Programming and Problem Solving 2024 - The Unofficial Guide

This comprehensive guide covers the Programming and Problem Solving (PoP) course, introducing F# programming and fundamental software development concepts. Each week's material builds upon the previous, progressing from basic syntax to advanced topics in functional programming.

Table of contents

F# Foundations: Week 1 Syntax and Concepts Guide

Description

This comprehensive guide covers the fundamental syntax and concepts of F# introduced in the first week of the Programming in Practice (PoP) course. It is designed to serve both beginners and experienced programmers, offering dual-level explanations for each topic.

The document provides a structured overview of F# basics, including:

1. Core syntax elements like value binding and function definition
2. Basic types and operators
3. Function application and tuple usage
4. Key F# features such as type inference and immutability
5. Introductory concepts in functional programming

Each section includes:

- Clear syntax examples
- Beginner-friendly explanations
- Insights for experienced programmers
- Practical code snippets demonstrating usage

This guide aims to establish a solid foundation in F# programming, enabling students to understand and write basic F# code. It serves as both a learning tool and a quick reference for the initial stages of F# development.

Whether you're new to programming or coming from another language, this document will help you grasp the essentials of F# syntax and its functional programming paradigm.

Use it alongside your course materials to reinforce your understanding and as a handy reference during coding exercises and projects.

Week 1 Overview: Programming Concepts from the Lecture

Table of Contents

Introduction to Programming

Programming is defined by Peter Naur as:

> ⚠ "Programming is the activity of matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer"

Naur also emphasizes that:

> ⚠ "Programming primarily must be the programmers' building up knowledge of a certain kind"

This definition highlights the importance of understanding the problem domain and translating real-world concepts into formal computer instructions.

Course Structure and Grading

- **Course Load**: 15 ECTS (~23 hours/week over 18 weeks)
- **Weekly Structure**:
  - Tuesdays 8-12: Lectures, exercises, live coding
  - Thursdays: 2-hour slot with Teaching Assistants (TAs)
- **Grading System**:
  - Pass/fail based on 4 assignments
  - Requirements:
    - Pass one of the first two assignments
    - Pass both of the last two assignments
  - Mandatory resubmission for all assignments
  - Re-exam: Oral exam if needed

Computer Basics

Computer Components

1. **CPU (Central Processing Unit)**
    - Function: Executes instructions
    - Core of the computer's processing power
2. **Memory**
    - Function: Stores instructions and data
    - Characteristics:
        - Transient (content disappears without power)
        - Organized as an array of bytes
3. **Disk**
    - Function: Permanent storage for instructions and data
    - Retains information even when power is off

Computer Architecture

- Main components (CPU, Memory, Disk) are connected via buses
- Peripherals include:
    - Screen
    - Keyboard
    - Network interfaces
    - Other input/output devices

Operating System

The operating system acts as an intermediary between hardware and software, providing:

1. Graphical User Interface (GUI)
2. File System management
3. Terminal emulator (console)

File System

- Hierarchical structure (tree-like)
- Root directory:
    - macOS: "/"
    - Windows: "C:\"
- Paths:
    - Absolute: Starts from the root (e.g., "/Users/username/Documents")
    - Relative: Starts from current directory (e.g., "Documents/project")

Terminal and Shell

- **Terminal**: Input/Output interface for text-based interactions
- **Shell**: Command interpreter (e.g., zsh on macOS, PowerShell on Windows)
- Basic commands:
    - `ls` or `dir`: List directory contents
    - `pwd` or `gl`: Print working directory
    - `mkdir`: Make directory
    - `cd`: Change directory
    - `touch` (macOS) or `ni` (Windows): Create new file
    - `rm -rf` (macOS) or `rmdir` (Windows): Remove directory

Editor

- Purpose: Write and edit file contents
- Recommended: Visual Studio Code (VS Code)
- VS Code can be configured as an Integrated Development Environment (IDE)

Programming Concepts

Program Components

A program consists of three main components:

