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:

```
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:

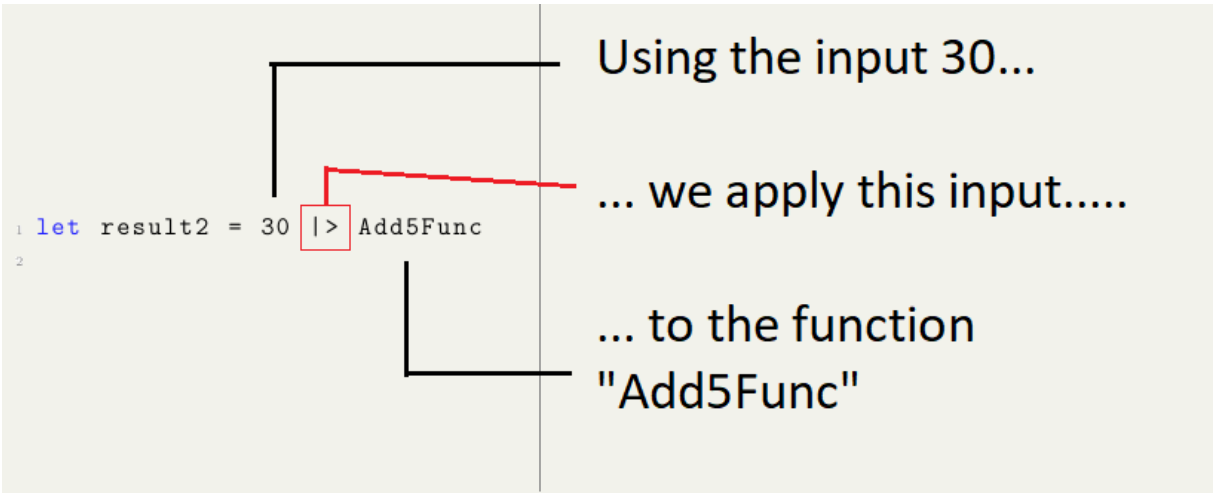
```
let Add5Func x = x + 5

let result1 = Add5Func 30
// 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.

```
let result2 = 30 |> Add5Func
// 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:

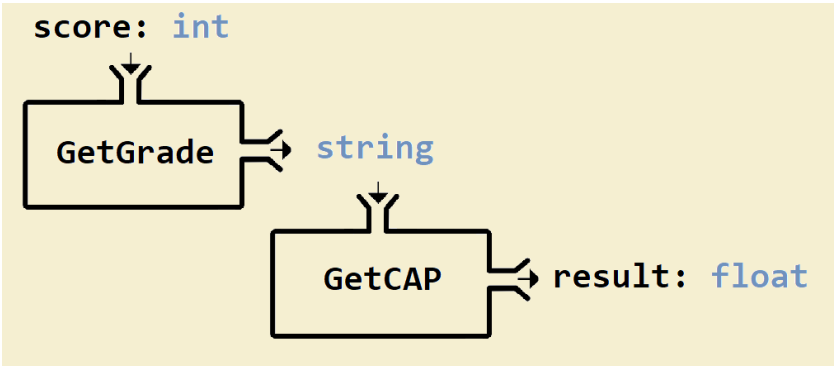
```
let GetGrade score =
    if score >= 90 then "A"
    else if score >= 70 then "B"
    else if score >= 50 then "C"
    else "D"

// For Singaporean University. (Maximum CAP 5.0)
let GetCAP grade =
    if grade = "A" then 5.0
    else if grade = "B" then 4.0
    else if grade = "C" then 3.0
    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:

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

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




```
let GetCAPfromScore1 score =
    let intermediateResult = GetGrade score
    let finalResult = GetCAP intermediateResult
    // return
    finalResult

let cap1 = GetCAPfromScore1 95
let cap2 = GetCAPfromScore1 85
```

Output:

