List in Haskell

Principles of Programming Languages

Lists

- In Haskell list is used to store the elements, and these elements are homogenous in nature which simply means only one type.
- It provides us some functions which allows to modify the elements of the list, also to merge two lists together by using the functions available in the list of Haskell.

List

- In Haskell, lists are a homogenous data structure.
- It stores several elements of the same type.
- That means that we can have a list of integers or a list of characters, but we can't have a list that has a few integers and then a few characters.
- Syntax:-

```
let varibale_name = [value1, value2, value3 ..]
```

- Example:- `let demo = ["val1", "val2", "val3", "so on"]`

List

- Note: We can use the let keyword to define a name right in GHCI. Doing let a = 1 inside GHCI is the equivalent of writing a = 1 in a script and then loading it.

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

Lists

- When taking apart a list, the main pieces are the head, the tail, and the end (represented by []).
- The tail is the rest of the list left over, after the head.
- The head is just the first element in a list:

```
GHCi> head [1,2,3]
1
GHCi> head [[1,2],[3,4],[5,6]]
[1,2]
```

- The tail is the rest of the list left over, after the head:

```
GHCi> tail [1,2,3]
[2,3]
GHCi> tail [3]
[]
```

List

- A common task is putting two lists together. This is done by using the ++ operator.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Two Constructors

[]

- Constructs an empty list.

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []     -- ((:) is right-associative)
-- or  = [12, 99, 37]          -- (syntactic sugar for lists)
```

(:)

- Pronounced cons, prepends elements to a list.
- Consing x (a value of type a) onto xs (a list of values of the same type a) creates a new list, whose head (the first element) is x, and tail (the rest of the elements) is xs.

Canonical Constructor - cons

- Putting something at the beginning of a list using the : operator also called the cons operator is instantaneous.
- The cons operator takes an element v and a list l and creates a new list v : l beginning with v followed by all the elements in l. The element v is called the head of the list, whereas the old list l is called the tail of the list.

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Cons

Note:

- [], [[]] and[[],[],[]] are all different things. The first one is an empty list, the seconds one is a list that contains one empty list, the third one is a list that contains three empty lists.

Cons

Under the hood, all lists in Haskell are represented as a bunch of consing operations, and the [...] notation is syntactic sugar (a feature of the programming language syntax designed solely to make things easier to read):

```
GHCi> 1:2:3:4:[]
[1,2,3,4]

GHCi> (1,2):(3,4):(5,6):[]
[(1,2),(3,4),(5,6)]
```

```
GHCi>['h','e','l','l','o']
"hello"
GHCi> 'h':'e':'l':'l':'o':[]
"hello"
```

cons

- We can create all of the lists shown earlier using repeated occurrences of : and one final [].
- Since : is right-associative, there is no need to write parentheses and 1 : (2 : []) can be safely written as 1 : 2 : [].

```
> 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : []
[0,1,2,3,4,5,6,7,8,9]
> True : False : False : []
[True,False,False]
```

```
> :type (:)
(:) :: a → [a] → [a]
```

** We see that (:) is a polymorphic function accepting an argument of type 'a' – the head of the list being created – and another argument of type [a] – the tail of the list being created – and returning a list of type [a].*

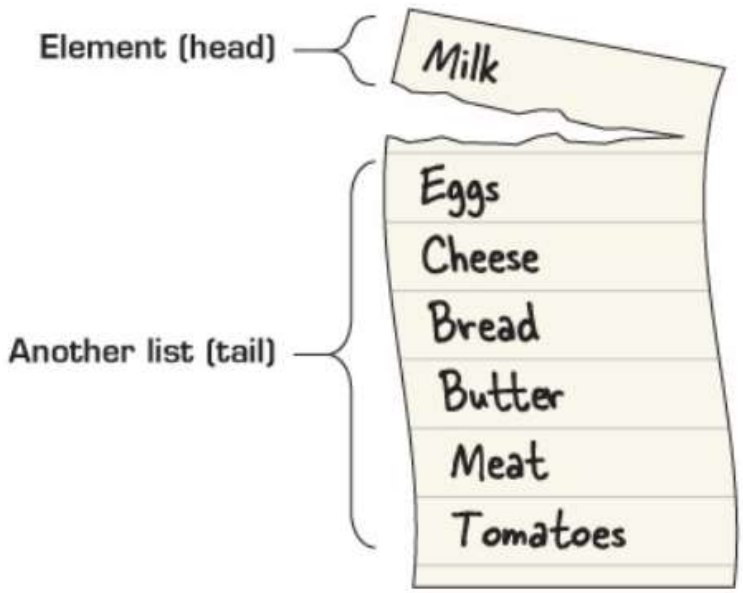
Cons

- By definition, a list is always a value consed with another list (which can also be an empty list).
- You could attach the value to the front of an existing list if you want.

