F#

Functional Programming

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

F# helps you in the daily development of the mainstream commercial business software. This tutorial provides a brief knowledge about F# and its features, and also provides the various structures and syntaxes of its methods and functions.

Audience

This tutorial has been designed for beginners in F#, providing the basic to advanced concepts of the subject.

Prerequisites

Before starting this tutorial you should be aware of the basic understanding of Functional Programming, C# and .Net.

Copyright & Disclaimer

Table of Contents

1. F# – Overview

F# is a functional programming language. To understand F# constructs, you need to read a couple of lines about the programming paradigm named **Functional Programming**.

Functional programming treats computer programs as mathematical functions. In functional programming, the focus would be on constants and functions, instead of variables and states. Because functions and constants are things that don't change.

In functional programming, you will write modular programs, i.e., the programs would consist of functions that will take other functions as input.

Programs written in functional programming language tend to be concise.

About F#

Following are the basic information about F#:

- It was developed in 2005 at Microsoft Research.
- It is a part of Microsoft's family of .Net language.
- It is a functional programming language.
- It is based on the functional programming language OCaml.

Features of F#

- It is .Net implementation of OCaml.
- It compiles .Net CLI (Common Language Interface) byte code or MSIL (Microsoft Intermediate Language) that runs on CLR (Common Language Runtime).
- It provides type inference.
- It provides rich pattern matching constructs.
- It has interactive scripting and debugging capabilities.
- It allows writing higher order functions.
- It provides well developed object model.

Uses of F#

F# is normally used in the following areas:

- Making scientific model
- Mathematical problem solving
- Artificial intelligence research work
- Financial modelling
- Graphic design
- CPU design
- Compiler programming
- Telecommunications

It is also used in CRUD apps, web pages, GUI games and other general purpose programs.

2. F# – Environment Setup

The tools required for F# programming are discussed in this chapter.

Integrated Development Environment (IDE) for F#

Microsoft provides Visual Studio 2013 for F# programming.

The free Visual Studio 2013 Community Edition is available from Microsoft's official website. Visual Studio 2013 Community and above comes with the Visual F# Tools. The Visual F# Tools include the command-line compiler (fsc.exe) and F# Interactive (fsi.exe).

Using these tools, you can write all kinds of F# programs from simple command-line applications to more complex applications. You can also write F# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler.

You can download it from Microsoft Visual Studio. It gets automatically installed in your machine.

Writing F# Programs on Linux

Please visit the F# official website for the latest instructions on getting the tools as a Debian package or compiling them directly from the source: http://fsharp.org/use/linux/.

Try it Option Online

We have set up the F# Programming environment online. You can easily compile and execute all the available examples online along with doing your theory work. It gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using the Try it option or use the url: http://www.compileonline.com/.

```
(* This is a comment *)
(* Sample Hello World program using F# *)
printfn "Hello World!"
```

For most of the examples given in this tutorial, you will find a Try it option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.

3. F# – Program Structure

F# is a Functional Programming language.

In F#, functions work like data types. You can declare and use a function in the same way like any other variable.

In general, an F# application does not have any specific entry point. The compiler executes all top-level statements in the file from top to bottom.

However, to follow procedural programming style, many applications keep a single top level statement that calls the main loop.

The following code shows a simple F# program:

```
open System

(* This is a
multi-line comment *)

// This is a single-line comment

let sign num =
    if num > 0 then "positive"
    elif num < 0 then "negative"
    else "zero"

let main() =
    Console.WriteLine("sign 5: {0}", (sign 5))

main()
```

When you compile and execute the program, it yields the following output:

```
sign 5: positive
```

Please note that:

- An F# code file might begin with a number of **open** statements that is used to import namespaces.
- The body of the files includes other functions that implement the business logic of the application.
- The main loop contains the top executable statements.

4. F# – Basic Syntax

You have seen the basic structure of an F# program, so it will be easy to understand other basic building blocks of the F# programming language.

Tokens in F#

An F# program consists of various tokens. A token could be a keyword, an identifier, a constant, a string literal, or a symbol. We can categorize F# tokens into two types:

- Keywords
- Symbol and Operators

F# Keywords

The following table shows the keywords and brief descriptions of the keywords. We will discuss the use of these keywords in subsequent chapters.

