7️⃣

Week 7: Functions

Defining

There are two ways to define functions in Haskell.

Conditionals

```
signum :: Int → Int
signum n =  if n < 0
                then -1
            else
                if n == 0
                    then 0
                else 1
```

Guarded Equations

```
abs n | n > 0 = n
      | n == 0 = n
      | otherwise = -n
```

Pattern Matching

```
not :: Bool → Bool
not False = True
not True = False
```

```
(&&):: Bool → Bool → Bool
True && True = True
True && False = False
```

```
False && True = False
False && False = False

True && True = True
_ && _ = False

False && _ = False
True && b = b
```

List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called “cons” that adds a new element to the start of a list.

ℹ️ [1, 2, 3] actually means 1:(2:(3:[]))

The cons operator can also be used in patterns, in which case it destructs a non-empty list.

```
head :: [a] → a
head (x:_) = x

tail:: [a] → [a]
tail (_:xs) = xs
```

Lambda Expression

A function can be constructed without giving it a name by using a lambda expression.

ℹ️ λx → x+1 is equal to \x → x + 1 in Haskell.

Why are they useful?

Lambda expressions can be used to give a formal meaning to functions defined using **currying**.

```
add x y = x + y
-- actually means --
add = \x → (\y → x+y)
```

Lambda expressions are also useful when defining functions that return functions as results.

```
compose f g x = f (g x)
compose f g = \x → f (g x)
```

Recursion

Why is recursion useful?

1. Some functions, such as factorial, are simpler to define in terms of themselves.
2. Properties of functions defined using recursion can be proven using the simple but powerful mathematical technique of induction.

Lists and Recursion

Recursion can also be done using lists.

Multiplication

```
product :: [Int] → Int
product [] = 1
product (x:xs) = x * product xs
```

Quick Sort Algorithm

The quick sort algorithm for sorting a list of integers can be specified by the following two rules:

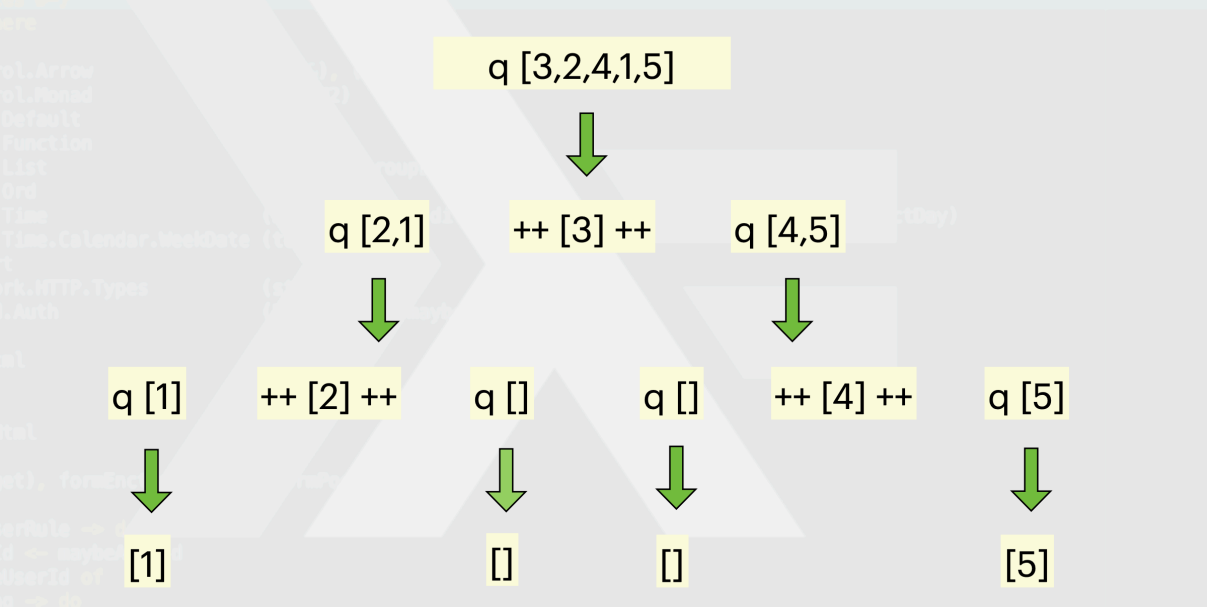
1. The empty list is already sorted.

2. Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

```
qsort :: [Int] → [Int]
qsort [] = []
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
                    ++ [x] ++
                    qsort [b | b ← xs, b > x]

-- NOTE: x:xs splits the list into head : rest
--       so this is why the other qsort lists work without using the head
--       which is added in ++ [x] ++
```

> ❗ This is probably the simplest implementation of quick sort in any programming language.



