

F# Tutorial  
**Pipe-Forward Operator**  
*February 12, 2018*

## 1 Syntax, variables, functions

### Key concepts:

1. Having a good text editor helps you code much easier.
2. (a) Once defined, a variable in F# cannot change value (unless "mutable" is used)  
(b) If you need an updated value, create a new one.
3. Different datatypes (e.g. integer and decimal-numbers) do not combine easily.
4. Defining and using functions in F# is slightly different from math notation/ other languages.
  - (a) F# automatically detects the type of the variables (e.g. integer, double, etc.) for a function.
  - (b) The variable types for a function will be enforced.

### 1.1 Setting Up

#### 1.1.1 Comments

You can use double-slash `//`, triple-slash `///`, or star-bracket `(* ..... *)` to make comments.

```
1 // These words are ignored.
2 /// These words are ignored.
3 (* These words are ignored. *)
4 let x = 1
5 let y = x + 5
```

#### 1.1.2 F# Interactive

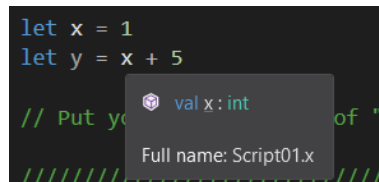
If you are using Visual Studio, you can run the code above by highlighting/selecting the code using your mouse, and press `ALT + ENTER`, or right-click and select **Execute** in **Interactive**.

INSERT PICTURE HERE.

### 1.1.3 Intellisense

If you are using Visual Studio or Visual Studio Code, you can put your mouse on top of the variable name `x` or `y`, and see that it is an `int` or integer.

This feature will help you identify what is each variable/function, and make coding easier for you.



## 1.2 Data Type

### 1.2.1 Common data types and printing

Some of the common types in F# are:

Keyword	Description	Print in output:
<code>int</code>	Integer	<code>%i</code>
<code>double</code> or <code>float</code>	Decimal numbers	<code>%f</code>
<code>string</code>	Words/Sentences	<code>%s</code>
<code>bool</code>	True/False	<code>%b</code>
-	Other objects	<code>%A</code> or <code>%O</code>

```
1 let name = "John"
2 let age = 21
3 let height = 170.5
4
5 printfn "My name is: %s" name
6
7 printfn "Name: %s. Age: %i. Height: %f." name age height
8
9 printfn "His height is: %.2f" height
10 // %.2f for showing two decimals.
```

Output:

```
1 // Output:
2 // My name is: John
3 // Name: John. Age: 21. Height: 170.500000
4 // His height is: 170.50
```

For example, in the second example, inside the string-format, there are `%s`, `%i`, `%f`. And so, we expect a string, integer, and decimal (in that order) after the string-format specification in order to completely print the result to the output console.

### 1.2.2 Equality and simple if-else

The `let ... = ...` combination is used to assigned a value to a variable. Other than this situation, the equal sign `=` is used for equality testing. `=`, `<>` are used for equality/inequality testing.

```
1 let valueToTest = 20
2 let isValueEqualToTwenty = (valueToTest = 20)
3
4 if isValueEqualToTwenty then
5     printfn "Yes, the value is Twenty"
6 else
7     printfn "No, the value is not Twenty"
8 // Output: "Yes, the value is Twenty"
9 ///////////////////////////////////////////////////
10
11 let inputUserName = "Jack"
12
13 if inputUserName = "John" then
14     printfn "Welcome back, John"
15 else
16     printfn "Access denied."
17 // Output: "Access denied."
```

In Java/C++, `==`, `!=` are used for comparison, and in Javascript, `===`, `!==` are used.

### 1.2.3 Immutability

In F#, variables are by default immutable/unchangeable. Once defined, the value of a variable cannot be changed. You can make a variable changeable/mutable using the keyword `mutable` and symbol `<-`, but this is highly discouraged. (If you use VisualStudio, then the color of the variable name will change color, warning you of potential mutable values)

```
// Warning: Do not use mutable value if possible!
//
// Using mutable value is a bad idea!
let mutable changableValue = 100
printfn "Original value is: %i" changableValue
// Output: "Original value is: 100"

changableValue <- 200
printfn "Updated value is: %i" changableValue
// Output: "Updated value is: 200"
```

If you try to update an immutable/unchangable value using `<-`, you will get an error.

```
// Uncomment the code below to see an error:
```

```
let immutableValue = 100
immutableValue <- 300
```

This value is not mutable. Consider using the mutable keyword, e.g. 'let mutable immutableValue = expression'.

## Benefit of immutable/unchangable values

Imagine the code below, with a mutable value `x`, and after thousands of lines of code later, you used `x`'s value again:

```
1 let mutable x = 100
2 //
3 // Thousands of lines of code later.....
4 // You have many lines of code in between.....
5 // It is hard to keep track.....
6 // Have you changed/updated x's value?
7 // Did you accidentally call any function that modify x?
8 // Can you guarantee x's value stay unchanged?
9 //
10 //
11 let y = x + 1
12 // What is the value of y?
13 //
14 // That depends on what happens between y's definition
15 // and x's definition.
```

---

On the other hand, if `x` is immutable/unchangable:

```
1 let x = 100
2 //
3 // Thousands of lines of code later.....
4 // You have many lines of code in between.....
5 // But because x is immutable/unchangable.....
6 // We can be sure that x stays constant.....
7 // And we can safely conclude that.....
8 //
9 let y = x + 1
10 // y = 101
```

Conclusion: Use immutable/unchangable value whenever possible. AVOID mutable/changable value whenever possible.

### 1.2.4 (+) Operator on the same type of variable

Integers, double, and string support the (+) operation:

```
1
2 let number1 = 40
3 let number2 = 55
4 let addTwoNumbers = number1 + number2
5
6 // Remark: "float" and "double" mean the same thing in F#.
7 let sqrtTwoApprox = 1.414
8 let piApprox = 3.1415926
9 let addTwoDecimals = sqrtTwoApprox + piApprox
10
11 let sentenceStart = "My school is "
12 let schoolName = "National University of Singapore"
13 let combinedSentence = sentenceStart + schoolName
```

However, you cannot add an integer with a decimal in F# directly using (+), and you cannot add/concatenate a string with a number directly using (+). If you use VisualStudio, then you may see an error similar to the one below.

```
////////////////////////////////////
// Cannot combine two different types using the "+" functions
// The code all below, if un-commented, will create error.

let addIntegerWithDecimal = 15 + 4.11
let combineStringWithInteger = "My age is: " + 21
```

The type 'int' does not match the type 'string'

Furthermore, some functions, like the square root `sqrt` and math exponent (`**`) only accepts decimal numbers:

```
1 let sqrtRootOfNine = sqrt 9.0
2 let twoToPowerOfFive = 2.0 ** 5.0
```

And it will cause error if you use them with integer input instead.

```
////////////////////////////////////
// Cannot combine two different types using the "+" functions
// The code all below, if un-commented, will create error.

let addIntegerWithDecimal = 15 + 4.11
let combineStringWithInteger = "My age is: " + 21
```

The type 'int' does not match the type 'string'

## 1.3 Functions

### 1.3.1 One variable functions

You can define functions using `let` followed by the inputs of your function.

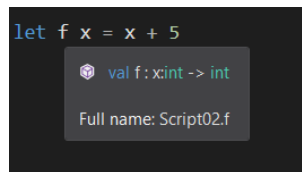
```
1 let f x = x + 5
2
3 let result1 = f 10
4 let result2 = f 20
```

Output:

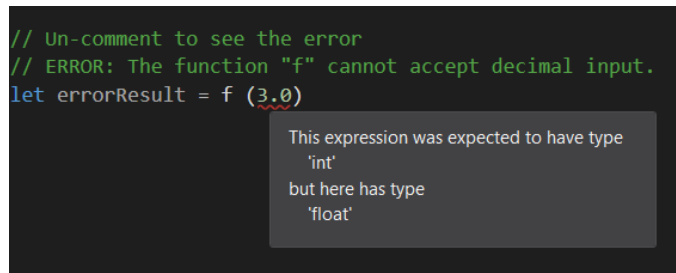
```
1 // val result1 : int = 15
2 // val result2 : int = 25
```

Notice the following:

1. To apply the function `f`, you do not need to use the math notation  $f(x)$ . You can apply the arguments by separating with a space.
2. If you hover your mouse on top of the function `f`, you will see that `f` is a function that accepts only integer `x` as the argument.



- (a) This is because in the function, `x` will be added (+) to the integer 5. We have seen before that we cannot use the symbol (+) to combine an integer with a decimal number directly. Hence, `x` has to be of type `int`.
- (b) As a consequence, if you try to input a decimal number to the function `f`, then it will fail:



3. As mentioned, F# automatically inferred that `x` is an integer. This is different from other languages (e.g. Java, C++) that needs you to specify the type of the variable (is it an integer? double? etc.)

So, you can spend less time on the tiny details (e.g. what is the variable type), and focus more on the correctness of your program.

Similarly, the following function accepts decimals/double only.

```
1 let DiscountFunc originalPrice = originalPrice * 0.8
2
3 let discountedPrice = DiscountFunc 399.99
4 printfn "New price: %.2f" discountedPrice
5 // Output: "New price: 319.99"
6
7 let anotherDiscount = DiscountFunc discountedPrice
8 printfn "New price: %.2f" anotherDiscount
9 // Output: "New price: 255.99"
```

Remark: The %.2f for printing 2 decimals.

