VIA University College

Programming Concepts and Languages

Spring 2024

Learning Objectives

- By the end of class today, you will be able to:
  - ✓ explain and implement simple F# programs using
    - ✓ function composition (>>, <<)
    - ✓ pipelining (|>, <|)
    - ✓ type definitions and discriminated unions

Get the size of a given folder

```
open System
open System.IO
let sizeOfFolder folder =
    // Get all files under the path
    let filesInFolder : string [] =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

    // Map those files to their corresponding FileInfo object
    let fileInfos : FileInfo [] =
        Array.map (fun (file : string) -> new FileInfo(file)) filesInFolder

    // Map those fileInfo objects to the file's size
    let fileSizes : int64 [] =
        Array.map (fun (info : FileInfo) -> info.Length) fileInfos

    // Total the file sizes
    let totalSize = Array.sum fileSizes
    // Return the total size of the files
    totalSize
```

Get the size of a given folder some issues

- Type inference system cannot determine the correct type automatically
  - must provide a type annotation in each lambda
- Unnecessary let statements
  - Just feeding the result from one computation to the next
- It looks a bit ugly
  - Takes more time to figure what is going on

```
open System
open System.IO
let sizeOfFolder folder =
    // Get all files under the path
    let filesInFolder : string [] =
        Directory.GetFiles(folder, "*.*",
SearchOption.AllDirectories)

    // Map those files to their corresponding FileInfo object
    let fileInfos : FileInfo [] =
        Array.map (fun (file : string) -> new FileInfo(file))
filesInFolder

    // Map those fileInfo objects to the file's size
    let fileSizes : int64 [] =
        Array.map (fun (info : FileInfo) -> info.Length)
fileInfos

    // Total the file sizes
    let totalSize = Array.sum fileSizes
    // Return the total size of the files
    totalSize
```

Pipelining

- |> is an infix polymorphic function which simply applies it's second argument to it's first.

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

```
'a -> ('a -> 'b) -> 'b
```

- Example: get a student name from a tuple:

```
let studentName = fst ("Mihai",123456)
```

- Using |>:

```
let studentName2 = ("Mihai",123456) |> fst
```

- This is called *pipelining*, and is a common style in F#
  - values flow through the functions in the pipeline from left to right.
  - the functions in the pipeline are often formed by partial applications.
  - at the start of the pipeline is the value to begin with `("Mihai",123456)`.

Pipelining - Example

- can continually reapply `|>` to chain functions together
  - Result of one function is piped into the next
  - needs a place holder variable to kick off the pipelining (e.g.: folder)
- Rewriting the `sizeOfFolder` function:

```
let sizeOfFolderPiped folder =
    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)
    let totalSize =
        folder
        |> getFiles
        |> Array.map (fun file -> new FileInfo(file))
        |> Array.map (fun info -> info.Length)
        |> Array.sum
    totalSize
```

- Mix of pipe-forward and currying.
  - |> takes a value and a function that only takes one parameter,
  - but map take two. This works because of partial application of functions

Function Composition I

- A well-known operation in mathematics, defined thus:

  $(f \circ g)(x) = g(f(x))$, for all x

- F# definition: Functional composition

```
(>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
let (>>) f g x = g (f x)
```

- Similar to the “forward pipe” operator |>: we have

```
x |> f |> g = (f >> g) x
```

Composition - Example

- (>>) joins two functions together
  - function on the left is called first
- Rewrite the `sizeOfFolder` function:

- .

More Examples

```
square x = x * x

toString (x : int) = x.ToString()

strLen (x : string) = x.Length

lenOfSquare = square >> toString >> strLen

square 125?   gives 15625

lenOfSquare 125?  gives 5
```

Backward Pipe and Composition

- Pipe-backward operator <|
  - accepts a function on the left and applies it to a value on the right.
- seems unnecessary:

```
let (<|) f x = f x
List.iter (printfn "%d") [1 .. 3]
List.iter (printfn "%d") <| [1 .. 3]
```

- it allows you to change precedence (the order in which functions are applied)
- arguments are evaluated left-to-right
- to call a function and pass the result to another function:
  - add parentheses around the expression or
  - use the pipe-backward operator
-

```
printfn "The result of sprintf is %s" (sprintf "(%d, %d)" 1 2)
printfn "The result of sprintf is %s" <| sprintf "(%d, %d)" 1 2
```

Backward composition <<

- backward composition <<
  - takes two functions and applies the right function first and then the left.
  - It is useful when you want to express ideas in reverse order.

```
let (<<) f g x = f(g x)
```

- Example: take the square of the negation of a number

```
let square x = x * x
let negate x = -x
(square >> negate) 10?  gives -100
(square << negate) 10?  gives 100
```

- Example 2: filter out empty lists in a list of lists.
  - << changes the way the code reads to the programmer:

```
[ [1]; []; [4;5;6]; [3;4]; []; []; []; [9] ] |> List.filter (not << List.isEmpty)
gives [[1]; [4;5;5]; [3;4]; [9]]
```

- The |>, <|, >>, and << operators serve as a way to clean up F# code.
- Avoid them if adding them would only add clutter or confusion.

Data Type Declarations I

- We can define our own data types in F# by:
- Type abbreviation:

```
type lineNumber = int
type entryIndex = words * (lineNum list)
// parametric/generic
type funAndInv<'a,'b> = ('a->'b)*('b->'a)
```

- Discriminated Unions:

```
type Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
```

  - type
    - Color is a type just like int, bool
  - constructors
    - Black, Blue, etc. are constructors just like true, [ ]
  - elements
    - elements of Color are the values Black, Blue, etc.
- Syntax rule:
  - names of user-defined constructors must start with Upper-case

Discriminated Unions I

- We can represent a value that may or may not exist.

- Option Type:

Signature	Name	Description	Example
'a option	Option type	An optional value	Some(3), None

```
let myFirstOption = Some(12)
let mySecondOption = "Excellent"
let myNoOption = None
```

- Option type is a simple discriminated union in the F# core library:

```
type Option<'a> =
    | Some of 'a
    | None
```

Discriminated Unions II

- defining deck of cards, a card's suit can be represented thus:

```
// Discriminated union for a card's suit
type Suit = | Heart | Diamond | Spade | Club
let suits = [ Heart; Diamond; Spade; Club ]
```

- data can be associated with each union case

```
// Discriminated union for playing cards
type PlayingCard =  | Ace   of Suit    | King  of Suit
     | Queen of Suit     | Jack  of Suit
     | ValueCard of int * Suit
```

Discriminated Unions III

- Generating a deck of cards: