Chapter 9 - Lists

*) The first purpose they serve is as a way to refer to & process a collection or plurality of values. The second is an ∞ series of values, usually generated by functions, which allows them to act as a stream datatype.

→ The list datatype

```
data [] a = [] | a : [a]
```

- `[]` → type constructor for lists
- `[]` → data constructor for empty list → nullary constructor, since it takes no arguments.
- `:` → infix operator called cons (short for construct). Here cons takes a value of type `a` and a list of type `[a]` and evaluates to `[a]`.

`a : [a]` constructs a value from two arguments, by adding the `a` to the front of the list `[a]`.

*) A singly linked list is seen as a fair description of a list in Haskell., although avg. case performance in some situations changes due to nonstrict evaluation.

↓

However it can contain ∞ data which makes it also work as a stream datatype, but one that has the option of ending the stream with the `[]` data constructor.

→ Pattern matching on lists : pattern matching on data constructors, Here we match on the first argument to the infix (:) constructor,

ignoring the rest of the list, and return that value

```
let myHead (x : _) = x
```

→ returns the head of any list.

pattern matching construct for the first element & the remaining list.

```
let myTail ( _ : xs) = xs
```

*) important to stay careful in such cases, since none of these functions is handling the empty list case. if an empty list is passed to these functions, then it can't pattern match.

↓

one solution is to put a base case on both functions handling the null list case

↓

better way to handle this might be to use maybe datatype. It makes the failure case explicit, and as programs get longer & more complex, that can be quite useful.

↓

```
safeTail :: [a] -> Maybe [a]

safeTail [] = Nothing
safeTail (x : []) = Nothing
safeTail (x : xs) = Just xs
```

↓

```
safeHead :: [a] → Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

→ List's syntactic sugar : Haskell has syntactic sugar to accomodate the use of lists, so that we can write :

```
>> [1, 2, 3] ++ [4]
>> [1, 2, 3, 4]
```

instead of :

```
>> (1 : 2 : 3 : []) ++ (4 : [])
```

↓ syntactic sugar is here to allow building lists in terms of successive applications of (:)

*) when we talk about lists, we often talk about them in terms of con cells & spines. The con cells are the list datatype's second data constructor, a : [a], the result of recursively prepending a value to more list. The con cell is a conceptual space that values may inhabit. The spine on the other hand is the connective structure that holds the con cells together and in place.

→ Using ranges to construct lists : [1..10]

ending element

two dots

starting element

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

*) The types of fns underlying this syntax are :

1) enumFrom , enumFrom :: Enum a
⇒ a → [a]

2) enumFromThen :: Enum a
⇒ a → a → [a]

3) enumFromTo :: Enum a
⇒ a → a → [a]

4) enumFromThenTo :: Enum a
⇒ a → a → a → [a]

*) All these fns require that the type being "ranged" have an instance of the Enum typeclass.

*) The first two functions, generate ∞ lists, if possible. For that to be possible, we must be ranging over a type that has no upperbound in its enumeration. Integer is such a type.

→ exercise : enumFromTo

```
eftBool x y
  | x > y  = []
  | x == y = [x]
  | x < y  = (x : eftBool (succ x) y)
```

→ Extracting portions of lists → useful fns. for extracting portions of a list and dividing lists into parts.

```
take :: Int → [a] → [a]
drop :: Int → [a] → [a]
splitAt :: Int → [a] → ([a], [a])
```

take : takes specified no. of elements out of a list → can be used successfully with ∞ lists to construct lists.

drop : drops the specified no. of elements from the beginning of the list

splitAt : cuts the list at the specified index

*) takeWhile & dropWhile perform the take & drop operations till the predicate holds true.

```
takeWhile :: (a → Bool) → [a] → [a]
dropWhile :: (a → Bool) → [a] → [a]
```

example : takeWhile (<3) [1..10]

takeWhile (<8) (enumFromTo 5 15)

```
>> takeWhile (== 'a') "abracadabra"
>> "a"
>> dropWhile (>6) [1..10]
>> [1,2,3,4,5,6,7,8,9,10]
```

→ because the condition failed for the first element.

→ Exercise : The fearful symmetry

1. myWords :: [Char] → [[Char]]

```
myWords [] = []
```

```
myWords x = [t1] ++ t2
    where t1 = takeWhile (/=' ') x
          t2 = myWords $ drop 1 (dropWhile (/=' ')
                                   x )
```

2. `myLines :: String → [String]`

```
myLines [] = []
myLines x = [t1] ++ t2
    where t1 = takeWhile (/='\n') x
          t2 = myLines $ drop 1 $ dropWhile (/='\n')
                                      x
```

3. `myBreaker :: String → char → [String]`