Keyword	Description
abstract	Indicates a method that either has no implementation in the type in which it is declared or that is virtual and has a default implementation.
and	Used in mutually recursive bindings, in property declarations, and with multiple constraints on generic parameters.
as	Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.
assert	Used to verify code during debugging.
base	Used as the name of the base class object.
begin	In verbose syntax, indicates the start of a code block.

class	In verbose syntax, indicates the start of a class definition.
default	Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.
delegate	Used to declare a delegate.
do	Used in looping constructs or to execute imperative code.
done	In verbose syntax, indicates the end of a block of code in a looping expression.
downcast	Used to convert to a type that is lower in the inheritance chain.
downto	In a **for** expression, used when counting in reverse.
elif	Used in conditional branching. A short form of else if.
else	Used in conditional branching.
end	In type definitions and type extensions, indicates the end of a section of member definitions. In verbose syntax, used to specify the end of a code block that starts with the begin keyword.
exception	Used to declare an exception type.
extern	Indicates that a declared program element is defined in another binary or assembly.
false	Used as a Boolean literal.

finally	Used together with try to introduce a block of code that executes regardless of whether an exception occurs.
for	Used in looping constructs.
fun	Used in lambda expressions, also known as anonymous functions.
function	Used as a shorter alternative to the fun keyword and a match expression in a lambda expression that has pattern matching on a single argument.
global	Used to reference the top-level .NET namespace.
if	Used in conditional branching constructs.
in	Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.
inherit	Used to specify a base class or base interface.
inline	Used to indicate a function that should be integrated directly into the caller's code.
interface	Used to declare and implement interfaces.
internal	Used to specify that a member is visible inside an assembly but not outside it.
lazy	Used to specify a computation that is to be performed only when a result is needed.
let	Used to associate, or bind, a name to a value or function.

let!	Used in asynchronous workflows to bind a name to the result of an asynchronous computation, or, in other computation expressions, used to bind a name to a result, which is of the computation type.
match	Used to branch by comparing a value to a pattern.
member	Used to declare a property or method in an object type.
module	Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.
mutable	Used to declare a variable, that is, a value that can be changed.
namespace	Used to associate a name with a group of related types and modules, to logically separate it from other code.
new	Used to declare, define, or invoke a constructor that creates or that can create an object. Also used in generic parameter constraints to indicate that a type must have a certain constructor.
not	Not actually a keyword. However, not struct in combination is used as a generic parameter constraint.
null	Indicates the absence of an object. Also used in generic parameter constraints.
of	Used in discriminated unions to indicate the type of categories of values, and in delegate and exception declarations.
open	Used to make the contents of a namespace or module available without qualification.

or	Used with Boolean conditions as a Boolean or operator. Equivalent to ||. Also used in member constraints.
override	Used to implement a version of an abstract or virtual method that differs from the base version.
private	Restricts access to a member to code in the same type or module.
public	Allows access to a member from outside the type.
rec	Used to indicate that a function is recursive.
return	Used to indicate a value to provide as the result of a computation expression.
return!	Used to indicate a computation expression that, when evaluated, provides the result of the containing computation expression.
select	Used in query expressions to specify what fields or columns to extract. Note that this is a contextual keyword, which means that it is not actually a reserved word and it only acts like a keyword in appropriate context.
static	Used to indicate a method or property that can be called without an instance of a type, or a value member that is shared among all instances of a type.
struct	Used to declare a structure type. Also used in generic parameter constraints. Used for OCaml compatibility in module definitions.
then	Used in conditional expressions.

	Also used to perform side effects after object construction.
to	Used in for loops to indicate a range.
true	Used as a Boolean literal.
try	Used to introduce a block of code that might generate an exception. Used together with with or finally.
type	Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.
upcast	Used to convert to a type that is higher in the inheritance chain.
use	Used instead of let for values that require Dispose to be called to free resources.
use!	Used instead of let! in asynchronous workflows and other computation expressions for values that require Dispose to be called to free resources.
val	Used in a signature to indicate a value, or in a type to declare a member, in limited situations.
void	Indicates the .NET void type. Used when interoperating with other .NET languages.
when	Used for Boolean conditions (*when guards*) on pattern matches and to introduce a constraint clause for a generic type parameter.
while	Introduces a looping construct.
with	Used together with the match keyword in pattern matching expressions. Also used in object expressions, record copying

	expressions, and type extensions to introduce member definitions, and to introduce exception handlers.
yield	Used in a sequence expression to produce a value for a sequence.
yield!	Used in a computation expression to append the result of a given computation expression to a collection of results for the containing computation expression.

Some reserved keywords came from the OCaml language:

asr	land	lor	lsl	lsr	lxor	mod	sig

Some other reserved keywords are kept for future expansion of F#.

atomic	break	checked	component	const	constraint	constructor
continue	eager	event	external	fixed	functor	include
method	mixin	object	parallel	process	protected	pure
sealed	tailcall	trait	virtual	volatile		

Comments in F#

F# provides two types of comments:

- One line comment starts with // symbol.
- Multi line comment starts with (* and ends with *).

