

F# Tutorial

Pipe-Forward Operator

February 10, 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

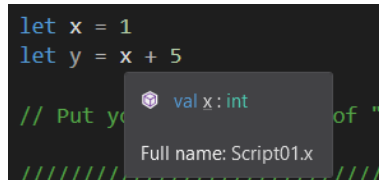
If you are using Visual Studio, you can run the code above by highlighting/selecting the code using your mouse, and press `ALT + ENTER`, or right-click and select **Execute** in **Interactive**.

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 // Output:
7 // My name is: John
8
9 printfn "Name: %s. Age: %i. Height: %f." name age height
10 // Output:
11 // Name: John. Age: 21. Height: 170.500000
12
13 printfn "His height is: %.2f" height
14 // Output:
15 // His height is: 170.50
16 //// Show only two decimal.
```

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 changable/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"
```

In addition, if you try to update an immutable/unchangeable 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'.

Why do we recommend immutable/unchangeable values:

Imagine you have a code below, where you have defined 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/unchangeable:

```
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/unchangeable.....
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
```

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

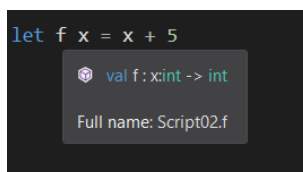
1.3.1 One variable functions

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

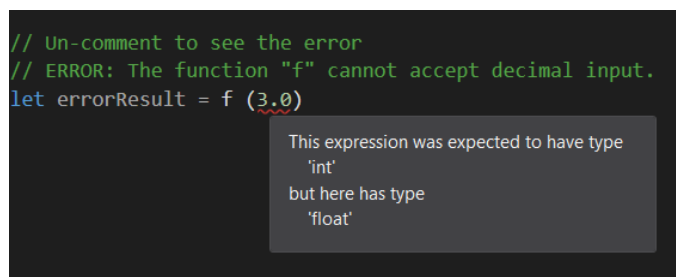
```
1 let f x = x + 5
2
3 let result1 = f 10
4 printfn "The result is: %i" result1
5 // Output: "The result is: 15"
6
7 let result2 = f 20
8 printfn "The result is: %i" result2
9 // Output: "The result is: 25"
```

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 DiscountBy20Percent originalPrice = originalPrice * 0.8
2
3 let discountedPrice = DiscountBy20Percent 399.99
4 printfn "The new price is: %.2f" discountedPrice
5 // Output: "The new price is: 319.99"
6
7 let anotherDiscount = DiscountBy20Percent discountedPrice
8 printfn "The new price is: %.2f" anotherDiscount
9 // Output: "The new price is: 255.99"
```

Remark: The %.2f is for formatting purposes when printing result, 2 decimals.

Here, if you try to put input an integer value to this function, then you will see an error:

INPUT ERROR PICTURE HERE!

And in order to use the function on integer values, you need to convert it to decimals using double or float function/keyword.

```
1 let convertedPrice = double 100
2 let decimalResult = DiscountBy20Percent convertedPrice
3 printfn "The new price is: %.2f" decimalResult
4 // Output: "The new price is: 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 printfn "The modified sentence is: %s" greeting1
7 // Output: "The modified sentence is: Hello John"
8
9 let greeting2 = AddGreeting "Mary"
10 printfn "The modified sentence is: %s" greeting2
11 // Output: "The modified sentence is: 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 printfn "The result is: %i" result3  
5 // Output: "The result is: 10"  
6  
7 let result4 = g 10 2  
8 printfn "The result is: %i" result4  
9 // Output: "The result is: 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 "The new balance is: %f" balance1
6 // Output: "The new balance is: 105000.00"
7
8 let balance2 = CalculateNewBalance 0.03 5000.00
9 printfn "The new balance is: %f" balance2
10 // Output: "The new balance is: 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 printfn "The result is: %i" result5
6 // Output: "The result is: 50"
7
8 // 3*1 + 4*1 + 5*1 = 12
9 let result6 = h 1 1 1
10 printfn "The result is: %i" result6
11 // Output: "The result is: 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 let DiscountBy20Percent originalPrice = originalPrice * 0.8
2 // Output: "The new price is: 255.99"

```