1. Data
2. Computations
3. Actions

Data

- Definition: Collection of bits interpreted as information
- Characteristics:
  - Values have types
  - Finite representation in memory
  - Values are immutable
- Types of values:
  - Primitive (e.g., integers, booleans, strings, floats)
  - Composite (e.g., tuples, lists)

Computation

- Definition: Transformation from input to output
- Implemented with pure functions

- Properties:
    - Deterministic (same input always produces same output)
    - No side effects

Actions

- Functions with side effects (e.g., I/O operations, file manipulations)
- Characteristics:
    - Results can change based on when or how many times they're run
    - More challenging to test compared to pure computations

Expressions

- Fundamental unit of computation
- Evaluated to compute a value
- In F#, everything is an expression

F# Basics

- **F# Interactive**: Read-Eval-Print Loop (REPL) environment
- Expressions are terminated with ";;"
- Binding (assigning names to values) uses "let" keyword
- Binding environment: Collection of active bindings
- F# scripts: Files containing F# code (usually with .fsx extension)

Problem Solving Strategies

1. **Use Examples**
    - Start with concrete input/output examples

- Consider edge cases (e.g., empty input, maximum values)

2. **Sequence of Steps**
   - Identify intermediate goals
   - Split the main problem into smaller sub-problems
3. **Ken's Detailed Method for Function Design**
   1. Write a brief description of the function's purpose
   2. Choose an appropriate name for the function
   3. Write down test examples (input and expected output)
   4. Determine the types of inputs and outputs
   5. Generate the function code (including any necessary helper functions)
   6. Write comprehensive test cases
   7. Provide concise documentation for the function

By following these strategies and understanding the core concepts, you'll be well-equipped to tackle programming problems in this course and beyond.

F# Syntax Covered in Week 1

1. Basic Syntax

Value Binding

```
let x = expression
```

Beginner: This is how you give a name to a value in F#. It's like saying "let x be equal to this expression."

Experienced: The `let` keyword introduces a binding. It's immutable by default, promoting functional programming principles. The right-hand side is evaluated eagerly.

Example:

```
let myNumber = 42
let myName = "Alice"
```

Function Definition

```
let functionName parameter1 parameter2 = expression
```

Beginner: This is how you create a function. You give it a name and tell it what inputs (parameters) it should expect.

Experienced: Functions in F# are first-class values. This syntax defines a function value. It's curried by default, allowing partial application.

Example:

```
let add x y = x + y
let greet name = printfn "Hello, %s!" name
```

Type Annotations

```
let functionName (parameter: type) = expression
```

Beginner: Sometimes you need to tell F# what type of data you're working with. This is how you do that.

Experienced: F# uses type inference, but explicit annotations can be used for clarity or to resolve ambiguities. They're also useful for documentation and type-driven development.

Example:

```
let square (x: int) = x * x
let convertToString (num: float) : string = string num
```

2. Basic Types

Integer Types

- `int`: 32-bit signed integer
- `int64` or `L`: 64-bit signed integer
- `byte` or `uy`: 8-bit unsigned integer
- `uint64`: 64-bit unsigned integer

Beginner: These are different ways to represent whole numbers. Some can be bigger than others or only positive.

Experienced: F# provides a range of integer types for different precision needs. The suffixes (`L`, `uy`) are type-specific literals that help with type inference.

Example:

```
let smallNumber = 42
let largeNumber = 9876543210L
let unsignedByte = 255uy
let unsignedLarge = 18446744073709551615UL
```

Floating-Point Types

- `float`: 64-bit double-precision floating-point number

Beginner: This is for numbers with decimal points.

Experienced: F# uses `float` as an alias for `System.Double`. It follows IEEE 754 standard for floating-point arithmetic.

Example:

```
let pi = 3.14159
let avogadro = 6.022e23
```

String Type

- `string`: Unicode text

Beginner: This is for text.

