F# Cheatsheet

A concise reference for the F# programming language, covering syntax, data types, control flow, and common libraries.

Basic Syntax & Types

Basic Syntax

Comments	`//` for single-line, `(* ... *)` for multi-line.
Value Binding	`let x = 5` (immutable). `let mutable y = 10` (mutable).
Function Definition	`let add x y = x + y`
Type Inference	F# infers types automatically.
Semicolons	Generally not required unless multiple expressions are on the same line.

Basic Data Types

Integer	`int` , `int8` , `int16` , `int32` , `int64` , `uint8` , `uint16` , `uint32` , `uint64`
Floating Point	`float` , `float32` (single-precision)
Boolean	`bool` (true or false)
String	`string` (immutable sequence of characters)
Character	`char` (single Unicode character)
Unit	`unit` (equivalent to `void` in C#)

Type Annotations

You can explicitly specify types using `:` .

Example: `let x: int = 5`

Function type annotation: `let add (x: int) (y: int) : int = x + y`

Control Flow

If/Then/Else

```
if condition then
 expression1
else
 expression2
```

Example:

```
let x = 10
if x > 5 then
 printfn "x is greater than 5"
else
 printfn "x is not greater than 5"
```

Match Expressions

Powerful pattern matching construct.

```
match value with
 | pattern1 -> expression1
 | pattern2 -> expression2
 | _ -> expressionN // Default case
```

Example:

```
let describeNumber x =
 match x with
 | 0 -> "Zero"
 | 1 -> "One"
 | _ -> "Many"
```

Loops

`for` loop

```
for i = start to end do
 expression
done
```

Example:

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

`while` loop

```
while condition do
 expression
done
```

Example:

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

Data Structures

Tuples

Definition	`(value1, value2, ...)`
Example	`let myTuple = (1, "hello", true)`
Accessing Elements	`let (a, b, c) = myTuple` or `fst myTuple` , `snd myTuple` (for 2-tuples).

Lists

Definition	`[value1; value2; ...]` (immutable)
Example	`let myList = [1; 2; 3; 4]`
Adding elements	`::` (cons operator) - adds to the beginning. `myList @ [5]` (appends a list)
Common functions	`List.map` , `List.filter` , `List.fold` , `List.iter`

Arrays

Definition	`[| value1; value2; ... |]` (mutable by default)
Example	`let myArray = [| 1; 2; 3; 4 |]`
Accessing elements	`myArray.[index]`
Slicing	`myArray.[1..3]`

Records

Definition	`type MyRecord = { Field1: type1; Field2: type2 }`
Example	`type Person = { Name: string; Age: int }` `let person1 = { Name = "Alice"; Age = 30 }`
Accessing fields	`person1.Name` , `person1.Age`

Discriminated Unions

Definition

```
type MyUnion =
 | Case1 of type1
 | Case2 of type2
 | Case3
```

Example

```
type Result =
 | Success of string
 | Failure of int

let result1 = Success
"Operation completed"
let result2 = Failure 500
```

Pattern Matching

```
match result1 with
 | Success msg -> printfn "%s"
msg
 | Failure code -> printfn
"Error code: %d" code
```

Functions

Function Basics

Definition	`let functionName parameter1 parameter2 = expression`
Example	`let square x = x * x`
Calling a Function	`square 5`

Lambda Expressions

Definition	`fun parameter -> expression`
Example	`let addOne = fun x -> x + 1`
Usage	`List.map addOne [1; 2; 3]`

Partial Application

Concept	Applying some arguments to a function, resulting in a new function that accepts the remaining arguments.

Example	`let add x y = x + y`
	`let addFive = add 5`
	`addFive 3 // Returns 8`

Function Composition

Operator	`>>` (forward composition), `<<` (backward composition)

Example	`let square x = x * x`
	`let addOne x = x + 1`
	`let squareThenAddOne = square >> addOne`
	`squareThenAddOne 5 // Returns 26`

Recursion

Keyword	`rec`

Example	`let rec factorial n =`
	`  if n <= 1 then 1`
	`  else n * factorial (n - 1)`