```
GHCi> 1:[2,3,4]
[1,2,3,4]
```

List

- The tail of a list with just one element is [], which marks the end of the list.
- This end of the list is just an empty list. But an empty list is different from other lists, as it has neither a head nor a tail. Calling head or tail on [] will result in an error.
- If you look at the head and tail, you can start to see the recursive nature of working with lists: a head is an element, and a tail is another list.



To be noted

- In Haskell every element of the list must be the same type.
- For example, you can cons the letter 'h' to the string "ello" because "ello" is just a list of characters and 'h' (single quotes) is a character:

- But you can’t cons "h" (double quotes) to "ello" because "h" is a list of one character and the values inside "ello" are individual characters. This becomes more obvious when you remove the syntactic sugar.

```
GHCi> 'h':"ello"
"hello"
```

Lists and lazy evaluation

There are many ways to quickly generate ranges of data. Here are some examples:

```
GHCi> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10]

GHCi> [1,3 .. 10]
[1,3,5,7,9]

GHCi> [1, 1.5 .. 5]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]

GHCi> [1,0 .. -10]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

Lists and lazy evaluation

- When an infinite list is defined and used in a function.
- Haskell uses a special form of evaluation called lazy evaluation. In lazy evaluation, no code is evaluated until it's needed.
- In the case of longList, none of the values in the list were needed for computation.

```
simple x = x
longList = [1 .. ]
stillLongList = simple longList
```

Common Functions in List

The !! operator

The !! operator takes a list and a number, returning the element at that location in the list. Lists in Haskell are indexed starting at 0.

```
GHCi> [1,2,3] !! 0
1
GHCi> "puppies" !! 4
'i'
GHCi> [1..10] !! 11
*** Exception: Prelude.!!: index too large
```

!! As prefix operator

- !! can also be used like a prefix function by wrapping it in parentheses.

```
GHCi> (!!) [1,2,3] 0
1
```

- Prefix notation is also useful for using operators as arguments to other functions.

```
GHCi> paExample1 = (!!) "dog"
GHCi> paExample1 2
'g'
GHCi> paExample2 = ("dog" !!)
GHCi> paExample2 2
'g'
```

Common Functions in List

length

- The length function gives you the length of the list!

```
GHCi> length [1..20]
20
GHCi> length [(10,20),(1,2),(15,16)]
3
GHCi> length "quicksand"
9
```

reverse

- Reverses the list

```
GHCi> reverse [1,2,3]
[3,2,1]
GHCi> reverse "cheese"
"eseehc"
```

- Reverse to make a basic palindrome checker

```
isPalindrome word = word == reverse word

GHCi> isPalindrome "cheese"
False
GHCi> isPalindrome "racecar"
True
GHCi> isPalindrome [1,2,3]
False
GHCi> isPalindome [1,2,1]
True
```

Common Functions in List

elem

- The elem function takes a value and a list and checks whether the value is in the list.

```
GHCi> elem 13 [0,13 .. 100]
True
GHCi> elem 'p' "cheese"
False
```

Example for elem

- For example, the function respond returns a different response depending on whether a string has an exclamation mark, as follows.

```
respond phrase = if '!' `elem` phrase
                 then "wow!"
                 else "uh.. okay"

GHCi> respond "hello"
"uh.. okay"
GHCi> respond "hello!"
"wow!"
```

Common Functions in List

take

- The take function takes a number and a list as arguments and then returns the first n elements of the list.

```
GHCi> take 5 [2,4..100]
[2,4,6,8,10]
GHCi> take 3 "wonderful"
"won"
```

- If you ask for more values then a list has, take gives you what it can, with no error:

```
GHCi> take 1000000 [1]
[1]
```

Example of take

take works best by being combined with other functions on lists. *For example, you can combine take with reverse to get the last n elements of a list.*

```
takeLast n aList = reverse (take n (reverse aList))

GHCi> takeLast 10 [1..100]
[91,92,93,94,95,96,97,98,99,100]
```

Common Functions in List

drop

- The drop function is similar to take, except it removes the first n elements of a list:

```
GHCi> drop 2 [1,2,3,4,5]
[3,4,5]
GHCi> drop 5 "very awesome"
"awesome"
```

zip

- zip is used when you want to combine two lists into tuple pairs. The arguments to zip are two lists. If one list happens to be longer, zip will stop whenever one of the two lists is empty:

```
GHCi> zip [1,2,3] [2,4,6]
[(1,2),(2,4),(3,6)]
GHCi> zip "dog" "rabbit"
[('d','r'),('o','a'),('g','b')]
GHCi> zip ['a' .. 'f'] [1 .. ]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
```

Common Functions in List

Last

- The last function is the opposite of the head function. It returns the last element in a list.