Experienced: Strings in F# are immutable and represent Unicode text. They're instances of `System.String`.

Example:

```
let greeting = "Hello, World!"
let multiLine = "This is a
multi-line string"
```

3. Operators

Arithmetic Operators

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division

Beginner: These work just like in math class.

Experienced: Operators in F# are actually functions. They can be overloaded and used in partial application.

Example:

```
let sum = 5 + 3
let difference = 10 - 7
let product = 4 * 6
let quotient = 15 / 3
```

String Concatenation

- `+`: Used to concatenate strings

Beginner: This is how you stick two pieces of text together.

Experienced: String concatenation with `+` is syntactic sugar. For performance-critical operations, consider using `StringBuilder`.

Example:

```
let firstName = "John"
let lastName = "Doe"
let fullName = firstName + " " + lastName
```

4. Function Application

```
functionName argument1 argument2
```

Beginner: This is how you use a function. You write its name and then give it the information it needs.

Experienced: Function application in F# is left-associative and doesn't require parentheses. This allows for clean, pipeline-style programming.

Example:

```
let add x y = x + y
let result = add 3 4  // result is 7
```

```
let greet name = printfn "Hello, %s!" name
greet "Alice"  // Prints: Hello, Alice!
```

5. Tuples

```
let tuple = (item1, item2)
```

Beginner: This is a way to group multiple values together.

Experienced: Tuples are structural types in F#. They're immutable and can be pattern-matched. They're often used for multiple return values.

Example:

```
let person = ("Alice", 30)
let point3D = (1.0, 2.0, 3.0)
```

Tuple Pattern Matching in Function Parameters

```
let functionName (a, b) = expression
```

Beginner: This is a way to break apart a tuple when you're defining a function.

Experienced: This syntax demonstrates F#'s pattern matching capabilities. It's a form of destructuring that can be used in various contexts, not just function parameters.

Example:

```
let addCoordinates (x, y) = x + y
let result = addCoordinates (3, 4)  // result is 7

let printPerson (name, age) = printfn "%s is %d years old" name age
printPerson ("Bob", 25)  // Prints: Bob is 25 years old
```

6. Comments

Beginner: These are notes in your code that F# ignores. They're for humans to read.

Experienced: F# supports single-line, multi-line, and XML documentation comments. XML comments can be used to generate documentation.

Example:

```
// This is a single-line comment

(*
   This is a
   multi-line comment
*)

/// <summary>
/// This is an XML documentation comment
/// </summary>
/// <param name="x">The first number to add</param>
/// <param name="y">The second number to add</param>
/// <returns>The sum of x and y</returns>
let add x y = x + y
```

7. Modules and Namespaces

Opening a Module

```
open ModuleName
```

Beginner: This is how you tell F# you want to use stuff from another part of the code.

Experienced: `open` brings a module's contents into scope. It's similar to `import` in other languages but with some unique behaviors in F#'s module system.

Example:

```
open System
let now = DateTime.Now
```

8. Basic I/O

Printing to Console

```
printfn "formatString %s %d" stringArg intArg
```

Beginner: This is how you make your program show text on the screen.

Experienced: `printfn` is a type-safe formatting function. It uses printf-style format strings and is resolved at compile-time.

Example:

```
let name = "Alice"
let age = 30
printfn "%s is %d years old" name age
// Prints: Alice is 30 years old
```

9. F# Interactive

Beginner: This is a tool where you can try out F# code piece by piece.

Experienced: F# Interactive (FSI) is a REPL that supports incremental compilation. It's valuable for exploratory programming and testing.

Example:

```
> let x = 42;;
val x : int = 42

> let double x = x * 2;;
val double : x:int -> int

> double 21;;
val it : int = 42
```

10. Type Inference

Beginner: F# is smart and can often figure out what type of data you're using without you telling it.

Experienced: F#'s type inference is based on Hindley-Milner type inference. It works across function boundaries and is a key feature of the language.

