# F#

## INTRODUCTION TO FSHARP

# OVERVIEW

- Strongly Typed programming language.
- Multi-Paradigm language mainly focuses on functional programming.
- Type Inference system.
- Syntax Customization/Domain-Specific Language
- Information Rich Programming through “Type Provider”

# BASIC SYNTAX

# BASIC SYNTAX

```
//type declaration
type Person = { Name: string; Age: int; Email: string }
//function declaration
let changeEmailTo newEmail person =
    { person with Email = newEmail }
//variable declaration
let ace = { Name = "Adison"; Age = 25; Email = "adison.prakongpan@gmail.com" }
let newEmail = "adison.prakongpan@bkk.jetabroad.com"
let ace = changeEmailTo newEmail ace
```

# FEATURES

- Type Inference
- Body Expression
- Pipeline & Curry
- Data Types
- Function Composition
- Matching Expression

# TYPE INFERENCE

# TYPE INFERENCE

- Determine types from coding/usage.
- Allow to code with less type annotation.

# TYPE INFERENCE

F#

```fsharp
let concat x y = x + y

[<EntryPoint>]
let main argv =
    let x = concat "starting" "..."
```

C#

```csharp
static string Concat(string x, string y)
{
    return x + y;
}

static void Main(string[] args)
{
    var x = Concat("starting", "...");
}
```

# TYPE INFERENCE

F#

```
let rec fib x =
    match x with
    | 0 -> 0
    | 1 -> 1
    | x -> fib (x - 1) + fib (x - 2)
```

C#

```
static int Fib(int x)
{
      switch (x)
      {
            case 0: return 0;
            case 1: return 1;
            default: return
                  Fib(x - 1) + Fib(x - 2);
      }
}
```

# TYPE INFERENCE

F#

```
let tupleArray = [
    (0, 1); (3, 4); (9, 30)
]
let tupleSeq = tupleArray :> seq<_>
```

C#

```
var tupleArray = new[]
{
    Tuple.Create(1, "2"),
    Tuple.Create(2, "4"),
    Tuple.Create(3, "6")
};
var tupleSeq =
    (IEnumerable<Tuple<int, string>>)
    tupleArray;
```

# BODY EXPRESSIONS

# BODY EXPRESSIONS

- Represent a block of statements returning values.
- Used to define/assign values to variables/data structure.

# BODY EXPRESSIONS

F#

```fsharp
let names = [| "Adison", "Ace" |]
let text =
    let sb = StringBuilder()
    for n in names do
        sb.Append(n).Append(", ")
    sb
```

C#

```csharp
var names = new[] { "Adison", "Ace" };
var sb = new StringBuilder();
foreach (var n in names)
{
    sb.Append(n).Append(", ");
}
var text = sb.ToString();
```

# BODY EXPRESSIONS

F#

```
let naturalNumbers =
    [| for i in 0..19 do yield i |]
```

C#

```
var naturalNumbers = new int[20];
for (var i = 0; i < naturalNumbers.
Length; i++)
{
    naturalNumbers[i] = i;
}
```

# DATA TYPES

# DATA TYPES

- Tuple
- Record
- Function

# DATA TYPES

```
//tuple
let person = "Adison", 25, “ace@home.th"
type Person = string * int * string
```

# DATA TYPES

```
//record
type Person = {
   Name: string
   Age: int
   Email: string
}
let ace =
    { Name = "Adison"; Age = 25; Email = “a@b.c" }
```

# DATA TYPES

```
//function: string -> Person -> Person
let changeEmailTo newEmail person =
    printfn "Changing email to %A..." newEmail
    { person with Email = newEmail }
type EmailModifier = string -> Person -> Person
```

# PIPELINE AND CURRY

# PIPELINE AND CURRY

- Enable to code function execution chaining more readable.
- Promote reusability in codes.

# PIPELINE AND CURRY

F#

```fsharp
let base2Expo =
    [ 0; 1; 2 ]
    |> Seq.map (pown 2)
    |> Seq.map (fun x -> x.ToString())
    |> String.concat "+"
```

C#

```csharp
var base2Expo =
    string.Join("+",
        new[] { 0, 1, 2 }
        .Select(x => Math.Pow(2, x))
        .Select(x => x.ToString())
    );
```

# FUNCTION COMPOSITION

# FUNCTION COMPOSITION

F#

```fsharp
let logInput x =
    printfn "%A" x
    x
let fibWithLog = fib << logInput
```

C#

```csharp
static void LogInput<T>(T x)
{
    Console.WriteLine(x);
}
var FibWithLog = (Func<int, int>)(x => {
    LogInput(x);
    return Fib(x);
});
```

# MATCH EXPRESSIONS

# MATCH EXPRESSIONS

```
match x with
| Some x when x < 30 -> printfn "[x:%A] < 30" x
| Some _ -> printfn "[x:%A] >= 30" x
| _ -> ()
```

# MATCH EXPRESSIONS

```
match p with
| name, _, _ when name = "" -> printfn "Unknown person."
| _, age, _ when y < 30 -> printfn “[age:%A] >= 30" y
| _ -> ()
```

# MATCH EXPRESSIONS

| Name | Description | Example |
|---|---|---|
| Constant pattern | Any numeric, character, or string literal, an enumeration constant, or a defined literal identifier | `1.0`, `"test"`, `30`, `Color.Red` |
| Identifier pattern | A case value of a discriminated union, an exception label, or an active pattern case | `Some(x)`<br>`Failure(msg)` |
| Variable pattern | *identifier* | `a` |
| **as** pattern | *pattern* as *identifier* | `(a, b) as tuple1` |
| OR pattern | *pattern1* \| *pattern2* | `([h] \| [h; _])` |
| AND pattern | *pattern1* & *pattern2* | `(a, b) & (_, "test")` |
| Cons pattern | *identifier* :: *list-identifier* | `h :: t` |
| List pattern | [ *pattern_1*; ... ; *pattern_n* ] | `[ a; b; c ]` |

# MATCH EXPRESSIONS

| Name | Description | Example |
|---|---|---|
| Array pattern | [\| *pattern_1*; ..; *pattern_n* ] | `[\| a; b; c \|]` |
| Parenthesized pattern | ( *pattern* ) | `( a )` |
| Tuple pattern | ( *pattern_1*, ... , *pattern_n* ) | `( a, b )` |
| Record pattern | { *identifier1* = *pattern_1*; ... ; *identifier_n* = *pattern_n* } | `{ Name = name; }` |
| Wildcard pattern | _ | `_` |
| Pattern together with type annotation | *pattern* : *type* | `a : int` |
| Type test pattern | :? *type* [ as *identifier* ] | `:? System.DateTime as dt` |
| Null pattern | null | `null` |

# Q&A