High Order Functions

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

```
twice :: (a → a) → a → a
twice f x = f (f x)
-- Takes a function as an argument
```

Capabilities

1. **Common programming idioms**, such as applying a function twice, can be encapsulated as general purpose higher-order functions.
2. **Special purpose languages** can be defined using higher-order functions, such as for list processing, interaction, or parsing.
3. **Algebraic properties** of higher-order functions can be used to reason about programs.

Composition

Composition Operator '.'

```
(f . g) x = f (g x)
-- f . g x not the same as f (g x)
-- Not all pairs can be composed output type of g must be input type of f
```

```
(.) :: (b → c) → (a → b) → (a →c)
-- input of f and output of g are of the same type: b.
-- f . g has input type a, same as g, and output type c, same as f.
```

ℹ️ Composition is associative: **f . (g . h) = (f . g) . h**

Forward Composition '>.>'

Order of composition is significant (f . g) means first apply g then apply f. It possible to define an operator that applies functions in the opposite order.

```
(>.>) :: (a → b) → (b → c) → ( a → c)
g >.> f = f . g
```

Application Operator '$'

Application of function f to argument e, f e, can also be explicit, f $ e. It is possible to use the application operator as an alternative to parentheses

```
flipV (flipH (rotate horse))
-- becomes
flipV $ flipH $ rotate horse
```

It can also be used as a function

```
zipWith ($) [sum, product] [[1,2], [3,4]]
```

Application and Composition

Suppose f has type Integer → Bool

- f . x means f composed with x, so x must have type s → Integer for some type s.
- f x means f applied to x so x must be of type Integer.
- f $ x also means f applied to x so x has type Integer

Map

The higher-order library function called map applies a function to every element of a list.

ℹ️ map :: (a → b) → [a] → [b]

```
> map (+1) [1,3,5,7] -- +1 is the function here to be applied to the list [2,4,6,8]
```

Definitions

Using list comprehension:

```
map f xs = [f x | x ← xs]
```

Using recursion:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Filter

The higher-order library function filter selects every element from a list that satisfies a predicate.

ℹ️ filter :: (a → Bool) → [a] → [a]

```
> filter even [1..10]
[2,4,6,8,10]
```

Definitions

Using list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Using recursion:

```
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
  | otherwise = filter p xs
```

General Definition

A number of functions on lists can be defined using the following simple pattern of recursion.

```
f [] = v
f (x:xs) = x Ω f xs
```

```
-- where f maps the empty list to a value v,
-- and any non-empty list to a function Ω applied to its head and f of its tail.
```

Examples

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs

and [] = True
and (x:xs) = x && and xs
```

Fold Right

The library function **foldr** encapsulates the simple pattern of recursion on the example above with the function Ω and value *v* as parameters.

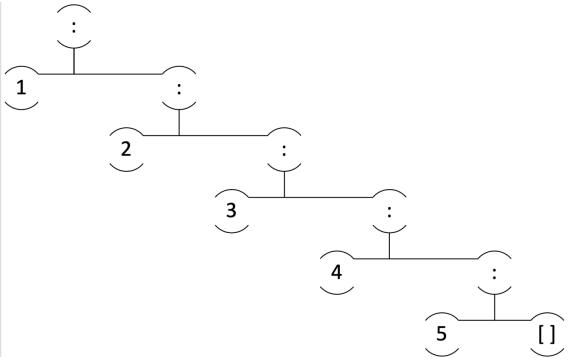
```
-- Examples
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
```

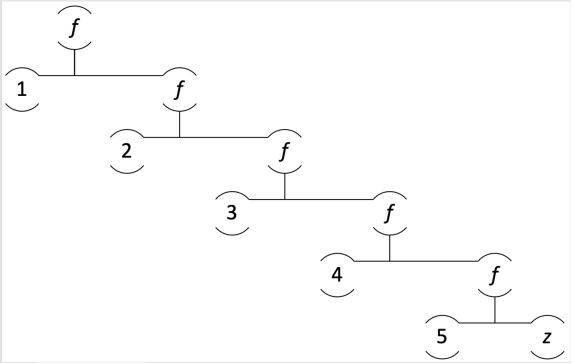
Type Definition

```
ℹ️ foldr :: (a → b → b) → b → [a] → b
```

Implementation

```
foldr Ω v [] = v
foldr Ω v (x:xs) = x Ω foldr (Ω) v xs
```





Example

```
sum [1,2,3]
> foldr (+) 0 [1,2,3]
> foldr (+) 0 (1:(2:(3:[])))
> 1 + (2 + (3 + 0))
> 6
```

Example - Combining

TASK: Sum of squares of positive integers in a list

List Comprehension

```
f :: [Int] → Int
f xs = sum [x*x | x←xs, x>0]
```

Recursion

```
f :: [Int] → Int
f [] = 0
f (x:xs) | x > 0 = (x*x) + f xs
         | otherwise = f xs
```

High Order Function

```
f :: [Int] → Int
f xs = foldr (+) 0 (map sqr (filter pos xs))
    where
    sqr x = x*x
    pos x = x > 0
```