Syntax in functions.

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) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

→ 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.

↓

addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)

addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

↓

writing a fst, and analogy for 3 elements vectors.

first :: (a, b, c) -> a

first (x, _, _) = x

second :: (a, b, c) -> b

second (_, y, _) = y

third :: (a, b, c) -> 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 syntatic 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 : zs` → 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 : _) = 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

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x : []) = "List has only one element" ++ show x
tell (x : y : []) = "List has only two elements" ++ show x
```

```
tell (x:_) = "list has many elements" ++ show x ++ show y
```

could have been written as [x, y]

→ implementing our own length function

```
length' :: (Num b) => a -> b
length' [] = 0
length' (_,xs) = 1 + length' xs
```

} again using recursion.

→ implementing our own sum function

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + 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)

↓

```
capital :: String -> String
capital "" = "empty string"
capital all@(x:xs) = "The first letter of" ++ all ++ " is " ++ [x]
```

can't be used for pattern matching

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) => a -> String
bmiTell bmi
    | bmi <= 18.5 = String 1
    | bmi <= 25.0 = String 2
    | bmi <= 30.5 = String 3
    | otherwise   = String 4
```

guards

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

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

otherwise — catch-all guard

The last guard to cover all cases.

↓

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) => a -> a -> Ordering

a `compare'` b
    | a > b  = GT
    | a == b = EQ
    | otherwise = LT
```

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) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, 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 -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
  where (f:_) = firstname
        (l:_) = lastname
```

→ pattern matching can be done here

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- 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 nested. 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 * r * h * pi
        topArea = pi * r^2
    in sideArea + 2 * topArea
```

let bindings

expression.

↓

format : let <binding> in expression

let : expressions themselves

where : syntatic constructs.

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

↓

to bind several variables inline, then we need to separate them using semicolons.

(let a = 100; b = 200; c = 300 in a*b*c, let foo = "Hey "; bar = "there!" in foo ++ bar)

↓

pattern matching can also be used with let bindings

```
(let (a,b,c) = (1,2,3) in a+b+c) * 100
```

↓

let bindings can also be used in list comprehensions.

```
calcBmis :: (RealFloat a) => [(a,a)] -> [a]
```

↓ list of tuples of a ↓ list of a

```
calcBmis xs = [bmi | (w,h) <- xs, let bmi = w/h^2]
```

↓

visible to the output fn, and the part after the binding.

```
calcBmis xs = [bmi | (w,h) <- xs, let bmi = w/h^2,
                                  bmi >= 25]
```

?

bmi can't be used here, since defined before the binding

↓

in is not used here, because the visibility is predefined here.

Though let can be used in a predicate ← and the names would be visible only to that predicate.

5. Case expressions: about taking a variable and then executing blocks of code for specific values of that variable. We have already handled this for function parameters directly pattern matched and then accordingly code being executed. This can be thought of as an extension, since this can be done anywhere in the function, rather than only at the start of the function definition

```
head' :: [a] → a
head' []  = error ""
head' (x:_) = x
```

↓ implemented through case

```
head' :: [a] → a
head' xs = case xs of [] → error ""
                      (x:_) → x
```

↓

```
case expression of pattern → result
                   pattern → result
                   pattern → result
                   ...
```