# Basics

## Function Syntax

```haskell
createList :: Int -> Int -> [Int]
createList x y = [x, y]
```

The first line denotes the types for input and output. All the types listed before the last one represent the types of the arguments that the function takes, the last type represents the return type of the function.

The second line specifies how it transforms its inputs into an output.

## Recursive Function

```haskell
factorial :: Integer -> Integer
factorial 0 = 1  -- Base case
factorial n = n * factorial (n - 1)  -- Recursive case
```

## Modify List

Let's say we have a list of integers and we want to update the fourth element of the list to a new specified value.

```haskell
updateFourthElement :: [Int] -> Int -> [Int]
updateFourthElement list newValue =
    if length list < 4
    then list  -- Return the original list if it has less than 4 elements
    else take 3 list ++ [newValue] ++ drop 4 list
```

The function takes a list of integers and a new value. If the list's length is shorter than 4 then we simply return the list. Else we take the first 3 elements of the list (take 3 list) put in the new value next and then we wrap it up with the rest of the list at the end, by dropping the first 4 elements of the list (drop 4 list). This way we «changed» the immutable list by creating a new one with the desired changes.

This can be used for the maze solving problem when updating the maze's cell from . to * if visited.

## Pattern Matching

Pattern matching in Haskell allows you to define functions with multiple cases, where each case handles different inputs based on their structure and content. This feature simplifies decision-making in functions by directly matching inputs against specified patterns.

Example

```haskell
describeNumber :: Int -> String
describeNumber n = case n of
    0 -> "Zero"
    1 -> "One"
    _ -> "Some number"  -- '_' is a wildcard that matches any value

-- Example Usage:
-- describeNumber 0 returns "Zero"
-- describeNumber 5 returns "Some number"
```

Regarding the maze solver, pattern matching can help to handle different type of cells ('S', 'E', '#', '.') to then decide whether to continue exploring or terminating.

## Type Maybe

The Maybe type in Haskell is a simple yet powerful way to handle operations that might fail or cases where a value might be absent without using exceptions. It encapsulates an optional value; a value of type Maybe foo either contains a value of type foo (represented as Just foo), or it contains nothing (Nothing).

Example

```haskell
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing  -- If the denominator is zero, return Nothing.
safeDivide x y = Just (x `div` y)  -- Otherwise, perform the division and wrap
the result in Just.
```

The return type Maybe Int states that the function might not always return a straightforward integer, but rather it could return Nothing if the division cannot be performed (e.g., division by zero).

In your maze solver, Maybe is an ideal choice for the function that seeks a path through the maze. When the function is terminated we can then handle the results of the maze solver function like this:

```haskell
case solveMaze maze of
    Just solvedMaze -> do
        putStrLn "Path found! Here is the solved maze:"
        mapM_ putStrLn solvedMaze
    Nothing -> putStrLn "No path found."
```