Init

- The init function is the opposite of the tail function. It returns all elements from a list except the last one.

```
Prelude> let a = [1,2,3]
Prelude> a
[1,2,3]
Prelude> last a
3
Prelude> init a
[1,2]
```

Common Functions in List

cycle

- Given a list, cycle repeats that list endlessly.
- For example, it's common in numerical computing to need a list of n ones. With cycle, this function is trivial to make.

cycle example

```
ones n = take n (cycle [1])

GHCi> ones 2
[1,1]
GHCi> ones 4
[1,1,1,1]
```

Common Functions in List

- Imagine you want to divide a list of files and put them on n number of servers, or similarly spilt up employees onto n teams.
- The general solution is to create a new function, assignToGroups, that takes a number of groups and a list, and then cycles through the groups, assigning members to them.

```
assignToGroups n aList = zip groups aList
  where groups = cycle [1..n]

GHCi> assignToGroups 3 ["file1.txt","file2.txt","file3.txt"
                       ,"file4.txt","file5.txt","file6.txt","file7.txt"
                       ,"file8.txt"]

[(1,"file1.txt"),(2,"file2.txt"),(3,"file3.txt"),(1,"file4.txt"),
(2,"file5.txt"),(3,"file6.txt"),(1,"file7.txt"),(2,"file8.txt")]

GHCi> assignToGroups 2 ["Bob","Kathy","Sue","Joan","Jim","Mike"]
[(1,"Bob"),(2,"Kathy"),(1,"Sue"),(2,"Joan"),(1,"Jim"),(2,"Mike")]
```

Common Functions in List

null

null checks if a list is empty. If it is empty, it returns True, otherwise it returns False. Use this function instead of xs == [] (if you have a list called xs)

Example

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

Common Functions in List

maximum

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

minimum

- returns the smallest.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

Common Functions in List

sum

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

product

- 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
```

Common Functions in List

repeat

- 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

- Use the replicate function if you want some number of the same element in a list.
- replicate 3 10 returns [10,10,10].

Common Functions in List

Words

- The words function will separate a string by its spaces.

Unwords

- The unwords function does the opposite of words.

```
Prelude> let a = "Free will is an illusion"
Prelude> words a
["Free","will","is","an","illusion"]
Prelude> let b = words a
Prelude> b
["Free","will","is","an","illusion"]
Prelude> unwords b
"Free will is an illusion"
```

Deconstructing list with pattern

Every list is either the empty list [] or has the form v : l for some head v and some tail l. For this reason [] and : are the canonical list constructors that can be used in patterns for defining functions that analyze and deconstruct lists.

Example:-

```
headOrZero :: [Int] → Int
headOrZero [] = 0
headOrZero (x : _) = x
```

```
> headOrZero []
0
> headOrZero [1, 2, 3]
1
```

returns the head of a list of integers if the list is not empty and returns 0 otherwise.

- The first equation matches the list against the pattern []: there is only one list that matches this pattern, namely the empty list.
- The second equation matches the list against the pattern (x : _).

Every non-empty list matches this pattern, since a non-empty list has a head (here captured by the pattern variable x) and a tail (which is matched here by _).

Deconstructing list with pattern

Example – length of a list

The idea is to scan a list until we find []. This is a typical case in which a recursive function is needed.

- the base case is represented by the empty list.
- the recursive case is represented by the non-empty list.

```
length :: [a] → Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Observe that length is a polymorphic function: it does not depend in any way on the type of the elements of the list it is applied to.

```
> length [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
10
> length [True, False, False]
3
> length (fromTo (-5) 5)
11
```

Comparing Lists

- Relational operators can compare lists of different lengths, provided that they have the same type.
- The lexicographic ordering of lists states that a list l is smaller if a list l' if either l is a prefix of l' or if the first element of l in which l and l' disagree is smaller than the corresponding element in l' .

```
> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] == fromTo 0 9
True
> fromTo 0 9 == []
False
> [True, False, False] /= []
True
```

```
> fromTo 0 9 < fromTo 0 100
True
> fromTo 0 9 < []
False
> [1, 2, 3, 4, 5] < [4, 5]
True
```

Next – List Comprehension