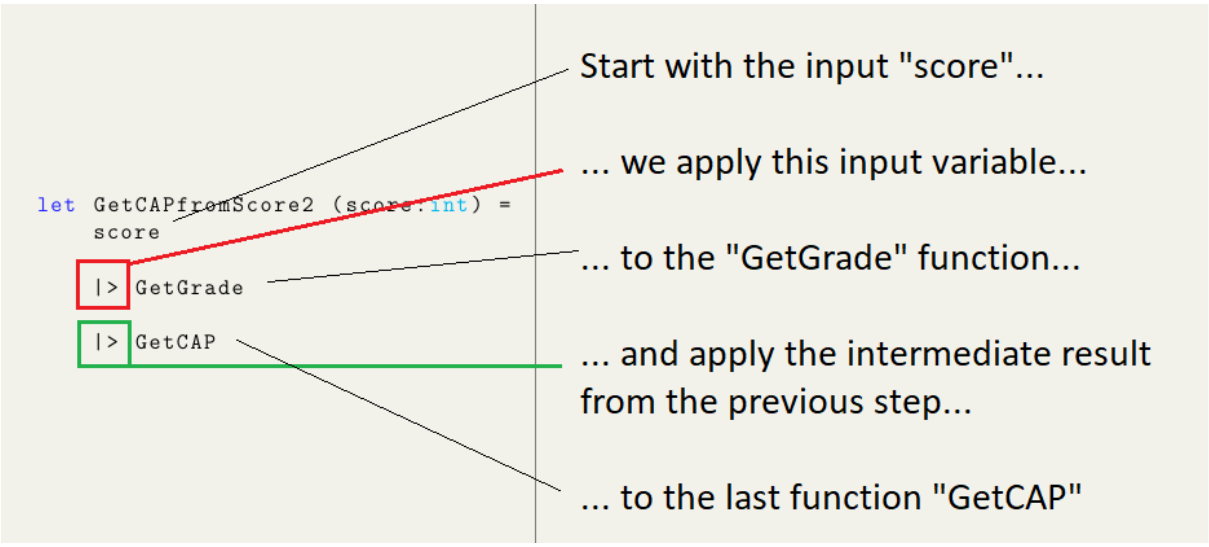
```
// val cap1 : float = 5.0
// val cap2 : float = 4.0
```

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:

```
// GetGrade: int   -> string
// GetCAP:            string ->  float
let GetCAPfromScore2 score =
    score
    |> GetGrade
    |> GetCAP

let cap3 = GetCAPfromScore2 95
// 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:

```
let CombinedFunction3Error score =
    score        // int
    |> GetCAP    // function: string -> float    // ERROR!
    |> GetGrade // function: int -> string      // ERROR!
"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 Exercise

2.4.1 Exercise 1

Lets say you have the following two functions:

```
let GetMeal money =
    if money > 59.90 then "LARGE SET"
    else if money >= 39.90 then "MEDIUM SET"
    else "SMALL SET"

let GetNumberPeople meal =
    if meal = "LARGE SET" then 6
    else if meal = "MEDIUM SET" then 4
    else 2
```

The signatures of the functions are:

```
GetMeal:          float -> string
GetNumberPeople:          string -> int
```

The long and complicated way of combining these functions are:

```
let NumFriendsToInvite money =
    let intermediateResult = GetMeal money
    let finalResult = GetNumberPeople intermediateResult
    // return
    finalResult
```

Try to rewrite the function using the pipe-forward operator `|>`

```
let NumFriendsToInvite money =
    money
    |> ......
    ......
```

2.4.2 Exercise 2

You are given the following functions:

```
let func1 x = (x < 0)
let func2 y = if y = true then 1.0 else 5.0
let func3 z = if z < 3.0 then "SMALL" else "LARGE"
```

The type signatures are:

```
func1 : int -> bool
func2 :       bool -> float
func3 :               float -> string
```

Since the output of one function is the input of another, we can do:

```
let CombinedFunction1 x =
    let intermediateResult1 = func1 x
    let intermediateResult2 = func2 intermediateResult1
    let finalResult = func3 intermediateResult2
    // return
    finalResult
```

Try combining these functions using the pipe forward operator `|>`

```
let CombinedFunction2 x =
    x
    |> ......
    ......
    ......
```

2.4.3 Exercise 3

In this example, you are given the following three functions with the following type signature:

```
List.average:
    List<double> -> double
GetPerformance:     double -> string
GetNumSharesToBuy:             string -> int
```