This function does not accept integer values:

INPUT ERROR PICTURE HERE!

We need to convert integer to decimal (using double or float) before using the function.

```
1 let convertedPrice = double 100
2 let decimalResult = DiscountFunc convertedPrice
3 printfn "New price: %.2f" decimalResult
4 // Output: "New price: 80.00"
```

---

Similarly, the following function accepts strings only.

```
1 // Define a function for string.
2 let AddGreeting name =
3     "Hello " + name
4
5 let greeting1 = AddGreeting "John"
6 let greeting2 = AddGreeting "Mary"
```

Output:

```
1 // val greeting1 : string = "Hello John"
2 // val greeting2 : string = "Hello Mary"
```

And it will cause error if you try to input an integer value to this function:

INPUT ERROR PICTURE HERE!

Exercise: Write a function that calculates the area of a circle of radius  $r$ .

```
1 let CircleArea r =  
2     //  
3     // ... INSERT YOUR CODE HERE ...  
4     // Hint: Use      "System.Math.PI"
```

### 1.3.2 Two variable functions

You can define a function that takes in two variables:

```
1 let g x y = 3 * x + y  
2  
3 let result3 = g 3 1  
4 let result4 = g 10 2  
  
1 // val result3 : int = 10  
2 // val result4 : int = 32
```

Notice the following:

1. To apply the function `g`, you do not need to use the math notation  $g(x, y)$  with brackets and commas. This is different from other programming languages (e.g. Java, C++). You can apply the arguments by separating with a space.
2. If you hover your mouse on top of `g`, as seen in this picture:

INPUT PICTURE HERE!

You will see that the variables `x`, `y` need to be integers.

- (a) This is because in the function, `x` will be multiplied with 3, and then later added with `y`. As seen before, the addition and multiplication symbol (+), (\*) only combined numbers of the same type (integers with integers, double with double)
- (b) As a consequence, if you input decimals into the function, it will fail:

INPUT PICTURE HERE!

3. Again, you can spend less time typing out the details (i.e. what are the types of `x`, `y`? Integer? Double?) and focus more on making your program/algorithm works, and make yourself more productive (compared to other programming languages)



Similarly, the following function accepts two decimal numbers:

```
1 let CalculateNewBalance interestRate principal =
2     principal * (1.0 + interestRate)
3
4 let balance1 = CalculateNewBalance 0.05 100000.00
5 printfn "New Balance: %f" balance1
6 // Output: "New Balance: 105000.00"
7
8 let balance2 = CalculateNewBalance 0.03 5000.00
9 printfn "New Balance: %f" balance2
10 // Output: "New Balance: 5150.00"
```

And it will cause error if you try to change one of the input into integer.

INPUT PICTURE HERE!

### 1.3.3 Multivariable functions

```
1 let h x y z = 3 * x + 4 * y + 5 * z
2
3 // 3*3 + 4*4 + 5*5 = 50
4 let result5 = h 3 4 5
5
6 // 3*1 + 4*1 + 5*1 = 12
7 let result6 = h 1 1 1
```

Output:

```
1 // val result5 : int = 50
2 // val result6 : int = 12
```

### 1.3.4 Default integers for +, \*

If you use (+), (\*) with no other information available in your function (e.g. an appearance of a decimal, string, etc.), then F# will assume the function variables as integers.

```
1 let AddThree x y z = x + y + z
2 let addThreeResult = AddThree 5 6 7
```

If you hover your mouse on top of `AddThree`, then you see that all the inputs are inferred to be integers.

If you want this function to work for decimals, then you will need to annotate/manually add in the type for one of the variables:

```
1 let AddThreeCustom (x:double) y z = x + y + z
```

Here, we are explicitly saying that `x` is a double. And since `y`, `z` interacts with `x` using (+), we can infer that `y`, `z` are also doubles (and we do not need to explicitly label them as decimal/doubles)

## 1.4 Scoping

### 1.4.1 Indenting

You can use a `let` inside a `let`, i.e. you can define a variable inside a variable. For example:

```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence  
4  
5 let combinedSentence1 = AddFriend "Jack"
```

Output:

```
1 // combinedSentence1 : string = "Jack and Mary are friends"
```

Notice that the two lines immediately after the `AddFriend` function has some spaces in front of each line. This means that those two lines are accessible only inside the `AddFriend` function.

So, you cannot access the `endOfSentence` variable outside of the function. The following code will not work:

```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence  
4  
5 // ERROR: "endOfSentence" is not accessible outside of "  
6   AddFriend"  
7 let x = endOfSentence  
8 "ERROR: endOfSentence is not accessible outside of  
9   AddFriend"
```

### 1.4.2 Reuse variable name

By carefully using indenting/spacing, you can repeatedly use the same variable name, as long as the spacing/indenting is such that the variables do not cause conflict with each other.

```
1 let DrinkFunction person =  
2     let endOfSentence = " likes to drink coffee."  
3     person + endOfSentence  
4  
5 let EatFunction person =  
6     let endOfSentence = " prefers eating chocolate."  
7     person + endOfSentence  
8  
9 printfn "%s" (DrinkFunction "Jack")  
10 // Output:
```

```

11 // "Jack likes to drink coffee."
12
13 printfn "%s" (EatFunction "Jill")
14 // Output:
15 // "Jill prefers eating chocolate."

```

The `endOfSentence` inside these two functions will not cause conflict with each other.

### 1.4.3 From top to bottom

F# code are read from top to bottom. For example, look at the following code:

```

1 let a = 5
2
3 let f1 b =
4     a + b
5
6 let f2 b =
7     a + a + b
8
9 printfn "%i" (f1 10)
10 printfn "%i" (f2 10)

```

Notice that there are no spacing/indenting before `let a = 5` and the definition of `f1`, `f2`. These variables and functions are equally indented, and so the value of `a` is accessible from `f1`, `f2`

However, the following code below will not be accepted, because `a` is defined later/down lower in the code, but it is incorrectly used before it is defined (i.e. above it).

```

1 // ERROR: "a" is not yet defined.
2 let f1 b =
3     a + b
4 "ERROR!"
5 // ERROR: "a" is not yet defined.
6 let f2 b =
7     a + a + b
8 "ERROR!"
9 // ERROR: "a" is defined too late! It is used above.
10 let a = 5

```

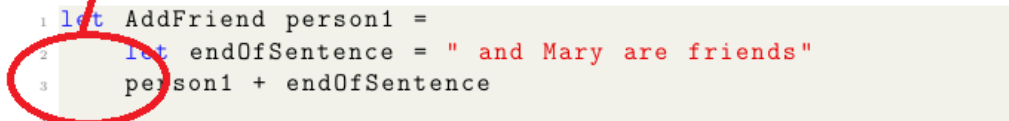
### 1.4.4 Warning: No TAB

In Python, you use `TAB` to indent the file. The `TAB` button will insert a special character.

However, in F#, you use blank spaces to do indenting. You should configure/adjust your IDE (e.g. VisualStudio, VisualStudioCode, etc.) so that it insert multiple blank spaces instead of a special character.

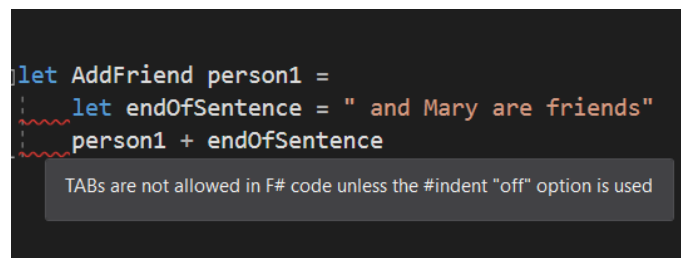
For example, the code below is indented using 4 spaces for the second and third line.

These are 4 blank spaces! Not the special character "TAB"



```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence
```

If you did not configure your IDE correctly, or if you copy-and-paste the special TAB character from another source (e.g. Notepad), then you may see the following error:



```
let AddFriend person1 =  
    let endOfSentence = " and Mary are friends"  
    person1 + endOfSentence
```

TABs are not allowed in F# code unless the #indent "off" option is used

## 1.5 Reset F# Interactive

Remember to reset your F# Interactive once in a while, so that you don't have too many previous variables (especially if you re-use the same variable names)

INSERT PICTURE HERE.

In Visual Studio, you can Right-click the interactive window, and select "Reset Interactive Session", or use the shortcut key CTRL + ALT + R

## 2 Pipe-forward

### Key Concept:

1. Coding in F# is similar to building LEGO.
  - Source: Scott Wlaschin
2. The output of one function is the input of the next function.

### 2.1 Introduction

F# has an operator, called the pipe-forward operator.

The definition of pipe-forward is:

```
1 let inline (|>) x f = f x
```

(The `inline` keyword is used to handle some special cases.) You do not need to worry about the definition. This operator is already implemented in F# by default.

