LISTS IN HASKELL

SOME MORE LIST NOTATIONS

List notation, such as [1,2,3], is in fact an abbreviation for a more basic form 1:2:3:[] where [ ] is the null list and (:) is the cons operator

Example:

ghci> 3:[]

[3]

ghci> 2:3:[]

[2,3]

ghci> 1:2:3:[]

[1,2,3]

FORMS AND KINDS OF LISTS

- Every list of type [a] takes one of three forms:
  - The undefined list undefined :: [a];
  - The empty list [] :: [a];
  - A list of the form x:xs where x :: a and xs :: [a].
- As a result there are three kinds of lists:
- A *finite* list, which is built from (:) and []; for example, 1:2:3:[]
- An *infinite* list, which is built from (:) alone; for example, [1..] is the infinite list of the non-negative integers.
- A *partial* list, which is built from (:) and undefined. For example, the list

filter (<4) [1..] is the partial list 1:2:3:undefined.

ghci> (filter odd [1,2,3,4,5])

[1,3,5]

ghci> (filter (<4) [1..])
[1,2,3

SET COMPREHENSIONS TO LIST COMPREHENSIONS

- Set comprehensions: Normally used for building more specific sets out of general sets.
- Example:

$$S = \{2 \cdot x \mid x \in N, x \leq 10\}.$$

  - The part before the pipe is called the output function,
  - **x** is the variable,
  - **N** is the input set
  - and **x <= 10** is the *predicate*

S contains the doubles of all natural numbers that satisfy the predicate.

- Example in Haskell:
  - *ghci> take 10 [2,4..]*
  - *[2,4,6,8,10,12,14,16,18,20]*

LIST COMPREHENSION

- Creates a list from one or more other lists

Example 1:

```
ghci> [x*x | x <- [1..5]]
[1,4,9,16,25]
```

Example 2:

```
ghci> [(i,j) | i <- [1..5], even i, j <- [i..5]]
[(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
```

Example 3:

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

LIST COMPREHENSION

- Predicates go after the binding parts and are separated from them by a comma.
- Let's say we want only the elements which, doubled, are greater than or equal to 12.

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

Weeding out lists by predicates is also called **filtering**

Try it yourself:

- How to obtain all numbers from 50 to 100 whose remainder when divided with the number 7 is 3?

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

A COMPREHENSION SCENARIO USING FUNCTION !

- Let's say we want a comprehension that replaces each odd number greater than 10 with **"BANG!"** and each odd number that's less than 10 with **"BOOM!"** . If a number isn't odd, we throw it out of our list.

```
--boomBangs.hs
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]

ghci> :l boomBangs.hs
[1 of 1] Compiling Main             ( boomBangs.hs, interpreted )
Ok, one module loaded.
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

INCLUDING SEVERAL PREDICATES

Example:

Obtain all numbers from 10 to 20 that are not 13, 15 or 19

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
```

```
[10,11,12,14,16,17,18,20]
```

DRAWING THE OUTPUT FROM SEVERAL LISTS

- When drawing from several lists, comprehensions produce all combinations of the given lists and then join them by the output function we supply.
- A list produced by a comprehension that draws from two lists of length 4 will have a length of 16, provided we don't filter them.
- Example: To obtain the product of all possible combinations of two lists:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

- Using Filter: To get all possible products that are more than 50

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

MORE EXAMPLES

- A list comprehension that combines a list of adjectives and a list of nouns

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy
frog","grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

- Nested list comprehensions:

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5], [1,2,3,4,5,6,7,8,9],
[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

SOME COMMON LIST FUNCTIONS USING COMPREHENSIONS : MAP

```
--mymap.hs
mymap f xs = [ f  x |  x <- xs ]
```

Mymap applies the function f to each element x of the list xs

```
ghci> :l mymap.hs
[1 of 1] Compiling Main             ( mymap.hs, interpreted )
Ok, one module loaded.
ghci> mymap abs [1,-2,3,-4,5]
[1,2,3,4,5]
ghci> mymap odd [1,2,3,4,5]
[True,False,True,False,True]
```

FILTER FUNCTION USING COMPREHENSION

```
--myfilter.hs
myfilter p xs = [x | x <- xs, p x]
```

myfilter applies the predicate p to each element x of the list xs

```
ghci> :l myfilter.hs
[1 of 1] Compiling Main             ( myfilter.hs, interpreted )
Ok, one module loaded.
ghci> myfilter odd [1,2,3,4,5]
[1,3,5]
ghci> myfilter (<4) [1,2,3,4,5]
[1,2,3]
```

CONCATENATION USING COMPREHENSION

```
--myconcat.hs
myconcat xss = [x | xs <- xss, x <- xs]

ghci> :l myconcat.hs
[1 of 1] Compiling Main             ( myconcat.hs, interpreted )
Ok, one module loaded.

ghci> myconcat [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]

ghci> myconcat [['a','b','c'],['d','e','f']]
"abcdef "

ghci> myconcat [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

xss is a list of lists

PYTHAGOREAN TRIAD FUNCTION - EXAMPLE

```
--Pythagorean triads function using list comprehension pythtriad.hs
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | x <- [1..n], y <- [1..n],
 z <- [1..n], x*x+y*y==z*z]


ghci> :l pythtriad.hs
[1 of 1] Compiling Main             ( pythtriad.hs, interpreted )
Ok, one module loaded.
ghci> triads 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

SOME BASIC OPERATIONS

- null : null :: [a] -> Bool

  ghci> null []

  True

  ghci> null [1,2,3,4,5]

  False

- head: head :: [a] -> a
  - head (x:xs) = x

    ghci> head [1,2,3,4,5]