```
myBreaker [] _ = []
myBreaker x c = [t1] ++ t2
    where t1 = takeWhile (/=c) x
          t2 = myBreaker (drop 1 $
                          dropWhile (/=c) x )
               c
```

more generalised version of 1 & 2.

→ **list comprehensions**: means of generating a new list from a list or lists. They come directly from the concept of set comprehensions in mathematics, including similar syntax.

↓

They must have atleast one list called the generator, that gives the input for the comprehension, i.e. provides the set of items from which the new list will be constructed.

↓

Elements that satisfy certain predicates might be included or excluded. and have certain functions be applied to them before including them in the list.

↓

for example : [x^2 | x ← [1..10]]

designates the separation b/w i/p & o/p.

o/p function applied to the elements of the list x

input set, a generator list & a variable that represents the elements that get drawn from the list

*) Adding predicates : must evaluate to Bool. The elements that pass the predicate are sent as argument to the o/p fn, & the result is appended to the meant to be list. for example

[x^2 | x ← [1..10], even x]

*) multiple generators : we can have list comprehensions with multiple generator lists as well, but the only thing to keep in mind in those situations, is that the rightmost list is exhausted first and the elements in the o/p list will contain a cartesian product of the involved generator lists.

→ list comprehensions with strings

*) elem : tells us whether an element is present in

the given list or not. evaluates to bool.

*) List comprehension to remove all lowercase letters from a string.

```
f s = [ x | x ← s, elem x ['A'..'Z'] ]
```

→ Exercises : Square Cube

```
mysqr = [x^2 | x ← [1..5]]
mycube = [x^3 | x ← [1..5]]
```

1. `[ (x,y) | x ← mysqr, y ← mycube ]`

2. `[(x,y) | x ← mysqr, y ← mycube, x<50, y<50]`

3. `length$ tupleList`

→ Spines & nonstrict evaluation :

lists are recursive series of cons cells, a : [a] terminated by the empty list, but we want a way to visualize this structure in order to understand the ways lists get processed.

↓

When we talk about data structures in Haskell, particularly lists, sequences & trees, we talk about them having a spine. Spine is the connective structure that ties the collection of values together.

↓

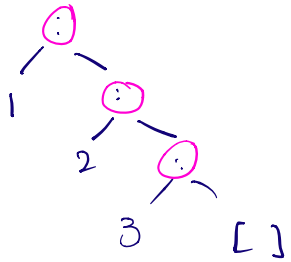
In the case of a list, spine is usually textually represented by the recursive cons (:) operators.

↓

Given the data: [1, 2, 3], we get a list that looks like 1 : 2 : 3 : []

or

1 : (2 : (3 : []))



↓

can cells contain the values. Because of this and the way non strict evaluation works, we can evaluate only the spine of the list without evaluating individual values. It is also possible to evaluate only part of the spine of a list & not the rest of it.

↓

Evaluation of the list in tree like representation is done down the spine. However, constructing the list, proceeds up the spine.

↓

Because Haskell's evaluation is nonstrict, the list isn't constructed until it's consumed - indeed, nothing is evaluated until it must be. Until it's consumed or we enforce strictness in some way there are a series of placeholders as a blueprint of the list that can be constructed when it's

needed.

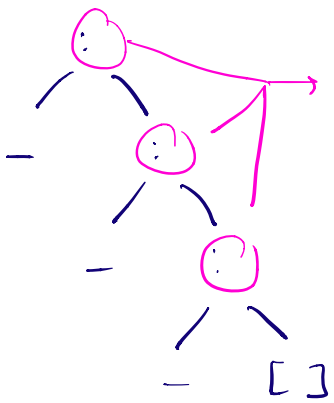
↓

'_' : to syntactically signify values we are ignoring & not evaluating. These represent the values contained by the Con cells.

↓

The spine is the recursive series of Cons constructors signified by (:)

↓



*) Sprint command in ghci to print variables. and see what has been evaluated already, with the underscore representing what hasn't been evaluated yet.

```
>> let blah = enumFromTo 'a' 'z'
>> :sprint blah.
blah = _
```

→ indicates that blah is totally unevaluated.

forcing evaluation of blah by operating on it using take function

```
>> take 1 blah
"a"
>> :sprint blah
blah = 'a' : _
```

→ evaluated a cons cell : and the first value 'a'.