### 2.2 Simple demonstration

Let us take a look at an example:

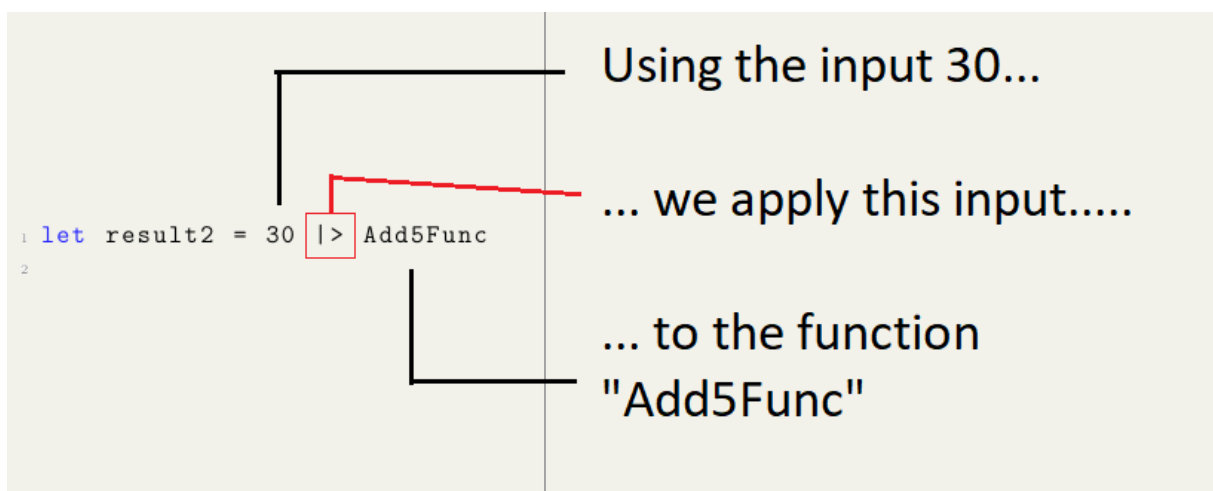
```
1 let Add5Func x = x + 5
2
3 let result1 = Add5Func 30
4 // val result1 : int = 35
```

Notice that the variable/input 30 is located after the function `Add5Func`.

However, with the new symbol `|>`, we can specify the variable/input first, and then the function that we want to apply it to.

```
1 let result2 = 30 |> Add5Func
2 // val result2 : int = 35
```

How this code should be interpreted is the following:



## 2.3 Why is this useful?

The reason why the symbol `|>` is useful is because it helps us to compose functions. Let's say that you are given these functions:

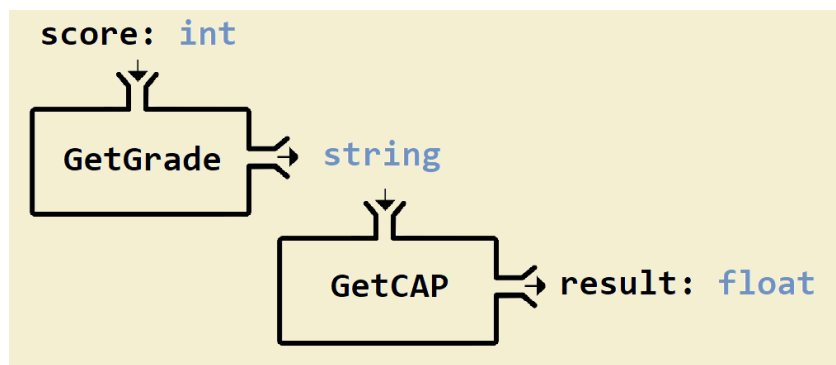
```
1 let GetGrade score =  
2   if score >= 90 then "A"  
3   else if score >= 70 then "B"  
4   else if score >= 50 then "C"  
5   else "D"  
6  
7 // For Singaporean University. (Maximum CAP 5.0)  
8 let GetCAP grade =  
9   if grade = "A" then 5.0  
10  else if grade = "B" then 4.0  
11  else if grade = "C" then 3.0  
12  else 2.0
```

Remark: In American universities, they use a maximum score/GPA of 4.0. In Singapore we use CAP 5.0.

We can take a look at the signatures of the functions:

```
1 GetGrade: int -> string  
2 GetCAP:   string -> float
```

So, we can use the result of the first function `GetGrade` as the input of a second function `GetCAP`.



```
1 let GetCAPfromScore1 score =  
2   let intermediateResult = GetGrade score  
3   let finalResult = GetCAP intermediateResult  
4   // return  
5   finalResult  
6  
7 let cap1 = GetCAPfromScore1 95  
8 let cap2 = GetCAPfromScore1 85
```

Output:

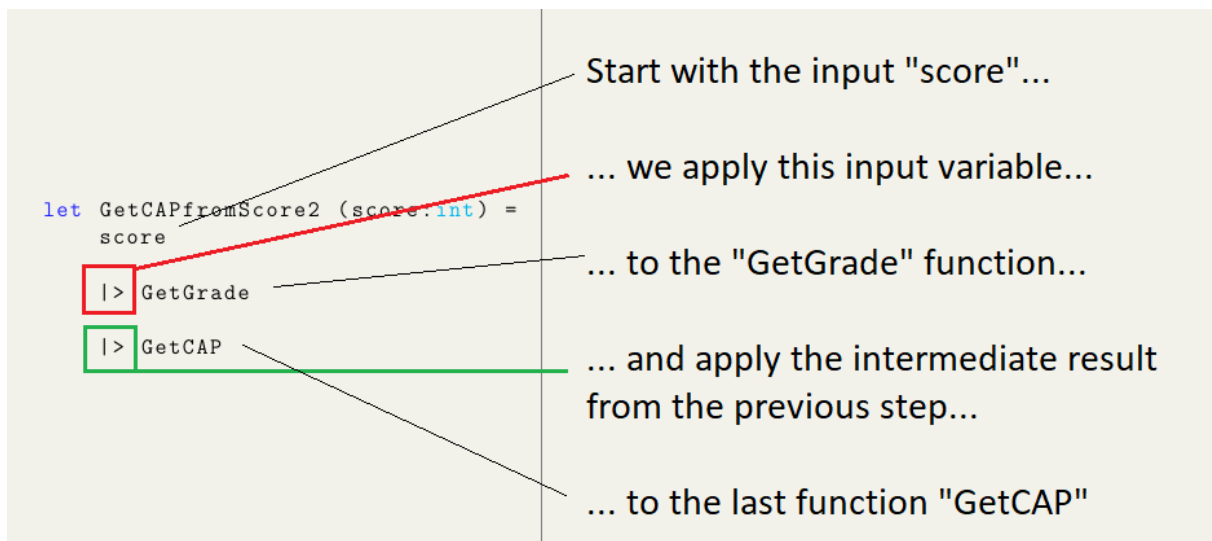
```
1 // val cap1 : float = 5.0
2 // val cap2 : float = 4.0
```

Notice that in the code above, we named out the intermediate steps/variables, i.e. `intermediateResult` and `finalResult`, even though it makes the code longer.

However, if we use the pipe-forward operator `|>`, we can simplify it as:

```
1 // GetGrade: int -> string
2 // GetCAP:      string -> float
3 let GetCAPfromScore2 score =
4     score
5     |> GetGrade
6     |> GetCAP
7
8 let cap3 = GetCAPfromScore2 95
9 // val cap3 : float = 5.0
```

How this code should be interpreted:



Remark: The code will not compile if we put the functions in the wrong order:

```
1 let CombinedFunction3Error score =
2     score // int
3     |> GetCAP // function: string -> float // ERROR!
4     |> GetGrade // function: int -> string // ERROR!
5 "ERROR!!!"
```

Because `score` is an `int`, but the function `GetCAP` only accepts `string` as input. Similarly, the intermediate result from `GetCAP` is `float`, but the function `GetGrade` only accepts `int`

## 2.4 More Examples

Let us consider another hypothetical example.

Imagine that you want to buy or sell a company's stock, based on the company's performance relative to the financial analyst's estimate.

1. F# has a built-in function, `List.average` to find the average of a list of numbers:

```
1 let average1 = List.average [1.0; 2.0; 3.0; 4.0; 5.0]
2 let average2 = List.average [80.0; 85.0; 90.0; 95.0;
    100.0]
```

2. You are provided another function, `GetPerformance`, that determines the condition of the company.

Assume that the actual profit of the company is \$ 6.0 billion for that year, then we say that the company is:

- OUTPERFORM: If actual profit exceed the analystEstimate by 5%
- UNDERPERFORM: If actual profit misses the analystEstimate by 5%
- NEUTRAL: If actual profit is within 5% of the analystEstimate

```
1 let GetPerformance analystAverageEstimate =
2     let actualProfit = 6.0
3     if actualProfit > analystAverageEstimate * 1.05
4         then "OUTPERFORM"
5     else if actualProfit < analystAverageEstimate * 0.95
6         then "UNDERPERFORM"
7     else
8         "NEUTRAL"
```

3. You are also provided another function, `GetNumSharesToBuy`, that determines how much additional shares to buy/sell depending on the company's condition:

- OUTPERFORM: Buy additional 1000 shares.
- UNDERPERFORM: Sell 1000 shares.
- NEUTRAL: Hold the same portfolio.