```

1 let DiscountBy20Percent originalPrice = originalPrice * 0.8
2 // Output: "The new price is: 255.99"

```

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"
6 printfn "The new sentence is: %s" combinedSentence1
7 // Output:
8 // The new sentence is: 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 "Drink preference: %s" (DrinkFunction "Jack")
10 // Output:
11 // "Drink preference: Jack likes to drink coffee."
12
13 printfn "Eating preference: %s" (EatFunction "Jill")
14 // Output:
15 // "Eating preference: 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

```

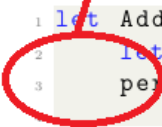
1.4.4 Warning: No TAB

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

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 printfn "Result is: %i" result1
5 // Output: "Result is: 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
3 printfn "Result is: %i" result2
4 // Output: "Result is: 35"
```

How this code should be interpreted is the following:

```
1 let result2 = 30 |> Add5Func
2
```

Using the input 30...

... we apply this input.....

... to the function
"Add5Func"

2.3 Why is this useful?

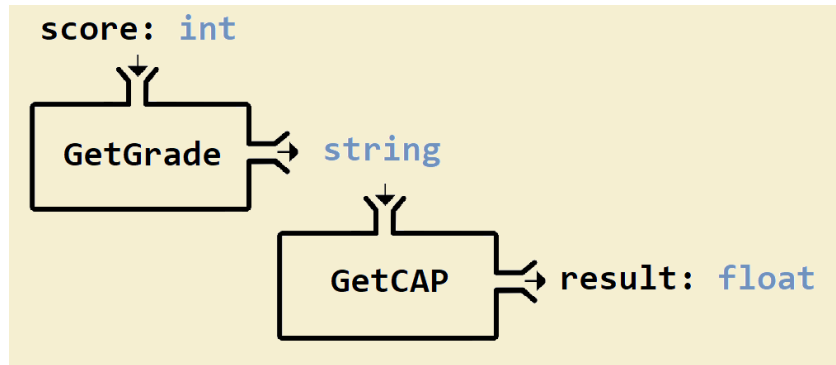
The reason why the symbol `|>` is useful is because it helps us to compose functions. Let's say that you are given these functions:

```
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
13
14 // For American University. (Maximum GPA 4.0)
15 let GetGPA grade =
16   if grade = "A" then 4.0
17   else if grade = "B" then 3.5
18   else if grade = "C" then 3.0
19   else 2.5
```

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`.



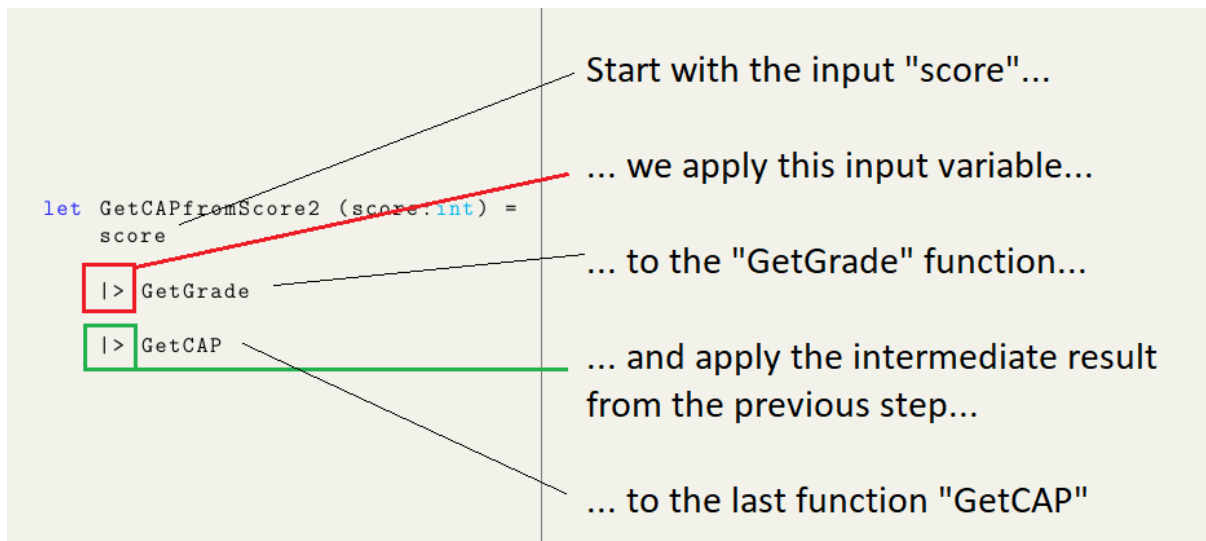
```
1 // GetGrade: int -> string
2 // GetCAP:      string -> float
3
4 let GetCAPfromScore1 score =
5     let intermediateResult = GetGrade score
6     let finalResult = GetCAP intermediateResult
7     // return
8     finalResult
9
10 let cap1 = GetCAPfromScore1 95
11 printfn "Your CAP is: %f" cap1
12 // Output: "Your CAP is: 5.0"
13
14 let cap2 = GetCAPfromScore1 85
15 printfn "Your CAP is: %f" cap2
16 // Output: "Your CAP is: 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
4 let GetCAPfromScore2 score =
5     score
6     |> GetGrade
7     |> GetCAP
8
9 let cap3 = GetCAPfromScore2 95
10 printfn "Your CAP is: %f" cap3
11 // Output: "Your CAP is: 5.0"
12
13 let cap4 = GetCAPfromScore2 85
14 printfn "Your CAP is: %f" cap4
15 // Output: "Your CAP is: 4.0"
```

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`, it cannot be the input for the function `GetCAP` (which accepts only `string`).

Similarly, the intermediate result from `GetCAP` is `float`, which cannot be the input for the function `GetGrade` (which accepts only `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.

```

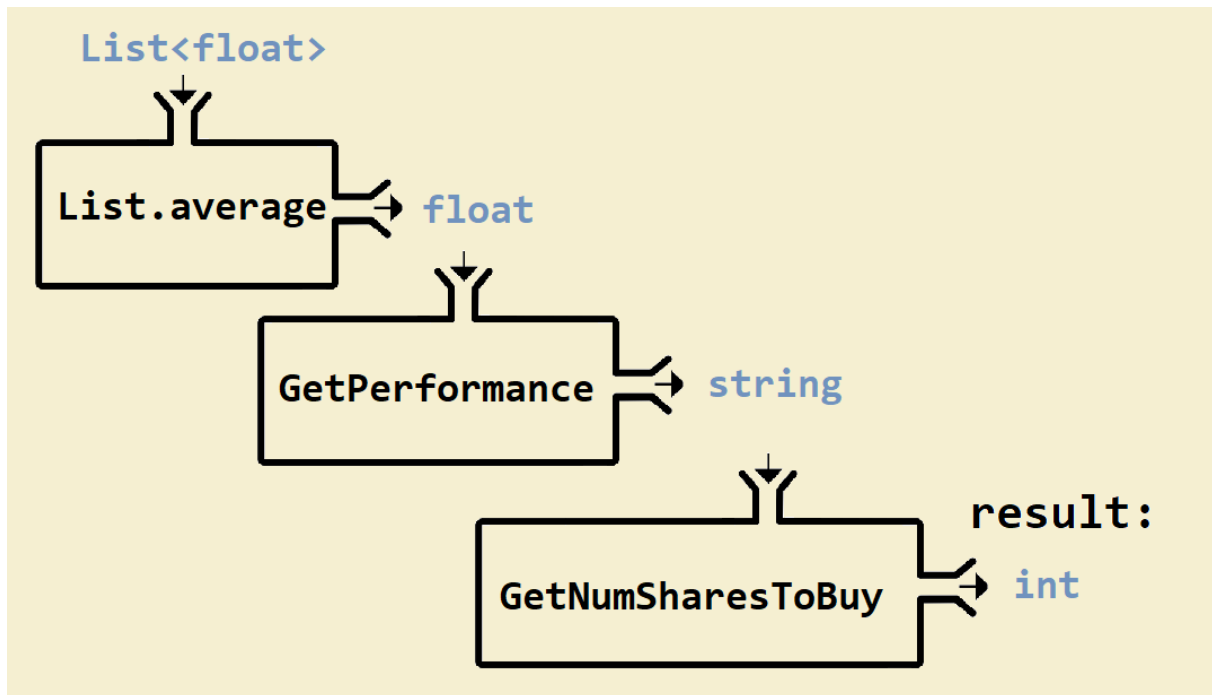
So, the function signatures are:

```

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   let intermediateResult2 =
6     GetPerformance intermediateResult1
7   let finalResult = GetNumSharesToBuy intermediateResult2
8   // output
9   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;
  2.0; 2.5]
2 printfn "Number of shares to buy(+) or sell(-): %i"
  numShares1
3 // Output:
4 // "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;
    10.0; 12.0; 10.5]
2 printfn "Number of shares to buy(+) or sell(-): %i"
    numShares2
3 // Output:
4 // "Number of shares to buy(+) or sell(-): -1000"
```