```
>> take 2 blah
"ab"
>> :sprint blah
blah = 'a' : 'b' : _
```

→ the first cons cells and values were already forced, the next values are forced here.

*) The length function is only strict in the spine, meaning it only forces evaluation of the spine of the list, not the values.

→ Spines are evaluated independently of values : values in Haskell get reduced to weak head normal form by default. By 'normal form', it means that the expr. is fully evaluated. Weak head normal form means the expr. is only evaluated as far as necessary to reach a data constructor.

↓

for an expr. in WHNF, further evaluation may be possible once another argument is provided. If no further i/ps are possible, then it is in WHNF & NF as well.

↓

(1, 2) : WHNF & NF → fully evaluated

↓

Anything in NF is also in WHNF.

(1, 1+1) : WHNF → (+) applied to its argument can be evaluated, but hasn't been yet.

\x → x * 10 : WHNF & NF → normal form, since '*' cannot be reduced further till the outer x has been applied.

↓

with nothing further to reduce, it is normal form

An expr. cannot be in NF or WHNF, if the outermost part of the expr is not a data constructor

*

"Papu" ++ "chan" : nothing → outermost component of the expr is a fn. (++), whose arguments are completely applied but hasn't been evaluated.

(1, "Papu" ++ "chan") : WHNF

*) When we define a list and define all its values, it is in NF and all its values are known, nothing left to evaluate at that point.

```
>> let num :: [Int]; num = [1, 2, 3]
>> :sprint num
num = [1, 2, 3]
```

*) When we construct lists through ranges or fns. the lists are in WHNF, but not NF. The compiler only evaluates the head or first node of the graph, but just the cons constructor, not the value or rest of the list it contains.

*) The evaluation that we say earlier on the list from 'a' to 'z' is an example of WHNF evaluation, since the list is created by a range & needs to evaluate as far as it needs to.

*) Evaluating to normal form would've meant recursing through the entire list, forcing not only the entire spine but also the values each con cell contains.

*) functions that are spine strict can force complete spine evaluation, even if they don't force evaluation of each value. Pattern matching is strict by default, so pattern matching on cons cells can mean forcing spine strictness if your function doesn't stop recursing the list. It can evaluate the spine only or the spine as well as the values that inhabit each cons cells, depending on context.

*) On the other hand, length is strict in the spine but not the values. This can be checked by

making one of the values of the list equal to bottom.

```
>> let x = [1, undefined, 3]
>> length x
   3
```

*) Our own length function,

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

(_) used to ignore values in our arguments or that are part of a pattern match.

We can't bind on the right.

*) sum fn. is strict both for spine & values. First the '+' operator is strict in both its arguments, so that will force evaluation of the values and the mySum xs. Therefore mySum will keep recursing until it hits the empty list & must stop. Then it will start going back up the spine of the list, summing the inhabitants as it goes. Looks something like this.

```
>> sum [1..5]
1 + (2 + (3 + (4 + (5 + 0))))
1 + (2 + (3 + (4 + 5)))
```

1 + (2 + (3 + 9))

1 + (2 + 12)

1 + 14

15

Haskell evaluation Strategies takes time And practice to get on top off & will be encountered multiple times throughout the book. Better to approach it in stages.

>> take 1 $ filter even [1, 2, 3, undefined]

[2]

>> let x = [x^y | x ← [1..5], y ← [2, undefined]]

>> take 1 x

→ Transforming lists of values: In truth, in part because of the non-strict evaluation in Haskell, we tend to use higher order fns. for transforming data rather than manually recursing over and over.

↓

fn. applied uniformly to all values in a list. map or fmap fns. can be used to access this functionality. map used with [] whereas fmap is defined in a typeclass named functor and can be applied to data other than lists. fmap can be used for [] as well.

↓

>> map (+1) [1, 2, 3, 4]

[2, 3, 4, 5]

```
>> :t map
map :: (a -> b) -> [a] -> [b]
>> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

*) Here's how map is defined in base:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

↓ the application of map.

```
map (+1) [1,2,3]
map (+1) (1:(2:(3:[])))
```

↓

```
(+1) 1 :
    map (+1) (2:(3:[]))
```

↓

```
(+1) 1 :
    (+1) 2 :
        map (+1) (3:[])
```

↓

```
(+1) 1 :
    (+1) 2 :
        (+1) 3 :
            map (+1) []
```

↓ base case gets triggered.

```
(+1) 1 :
    (+1) 2 :
        (+1) 3 : []
```

→

```
2 : 3 : 4 : []
```

↑

```
2 : 3 : (+1) 3 : []
```

↑

```
2 : ((+1) 2 : ((+1) 3 : []))
```

↑ reduction

*) These representations do not account for nonstrict evaluation. Crucially, map doesn't traverse the whole list & apply the function immediately. The function is applied to the values that are forced out of the list one by one. example

```
>> take 2 $ map (+1) [1, 2, undefined]
[2, 3]
```

*) Strictness doesn't proceed only outside-in. We can have lazily evaluated code (eg. map) wrapped around a strict core (+). In fact, we can choose to apply laziness and strictness in how we evaluate the spine or the leaves independently. Common mantra for performance sensitive code in Haskell is, "lazy in the spine, strict in the leaves".

