# Lec 13: Active Patterns and More

## CS220: Programming Principles

Sang Kil Cha

# Active Patterns

# Active Patterns?

F# has a feature called ***active patterns*** that allows you to define custom patterns for use in pattern matching, which is particularly useful for parsing and interpreting data.

# Motivation

Suppose we take a user input as a string and we want to parse it into `MyValue`.

```
type MyValue =
  | Bool of bool
  | Integer of int
  | Float of float
  | String of string

// We want to parse a string into MyValue
val parse: string -> MyValue
```

# Implementing `parse`

```
let parse str =
  match str with
  | "true" -> Bool true
  | "false" -> Bool false
  | _ ->
    match Int32.TryParse str with
    | true, n -> Integer n
    | false, _ ->
      match Double.TryParse str with
      | true, f -> Float f
      | false, _ -> String str
```

This is not so elegant (not easy to read) because we have to use nested `match` expressions.

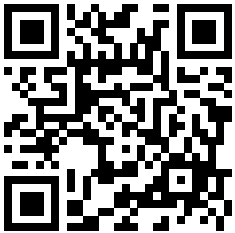
# Number One Principle

Always make your code readable and ***concise***!

# Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.



# Refactoring?

We can potentially create a function for each value type, but still quite verbose.

```
let parseBool = function
  | "true" -> Bool true |> Some
  | "false" -> Bool false |> Some
  | _ -> None

...

let parse str =
  match parseBool str with
  | Some v -> v
  | None ->
    match parseInt str with
    | Some v -> v
    | None -> ...
```

# Using Active Patterns

By defining active patterns, we can do this:

```
let parse' = function
  | BoolPattern b -> Bool b
  | IntegerPattern n -> Integer n
  | FloatPattern f -> Float f
  | s -> String s
```

# Using Active Patterns

By defining active patterns, we can do this:

```
let parse' = function
  | BoolPattern b -> Bool b
  | IntegerPattern n -> Integer n
  | FloatPattern f -> Float f
  | s -> String s
```

How do we define those patterns?

# Active Patterns

Active patterns with _ are called ***partial*** active patterns, which use an option value to represent if the type is satisfied or not.

```
let (|BoolPattern|_|) = function
  | "true" -> Some true
  | "false" -> Some false
  | _ -> None

let (|IntegerPattern|_|) (str: string) =
  match Int32.TryParse str with
  | true, n -> Some n
  | _ -> None

let (|FloatPattern|_|) (str: string) =
  match Double.TryParse str with
  | true, f -> Some f
  | _ -> None
```

# The Key Takeaway?

We can ***hide the details*** of parsing logic and make the code more readable by using active patterns.

# Understanding `TryParse`

```
System.Int32.TryParse(s: string, result: byref<int>) : bool
```

With `byref`, we can pass a reference to a variable to a function, and the function can modify the value of the variable.

```
let f (x: byref<int>) =
  x <- 42

let mutable myvalue = 0
f &myvalue
```

# Power of Pattern Matching

```
// C# style
let mutable x = 0
System.Int32.TryParse ("42", &x) |> ignore
printfn "%d" x

// F# style
let status, v = System.Int32.TryParse "42"
printfn "%b, %d" status v
```

For further information about this:
https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/byrefs.

# Other Example

Say we want to parse an integer and classify it as either `Even` or `Odd`.

```
let printEvenOrOdd n =
  if n % 2 = 0 then printfn "even"
  else printfn "odd"
```

# Using Active Patterns

```
let (|Even|Odd|) n = if n % 2 = 0 then Even else Odd

let printEvenOrOdd' = function
  | Even -> printfn "even"
  | Odd -> printfn "odd"
```

# Single-Case Active Patterns

Single-case active patterns are particularly useful for extracting partial data from complex data structures.

```
type Person = {
  FirstName: string
  LastName: string
  Email: string
}

let (|FullName|) person =
  $"{person.FirstName} {person.LastName}"

let (|Contact|) person =
  $"{person.FirstName} {person.LastName} <{person.Email}>"
```

