VIA University College

Programming Concepts and Languages

Spring 2024

Learning Objectives

- explain the main features of functional programming
- explain the background features of F#
- define functional programs using F# in Visual studio Code

What is Functional Programming?

- At a high level: functional programming focuses on building functions.
- The programmer declares what the program does by defining a function that maps inputs to outputs.
- Complex functions are built by composing simpler functions.
  - let square x = x*x
  - let sumSqr(x,y) = square x + square y
- i.e. functions in the mathematical sense:
  - In particular, variables are not modified by the code.
  - Instead, variables are just names for values.

Functional Programming–Continuum I

- FP departs from the imperative model by retaining the mathematical form of variables.
- This means that modifying variables is not allowed.
- Instead of evaluating a formula many times, it is placed in a function, and the function is called many times as in mathematics.

- Instead of loops , recursive functions are emphasized, as well as applying functions to each element in a list or collection.
- There is also an emphasis on writing simple but general functions. E.g., to sum the squares of the numbers from 1 to 10:

```
let k = sum [for x in 1..10 -> x*x]
```

Functional Programming – Continuum II

- In functional languages we have:
  - functions (first-class citizens like other types)
  - recursion (instead of loops)
  - expressive data types (much more than numbers)
  - advanced data structures

Functions

- Functions, mathematically: sets of pairs where no two pairs can have the same first component:

  `f ={(1, 17), (2, 32), (3, 4711)}`

  `f(2) = 32`

- Or, given the same argument, the function always returns the same value
- (cannot have `f(2) = 32 and f(2) = 33` at the same time)
- Functions model determinism: that outputs depend predictably on inputs

Define a function

- Defining a function by writing all pairs can be very tedious
- usually defined by simple rules:

  $simple(x, y, z) = x \cdot (y + z)$
- rules express abstraction: that a similar pattern holds for many different inputs
- ("For all x, y, z, $simple(x, y, z)$ equals $x \cdot (y + z)$")
- Abstraction makes definitions shorter and easier to grasp

Recursion

- Mathematical functions are often specified by recursive rules
- Recursion means that a defined entity refers to itself in the definition
- **Example**: the factorial function “!” on natural numbers

  $0! = 1$

  $n! = n \cdot (n - 1)!, \; n > 0$

- Recursion corresponds to loops in ordinary programming

Pure Functional Languages

- Pure functional languages implement mathematical functions
- A functional language is pure if there are no side-effects
- A side effect means that a function call does something more than just returning a value
- An example in Python:

```
def f(x):
      global num
      num += 1
      return num + x
print(f(3))
```

Side Effect I

- Side effect for f: global variable num is incremented for each call
- This means that f returns different values for different calls, even when called with the same argument
- Much harder to reason mathematically about such functions: for instance,
- `f(17) + f(17)≠ 2*f(17)`
- Side effects requires a more complex model, and thus makes it harder to understand the software

```
def f(x):
    global num
    num += 1
    return num + x
print(f(3))
```

Side Effect II

- In pure functional languages, functions are specified by side-effect free (declarations)
- In F#:

```
let simple x y z = x*(y + z)
```

- Each rule defines a calculation for any actual arguments:

```
simple 3 9 5 ⇒ 3*(9 + 5)
             ⇒ 3*14
             ⇒ 42
```

- Just put actual arguments into the right-hand side and that's it!

Try this

- Calculate `simple(simple 2 3 4) 5 6`
- Note that we can do the calculation in a different order
- Do we get the same result?

Side Effect III

- The mathematical view makes it possible to prove properties of programs
- When calculating, all intermediate results are mathematically equal :

```
simple 3 9 5 = 3*(9 + 5) = 3*14 = 42
```

- For instance, prove that:

`simple x y z = simple x z y` for all `x, y, z`:

```
simple x y z = x*(y + z)
             = x*(z + y)
             = simple x z y
```

- We cannot do this for functions with side-effects

Expressions, Values, and Types

- Calculation is performed on expressions:

```
simple (simple 2 3 4) 5 6
```

- Expressions are calculated into values:

```
simple (simple 2 3 4) 5 6 ⇒ 154
```

- Values are also expressions, which cannot be calculated any further
- Types represent sets of values (eg. integers and floating-point numbers)

What is F#?

- F# being a functional-first language
- a **functional** programming language for **.NET** and runs on the CLI
- 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