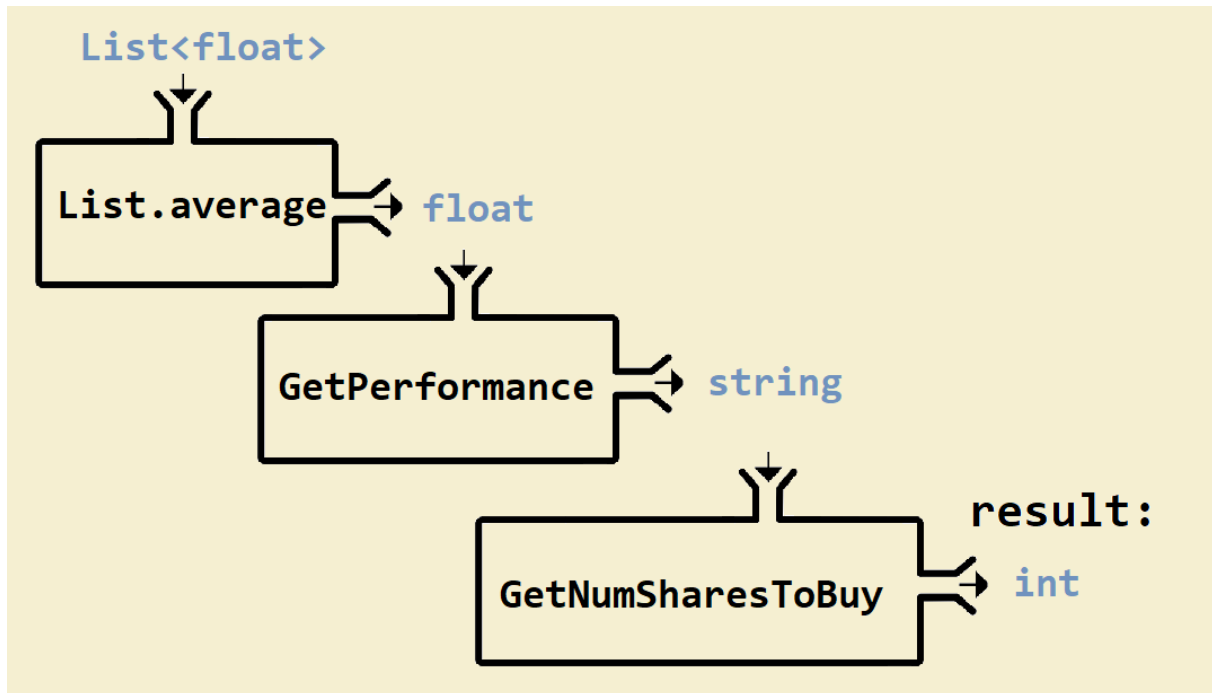
```
1 let GetNumSharesToBuy performance =
2     if performance = "OUTPERFORM" then
3         1000 // buy 1000 shares
4     else if performance = "UNDERPERFORM" then
5         -1000 // sell 1000 shares
6     else
7         0 // hold.
```



So, the function signatures are:

```
1 List.average :  
2   List<double> -> double  
3 GetPerformance :      double -> string  
4 GetNumSharesToBuy :   string -> int
```

In this carefully crafted example, notice that the result of the one function can act as the input to the other function.



So, we can combine them into a big function:

```
1 // Assume the profit is already known to be $6.0 billion,  
  and written in "GetPerformance"  
2 let GetNumSharesFromEstimate1 individualEstimates =  
3   let intermediateResult1 =  
4     List.average individualEstimates  
5  
6   let intermediateResult2 =  
7     GetPerformance intermediateResult1  
8  
9   let finalResult = GetNumSharesToBuy intermediateResult2  
10  // output  
11  finalResult
```

Notice that the code above uses a lot of temporary variables `intermediateResult1`, etc. which makes the code unnecessarily longer.

Usage example:

1. In this example, the actual profit (6.0 billion) exceeds all the financial analyst's prediction, which means this is good news.

```
1 let numShares1 = GetNumSharesFromEstimate1 [4.0; 5.0; 3.0;
2         2.0; 2.5]
3 printfn "Number of shares to buy(+) or sell(-): %i"
4         numShares1
// Output:
// "Number of shares to buy(+) or sell(-): 1000"
```

2. In this example, the actual profit (6.0 billion) misses all the financial analyst's prediction, which means this is bad news.

```
1 let numShares2 = GetNumSharesFromEstimate1 [8.0; 7.0;
2         10.0; 12.0; 10.5]
3 printfn "Number of shares to buy(+) or sell(-): %i"
4         numShares2
// Output:
// "Number of shares to buy(+) or sell(-): -1000"
```

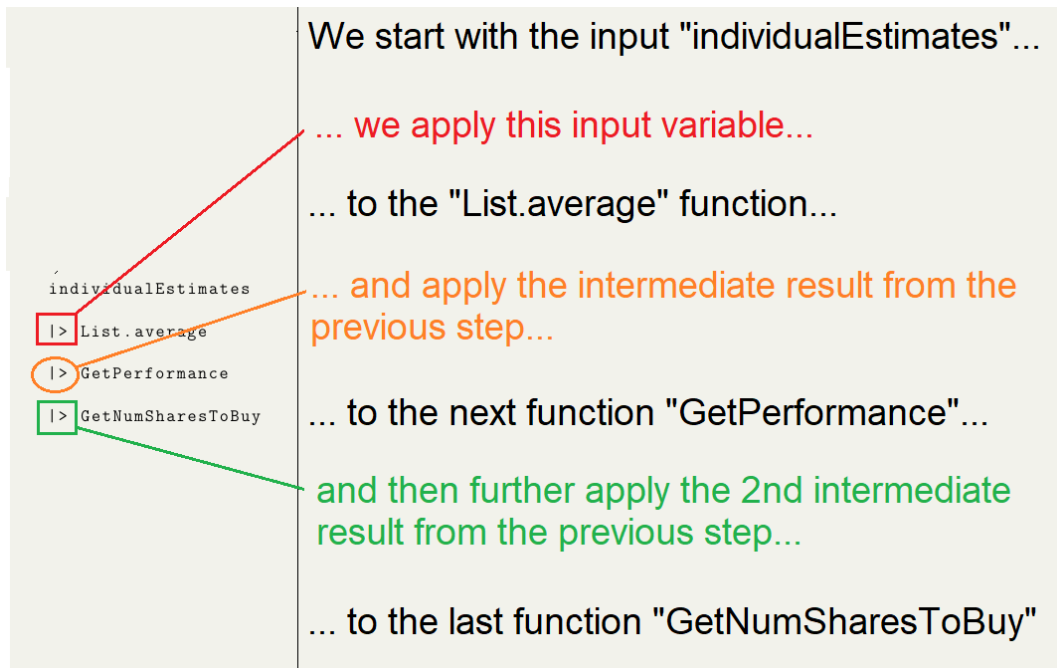
The following is the original code:

```
1 // Assume the profit is already known to be $6.0 billion,
2   and written in "GetPerformance"
3 let GetNumSharesFromEstimate1 individualEstimates =
4     let intermediateResult1 =
5         List.average individualEstimates
6     let intermediateResult2 =
7         GetPerformance intermediateResult1
8     let finalResult = GetNumSharesToBuy intermediateResult2
9     // output
10    finalResult
```

And the following is the simplified version with the pipe-forward operator |>

```
1 let GetNumSharesFromEstimate2 individualEstimates =
2     individualEstimates
3     |> List.average
4     |> GetPerformance
5     |> GetNumSharesToBuy
```

How this code should be interpreted:



## 2.5 Benefits

The benefits of using the pipe-forward operator `|>`:

1. You can remove unnecessary clutter/words on our computer screen. You do not need explicitly write out intermediate result, and we can focus more on the internal logic/calculations (and reserve the naming for variables/results that are truly important).
2. It is easier to follow instructions than to reason mathematically. Consider the following two statements:

$$y = h(g(f(x)))$$

```
1 Start with variable x.  
2 Step 1: Use function f.  
3 Step 2: Use function g.  
4 Step 3: Use function h.
```

Most common languages\* are written from left-to-right, and top-to-bottom. So, the conventional mathematical notation  $h(g(f(x)))$  is not very natural to most languages. Whereas in the second case, it gives us a simple step-by-step instructions on how to get our final result.

This makes it easier to non-programmers to understand your code (e.g. if you work with a manager or a trader); it makes it easier for you to understand your own code (e.g. if you re-visit some code that you have written 1 year ago).

\*Exception: Hebrew and Arabic.

Once you get used to this syntax, you may find other traditional programming language, e.g. Java/ C++ to be a bit verbose/too long.

## 2.6 Intellisense

In actual code development, we will do things step by step (instead of collecting everything together and chain everything using `|>`).

1. We will first start off like this:

```
1 let myFunction1 (individualEstimates: List<float>) =  
2     individualEstimates  
3     |> List.average
```

If you are using VisualStudio or VisualStudioCode, hover your mouse over `myFunction1` to see the type signature:

```
1 List<float> -> float
```

2. Next, let's add one more line:

```
1 let myFunction2 individualEstimates =  
2     individualEstimates  
3     |> List.average  
4     |> GetPerformance
```

We know that until the `List.average` step, we have an intermediate result of type `float`. So, we want the next function, `GetPerformance`, to take in `float` as an input.

Hover your mouse over `myFunction2` to see the type signature:

```
1 List<float> -> string
```

3. Finally, let's add one more line:

```
1 let myFunction3 individualEstimates =  
2     individualEstimates  
3     |> List.average  
4     |> GetPerformance  
5     |> GetNumSharesToBuy
```

The new function, `GetNumSharesToBuy`, should ideally accept `string` as its input (which it does). And if we hover your mouse over `myFunction3` to see the type signature:

```
1 List<float> -> int
```

## 2.7 Exercise

Scenario: Assume that you are in a trading firm, and you want to manage your employees based on their performance.