# Two Different Representations

```
let printFullName = function
  | FullName n -> printfn "%s" n

let printContactInfo = function
  | Contact s -> printfn "%s" s

let p = { FirstName = "Alice"
          LastName = "Kim"
          Email = "alice.kim@xyz.com" }

printFullName p
printContactInfo p
```

# Disjoint or Not

Some active patterns may overlap. Consider we are finding square and cube numbers. A number can be both a square and a cube number, e.g., 64.

```
let err = 1.e-10

// check if the fractional part is small
let isNearlyIntegral (x: float) = abs (x - round x) < err

let (|Square|_|) (x: int) =
  if isNearlyIntegral (sqrt (float x)) then Some(x)
  else None

let (|Cube|_|) (x: int) =
  if isNearlyIntegral ((float x) ** (1.0 / 3.0)) then Some(x)
  else None
```

# Finding Square and Cube Numbers (cont'd)

```
let findSquareCubes n =
    match n with
    | Cube n & Square _ -> printfn "%d is a cube and a square" n
    | Cube n -> printfn "%d is a cube" n
    | Square n -> printfn "%d is a square" n
    | _ -> ()

[ 1 .. 1000 ] |> List.iter findSquareCubes
```

# Finding Square and Cube Numbers (cont'd)

```
let findSquareCubes n =
    match n with
    | Cube n & Square _ -> printfn "%d is a cube and a square" n
    | Cube n -> printfn "%d is a cube" n
    | Square n -> printfn "%d is a square" n
    | _ -> ()

[ 1 .. 1000 ] |> List.iter findSquareCubes
```

How about performance? Can you spot a problem from the above?

# Parameterized Active Patterns[1]

```
open System.Text.RegularExpressions

let (|FirstRegexGroup|_|) pattern input =
  let m = Regex.Match (input, pattern)
  if m.Success then Some m.Groups[1].Value else None

let testURL str =
  match str with
  | FirstRegexGroup "http://(.*?)/.*" domain ->
    printfn  "http domain %s" domain
  | FirstRegexGroup "mailto:.*?@(.*)" domain ->
    printfn  "email domain %s" domain
  | _ -> printfn "unknown pattern"
```

[1] To learn more about regex, visit
https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions

# I/O

# Input/Output

Open `System.IO` namespace to use I/O functions!

# Reading from a File

- `File.ReadAllText` reads the contents of a file into a string.
- `File.ReadAllLines` reads the contents of a file into a string array.
- `File.ReadAllBytes` reads the contents of a file into a byte array.

# Writing to a File

- `File.WriteAllText` writes a string to a file.
- `File.WriteAllLines` writes a string array to a file.
- `File.WriteAllBytes` writes a byte array to a file.

For example, `File.WriteAllText (fileName, text)` writes the text to the file.

# Console I/O

```
open System

Console.Write "What is your name? "
let name = Console.ReadLine ()
Console.WriteLine $"Hi, {name}!"
```

# In-Class Activity #13

# Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

# Implementing a Simple User Database

Our user data type is defined in `User.fs` as follows:

```
type User = {
  Name: string
  Password: string
}
```

And our DB is simply a list of users as defined in `DB.fs`:

```
type DB = User list
```

# Problem

Our goal here is to implement the `initializeFromCSV` function that takes in a CSV file, which stores a name-password pair per line, and returns a newly created DB. We perform a simple password validation check: the password must be at least 8 characters long and contains both letters and digits. If the password does not meet the requirement, we simply ignore the user.

The CSV file is located at `data/users.csv`.

# Conclusion

# Summary

1. Active patterns are a powerful abstraction feature in F#.
2. Hiding the details (i.e., abstraction) is the key to making the code more readable and F# provides many good ways to do so compared to other languages.

# Question?