Example:

```
let numbers = [1; 2; 3; 4; 5]  // F# infers this is a list of integers
let square x = x * x  // F# infers x is a number (generic)
```

11. Immutability

Beginner: Once you give something a value in F#, it doesn't change unless you explicitly say it can.

Experienced: Immutability is a core principle in F#, promoting functional programming paradigms. It aids in creating thread-safe and easier-to-reason-about code.

Example:

```
let x = 5
// x <- 10  // This would cause a compilation error
let y = x + 5  // This creates a new binding, y, with value 10
```

12. Function Composition

Beginner: You can combine simple functions to make more complex ones.

Experienced: F# provides the `>>` and `<<` operators for forward and backward function composition, enabling point-free style programming. We are not allowed to use this yet though.

Example:

```
let add1 x = x + 1
let double x = x * 2
let add1ThenDouble x = double (add1 x)
```

13. Partial Application

Beginner: You can use a function even if you don't give it all the information it usually needs.

Experienced: Partial application is a consequence of currying in F#. It's powerful for creating specialized functions from more general ones and is fundamental to many functional programming patterns.

Example:

```
let add x y = x + y
let add5 = add 5  // Partially applied function
let result = add5 3  // result is 8
```

Week 1 Hands-On: F# Exercises

1. Basic F# Operations

Integer Addition

```
let addInt (a: int) (b: int) = a + b
```

This function adds two integers.

Float Addition

```
let addFloat (a: float) (b: float) = a + b
```

This function adds two floating-point numbers.

String Concatenation

```
let concatStrings (a: string) (b: string) = a + b
```

This function concatenates two strings.

Byte Addition

```
let addByte (a: byte) (b: byte) = a + b
```

This function adds two byte values.

Unsigned Long Integer Addition

```
let addUnsignedLong (a: uint64) (b: uint64) = a + b
```

This function adds two unsigned 64-bit integers.

Variable Bindings and Overflow Examples

```
let x = 3 + 6
let y = 3L + 6L
```

```
let canAddXY = false
let overflowOne = 255uy + 1uy = 0uy
let overflowTwo = 255uy + 2uy = 1uy
```

These lines demonstrate variable binding and overflow behavior with byte operations.

2. Geometric Calculations

Circle Area

```
let circleArea radius = Math.PI * radius * radius
```

Calculates the area of a circle given its radius.

Circle Perimeter

```
let circlePerimeter radius = 2. * Math.PI * radius
```

Calculates the perimeter of a circle given its radius.

Square Area

```
let squareArea (length: float) = length * length
```

Calculates the area of a square given the length of one side.

Square Perimeter

```
let squarePerimeter length = 4. * length
```

Calculates the perimeter of a square given the length of one side.

3. Average Functions

Integer Average

```
let avg x y = (x + y) / 2
```

Calculates the average of two numbers (note: this performs integer division).

Float Average

```
let avgFloat x y = (x + y) / 2.
```

Calculates the average of two floating-point numbers.

4. Rational Number Operations

Rational numbers are represented as tuples `(numerator, denominator)`.

Addition of Rationals

```
let qplus (a, b) (c, d) = ((a * d + b * c), (b * d))
```

Subtraction of Rationals

```
let qminus (a, b) (c, d) = ((a * d - b * c), (b * d))
```

Multiplication of Rationals

```
let qmult (a, b) (c, d) = ((a * c), (b * d))
```

Division of Rationals

```
let qdiv (a, b) (c, d) = ((a * d), (b * c))
```

String Representation of Rationals

```
let toString (a: int, b: int) = string a + "/" + string b
```

Example Usage

```
printfn "1/2 + 1/3 = %s" (toString (qplus (1, 2) (1, 3)))
```

This line demonstrates how to use the rational number functions and print the result.

Notes

- The rational number operations do not include simplification or error handling for division by zero.
- The `avg` function uses integer division, which may lead to unexpected results for odd sums.
- The geometric functions use `float` type for precision in calculations.