You are given the following functions:

1. The F# build-in function, `List.sum` that finds the sum of a list of doubles/decimals.

```
1 let sum1 = List.sum [1.0; 2.0; 3.0; 4.0; 5.0] // sum
   from 1 to 5
2 let sum2 = List.sum [1.0 .. 100.0]           // sum
   from 1 to 100
```

2. Another function, `GetStatus`, that determines how well is the trader

- TOP TRADER: Profit exceeds \$ 10.0 million.
- HUGE LOSSES: Loses \$3.0 million.
- NORMAL TRADER: Remaining cases

```
1 let GetStatus profit =
2     if profit > 10.0 then
3         "TOP TRADER"
4     else if profit < -3.0 then
5         "HUGE LOSSES"
6     else
7         "NORMAL TRADER"
```

3. Another function, `GetBonus`, that determines how many months of bonus is given to the trader.

- TOP TRADER: 24 months bonus (i.e. 2 years bonus)
- HUGE LOSSES: 6 months bonus (i.e. half year bonus)
- NORMAL TRADER: 0 months bonus (i.e. no bonus)

```
1 let GetBonus status =
2     if status = "TOP TRADER" then
3         24 // 24-month, i.e. 2 year bonus.
4     else if status = "NORMAL TRADER" then
5         6 // 6-month, i.e. half year bonus.
6     else
7         0 // No bonus.
```

Again, the output of one function is the input of the next function:

```
1 List.sum : List<double> -> double
2 GetStatus:           double -> string
3 GetBonus:            string -> int

1 let GetBonusFromTrades1 listOfTrades =
2     let intermediateResult1 = List.sum listOfTrades
3     let intermediateResult2 = GetStatus intermediateResult1
4     let finalResult = GetBonus intermediateResult2
5     // output
6     finalResult
```

Try to re-implement the function above using the pipe-forward operator |>.

```
1 let GetBonusFromTrades2 (listOfTrades: List<double>) =
2
3
4
5
6
7
8
9
10
11
12     // implement the function above.
```

Examples of use cases:

1. This trader helped the company earned some money.

```
1 let bonus1 =
2     GetBonusFromTrades2 [1.0; -2.0; 0.5; 0.3; 0.4; 0.2]
3 printfn "He received a bonus of %i months" bonus1
```

2. This trader made one huge profitable deal, with other tiny losses.

```
1 let bonus2 =
2     GetBonusFromTrades2 [-2.0; -1.0; -0.5; 30.0; -1.0]
3 printfn "She received a bonus of %i months" bonus2
```

## 2.8 Function with same input and output type

The mathematical term is called *endomorphism*.

In all the above examples, we have chosen functions that have different input and output types, so that it is obvious which function comes after which one.

Sometimes, you may face with functions that have the same input and output type. For example:

```
1 let Square x = x * x
2 let Cube x = x * x * x
3 let Add5 x = x + 5
4
5 // Square: int -> int
6 // Cube : int -> int
7 // Add5  : int -> int
```

All of these functions are `int -> int`, and so you may compose them in different orders, or you may apply the same function multiple times, which may cause the function to completely change.

### 1. Example 1

$$f_1(x) = (x^2 + 5)^3$$

```
1 let f1 x =
2     x
3     |> Square
4     |> Add5
5     |> Cube
6
7 // (1^2 + 5) ^ 3 = 216
8 let demo1 = f1 1
9
10 // (2^2 + 5) ^ 3 = 729
11 let demo2 = f1 2
```

Output:

```
1 // val demo1 : int = 216
2 // val demo2 : int = 729
```

## 2. Example 2

$$f_2(x) = (x^2)^3 + 5$$

```
1 let f2 x =  
2     x  
3     |> Square  
4     |> Cube  
5     |> Add5  
6  
7 // (1^2)^3 + 5 = 6  
8 let demo3 = f2 1  
9  
10 // (2^2)^3 + 5 = 71  
11 let demo4 = f2 2
```

Output:

```
1 // val demo3 : int = 6  
2 // val demo4 : int = 69
```

## 3. Exercise:

Try to implement the following function using pipe-forward:

$$f_3(x) = [(x + 5)^2 + 5]^3$$

```
1 //let Square x = x * x  
2 //let Cube x = x * x * x  
3 //let Add5 x = x + 5  
4 let f3 x =  
5  
6  
7  
8  
9  
10     // IMPLEMENT YOUR FUNCTION ABOVE  
11  
12 // Testing:  
13 // [ (1+5)^2 + 5 ]^3 = 68921  
14 let demo5 = f3 1  
15  
16 // [ (2+5)^2 + 5 ]^3 = 157464  
17 let demo6 = f3 2
```



## 3 List in F#

Key Concept:

1. Introduce basic `List` functions
  - (a) `List.filter`
  - (b) `List.map`
2. Code in F# are very easy to understand (thanks for pipe-forward operator `|>` and the F# language design)
3. Anonymous functions / lambda function also helps.
  - You are defining a function at the exact location where it is most useful. So it boosts productivity.
  - `fun` is a keyword in F#!

### 3.1 Creating a list

You can create a list of integers/ float / string using the following notations:

```
1 let list1 = [1 .. 100]
2 let list2 = [50 .. 80]
3 let list3 = [1 .. 2 .. 100]
4
5 let list4 = [1.0 .. 100.0]
6 let list5 = [0.0 .. 0.05 .. 1.0]
7
8 let list6 = [1; 20; 50; 100; 55; 5; 10]
9 let list7 = [1.0; 6.0; 5.0; 10.0; 3.0; 2.0]
10
11 let list8 = ["ABC"; "DEF"; "GHI"; "JKL"; "MNO"]
```

The `;` is used to separate different elements, and `[a .. b]`, `[a .. diff .. b]` is used to specify any increasing/decreasing pattern.

If you hover your mouse on top of those variables (using VisualStudio or VisualStudioCode), you will see the types are `int list`, `float list`, etc. An alternate notation would be `List<int>`, `List<double>`, etc.

Warning: You cannot create a list with different types, e.g. the example below tries to create a list with a string, an integer, and a decimal/float.

```
1 let listError = ["ABC"; 123; 400.0]
2 // ERROR! Cannot define different type in the same list!
```

## 3.2 List.filter

Here is a simple function that returns true/false, depending on whether  $x$  is divisible by 2:

```
1 let IsItEven x = (x % 2 = 0)
2
3 let trueOrFalse1 = IsItEven 10
4 let trueOrFalse2 = IsItEven 3
```

Remark:  $x \% 2$  means the remainder after we divide  $x$  by 2.

We can use this function together with `List.filter`:

```
1 let result1 = List.filter IsItEven [1 .. 100]
2 // Output:
3 // [2; 4; 6; .....; 98; 100]
```

The `List.filter` function filters a list, and only select the elements which satisfy some requirement; the requirement is specified through a function `IsItEven`.

Alternatively, because the definition of `IsItEven` is quite easy, we can even implement it immediately after `List.filter`, at the point where we need it the most.

```
1 let result2 = List.filter (fun x -> x % 2 = 0) [1 .. 100]
2 // Output:
3 // [2; 4; 6; .....; 98; 100]
```

The notation `(fun x -> x % 2 = 0)` is used to define anonymous/lambda function, i.e. functions that are easy to define, that we do not need to give it a name, e.g. `IsItEven`.

Benefits:

- We define this function using the `fun` keyword at exactly where it is used.
- If we define too many custom functions, e.g. `IsItEven`, then it will be hard to keep track when we have 1000+ functions, and we will lose productivity.

Remark: The code `(fun x -> x % 2 = 0)` represents a “thing”, and that “thing” is a function, just like `IsItEven` is a function.

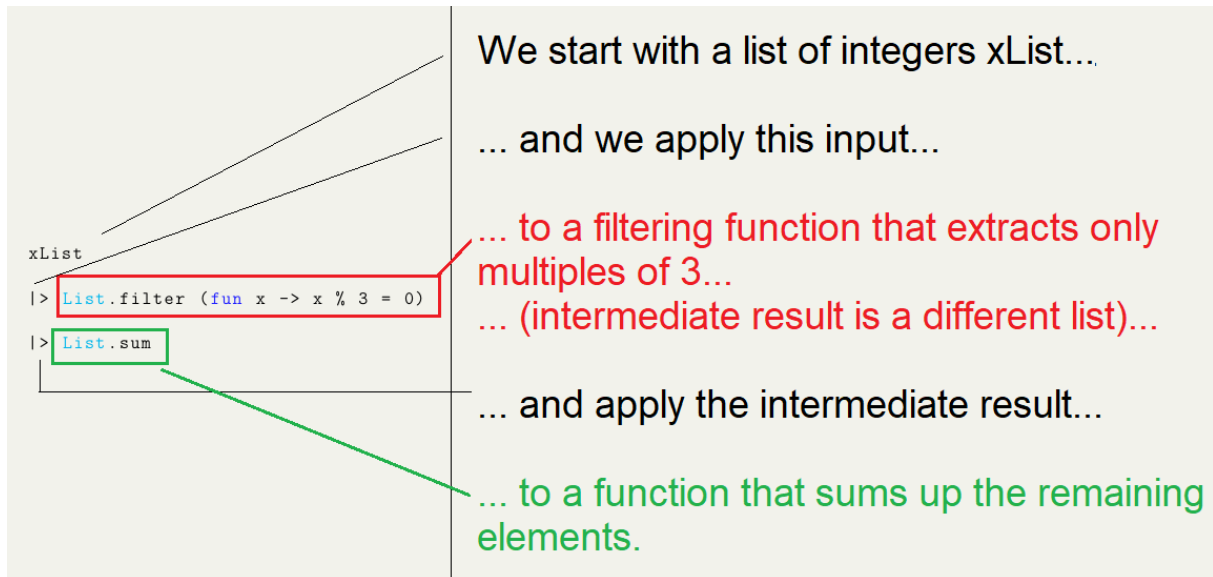
Side note: `fun` is a keyword in the F#! Programming in F# is very fun!