The functions are:

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

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

2. 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)

```
let GetPerformance analystAverageEstimate =
    let actualProfit = 6.0
    if actualProfit > analystAverageEstimate * 1.05
        then "OUTPERFORM"
    else if actualProfit < analystAverageEstimate * 0.95
        then "UNDERPERFORM"
    else
        "NEUTRAL"
```

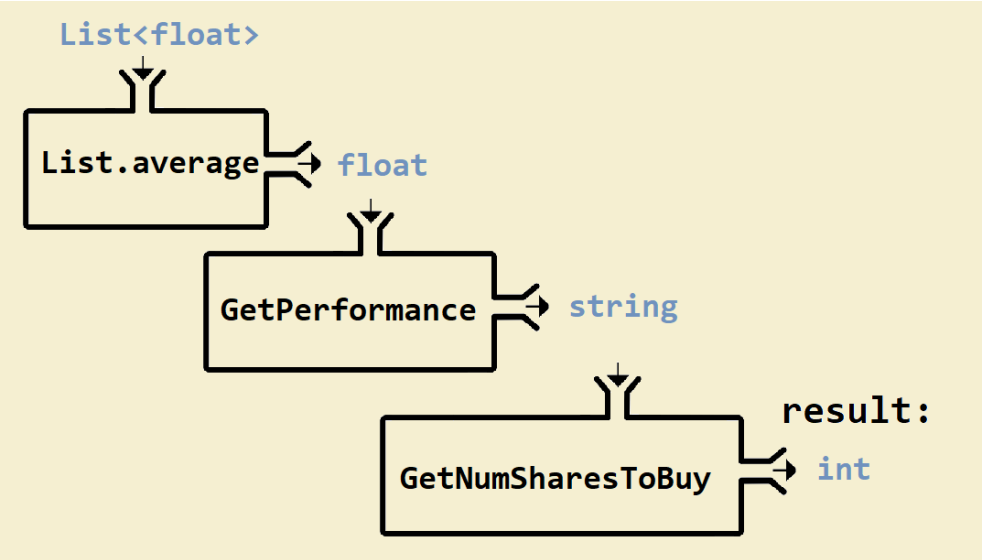
3. Another function, `GetNumSharesToBuy`, that determines how much additional shares to buy/sell depending on the company's condition.

```
let GetNumSharesToBuy performance =
    if performance = "OUTPERFORM" then
        1000      // buy 1000 shares
    else if performance = "UNDERPERFORM" then
        -1000     // sell 1000 shares
    else
        0        // hold.
```

Again, the function signatures are:

```
List.average:
    List<double> -> double
GetPerformance:      double -> string
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:

```
// Assume the profit is already known to be $6.0 billion,
   and written in "GetPerformance"
let GetNumSharesFromEstimate1 individualEstimates =
    let intermediateResult1 =
        List.average individualEstimates

    let intermediateResult2 =
        GetPerformance intermediateResult1

    let finalResult = GetNumSharesToBuy intermediateResult2
    // output
    finalResult
```

Try to write a simplified version with the pipe-forward operator `|>`

```
let GetNumSharesFromEstimate2 individualEstimates =
    individualEstimates
    |> ......
    ......
    ......
```

Compare your solution with the picture in the next page.

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:

```
let myFunction1 (individualEstimates: List<float>) =
    individualEstimates
    |> List.average
//
//
```

In VisualStudio, hover your mouse over `myFunction1` to see the type signature:

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

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

