,3,4]
ghci> reverse [1,2,3,4]
[4,3,2,1]
```

- Strings in Haskell are lists of characters

```
ghci> "He" ++ "llo"
"Hello"
```

Lists

- Since `List` is a type, a list of lists is allowed:

```
ghci> ["Hello", "World", "!!!"]
["Hello","World","!!!"]
ghci> [[1,2], [], [3,4,5]]
[[1,2],[],[3,4,5]]
```

- Reminder: `[]`, `[[]]`, are `[[],[],[]]` are all different
  - `[]` is the empty list

```
ghci> null []
True
```

  - `[[]]` is not empty but contain the empty list as its only element

```
ghci> null [[]]
False
ghci> null (head [[]])
True
```

  - `[[],[],[]]` has three elements and they are all the empty lists

```
ghci> null [[],[],[]]
False
```

Accessing List Elements

- To access an element in a list, use the `!!` operator
- The first index is 0

```
ghci> "Hello World" !! 6
'W'
ghci> [3,5,2,4,1] !! 3
4
ghci> [3,5,2,4,1] !! 5
*** Exception: Prelude.!!: index too large
```

Comparing Lists

- Lists can be compared if the items they contain can be compared
- Using ==, /=, <, <=, >, and >=, lists are compared in lexicographical order
- The order of the two lists is determined by the order of the first pair of differing elements:

```
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4,3]
False
ghci> [3,4,2] < [3,4,3]
True
ghci> [3,4,2] == [3,4,2]
True
```

- Nonempty list is considered to be greater than an empty list

```
ghci> [1,2,3] < [1,2]
False
ghci> [1,2] < [1,2,3]
True
```

More List Operations

- The `last` function returns a list's last element

```
ghci> last [1,2,3,4,5]
5
ghci> last [1]
1
ghci> last []
*** Exception: Prelude.last: empty list
```

- The `init` function takes a list and return everything except its last element

```
ghci> init [1,2,3,4,5]
[1,2,3,4]
ghci> init [1]
[]
ghci> init []
*** Exception: Prelude.init: empty list
```

- The `length` function takes a list and returns its length

```
ghci> length [1,2,3,4,5]
5
ghci> length []
0
```

More List Operations

- The `take` functions takes a number and a list and extract the specified number of elements from the beginning of the list

```
ghci> take 3 [1,2,3,4,5]
[1,2,3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [1,2,3]
[]
```

  - Take more than the number of elements results in the original list
  - Take 0 elements results in the empty list

- The `drop` functions drop the specified number of elements from the beginning and returns the rest

```
ghci> drop 3 [1,2,3,4,5]
[4,5]
ghci> drop 5 [1,2]
[]
ghci> drop 0 [1,2,3]
[1,2,3]
```

  - Drop more than the number of elements results in the empty list
  - Drop 0 elements results in the original list

More List Operations

- If elements in a list can be compared, the `maximum` function returns the largest element on the list

```
ghci> maximum [3,5,2,4,1]
5
ghci> maximum [3]
3
ghci> maximum "Hello"
'o'
ghci> maximum []
*** Exception: Prelude.maximum: empty list
```

- Similarly, the `minimum` function returns the smallest element on the list

```
ghci> minimum [3,5,2,4,1]
1
ghci> minimum [5]
5
ghci> minimum "Hello"
'H'
ghci> minimum []
*** Exception: Prelude.minimum: empty list
```

More List Operations

- The `sum` function takes a list of numbers and returns their sum

```
ghci> sum [3,5,2,4,1]
15
ghci> sum [3.1, 5.2, 4.3, 5.4, 1.5]
19.5
ghci> sum []
0
```

- The `product` functions takes a list of number and returns their product

```
ghci> product [3,5,2,4,1]
120
ghci> product [3.1, 5.2, 4.3, 5.4, 1.5]
561.4596000000001
ghci> product []
1
```

More List Operations

- The `elem` function takes an item and a list and tell us if the item is an element of the list

```
ghci> elem 5 [3,5,2,4,1]
True
ghci> elem 12 [3,5,2,4,1]
False
ghci> elem 7 []
False
ghci> elem 'e' "Hello"
True
ghci> 2 `elem` [3,5,2,4,1]
True
ghci> 'W' `elem` "Hello"
False
```

- Surround the prefix operator by ` and ` will turn it into an infix operator

Ranges

- Ranges are used to make lists composed of elements that can be **enumerated** or counted off in order
  - Numbers can be enumerated: `1, 2, 3, 4, ...`
  - Characters can be enumerated: `'A', 'B', 'C', ...`
- Here are some examples:

```
ghci> [5..27]
[5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
ghci> ['D'..'W']
"DEFGHIJKLMNOPQRSTUVW"
ghci> ['a'..'r']
"abcdefghijklmnopqr"
ghci> [12..4]
[]
```

- By default, it tries to count up

Ranges

- Step between items can be specified in a range
- Simply separate the first two elements with a command and specifying the upper/lower limit

