VIA University College

Programming Concepts and Languages

Spring 2024

Question 1

- What is the value of evenList1?

```
isEven x = (x % 2 = 0)
evenList1 = List.map isEven [0;1;2;3;4]
```

A. [0; 1; 2; 3; 4]

B. [0; 2; 4]

C. [true; false; true; false; true]

D. false

Question 2

- What is the value of evenList2 ?

```
isEven x = (x % 2 = 0)
evenList2 = List.filter isEven [0;1;2;3;4]
```

A. [0; 1; 2; 3; 4]

B. [0; 2; 4]

C. [true; false; true; false; true]

D. false

Question 1

- What is the value of evenList1 ?

```
isEven x = (x % 2 = 0)
evenList1 = List.map isEven [0;1;2;3;4]
```

A. [0; 1; 2; 3; 4]

B. [0; 2; 4]

C. [true; false; true; false; true]

D. false

Question 2

- What is the value of evenList2 ?

```
isEven x = (x % 2 = 0)
evenList1 = List.filter isEven [0;1;2;3;4]
```

A. [0; 1; 2; 3; 4]

B. [0; 2; 4]

C. [true; false; true; false; true]

D. false

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 ∘ 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
- Rewriting the `sizeOfFolder` function:

```
open System.IO
let sizeOfFolderComposed (* No Parameters!*) =
   let getFiles folder =
     Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)
    // The result of this expression is a function that takes
    // one parameter, which will be passed to getFiles and piped
    // through the following functions.
   getFiles
   >> Array.map (fun file -> new FileInfo(file))
   >> Array.map (fun info -> info.Length)
   >> Array.sum
//sizeOfFolderComposed : (string -> int64)
```

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;6]; [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:

```
// Use list comprehension to generate a deck of cards.
let deckOfCards =
    [
        for suit in [ Spade; Club; Heart; Diamond ] do
            yield Ace(suit)
            yield King(suit)
            yield Queen(suit)
            yield Jack(suit)
            for value in 2 .. 10 do
                yield ValueCard(value, suit)
    ]
```

Discriminated Unions IV

- Generating a deck of cards:

```
// Use list comprehension to generate a deck of cards.
let deckOfCards =
    [
        for suit in [ Spade; Club; Heart; Diamond ] do
            yield Ace(suit)
            yield King(suit)
            yield Queen(suit)
            yield Jack(suit)
            for value in 2 .. 10 do
                yield ValueCard(value, suit)
    ]
```

```
val deckOfCards: PlayingCard list =
  [Ace Spade; King Spade; Queen Spade; Jack Spade; ValueCard (2, Spade);
   ValueCard (3, Spade); ValueCard (4, Spade); ValueCard (5, Spade);
   ValueCard (6, Spade); ValueCard (7, Spade); ValueCard (8, Spade);
   ValueCard (9, Spade); ValueCard (10, Spade); Ace Club; King Club;
   Queen Club; Jack Club; ValueCard (2, Club); ValueCard (3, Club);
   ValueCard (4, Club); ValueCard (5, Club); ValueCard (6, Club);
   ValueCard (7, Club); ValueCard (8, Club); ValueCard (9, Club);
   ValueCard (10, Club); Ace Heart; King Heart; Queen Heart; Jack Heart;
   ValueCard (2, Heart); ValueCard (3, Heart); ValueCard (4, Heart);
   ValueCard (5, Heart); ValueCard (6, Heart); ValueCard (7, Heart);
   ValueCard (8, Heart); ValueCard (9, Heart); ValueCard (10, Heart);
   Ace Diamond; King Diamond; Queen Diamond; Jack Diamond;
   ValueCard (2, Diamond); ValueCard (3, Diamond); ValueCard (4, Diamond);
   ValueCard (5, Diamond); ValueCard (6, Diamond); ValueCard (7, Diamond);
   ValueCard (8, Diamond); ValueCard (9, Diamond); ValueCard (10, Diamond)]
```

Tree Structures

- Discriminated unions are ideal for tree-like data structures
- Example:
  - binary tree and a function for traversing the tree
  - constructors carry values through “of”

```
type BinaryTree =
      | Node of int * BinaryTree * BinaryTree
      | Empty

let rec printInOrder tree =
    match tree with
    | Node (data, left, right)
        -> printInOrder left
           printfn "Node %d" data
           printInOrder right
    | Empty  -> ()
```

```
      2
    /   \
   1     4
        /   \
       3     5
```

```
binTree =
  Node(2,
      Node(1, Empty, Empty),
      Node(4,
           Node(3, Empty, Empty),
           Node(5, Empty, Empty)
      )
  )
```

Pattern Match – Discriminated Unions

- Use case labels as patterns
- Example:
  - describe a pair of cards in a game of poker

```
let describeHoleCards cards =
    match cards with
    | []
    | [_]
        -> failwith "Too few cards."
    | cards when List.length cards > 2
        -> failwith "Too many cards."

    | [ Ace(_);  Ace(_)  ] -> "Pocket Rockets"
    | [ King(_); King(_) ] -> "Cowboys"

    | [ ValueCard(2, _); ValueCard(2, _)]  -> "Ducks"

    | [ Queen(_); Queen(_) ]
    | [ Jack(_);  Jack(_)  ]
        -> "Pair of face cards"
    | [ ValueCard(x, _); ValueCard(y, _) ] when x = y   -> "A Pair"

    | [ first; second ]    -> sprintf "Two cards: %A and %A" first second
```

Pattern Match–Recursive DUs

- Can use nested pattern matching
- Example:
  - describe an organization and its employees

```
type Employee = Manager of string * Employee list | Worker  of string

let rec printOrganization worker =
    match worker with
     | Worker(name) -> printfn "Employee %s" name
     // Manager with a worker list with one element
     | Manager(managerName, [ Worker(employeeName) ] )
        -> printfn "Manager %s with Worker %s" managerName employeeName
     // Manager with a worker list of two elements
     | Manager(managerName, [ Worker(employee1); Worker(employee2) ] )
        -> printfn
            "Manager %s with two workers %s and %s"
            managerName employee1 employee2
     // Manager with a list of workers
     | Manager(managerName, workers)
        -> printfn "Manager %s with workers..." managerName
           workers |> List.iter printOrganization
```

Issues with Discriminated Unions (also tuples)

- Discriminated unions are great but …
  - How do we get values out of discriminated unions?
  - No meaning associated with the values
- Consider describing a person:

```
type Person = | Person of string * string * int
let alex = Person("Alex", "Alexy", 22)
let juan = Person("Juan", "Marty", 21)
```

- Are its two string fields referring to the first and then last name, or the last and then first name?
- Records allows us o organize values into a type, as well as name those values through fields.

Record Types

- F# also has *records* (similar to simple objects)
- Basically, a record is a tuple where every field has a name
- Access is by “dot” notation
- Record fields can *not* be accessed by pattern matching

Constructing and Using Records

- Define a record type

```
type PersonRecord = {FirstName : string; LastLast : string; Age : int}
```

- Construct a record

```
him = {FirstName = “Himal"; LastName = “Sharmy"; Age = 21}
```

- Use .field to access record fields

```
printfn "%s is %d years old." him.FirstName him.Age
```

- .