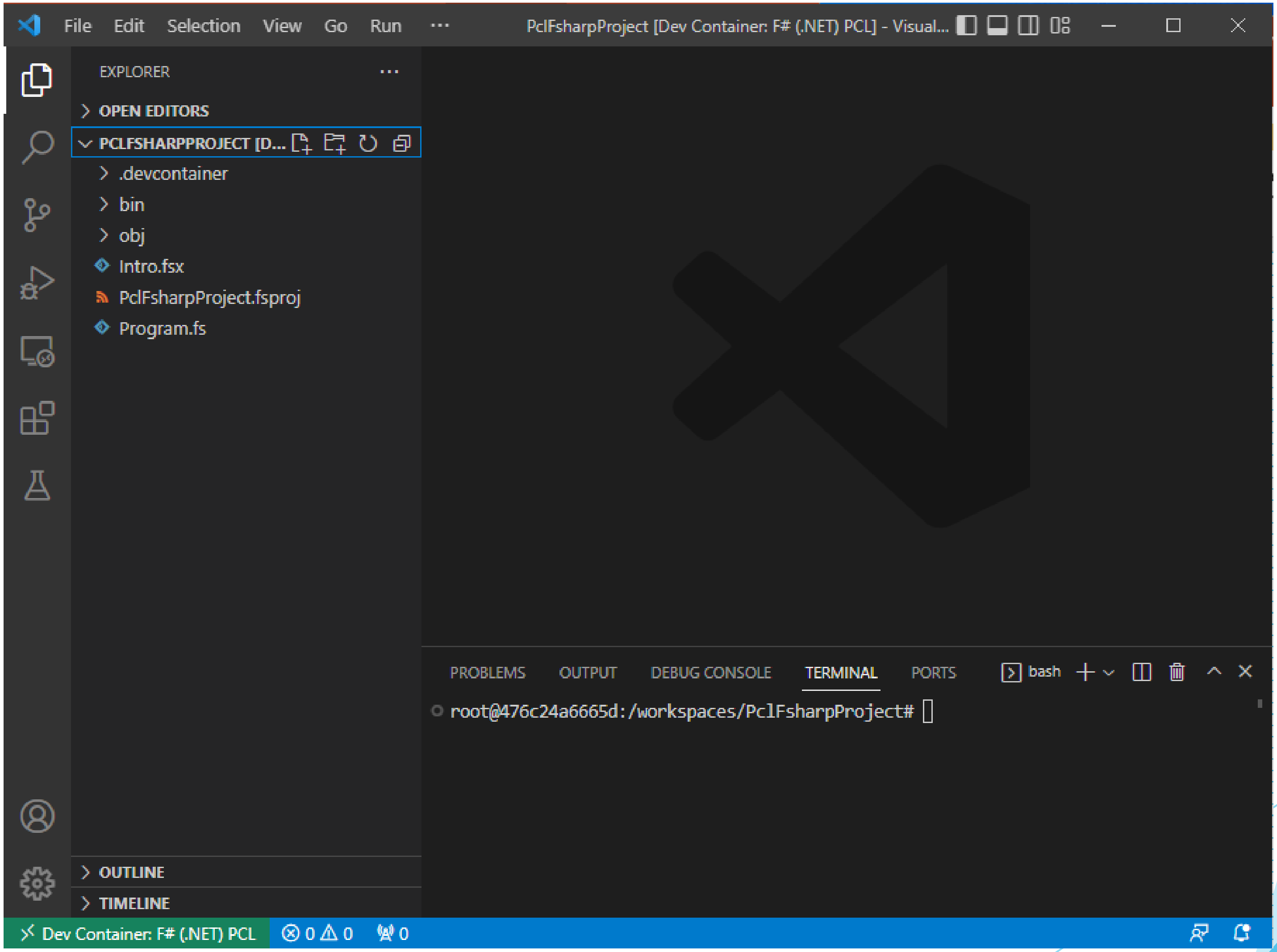
F# Features I

- It is *strict* or *call-by-value*
  - Arguments to functions are evaluated before starting to evaluate the body of the function.
- It is *Impure*: functions may have side effects
  - Generally side effects are only used where necessary.
- It has a strong static type system.
  - Types are checked (and inferred) at compile time.

F# Features II

- It is *polymorphic*.
  - functions can be applied to values of more than one type.
- It is *higher order*.
  - functions are first-class citizens.
  - i.e., they can be used just like other data types.
- It has *automatic storage management*
  - all issues of data representation and store re-use are handled by the implementation
- It has algebraic datatypes, exceptions.
- It has workflows and (.NET) objects

F# Project - VS Code



F# on Visual Studio Code

DotNetLibraryDemo.fs 1

FSharpDemo1 > DotNetLibraryDemo.fs > {} DotNetLibraryDemo > httpGet2

```fsharp
    html

let httpGet2 (url:string) = // string -> Async<string>
    async {
        let httpClient = new System.Net.Http.HttpClient()
        let! response = httpClient.GetAsync(url) |> Async.AwaitTask
        response.EnsureSuccessStatusCode () |> ignore
        let! content = response.Content.ReadAsStringAsync() |> Async.AwaitTask
        return content
    }

let viaStr = httpGet("https://www.via.dk/") // string
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
        <div id="" class="o-headline__content">
            <h1 class="o-headline__heading">
            Bliv studerende p&#229; VIA University College


            </h1>

        </div>


    </section>





<!-- Design 2019 -->
<section id="" class="m-abstract-section">
    <p id="regions_0_content_1_Manchet" class="a-abstract">
            Gør ligesom 40.000 andre. Læs en professionsbachelor, eller bliv efter- og videreuddannet hos os. Vi
```

Let's start with our first F# project

F# Syntax and examples "let" binding

```
open System

/// A very simple constant integer
let i1 = 1
/// A second very simple constant integer
let i2 = 2
/// Add two integers
let i3 = i1 + i2
/// A function on integers
let f x = 2*x*x-5*x+3
/// The result of a simple computation
let result = f (i3 + 4)
```

F# examples: Recursion and Tuples

```
/// Compute the factorial of an integer recursively
let rec factorial x =
   if x <= 1 then
       1
   else
      x * factorial (x - 1)



// A simple pair of two integers
let pointA = (32, 42)
// A simple tuple of an integer, a string and a double-
precision floating point number
let dataB = (1, ”Thursday",25.15)
/// A function that swaps the order of two values in a tuple
let swap (a, b) = (b, a)
/// The result of swapping pointA
let pointB = swap pointA
```

F# examples: Lists

```
/// The empty list
let listA = [ ]
/// A list with 3 integers
let lst3 = [1; 2; 3]
let newList = 0 :: lst3
/// A list of odd integers
let oddNums = [1; 3; 5; 7; 9]
/// A list of even integers
let evenNums = [2; 4; 6; 8; 10]
/// A list with 3 integers, note head::tail
constructs a list
let listC = 1 :: [2; 3]
/// A list of characters
let vowels = ['a'; 'e'; 'i'; 'o'; 'u']

/// The squares of the first 10 integers in a list
let squaresOfOneToTen = [for n in 1..10 -> n*n]
```