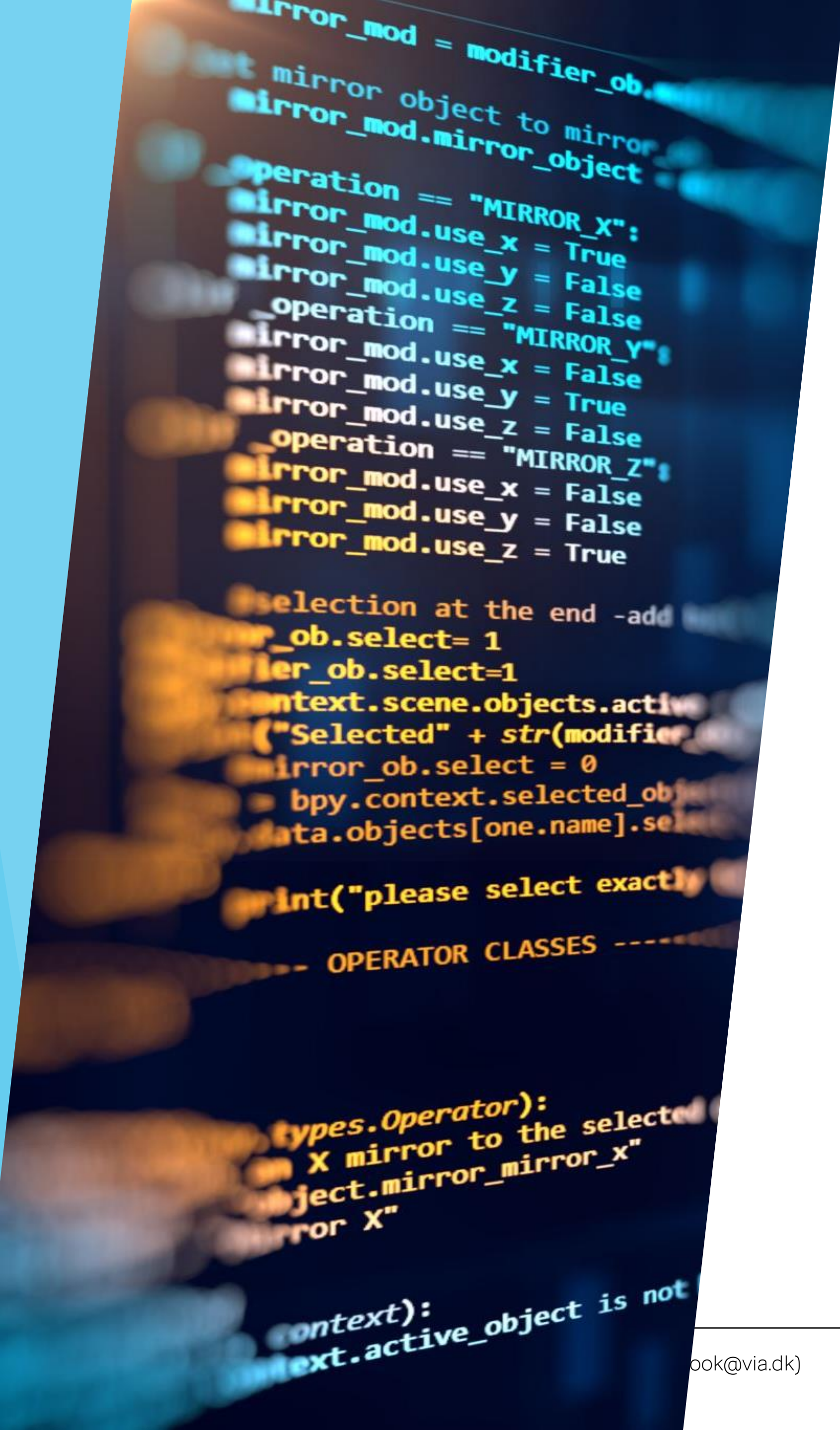


Programming Concepts and Languages

Spring 2024

Learning Objectives

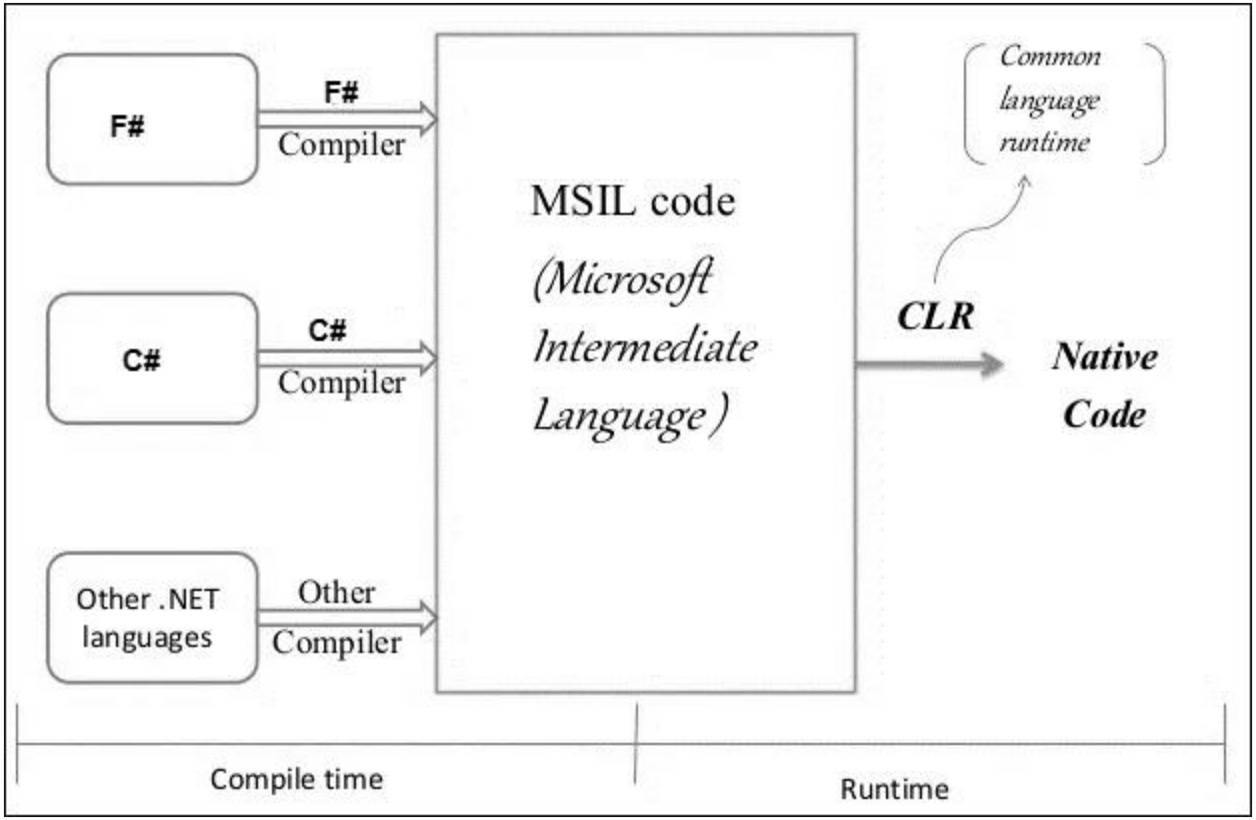
- By the end of this session, you should be able to:
  - ✓ explain F# Imperative Programming and implement simple programs using the following:
    - ✓ mutable data, variables and mutable records
    - ✓ Units of measure
    - ✓ Arrays and control structures
  - ✓ explain F# Object-Oriented Programming and implement simple programs:
    - ✓ using classes
    - ✓ construct and use a class
    - ✓ call methods and properties
  - ✓ explain F# Concurrent/Parallel Programming
    - ✓ explain the basic .NET Thread, Async framework
    - ✓ Type Provider - WorldBankData
    - ✓ implement concurrent programs using the mailboxprocessor/agent/actor model

Recall: F# is …

- a **functional** programming language
- a functional programming language for **.NET**.
- a functional and **object oriented** programming language for .NET
- a functional, object oriented and **imperative** programming language for .NET

- a functional, object oriented, imperative and **explorative** programming language for .NET
- a **multi-paradigm** programming language for .NET

Recall: CLR …

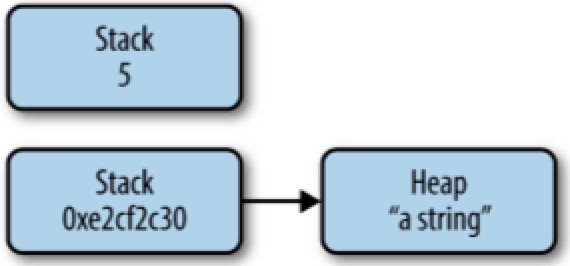

F#
F#
Compiler
C#
C#
Compiler
Other .NET languages
Other
Compiler
MSIL code
(Microsoft Intermediate Language)
Common language runtime
CLR
Native Code
Compile time
Runtime

Imperative Programming in F#

- F# has:
  - mutable data (that can be overwritten),
  - imperative control structures (loops, conditionals), and
  - iteration over sequences, lists, and arrays (similar to loops)
- **Mutable Variables**
- F# has mutable variables
  - contents can be changed
  - declared with keyword mutable:
  - `let mutable x = 5`
- Can be of any type:
  - `let mutable f = fun x -> x + 1`
- To change the contents of a mutable value
  - use the left arrow operator, **<-**

Value Types vs Reference Types

- Values stored on the stack are known as value types – fixed size
- values stored on the heap are known as reference types – pointer on the stack



- Reference Type Aliasing
  - two reference types point to the same memory address on the heap
  - modifying one value will silently modify the other because they both point to the same memory address
- Example: Array
  - modifying one value will silently modify the other because they both point to the same memory address
- .

```
let x = [|0|]
let y = x ???     x = [|0|] and y = [|0|]
x.[0] <- 3 ???    x = [|3|] and y = [|3|]
```

Reference Cells

- The ref type (ref cell) allows storing of mutable data on the heap
- enables us to bypass limitations with mutable values that are stored on the stack
- to retrieve the value of a ref cell, use the !
- to set the value, use the <- operator.
- The ref function takes a value and returns a copy of it wrapped in a ref cell

Using ref cells to mutate data

- Example:

```
let mutable planets =
    [
        "Mercury";  "Venus";     "Earth";
        "Mars";     "Jupiter";   "Saturn";
        "Uranus";   "Neptune";   "Pluto"]
```

- filter all planets not “Earth”
- get the value of the planets ref cell
- assign the new value using `<-`

```
planets <- planets |> List.filter (fun p -> p <> "Earth")
```

Mutable Records

- to use records with the imperative style
  - a record with a mutable field Miles, which can be modified

```
type MutableCar = { Make : string; Model : string;
mutable Miles : int }
let driveForASeason car =
        let rng = new Random()
        car.Miles <- car.Miles + rng.Next() % 10000
```

- one can now update record fields without being forced to clone the entire record.

```
let tesla = { Make = “Tesla"; Model = “Model X"; Miles = 0 }
let bmw = {Make = “BMW”; Model = “i4”; Miles = 50000}
```

Units of Measure

- Units of measure:
  - allow you to pass along unit information with a floating-point value
  - float, float32, decimal or signed integer types
  - in order to prevent an entire class of software defects
- Example - Fahrenheit to Celsius with units of measure

```
[<Measure>]
type fahrenheit
let printTemperature (temp : float<fahrenheit>) =
    if temp < 32.0<_>  then printfn "Below Freezing!"
    elif temp < 65.0<_>  then printfn "Cold"
    elif temp < 75.0<_>  then printfn "Just right!"
    elif temp < 100.0<_> then printfn "Hot!"
    else printfn "Scorching!"
let horsens = 59.0<fahrenheit>
```

- the function only accepts fahrenheit values.
- Calling the function with an invalid unit of measure will result in a compile-time error
- .

```
[<Measure>]
type celsius
let viborg = 12.0<celsius>
```

Arrays

- Arrays are mutable in F#
- can be constructed using array comprehensions

```
let perfectSquares = [| for i in 1 .. 7 -> i * i |]
```

- Array elements can be updated similarly to mutable record fields:

```
let a = [|1; 3; 5|]
```

- What's the value of: `a.[1] <- 7 + a.[1]`

```
a = [|1; 10; 5|]
```

- Array module has methods: like List and Seq
  - Array.iter, Array.map, Array.fold, Array.tryFind, etc.

Control Structures in F#

- F# has conditionals and loops
- The conditional statement is just the usual if-then-else:

```
if b then s1 else s2
```

- It first evaluates b, then s1 or s2 depending on the outcome of b
- With side effects, it works the same as an imperative if-then-else

While Loops

- F# has a quite conventional while loop construct

```
let mutable i = 0
while i < 5 do
   i <- i + 1
   printfn "i = %d" i
```

For Loops

- The simplest kinds of for loop:

```
for i = 1 to 5 do
    printfn "%d" i

for i = 5 downto 1 do
    printfn "%d" i
```

- The first form increments `i` by 1, the second decrements it by 1
- Numerical for loops are only supported with integers as the counter.
- if you need to loop more than System.Int32.MaxValue times
  - use enumerable for loops.

```
for i in [1 .. 5] do
    printfn "%d" i
```

For Loops with Pattern Matching

```
type Pet =
     | Cat of string * int // Name, Lives
     | Dog of string       // Name

let famousPets =
    [Dog("Lassie"); Cat("Felix", 9); Dog("Rin Tin Tin")]

for Dog(name) in famousPets do
    printfn "%s was a famous dog." name
```

- What's the output?

```
Lassie was a famous dog.
Rin Tin Tin was a famous dog
```

- The for loop iterates through a list but only executes when the element in the list is an instance of the Dog union case

.NET Interoperability

- The .NET BCL (Base Class Library) is built in an object-oriented way, so the ability to work with existing classes is essential for the interoperability.
- Many (in fact almost all) of the classes are also mutable
- Example:
  - the mutable generic ResizeArray<T> type from the BCL (ResizeArray is an alias for a type System.Collections. Generic.List to avoid a confusion with the F# list type):

```
let lst = new ResizeArray<string>()
lst.Add(″programming")
lst.Add(″paradigm")
Seq.toList list gives [‶programming"; ‶paradigm"]
```

- F# also provides a way for declaring its own classes (called *object types* in F#)
  - compiled into CLR classes or interfaces and therefore the types can be accessed from any other .NET language as well as used to extend classes written in other .NET languages.

Should I use O-O features of F#?

- In favour
  - a direct port from C# to F# without refactoring.
  - use F# primarily as an OO language, maybe as an alternative to C#.
  - need to integrate with other .NET languages
- Against
  - As a beginner coming from an imperative language, classes and similar concepts can hinder your understanding of functional programming.
  - Classes do not have the convenient "out of the box" features that the "pure" F# data
  - Types have, such as built-in equality and comparison, pretty printing, etc.
  - Classes and methods do not play well with the type inference system and higher order functions.
- Hybrid approach using pure F# types and functions, but occasionally using classes and interfaces when you need polymorphism would be the best in most cases.

Object-Oriented Programming

- F#, like other CLI languages, can use CLI types and objects through object programming.
- F# support for object programming in expressions includes:
  - Dot-notation (e.g., `x.Name`)
  - Object expressions (e.g., `{ new obj() with member x.ToString() = "hello" }`)
  - Object construction (e.g., `new Form()`)
  - named arguments (e.g., `x.Method(someArgument=1)`)
  - Named setters (e.g., `new Form(Text="Hello")`)
  - Optional arguments (e.g., `x.Method(OptionalArgument=1)`)

Defining a Class

- F# object type definitions can be class, struct, interface, enum or delegate type definitions, corresponding to the definition forms found in the C#.

- Example: a student class with a constructor taking a name, number and age, and declaring three properties (class members).

```
/// A simple student class/object type  definition
type Student(name:string, number:int, age:int) =
    member this.Name = name
    member this.Number = number
    member this.Age = age
```

Constructing and Using a Class

```
type MyViaStudy(intStarsParam:int, strStatusParam:string) =
        member this.DreamStar = 5
        member this.IncrementStars x = x + 1
```

- use the new keyword and pass in the arguments to the constructor.

```
let myVSInstance = new MyViaStudy(5,"Great!")
```

- eliminate the new and call the constructor function on its own

```
let myVSInstance = MyViaStudy(5,"Great!")
```

- Calling methods and properties

```
myVSInstance.DreamStar
myVSInstance.IncrementStars 2
```

Concurrent Programming

- What is concurrency?
  - several things happening at the same time, and maybe interacting with each other
  - spawn independent processes, which live independent lives
  - writing concurrent code that is *correct* is extremely hard!
- How can F# help us with this paradigm?
  - ✓ F# can use .NET Thread, as well as “asynchronous workflows”
  - ✓ F# has a built-in implementation of the actor model (aka MailboxProcessor)
  - F# can use the :NET Task Parallel Library to manage true CPU parallelism
  - F# has a built-in support for a functional approach that treats events as “streams“.

