Pattern Matching:

- **Pattern matching** means that you write down a literal at the place where you're supposed to write the parameter(s).
- E.g.

```
factorial :: Int -> Int
factorial 0 = 1 -- An example of pattern matching.
factorial n = n * factorial(n - 1)
```

```
*Main> factorial 5
120
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci 0 = 0 -- Example of pattern matching.
fibanocci 1 = 1 -- Example of pattern matching.
fibanocci n = fibanocci(n-1) + fibanocci(n-2)
```

```
*Main> fibanocci 5
8
```

Case Expressions:

- Allows us to write control flows on data types.
- Matches from top to bottom.
- **Note:** The pattern **_** means match anything.
- Syntax:

```
case <expr> of
  <pattern1> -> <result1>
  <pattern2> -> <result2>
  ...
  <patternN> -> <resultN>
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci x = case x of
    0 -> 0
    1 -> 1
    n -> fibanocci n1 + fibanocci n2
        where
            n1 = n - 1
            n2 = n - 2
```

```
*Main> fibanocci 5
5
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci x = case x of
    0 -> 0
    1 -> 1
    n -> let
        n1 = n - 1
        n2 = n - 2
        in
            fibanocci n1 + fibanocci n2
```

```
*Main> fibanocci 5
5
```

If statements:

- Syntax:

  if (condition)
  then (value)
  else if (condition)
  then (value)
  else
  (value)

- E.g.

```
factorial :: Int -> Int
factorial n =
    if n < 2
        then 1
    else
        n * factorial(n-1)
```

```
*Main> factorial 5
120
```

- **Note:** The else if is conditional, but you must have the if and the else.

type, term, value:

Those two things are called:

```
f (x*2 + 1) :: Integer
^^^^^^^^^^^    ^^^^^^^
   term          type
```

One more example:

```
h . g :: Char -> Bool
^^^^^    ^^^^^^^^^^^^
term          type
```

- **Note: term** is also widely known as **expression**.

- **Note:** 5+4 is a **term**; the result of evaluating it, 9, is a **value**.
  I.e. **term** is your code and **value** is the result of the term.

Synthesis and Evaluation:

- **Synthesis** is how you write code.
- **Evaluation** is how the computer runs your code.
- For synthesis, using induction can help you write the code.
- E.g. Consider the factorial code below:

```
factorial :: Int -> Int
factorial 0 = 1 -- An example of pattern matching.
factorial n = n * factorial(n - 1)
```

Here is the mindset of how to write it:

WTP: For all natural n: Factorial n = n!

Base case:

WTP: Factorial 0 = 0!

Notice that 0! = 1, so if I code up Factorial 0 = 1, I get Factorial 0 = 0!.

Induction step:

Let natural n ≥ 1 be given.

Induction hypothesis: Factorial (n-1) = (n-1)!

WTP: Factorial n = n!

Notice that n! = n*(n-1)!

= n * Factorial (n-1) by I.H.

So if I code up Factorial n = n * Factorial (n-1), I get Factorial n = n!.

Here is the evaluation of factorial 3:

→ 3 * factorial(3 - 1)
→ 3 * factorial(2)
→ 3 * (2 * factorial(2 - 1))
→ 3 * (2 * factorial(1))
→ 3 * (2 * (1 * factorial(1-1)))
→ 3 * (2 * (1 * (factorial(0))))
→ 3 * (2 * (1 * 1))
→ 3 * 2
→ 6

Guards:

- Denoted by "|"
- We use | to say alternatively.

- E.g.

```
absolute :: Integer -> Integer
absolute x
    | x < 0 = (-x)
    | otherwise = x
```

```
*Main> absolute (-5)
5
```

- E.g.

```
factorial :: Int -> Int
factorial n
    |n == 0 = 1
    |otherwise = n * factorial(n-1)
```

```
*Main> factorial 5
120
```

Lists:

- Some types of lists are [Integer], [Bool], [] Integer, [] Bool, etc.
- An empty list is denoted as [].
- A list literal is denoted like [4, 1, 6].
  Note: Remember that Haskell makes lists in this way:
  (4 : (1 : (6 : ([])))) or 4 : 1 : 6 : []
  The parentheses are optional.
- Formally (recursive definition as in CSCB36): a list is one of:
  - []
  - <an item here> : <a list here>
- **Note:** These are singly-linked lists. These are not arrays.
- **Note:** Lists are immutable in Haskell.
- E.g. Insertion Sort:
  Strategy: Have a helper function insert.
  Take element e and list xs. xs is assumed to have been sorted in increasing order.
  Put e into the “right place” in xs so the whole is still sorted.
  E.g. insert 4 [1,3,5,8,9,10] = [1,3,4,5,8,9,10]

Here's the code:

```haskell
insert :: Integer -> [Integer] -> [Integer]

-- Structural induction on xs.

-- Base case: xs is empty. Answer is [e] aka e:[]
insert e [] = [e]  -- e : []

-- Induction step: Suppose xs has the form x:xt (and xt is shorter than xs).
-- E.g., xs = [1,3,5,8], x = 1, xt = [3,5,8].
-- Induction hypothesis: insert e xt = put e into the right place in xt.
insert e xs@(x:xt)
    -- xs@(x:xt) is an "as-pattern", "xs as (x:xt)",
    -- so xs refers to the whole list, and it also matches x:xt
    --
    -- If e <= x, then e should be put before x, in fact all of xs, and be done.
    -- E.g., insert 1 (10 : xs) = 1 : (10 : xs)

    | e <= x = e : xs

    -- Otherwise, the answer should go like:
    -- x, followed by whatever is putting e into the right place in xt.
    -- i.e.,
    -- x, followed by insert e xt (because IH)
    -- E.g., insert 25 (10 : xt) = 10 : (insert 25 xt)

    | otherwise = x : insert e xt

insertionSort :: [Integer] -> [Integer]
insertionSort [] = []
insertionSort (x:xt) = insert x (insertionSort xt)
```

E.g.

```
*Main> insertionSort [3,2,1]
[1,2,3]
```