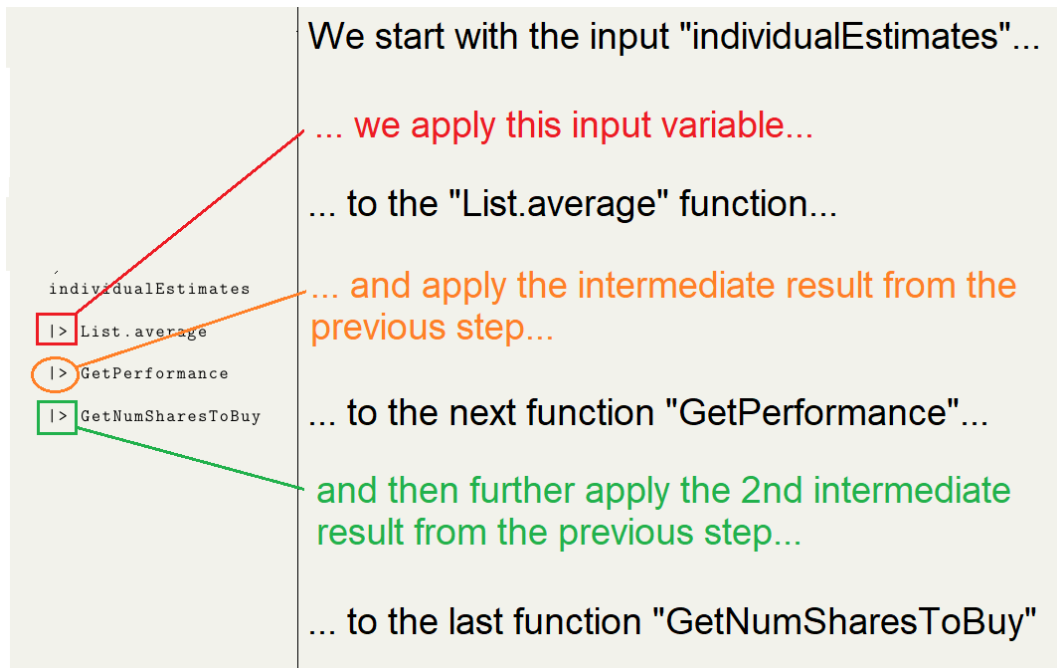
The following is the original code:

```
1 // Assume the profit is already known to be $6.0 billion,
    and written in "GetPerformance"
2 let GetNumSharesFromEstimate1 individualEstimates =
3     let intermediateResult1 =
4         List.average individualEstimates
5     let intermediateResult2 =
6         GetPerformance intermediateResult1
7     let finalResult = GetNumSharesToBuy intermediateResult2
8     // output
9     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
13    // 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

```

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!