```
let myFunction2 individualEstimates =
    individualEstimates
    |> List.average
    |> GetPerformance
//
```

The next function, `GetPerformance`, takes in `float` as input, and returns `string`. Hover your mouse over `myFunction2` to see the type signature:

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

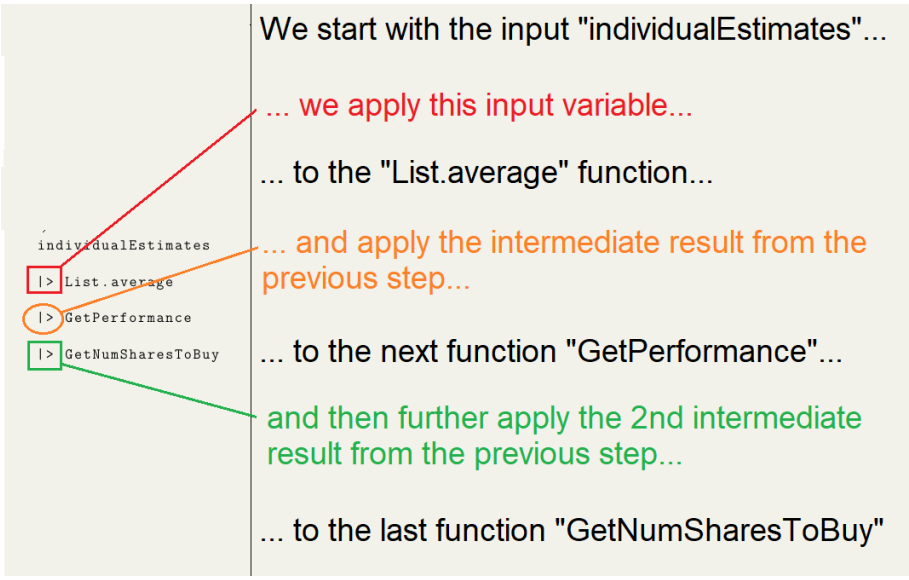
3. Finally, let's add one more line:

```
let myFunction3 individualEstimates =
    individualEstimates
    |> List.average
    |> GetPerformance
    |> GetNumSharesToBuy
```

The new function, `GetNumSharesToBuy`, accepts `string` as its input and returns `int`. Hover your mouse over `myFunction3` to see the type signature:

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

Solution and Interpretation to Exercise



2.4.4 Exercise 4

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.

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

```
let GetStatus profit =
    if profit > 10.0 then
        "TOP TRADER"
    else if profit < -3.0 then
        "HUGE LOSSES"
    else
        "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)
   - `NORMAL TRADER`: 6 months bonus (i.e. half year bonus)
   - `HUGE LOSSES`: 0 months bonus (i.e. no bonus)

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

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

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

```
let GetBonusFromTrades1 listOfTrades =
    let intermediateResult1 = List.sum listOfTrades
    let intermediateResult2 = GetStatus intermediateResult1
    let finalResult = GetBonus intermediateResult2
    // output
    finalResult
```

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

```
let GetBonusFromTrades2 (listOfTrades: List<double>) =









    // implement the function above.
```

Examples of use cases:

1. This trader helped the company earned some money.

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

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

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

2.5 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:

```
let Square x = x * x
let Cube x = x * x * x
let Add5 x = x + 5

// Square: int -> int
// Cube  : int -> int
// 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

$$f_1(x) = \left(x^2 + 5\right)^3$$

```
let f1 x =
    x
    |> Square
    |> Add5
    |> Cube

// (1^2 + 5) ^ 3 = 216
let demo1 = f1 1

// (2^2 + 5) ^ 3 = 729
let demo2 = f1 2
```

Output:

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

2. Example 2

$$f_2(x) = \left(x^2\right)^3 + 5$$

```
let f2 x =
    x
    |> Square
    |> Cube
    |> Add5

// (1^2)^3 + 5 = 6
let demo3 = f2 1

// (2^2)^3 + 5 = 71
let demo4 = f2 2
```

Output:

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

3. Exercise:

Try to implement the following function using pipe-forward:

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

```
//let Square x = x * x
//let Cube x = x * x * x
//let Add5 x = x + 5
let f3 x =









    // IMPLEMENT YOUR FUNCTION ABOVE

// Testing:
// [   (1+5)^2   + 5   ]^3 = 68921
let demo5 = f3 1

// [   (2+5)^2   + 5   ]^3 = 157464
let demo6 = f3 2
```

2.6 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 truely important).

2. It is easier to follow instructions than to reason mathematically. Consider the following two statements:

$$y = h\left(g\left(f\left(x\right)\right)\right)$$

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

   Most common languages* are written from left-to-right, and top-to-bottom. So, the conventional mathematical notation $h\left(g\left(f\left(x\right)\right)\right)$ 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.