## List.filter and Pipe-Forward |>

Let us look at the following function:

```
1 let SumMultiplesOfThree xList =  
2   xList  
3   |> List.filter (fun x -> x % 3 = 0)  
4   |> List.sum
```

How to interpret this function:



So, F# is able to express all of these calculations with just 3 lines of code, which is quite elegant, maybe similar to Python code (in style), compared to other more traditional languages (Java/C++) which we need to write longer.

Using this function:

```
1 // 3 + 6 + 9 + ... + 99 = 1683  
2 let result3 = SumMultiplesOfThree [1 .. 100]  
3  
4 // 3 + 6 + 9 + ... + 198 = 6633  
5 let result4 = SumMultiplesOfThree [1 .. 200]
```

Output:

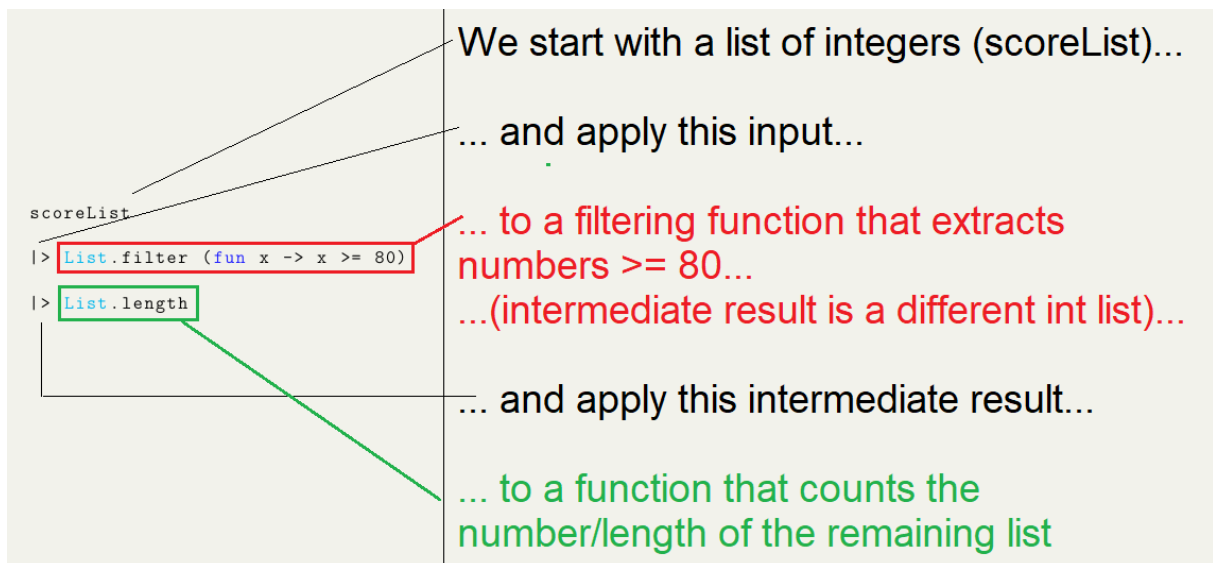
```
1 // val result3 : int = 1683  
2 // val result4 : int = 6633
```

## Another example

Let's say you want to find out how many students in your class got at least 80 points in an exam.

```
1 let CountGreaterThan80 scoreList =  
2   scoreList  
3   |> List.filter (fun x -> x >= 80)  
4   |> List.length
```

How to interpret this function:



Using this function:

```
1 let result5 =  
2   CountGreaterThan80 [60; 65; 70; 75; 80; 85; 90; 95]  
3 printfn "%i students scored 80 or above." result5
```

Output:

```
1 // "4 students scored 80 or above."
```

## Another example

This function adds up all multiples of 3, e.g. 3, 6, 9, ..., but ignore all multiples of 5, e.g. 5, 10, 15, 20, 25, 30, ...

```
1 let SumMultiplesOf3ButNot5 xList =
2     xList
3     |> List.filter (fun x -> (x % 3 = 0) && (x % 5 <> 0))
4     |> List.sum
5
6 let result6 = SumMultiplesOf3ButNot5 [1 .. 100]
```

Output:

```
1 // val result6 : int = 1368
```

Remark:

- $(x \% 3 = 0)$ : is x divisible by 3?
- $(x \% 5 <> 0)$ : is x NOT a multiple of 5?

## Exercise

Implement a function that sums up all multiples of 3 or 5 in a list.

```
1 let SumMultiplesOf3Or5 xList =
2
3
4
5
6     // Hint:  || means OR, && means AND
7
8     // From 1 to 10, the multiples of 3 or 5 are 3, 5, 6, 9, 10
9     // 3 + 5 + 6 + 9 + 10 = 33
10 let result7 = SumMultiplesOf3Or5 [1 .. 10]
11
12 let result8 = SumMultiplesOf3Or5 [1 .. 999]
```

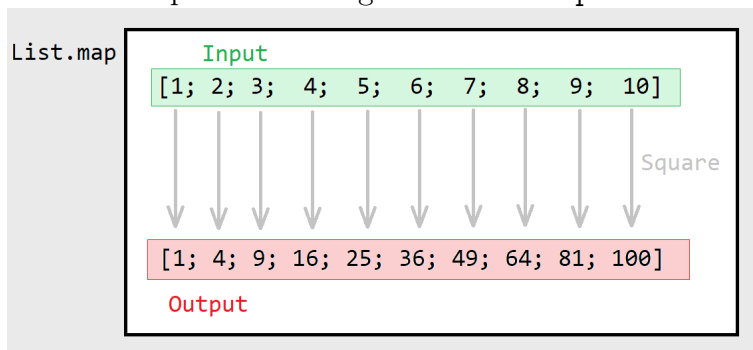
Remark: After you have completed this question, you can create an account and submit your solution here for personal achievement/accomplishment.

<https://projecteuler.net/problem=1>

### 3.3 List.map

```
1 let Square x = x * x
2 let result9 = List.map Square [1 .. 10]
```

The `List.map` function transform each individual element of a list using some transformation. The transformation is specified through a function `Square`.



Alternatively, we can use the `fun` keyword to define the `Square` function

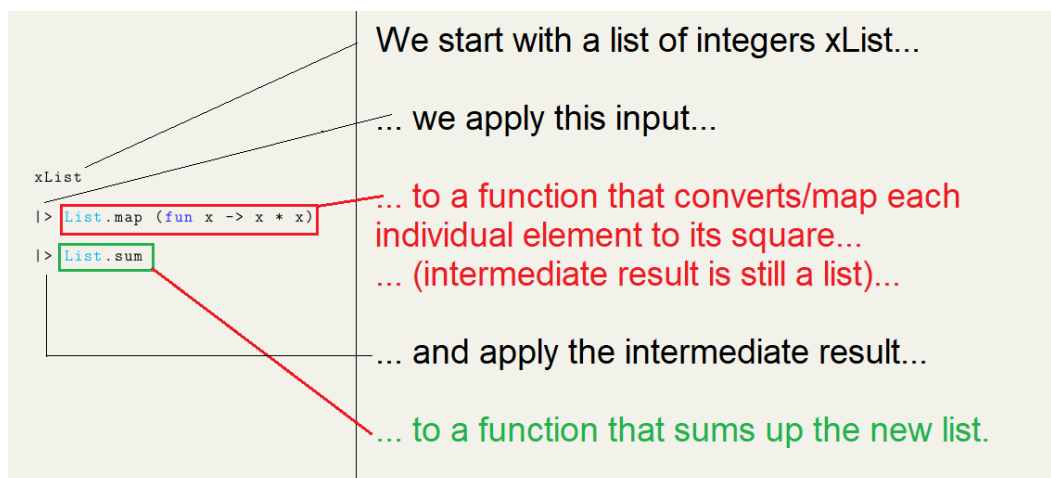
```
1 let result10 = List.map (fun x -> x * x) [1 .. 100]
```

#### List.map and Pipe-Forward |>

Let us look at an example:

```
1 let SumOfSquares xList =
2     xList
3     |> List.map (fun x -> x * x)
4     |> List.sum
5
6 // 1^2 + 2^2 + 3^2 + 4^2 + ... + 10^2 = 385
7 let result11 = SumOfSquares [1 .. 10]
```

How to interpret the code:



## Exercise

There are two supermarkets in town. One of them want to round the prices of each individual goods to the nearest dollar (might round-up or round-down). The other want to round DOWN the prices of each individual goods to the nearest dollar.