    1

- tail: tail :: [a] -> [a]
  - tail (x:xs) = xs

    ghci> tail [1,2,3,4,5]

    [2,3,4,5]

Haskell reports an error if we try to evaluate head [] or tail []

ghci> head []

*** Exception: Prelude.head: empty list

ghci> tail []

*** Exception: Prelude.tail: empty list

SOME BASIC OPERATIONS

- <u>last:</u> last :: [a] -> a
  - last [x] = x

    ghci> last [1]

    1
  - last (x:y:ys) = last (y:ys)

    ghci> last [1,2,3,4,5]

    5
- <u>Concatenation:</u> (++) :: [a] -> [a] -> [a]
  - [] ++ ys = ys

    ghci> [] ++ [1,2,3,4,5]

    [1,2,3,4,5]

SOME BASIC OPERATIONS

- Concatenation: (++) :: [a] -> [a] -> [a]
  - [] ++ ys = ys
  - (x:xs) ++ ys = x:(xs ++ ys)

    ghci> [1,2,3,4,5] ++ [6,7,8,9,10]

    [1,2,3,4,5,6,7,8,9,10]
- length: length :: [a] -> Int
  - length [] = 0

    ghci> length []

    0
  - length (x:xs) = 1 + length xs

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

    5

DEFINING CONCAT USING PATTERN MATCHING

```
--concat1 using pattern matching
concat1 :: [[a]] -> [a]
concat1 [] = []
concat1 (xs:xss) = xs ++ concat1 xss
```

The concat function takes a list of lists, all of the same type, and concatenates them into a single list

```
ghci> :l concat1.hs
[1 of 1] Compiling Main             ( concat1.hs, interpreted )
Ok, one module loaded.
ghci> concat1 []
[]
ghci> concat1 [[1,2,3,4], [5,6,7,8]]
[1,2,3,4,5,6,7,8]
ghci> concat1 [[1,2,3,4], [5,6,7,8],[9,10,11,12]]
[1,2,3,4,5,6,7,8,9,10,11,12]
```

DEFINING MAP USING PATTERN MATCHING

```
--map1 using pattern matching
map1 :: (a -> b) -> [a] -> [b]
map1 f [] = []
map1 f (x:xs) = f x:map1 f xs


ghci> :l map1.hs
[1 of 1] Compiling Main             ( map1.hs, interpreted )
Ok, one module loaded.
ghci> map1 abs [1,-2,3,-4,5]
[1,2,3,4,5]
ghci> map1 odd [1,2,3,4,5]
[True,False,True,False,True]
```

The signature tells us that map takes two arguments. The first is a function that takes a value of one type, a, and returns a value of another type, b.

DEFINING FILTER USING PATTERN MATCHING

```
--filter1 using pattern matching
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 p [] = []
filter1 p (x:xs) = if p x then x:filter1 p xs
else filter1 p xs
```

The filter function takes a predicate and applies it to every element in its input list, returning a list of only those for which the predicate evaluates to True.

```
ghci> :l filter1.hs
[1 of 1] Compiling Main             ( filter1.hs, interpreted )
Ok, one module loaded.
ghci> filter1 odd [1,2,3,4,5]
[1,3,5]
ghci> filter1 even [1,2,3,4,5]
[2,4]
ghci> filter1 (<=3) [1,2,3,4,5]
[1,2,3]
```

ZIP AND ZIPWITH

- The zip function takes two lists and “zips” them into a single list of pairs
- The resulting list is the same length as the shorter of the two inputs

```
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]
ghci> zip [12,72,93] "zippity"
[(12,'z'),(72,'i'),(93,'p')]
```

- zipWith takes two lists and applies a function to each pair of
- elements, generating a list that is the same length as the shorter of the two

```
ghci> :type zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
ghci> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

STANDARD PRELUDE DEFINITIONS OF ZIP AND ZIPWITH

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _ _ = []


zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

MORE LIST OPERATORS AND FUNCTIONS

- If you want to get an element out of a list by index, use **!!** . The indices start at 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

- Lists can also contain lists. They can also contain lists that contain lists that contain lists …

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

But if you try to get the
sixth element from a list
that only has four elements,
you'll get an error
so be careful!

LIST COMPARISONS

- Lists can be compared if the stuff they contain can be compared
- When using **<** , **<=** , **>** and **>=** to compare lists, they are compared in lexicographical order
- First the heads are compared
- If they are equal then the second elements are compared, etc.

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

SOME MORE BASIC FUNCTIONS TO OPERATE ON LISTS

- Init: takes a list and returns everything except its last element

  ghci> init [5,4,3,2,1]

  [5,4,3,2]

- reverse reverses a list:

  ghci> reverse [5,4,3,2,1]

  [1,2,3,4,5]

- **maximum** takes a list of stuff that can be put in some kind of order and returns the biggest element and **minimum** returns the smallest.

  ghci> minimum [8,4,2,1,5,6]

  1

  ghci> maximum [1,9,2,3,4]

  9

SOME MORE BASIC FUNCTIONS TO OPERATE ON LISTS

- **Sum** takes a list of numbers and returns their sum.

- **Product** takes a list of numbers and returns their product.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

SOME MORE BASIC FUNCTIONS TO OPERATE ON LISTS

- **elem** takes an element and a list of items and says if the element is an item in the list
- elem is usually called as an infix function for easy readability

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

- **cycle** takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

SOME MORE BASIC FUNCTIONS TO OPERATE ON LISTS

- **repeat** takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

  ghci> take 10 (repeat 5)

  [5,5,5,5,5,5,5,5,5,5]

- **replicate** function also replicates some number of the same element in a list:

  ghci> replicate 3 10

  [10,10,10]