*) We can use map & fmap with other fns and list types as well.

```
map (take 3) [[1..5], [1..5], [1..5]]
```

*) if-then-else over a list using an anonymous fn.

```
map (\x → if x >= 3 then (-x) else (x)) [1..10]
```

→ Filtering lists of values: takes a list as i/p and returns a new list consisting solely of the values in the i/p list that meet a certain condition. example

filter even [1..10] → the list to work with

the predicate

```
filter :: (a → Bool) → [a] → [a]
filter _ [] = []
filter pred (x:xs)
  | pred x    = x : filter pred xs
  | otherwise = filter pred xs
```

Again iterating over the fact that a new list is being created, rather than mutating over the previous one

→ Zipping lists → means of combining values from multiple lists into a single list. Related fns. like zipWith allows us to use a combining fn. to produce a list of results from two lists.

```
>> :t zip
zip :: [a] → [b] → [(a,b)]
>> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
```

*) one thing to note is that zip stops as soon as one of the lists runs out of values. i.e zip proceeds until the shortest list ends.

```
>> zip [1,2] [4,5,6]
[(1,4),(2,5)]
```

*) unzip can be used to recover the lists as they were in their original form. Returns a tuple of lists. Obviously information can be lost in this process.

*) zipwith can also be used to create an output list by applying the fn. to the values in the i/p lists.

```
zipwith :: (a → b → c)
        → [a] → [b] → [c]
```

```
>> zipwith (+) [1, 2, 3] [10, 11, 12]
[11, 13, 15]
>> zipwith (==) ['a' .. 'f'] ['a' .. 'm']
```

Data.Char → import in source file before proceeding

1. :t isUpper :: Char → Bool
   :t toUpper :: Char → Char

2. filter (isUpper) "HbEfLrLxO"

3. capitalCase :: String → String
   capitalCase (x : xs) = toUpper x : xs

4. capitalCaseRecursive :: String → String
   capitalCaseRecursive [] = []
   capitalCaseRecursive (x : xs) = toUpper x : capitalCaseRecursive xs

5. capitalHead :: String → Char

capitalHead = toupper . head

26 characters of the english alphabet

→ Caesar cipher

↓

write fns to encrypt & decrypt text using Caesar's cipher

ord , chr

ord → unicode to int

chr → int to unicode.

a e d z v - 23

```
encrypt :: String → Int → String
encrypt [] _ = []
encrypt (x:xs) p = (shift x p) : (encrypt xs p)

shift :: Char → Int → Char
shift x p = chr $ (((ord x - 97) + rp) `mod` 26) + 97
        where rp = mod p 26

decrypt :: String → Int → String
decrypt xs p = encrypt xs (-p)
```

---

① myOr :: [Bool] → Bool

```
myOr [] = False
myOr (x:xs) = x || myOr xs
```

② myAny :: (a → Bool) → [a] → Bool

```
myAny f (x:xs) = f x || myAny f xs
```

③ myElem :: Eq a ⇒ a → [a] → Bool

myElem _ [] = False

myElem y (x:xs) = (x == y) || myElem y xs

myElem' :: Eq a ⇒ a → [a] → Bool

myElem' x xs = any (x ==) xs

any: if any element satisfies the predicate in a foldable structure

④ myReverse :: [a] → [a]

myReverse [] = []

myReverse (x:xs) = myReverse xs ++ [x]

⑤ squish :: [[a]] → [a]

squish [] = []

squish (x:xs) = x ++ squish xs

= "a" "b" "c"

abcd

"a", "b", "c", "d"

a ++ ["b", "c", "d"]

"a" ++ "bcd"

⑥ squishMap :: (a → [b]) → [a] → [b]

squishMap (\x → "wo" ++ [x] ++ "hoo!") "123"

the applied function returns a list & that needs to be concatenated

↓

it's the function's responsibility to return the list

```
squishMap _ [] = []
squishMap f (x:xs) = f x ++ squishMap f xs
```

⑦ `squishAgain :: [[a]] → [a]`

```
squishAgain x = squishMap (/x → x) x
```

⑧ `myMaximumBy :: (a → a → Ordering) → [a] → a`

```
myMaximumBy _ [] = []
myMaximumBy f (x:xs) = f x (myMaximum f xs)
```