The functions `System.Math.Floor`, `System.Math.Round*` are used round the prices:

```
1 let originalPrice1 = 1.35
2 let originalPrice2 = 3.99
3
4 let newPrice1 = originalPrice1 |> System.Math.Floor
5 let newPrice2 = originalPrice2 |> System.Math.Floor
6
7 // Temporary ignore decimal numbers like 1.50, 2.50.
8 let newPrice3 = originalPrice1 |> System.Math.Round
9 let newPrice4 = originalPrice2 |> System.Math.Round
```

Output:

```
1 val newPrice1 : float = 1.0
2 val newPrice2 : float = 3.0
3
4 val newPrice3 : float = 1.0
5 val newPrice4 : float = 4.0
```

\*Remark: We will temporary ignore decimals like 1.50, 2.50, because F# uses “Banker’s Rounding” when tie-breaking is required. (Google it for more info)

1. Write a function that accepts a list of prices of the original products, and computes the final price of everything after each item are individually rounded-down.

```
1 // Round the prices to closest integer (ignore 1.50, 2.50,
   // etc.)
2 let TotalPriceAfterRoundDown priceList =
3
4
5
6     // Implement your function here.
```

2. Write a function that accepts a list of prices of the original products, and computes the final price of everything after each item are individually rounded to the nearest integer (ignore 1.50, 2.50, etc.).

```
1 let TotalPriceAfterRound priceList =
2
3
4
5     // Implement your function here.
```

## Application: Sample Variance

We will try to implement the sample variance function (VAR.S in Excel 2010 or later, or see <https://www.miniwebtool.com/sample-variance-calculator/>).

$$\text{Sample Variance} = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2$$

Remark: It is divided by  $N - 1$ , not  $N$ , because of statistics reasons (Bessel's correction).

```
1 let SampleVariance xList =
2   let N =
3     xList
4     |> List.length
5     |> double           // The "double" function
6
7   let average =
8     xList
9     |> List.average
10
11  // return this:
12  xList
13  |> List.map (fun x -> (x - average) ** 2.0)
14  |> List.sum
15  |> fun final -> final / (N - 1.0)
```

Remark:

1. We use the `double` function to convert an integer to decimal (you can also use it to process `string` to decimals, if the `string` is well defined)
2. At the last step, we divide by  $(N - 1.0)$  and not  $(N - 1)$  because we are working with decimals.
3. The compiler knows `xList` is a `float list` or `List<float>`, because at some point it interacted with `** 2.0`.

```
1 let result12 = SampleVariance [1.0 .. 7.0]
2 // val result12 : double = 4.666666667
```



## Exercise

Given a list of integers  $x_1, x_2, \dots, x_n$ , write a function that calculates the following:

$$\left(\sum_{i=1}^n x_i\right)^2 - \left(\sum_{i=1}^n x_i^2\right)$$

If you want, you can use the following hint:

```
1 let ProjectEulerProblem6 xList =
2   // if xList = [a;b;c], calculate a^2 + b^2 + c^2
3   let sumOfSquares =
4
5
6
7
8   // if xList = [a;b;c], calculate a + b + c
9   let sum =
10
11
12
13
14   // return
15   (sum * sum) - sumOfSquares
```

To use the function:

```
1 let result13 = ProjectEulerProblem6 [1 .. 100]
2 printfn "Answer for ProjectEuler Problem6 is: %i" result13
```

Remark: After you have completed this question, you can create an account and submit your solution here for personal achievement/accomplishment.

<https://projecteuler.net/problem=6>

## Exercise

You are given the following function that determines whether a positive integer  $x$  is a prime number or not. You can just directly use it. You do not need to implement it yourself.

```
1 let IsPrime x =
2   let squareRoot = x |> double |> sqrt |> int
3   if x = 1 then false
4   else if x = 2 then true
5   else if x % 2 = 0 then false
6   else
7     [3 .. 2 .. squareRoot]
8     |> List.forall (fun i -> x%i <> 0)
9
10  // val IsPrime: x:int -> bool
```

Reminder: You can directly use the `IsPrime` function in the previous page. You do not need to re-implement it again.

Write a function that takes in a list of positive integers, and sums all the prime numbers.

```
1 let Problem10_Version1 (xList: List<int>) =  
2  
3  
4  
5  
6  
7  
8 // test:  
9 let result1 = Problem10_Version1 [1 .. 9]  
10 // 2 + 3 + 5 + 7 = 17.  
11  
12 let result2 = Problem10_Version1 [1 .. 99]  
13 // 2 + 3 + 5 + 7 + 11 + ... + 83 + 89 + 97 = 1060.
```

However, if you try the following, you may potentially encounter an error:

```
1 let result3 = Problem10_Version1 [2 .. 2000000]  
2 // ERROR!  
3 // System.OverflowException: Arithmetic operation resulted  
  in an overflow.
```

This is because there are too many numbers to sum up, and the `int` datatype cannot handle large sums.

A workaround is to use the `System.Numerics.BigInteger` (Java also has such `BigInteger` standard library). We convert each individual prime number into a `BigInteger` before sum them all up.

```
1 open System.Numerics  
2  
3 let SumAllPrimes xList =  
4     xList  
5     |> List.filter (IsPrime)  
6     |> List.map (BigInteger)  
7     |> List.sum  
8  
9 // Remark: The code below can take 10 seconds, as this is  
  not the most optimal algorithm.  
10 let result17 = SumAllPrimes [2 .. 2000000]  
11 printfn "The sum of all primes from 2 to 2000000 is: %A"  
    result17
```

Remark: After you have completed this question, you can create an account and submit your solution here for personal achievement/accomplishment.

<https://projecteuler.net/problem=10>

## Optional Exercise (Try this at home)

<https://projecteuler.net/problem=3>

Given an integer  $Z$ , write a function that finds the largest prime factor of  $Z$ . e.g. The prime factors of 13195 are 5, 7, 13, 29, and so the largest for 13195 is 29.

## Problem Analysis

To find the largest prime factor for  $Z$ , we will first try to find all factors of  $Z$  (not necessarily prime factors) between 1 and  $\sqrt{Z}$ .

Let  $S_1 = \{a_1, \dots, a_n\}$  be all the factors of  $Z$  (not necessarily prime factors) between 1 and  $\sqrt{Z}$ . This set will always contain at least one element:  $a_1 = 1$ .

Let  $S_2 = \left\{\frac{Z}{a_1}, \dots, \frac{Z}{a_n}\right\}$ . These are all the factors of  $Z$  between  $\sqrt{Z}$  and  $Z$ . This set will always contain at least one element:  $\frac{Z}{a_1} = Z$ .

And so,  $S_1 \cup S_2 = \left\{a_1, \dots, a_n, \frac{Z}{a_1}, \dots, \frac{Z}{a_n}\right\}$  are all the factor of  $Z$  (not necessarily prime factors). And we just need to find that out of this set  $S_1 \cup S_2$ , which number is the largest prime number.

1. Example:  $3 \times 7 = 21$ . The largest prime factor is  $7 > \sqrt{21} \approx 4.58$ .
2. Example:  $6 \times 11 = 66$ . The largest prime factor is  $11 > \sqrt{66} \approx 8.12$ .

## Hints:

1. The function `IsPrimeBigInteger` is implemented for you. This is a slightly modified version of the `IsPrime` function that takes in a `BigInteger` as an input (as compared to an `int`). No need to re-implement this function.

```
1 let IsPrimeBigInteger x =  
2   let squareRoot = x |> double |> sqrt |> BigInteger  
3   if x = BigInteger(1) then false  
4   else if x = BigInteger(2) then true  
5   else if x % BigInteger(2) = BigInteger(0) then false  
6   else  
7       [BigInteger(3) .. BigInteger(2) .. squareRoot]  
8       |> List.forall (fun i -> x%i <> BigInteger(0))
```

2. To determine whether  $x$  is divisible by  $y$ , when  $x$ ,  $y$  are both `BigIntegers`, do the following:

```
1 x % y = BigInteger(0)
```

We cannot do  $x \% y = 0$ , because we cannot directly compare a `BigInteger` with an integer.

```

1 open System.Numerics
2
3 let FindLargestPrimeFactor (Z: BigInteger) =
4     let approxSqrt = Z |> double |> sqrt |> BigInteger
5
6     // Find factors of Z between [2 .. sqrt(Z)]
7     // Not necessarily prime factors.
8     let list1 =
9         [BigInteger(2) .. approxSqrt]
10        |> .....
11
12        // Complete the logic above
13
14    // Produce another list such that:
15    // For each element "a" in list1, it gives "Z / a"
16    let list2 =
17        list1
18        |> .....
19
20        // Complete the logic above
21
22    // List.append combines the two lists.
23    let combinedList =
24        List.append list1 list2
25
26    // Choose only prime numbers from the combinedList, and
27    // find the maximum using List.max
28    combinedList
29    |> .....
30
31    // Complete the logic above

```

```

1 let number1 = BigInteger(21)
2 let result18 = FindLargestPrimeFactor number1
3 // Expect result: 7
4
5 let number2 = BigInteger(66)
6 let result19 = FindLargestPrimeFactor number2
7 // Expect result: 11
8
9 let number3 = BigInteger.Parse("600851475143")
10 let result20 = FindLargestPrimeFactor number3

```

Remark: After you have completed this question, you can create an account and submit your solution online for personal achievement/accomplishment.

