

Syntax in function.

1. Pattern matching → consists of specifying patterns to which some data should conform and then checking to see if it does.

Deconstructing the data according to these patterns.



when defining functions, we can define separate fn. bodies for different patterns

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```



Factorial fn:

factorial :: (Integral a) \Rightarrow a \rightarrow a

factorial 0 = 1

factorial n = n * factorial (n-1) \rightarrow recursion



↓
important concept
in Haskell.

To remember : when making patterns, we should always include a catch-all pattern so that our program doesn't crash if we get some unexpected o/p.



pattern matching can also be applied on tuples. For example take two tuples of two size and then adding their parts separately to return a tuple.

↓

add vectors $\therefore (\text{Num} a) \Rightarrow (a, a) \rightarrow (a, a)$

add vectors $(x_1, y_1)(x_2, y_2) = (x_1 + x_2, y_1 + y_2)$

↓

writing a fst , and analogy for 3 elements vectors.

first $\therefore (a, b, c) \rightarrow a$

first $(x, -, -) = x$

second $\therefore (a, b, c) \rightarrow b$

second $(-, y, -) = y$

third $\therefore (a, b, c) \rightarrow c$

third $(-, -, z) = z$

↓

pattern matching on list comprehensions.

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

↓

lists themselves can be used in pattern matching .

we can match with the empty list [] or any pattern that involves `:' and empty list.

↓

But since [1, 2, 3] is syntactic sugar for 1 : 2 : 3 : [], we can use the former pattern as well .

↓

A pattern like x : xs binds the head of the list to x

and tail to xs.

↓

used a lot, especially for recursive functions.

but patterns that have `:` in them only match against lists of length 1 or more.

↓

wanting to bind the first three elements to variables

$a : b : c : \dots$ → will only match for lists with
more than 3 or equal to 3
elements.

→ own implementation of head function

head' :: [a] → a

head' [] = error "Can't call head on an empty list,
dummy!"

head' (x : _)

↓

need to surround with ()

takes a string
and generates
a runtime
error

↓

causes the program
to crash.

→ function to report some of the first elements of a list

tail :: (Show a) ⇒ [a] → string

tail [] = "The list is empty"

tail (x : l) = "List has only one element" ++ show x

tail (x : y : l) = "List has only two elements" ++ show x

↙

+ show y

tell $(x : -) =$ "list has many elements" ++ show x

could have been written as $[x, y]$

→ implementing our own length function

$\text{length}' :: (\text{Num } b) \Rightarrow a \rightarrow b$

$\text{length}' [] = 0$

$\text{length}' (-, xs) = 1 + \text{length}' xs$

} again using recursion.

→ implementing our own sum function

$\text{sum}' :: (\text{Num } a) \Rightarrow [a] \rightarrow a$

$\text{sum}' [] = 0$

$\text{sum}' (x : xs) = x + \text{sum}' xs$

→ patterns is a concept in Haskell wherein we can keep a reference to the whole argument which was being deconstructed due to the pattern matching.

↓

Done by placing @ in front of a pattern. for instance
the pattern $xs @ (x : y : xs)$

↓

$\text{capital} :: \text{String} \rightarrow \text{String}$

can't be used for pattern matching

$\text{capital} :: \text{String} = \text{"empty string"}$

$\text{capital all} @ (x : xs) = \text{"The first letter of " } ++ \text{all } ++ \text{" is " } + [x]$

↑

2. Guards: way of testing whether some property of a value (or several of them) are true or false.



↓
very similar to if construct

↓

bmiTell :: (RealFloat a) \Rightarrow a \rightarrow String

bmiTell bmi

① bmi ≤ 18.5 = "String 1"

① bmi ≤ 25.0 = "String 2"

① bmi ≤ 30.5 = "String 3"

guards

pipes that follow
a function's name
and its parameters.

① otherwise String 4

The last guard to cover all cases.

catch-all guard

→ basically a boolean expression. The function body corresponding to it is used if it evaluates to true.

↓

if all the guards of a function evaluate to false

(and we haven't provided an otherwise),
evaluation falls through to the next pattern

↓

If no suitable pattern or guard is found, an error is thrown.

↓

Works with multiple parameters as well
individual pattern matching for each parameter.

→ compare function using guards

compare' :: (Ord a) \Rightarrow a \rightarrow a \rightarrow Ordering

a `compare'` b

| a > b = LT

| a == b = EQ

| otherwise = GT

functions can be
defined infix as
well.

3. where : names and functions defined with 'where' are visible across the guards and give us the advantage of not having to repeat ourselves.

```
bmiTell :: (RealFloat a)  $\Rightarrow$  a  $\rightarrow$  a  $\rightarrow$  String
bmiTell weight height
| bmi <= skinny = "You're underweight, you emo, you!"
| bmi <= normal = "You're supposedly normal. Pfffft, I bet you're ugly!"
| bmi <= fat = "You're fat! Lose some weight, fatty!"
| otherwise = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
    skinny = 18.5
    normal = 25.0
    fat = 30.0 → alignment is important
```

prevents the namespace from getting polluted.

```
initials :: String  $\rightarrow$  String  $\rightarrow$  String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
where (f:_ ) = firstname
      (l:_ ) = lastname → pattern matching can be done here
```

```
calcBmis :: (RealFloat a)  $\Rightarrow$  [(a, a)]  $\rightarrow$  [a]
calcBmis xs = [bmi w h | (w, h)  $\in$  xs]
where bmi weight height = weight / height ^ 2 → function in where block.
```

gets a list of heights and weights and returns a list of bmis.

where bindings can also be needed. Common practice to make a function and define some helper function in its where clause and then give it to those functions as well, with its own where clause.

4. Let . → similar to where bindings → let bindings let us bind to variables anywhere and are expressions themselves.

cylinder := (RealFloat a) ⇒ a → a → a

cylinder r h =

let sideArea = $2\pi r \times h + \pi r^2$ } let bindings
in topArea = πr^2
in sideArea + 2 * topArea . } expression.

↓

format : let <binding> in expression

let : expressions themselves

where: syntactic constructs.

[let square x = x * x in (square 5, square 3, square 4)]