```
ghci> [0,2..15]
[0,2,4,6,8,10,12,14]
ghci> [12,11..4]
[12,11,10,9,8,7,6,5,4]
ghci> [1,11..99]
[1,11,21,31,41,51,61,71,81,91]
```

- An infinite list is allowed in Haskell by not specifying the upper/lower limit

```
ghci> [2,4..]
```

- **The above statement will not stop**

Ranges

- Because of the lazy evaluation, infinite lists are useful
- Suppose we want to create a list of the first 15 multiple of 17

```
ghci> [17,34..17*15]
[17,34,51,68,85,102,119,136,153,170,187,204,221,238,255]
```

- We can use infinite list with the `take` function as well

```
ghci> take 15 [17,34..]
[17,34,51,68,85,102,119,136,153,170,187,204,221,238,255]
```

More List Operations

- The `cycle` function takes a list and replicates its elements **indefinitely** which results in an infinite list

```
ghci> take 18 (cycle [1,2,3,4])
[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2]
ghci> cycle []
*** Exception: Prelude.cycle: empty list
```

- The `repeat` function takes an element and produces an infinite list of just that element

```
ghci> take 12 (repeat 7)
[7,7,7,7,7,7,7,7,7,7,7,7]
```

- The `replicate` function can be used to generate a list composed of a single item

```
ghci> replicate 5 12
[12,12,12,12,12]
```

  - The `replicate` function does not generate an infinite list

List Comprehensions

- **List comprehensions** are a way to filter, transform, and combine lists
- They are very similar to **set comprehensions**
- In this discussion, a set is a list (ordered and duplicates are allowed)
- Suppose we have $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- Consider the following set:

$$\{x \mid x \in A\}$$

- We can express the above expression in Haskell as:

```
ghci> [x | x <- [1..10]]
[1,2,3,4,5,6,7,8,9,10]
```

List Comprehensions

- `x <- [1..10]` means that `x` takes on the value of each element that is drawn from the list `[1..10]`
- In other words, we **bind** each element from `[1..10]` to `x`
- The part before the vertical pipe (`|`) is the **output** of the list comprehension
- The output is the part where we specify how we want the elements that we have drawn to be reflected int he resulting list

```
ghci> [x | x <- [1..10]]
[1,2,3,4,5,6,7,8,9,10]
ghci> [x * 2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
ghci> [x / 2 | x <- [1..10]]
[0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
ghci> [replicate 2 x | x <- [1..10]]
[[1,1],[2,2],[3,3],[4,4],[5,5],[6,6],[7,7],[8,8],[9,9],[10,10]]
```

Predicates in List Comprehensions

- Predicates are expressions that have type `Bool`
- Predicates go at the end of the list comprehension and are separated from the rest of the comprehension by a comma

```
ghci> [x | x <- [1..10], x < 5]
[1,2,3,4]
ghci> [x | x <- [1..10], x >= 5]
[5,6,7,8,9,10]
ghci> [x | x <- [1..10], x >= 3, x <= 8]
[3,4,5,6,7,8]
ghci> [x | x <- [1..10], mod x 2 == 0]
[2,4,6,8,10]
```

- The predicate part in a list comprehension is also called **filtering**

List Comprehensions

- We can have multiple takes in a list comprehension:

```
ghci> [x + y | x <- [1..3], y <- [8..11]]
[9,10,11,12,10,11,12,13,11,12,13,14]
ghci> [(x,y) | x <- [1..3], y <- [6..8]]
[(1,6),(1,7),(1,8),(2,6),(2,7),(2,8),(3,6),(3,7),(3,8)]
ghci> [x + y | x <- [1..3], y <- [8..11], x + y < 11]
[9,10,10]
```

- More examples:

```
ghci> greeting = ["Hello", "Goodbye"]
ghci> people = ["Alice", "Bob", "Carol"]
ghci> [x ++ " " ++ y | x <- greeting, y <- people]
["Hello Alice","Hello Bob","Hello Carol","Goodbye Alice",
 "Goodbye Bob","Goodbye Carol"]
```

- In some version of `GHCi`, you may have to use the `let` keyword to bind variable `greeting` and `people`

List Comprehensions

- A list comprehension is an expression
- A function can be expressed by a list comprehension

```
ghci> evenOdd xs = [if mod x 2 == 0 then "even" else "odd" | x <- xs]
ghci> :t evenOdd
evenOdd :: Integral a => [a] -> [[Char]]
ghci> evenOdd [3,5,4,1,2]
["odd","odd","even","odd","even"]
```

```
ghci> onlyUpper xs = [x | x <- xs, elem x ['A'..'Z']]
ghci> onlyUpper "Hello World"
"HW"
```

- The output or a predicate can also be a list comprehension:

```
ghci> [[x | x <- [3..5], x < y] | y <- [4..6]]
[[3],[3,4],[3,4,5]]
```

- Output of a list comprehension can be a constant:

```
ghci> [1 | _ <- [4..10]]
[1,1,1,1,1,1,1]
```

- Since we do not need elements taken from the list `[4..10]`, we can use _