## 4 Tuples, Records, Discriminated Union

Key concept:

1. Tuple is a good data-structure for many things:
  - (a) To represents terms that goes together, e.g. 2-D coordinates, Year-Month, etc.
  - (b) For pattern matching, especially for pairwise operation or grouping operation.
  - (c) For testing out ideas quickly. (and use other data-structures after you have done testing)
2. You can directly extract the content of a tuple, and use underscore “\_” to ignore any part of the tuple that you don’t need.

### 4.1 Tuples

A 2D-coordinate may look like this:

```
1 let point1 = (1.0, 2.0)
2 let point2 = (3.0, 4.0)
```

Hover your mouse on top of these two objects. Notice that the signature is `float * float`. So these points have two coordinates, each of them are `float` or `double`

```
1 let DistanceFromOrigin point =
2     let (x,y) = point          // Data extraction process!
3     sqrt (x ** 2.0 + y ** 2.0)
4
5 let distance1 = DistanceFromOrigin point1
6 printfn "The first point is distance %f away from origin"
   distance1
7
8 let distance2 = DistanceFromOrigin point2
9 printfn "The second point is distance %f away from origin"
   distance2
```

Output:

```
1 // The first point is distance 2.236068 away from origin
2 // The second point is distance 5.000000 away from origin
```

Notice that we have an extraction process `let (x,y) = point` that helps us extract the contents of `point` (and save the contents into the variables `x,y`). In fact, we can directly do the extraction process in the function definition:

```
1 let DistanceFromOrigin2 (x,y) =
2     sqrt (x ** 2.0 + y ** 2.0)
```

## Tuples of Different Type

We can mix tuples of different type (compared to list, which cannot contain elements of different types).

```
1 let mixedTuple1 = (1.0, "HELLO")
2 let mixedTuple2 = (1, "Hello", true)
```

If you hover your mouse on top of these, you will see that:

- The first tuple has signature `float * string`
- The second tuple has signature `int * string * bool`

As before, we can extract the contents of the tuple using `let`.

```
1 let (extractedDecimal, extractedString) = mixedTuple1
2 printfn "The extracted decimal is: %f" extractedDecimal
3 printfn "The extracted string is: %s" extractedString
4
5 let (extractedInteger, extractedString2, extractedBool) =
    mixedTuple2
6 printfn "The extracted integer is: %i" extractedInteger
7 printfn "The extracted string is: %s" extractedString2
8 printfn "The extracted boolean is: %b" extractedBool
```

Output:

```
1 // The extracted decimal is: 1.000000
2 // The extracted string is: HELLO
3
4 // The extracted integer is: 1
5 // The extracted string is: Hello
6 // The extracted boolean is: true
```

If you only want to extract part of a tuple, you can use the underline “`_`” to ignore any part of the tuple that you don’t need.

```
1 let personalInfo = ("John", 21, 170.0)
2
3 let (extractedName, _, _) = personalInfo
4
5 printfn "The extracted name is: %s" extractedName
6 // The extracted name is: John
```

## Example

You are given data about the number of student in each class. The data is saved in a `List<string * int>`. e.g. in the first list, Class A has 50 students, Class B has 40 students, etc.

```
1 let studentList =  
2   [("A",50); ("B", 40); ("C", 45); ("D", 48)]  
3 let studentList2 =  
4   [("A", 40); ("B", 30); ("C", 20); ("D", 25); ("E", 29);  
   ("F", 50)]
```

The following function helps to find the total number of students in those school:

```
1 let TotalStudent (studentList: List<string * int>) =  
2   studentList  
3   |> List.map (fun classInfo ->  
4     let (_,numStudent) = classInfo  
5     numStudent  
6   )  
7   |> List.sum  
8  
9 let totalStudent1 = TotalStudent studentList
```

Output:

```
1 // val totalStudent1 : int = 183
```

Of course, we can directly do the extraction process in the function definition:

```
1 let TotalStudentVersion2 (studentList: List<string * int>)  
2   =  
3   studentList  
4   |> List.map (fun (_,numStudent) -> numStudent)  
5   |> List.sum  
6 let totalStudent2 = TotalStudentVersion2 studentList
```

Output:

```
1 // val totalStudent2 : int = 194
```

## Exercise

You are given data about how each student score in a class. e.g. In this class, Ali scored 85.0 points, Baba scored 95.0 points, etc.

```
1 let classScore1 =  
2   [("Ali", 85.0); ("Baba", 95.0); ("Charlie", 87.0); ("  
   Dan", 92.0); ("Emily", 96.0); ("Fiona", 92.0)]
```

Write a function that accepts a list of names with their scores, and return the class average.

```
1 let ClassAverage (scores: List<string * double>) =  
2  
3  
4  
5  
6  
7  
8  
9   // Implement your function here.  
10  // Hint: List.map and List.average
```

## Example

A country currently wants to implement a new tax system:

- COMMON: 5% tax
- IMPORTS: 10% tax
- ALCOHOL: 20% tax

A supermarket wants currently saves the data in a `List<string * double * string>`, where the first `string` is the product, the `double` is the original price before tax, and the last `string` is the product code. e.g.

```
1 let productList1 =  
2   [("Bread", 2.40, "COMMON");  
3    ("Beer", 10.20, "ALCOHOL");  
4    ("Swiss Chocolate", 8.20, "IMPORTS");  
5    ("Rice", 20.50, "COMMON");  
6    ("Red Wine", 30.00, "ALCOHOL");  
7    ("Australian Beef", 18.50, "IMPORTS")]
```



The following code will help calculate the total price after tax:

```
1 let TotalAfterTax (productList: List<string * double *  
   string> ) =  
2     productList  
3     |> List.map (fun tuple ->  
4         let (_,priceBeforeTax,productType) = tuple  
5         // Data Extraction above!  
6  
7         if productType = "COMMON" then  
8             1.05 * priceBeforeTax  
9         else if productType = "ALCOHOL" then  
10            1.20 * priceBeforeTax  
11        else  
12            1.10 * priceBeforeTax  
13    )  
14    |> List.sum  
15  
16 let totalPrice = TotalAfterTax productList  
17 printfn "The final price after tax is: %.2f" totalPrice
```

Output:

```
1 The final price after tax is: 101.66
```

Again, we can move the extraction process into the function definition:

```
1 let TotalAfterTaxVersion2 (productList: List<string *  
   double * string> ) =  
2     productList  
3     |> List.map (fun (_,priceBeforeTax,productType) ->  
4         if productType = "COMMON" then  
5             1.05 * priceBeforeTax  
6         else if productType = "ALCOHOL" then  
7             1.20 * priceBeforeTax  
8         else  
9             1.10 * priceBeforeTax  
10    )  
11    |> List.sum
```

Notice that the values are extracted immediately after the `fun` keyword.

## Exercise

A clothing store is planning to do a discount sale:

- CLEARANCE: 50% off.
- SHIRT: 30% off.
- JEANS: 20% off.

You are given a `List<string * double>` that represents an item's product code and their original price. e.g. the customer below bought a clearance item, two shirts and two jeans.

```
1 let listOfClothes =  
2   [ ("CLEARANCE", 70.0); ("SHIRT", 20.0); ("SHIRT", 40.0)  
   ; ("JEANS", 55.0); ("JEANS", 79.9)]
```

Write a function that takes a list of items and their original price, and return the total price after discount.

```
1 let TotalAfterDiscount (priceList: List<string * double>) =  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12   // Implement your function here.
```

The expected final price after discount is \$184.92

### 4.2 List.allPairs

### 4.3 Record Type

### 4.4 Discriminated Union

# A Appendix

## A.1 Optional Topics

### A.1.1 inline functions

On some occasion, if you need to use the same function on different type which supports (\*), then you can use the inline keyword.

```
1 let inline Product x y = x * y
2
3 let multiply2Int = Product 2 3
4 printfn "Multiply the two numbers gives: %i" multiply2Int
5 // Output: "Multiply the two numbers gives: 6"
6
7 let multiply2Double = Product 3.0 4.0
8 printfn "Multiply the two numbers gives: %f"
   multiply2Double
9 // Output: "Multiply the two numbers gives: 12.000000"
```

However, not every datatype supports multiplication (\*)

```
1 let multiply2WordsError = Product "word1" "word2"
2 "ERROR!!!!!!!!!!"
```

INPUT PICTURE HERE!

---

Similarly, we can do this:

```
1 let inline CustomAdd x y z = x + y + z
2 let add3IntegerResult = CustomAdd 4 5 6
3 printfn "Adding the three integers give: %i"
   add3IntegerResult
4 // Output: "Adding the three integers give: 15"
5
6 let add3StringResult = CustomAdd "John " "F." " Kennedy"
7 printfn "Concatenate the three strings give: %s"
   add3StringResult
8 // Output:
9 // "Concatenate the three strings give: John F. Kennedy"
10
11 let add3DecimalResult = CustomAdd 10.3 10.2 10.1
12 printfn "Adding the three decimals give: %f"
   add3DecimalResult
13 // Output: "Adding the three decimals give: 30.600000"
```

However, not every datatype supports addition (+)

```
1 let add3BooleanError = CustomAdd true false false
2 "ERROR!!!!!!!!!!"
```

INPUT PICTURE HERE!