Parallel/Concurrent Programming

- F# has some support for parallel and concurrent processing:
- The System.Threading library gives threads
- A data type Async<'a> for asynchronous (concurrent) workflows (a kind of computation expressions)
- The System.Threading.Tasks library yields task parallelism
- Array.Parallel module provides data parallel operations on arrays

Worth Knowing

- When we use async { } we are creating objects of the type Async<‘a>
  - This is just a type that represents an asynchronous computation with a result of type ‘a
- Array.map is actually creating an Async for each member of our array
- Async.Parallel has type seq<Async<'a>> -> Async<'a []>
- Async.RunSynchronously is typed Async<'a> -> 'a

Working with Threads

- Asynchronous and parallel programming is done using *threads*
  - Spawn a new thread with:
    - a new instance of System.Threading.Thread
    - pass a Thread.Start delegate (or lambda) to its constructor
    - call the Thread object's Start method
- Example: Threads that each counts to five

```
open System
open System.Threading
  // What will execute on each thread
  let threadBody() =
    for i in 1 .. 5 do
      Thread.Sleep(100) // Wait 1/10 of a second
      printfn "[Thread %d] %d..."
        Thread.CurrentThread.ManagedThreadId
        i
```

- Spawn

```
let spawnThread() =
    let thread = new Thread(threadBody)
    thread.Start()
```

Type Providers

- Type provider is like an adapter that loads data respecting its schema and then turns this data and schema into types of target programming language
  - FSharp.Data includes Type Providers for JSON, XML, CSV, and HTML document formats and resources.
  - SQLProvider provides strongly typed access to relation databases through object mapping and F# LINQ queries.
  - FSharp.Data.SqlClient has a set of type providers for compile-time checked embedding of T-SQL in F#.
  - Azure Storage Type provider provides types for Azure Blobs, Tables, and Queues.
  - FSharp.Data.GraphQL contains the GraphQLProvider, which provides types based on a GraphQL server specified by URL.

Html Type Provider `HtmlProvider<>`

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
BeanShell: Original purpose:Application, scripting  functional:Yes
C++: Original purpose:Application, system  functional:Yes
C#: Original purpose:Application, RAD, business, client-side, general, server-side, web, game programm
ing  functional:Yes[18]
Clarion: Original purpose:General, business, web  functional:Yes[20]
Clean: Original purpose:General  functional:Yes
Clojure: Original purpose:General  functional:Yes
Cobra: Original purpose:Application, business, general, web  functional:Yes
Common Lisp: Original purpose:General  functional:Yes
Crystal: Original purpose:General purpose  functional:Yes
Curry: Original purpose:Application  functional:Yes
Cython: Original purpose:Application, general, numerical computing  functional:Yes
D: Original purpose:Application, system  functional:Yes
Dart: Original purpose:Application, web, server-side, mobile, IoT  functional:Yes
Delphi / Object Pascal: Original purpose:General purpose  functional:Yes
Dylan: Original purpose:Application  functional:Yes
Eiffel: Original purpose:General, application, business, client-side, server-side, web (EWF)  function
al:Yes[23][24]
Elixir: Original purpose:Application, distributed  functional:Yes
Erlang: Original purpose:Application, distributed  functional:Yes
FP: Original purpose:  functional:Yes
F#: Original purpose:Application  functional:Yes
Forth: Original purpose:General  functional:Yes
Fortran: Original purpose:Application, numerical computing  functional:Yes
```

pwsh
F# In...
F# In...

Code - Sign in type ProgrammingLanguages Ln 16, Col 1 (296 selected) Spaces: 4 UTF-8 CRLF F# Duet AI

Type Providers:

- WorldBankData.GetDataContext()
  - Nordic CO2 emissions