A Basic Program and Application Entry Point in F#

Generally, you don't have any explicit entry point for F# programs. When you compile an F# application, the last file provided to the compiler becomes the entry point and all top level statements in that file are executed from top to bottom.

A well-written program should have a single top-level statement that would call the main loop of the program.

A very minimalistic F# program that would display 'Hello World' on the screen:

```
(* This is a comment *)

(* Sample Hello World program using F# *)

printfn "Hello World!"
```

When you compile and execute the program, it yields the following output:

```
Hello World!
```

5. F# – Data Types

The data types in F# can be classified as follows:

- Integral types
- Floating point types
- Text types
- Other types

Integral Data Types

The following table provides the integral data types of F#. These are basically integer data types.

F# Type	Size	Range	Example	Remarks
sbyte	1 byte	-128 to 127	42y -11y	8-bit signed integer
byte	1 byte	0 to 255	42uy 200uy	8-bit unsigned integer
int16	2 bytes	-32768 to 32767	42s -11s	16-bit signed integer
uint16	2 bytes	0 to 65,535	42us 200us	16-bit unsigned integer
int/int32	4 bytes	-2,147,483,648 to 2,147,483,647	42 -11	32-bit signed integer

uint32	4 bytes	0 to 4,294,967,295	42u 200u	32-bit unsigned integer
int64	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	42L -11L	64-bit signed integer
uint64	8 bytes	0 to 18,446,744,073,709,551,615	42UL 200UL	64-bit unsigned integer
bigint	At least 4 bytes	any integer	42I 1499999 9999999 9999999 9999999 9999I	arbitrary precision integer

Example

```
(* single byte integer *)
let x = 268.97f
let y = 312.58f
let z = x + y

printfn "x: %f” x
printfn "y: %f” y
printfn "z: %f” z

(* unsigned 8-bit natural number *)

let p = 2uy
let q = 4uy
let r = p + q
```

```
printfn "p: %i" p
printfn "q: %i" q
printfn "r: %i" r

(* signed 16-bit integer *)
let a = 12s
let b = 24s
let c = a + b

printfn "a: %i" a
printfn "b: %i" b
printfn "c: %i" c

(* signed 32-bit integer *)

let d = 212l
let e = 504l
let f = d + e

printfn "d: %i" d
printfn "e: %i" e
printfn "f: %i" f
```

When you compile and execute the program, it yields the following output:

```
x: 1
y: 2
z: 3
p: 2
q: 4
r: 6
a: 12
b: 24
```

```
c: 36
d: 212
e: 504
f: 716
```

Floating Point Data Types

The following table provides the floating point data types of F#.

F# Type	Size	Range	Example	Remarks
float32	4 bytes	±1.5e-45 to ±3.4e38	42.0F -11.0F	32-bit signed floating point number (7 significant digits)
float	8 bytes	±5.0e-324 to ±1.7e308	42.0 -11.0	64-bit signed floating point number (15-16 significant digits)
decimal	16 bytes	±1.0e-28 to ±7.9e28	42.0M -11.0M	128-bit signed floating point number (28-29 significant digits)
BigRational	At least 4 bytes	Any rational number.	42N -11N	Arbitrary precision rational number. Using this type requires a reference to FSharp.PowerPack.dll.

Example

```
(* 32-bit signed floating point number *)
(* 7 significant digits *)
let d = 212.098f
let e = 504.768f
let f = d + e

printfn "d: %f" d
printfn "e: %f" e
printfn "f: %f" f
```

```
(* 64-bit signed floating point number *)
(* 15-16 significant digits *)
let x = 21290.098
let y = 50446.768
let z = x + y

printfn "x: %g" x
printfn "y: %g" y
printfn "z: %g" z
```

When you compile and execute the program, it yields the following output:

```
d: 212.098000
e: 504.768000
f: 716.866000
x: 21290.1
y: 50446.8
z: 71736.9
```

Text Data Types

The following table provides the text data types of F#.

F# Type	Size	Range	Example	Remarks
char	2 bytes	U+0000 to U+ffff	'x' '\t'	Single unicode characters
string	20 + (2 * string's length) bytes	0 to about 2 billion characters	"Hello" "World"	Unicode text

Example

```
let choice = 'y'
```

```
let name = "Zara Ali"
let org = "Tutorials Point"

printfn "Choice: %c" choice
printfn "Name: %s" name
printfn "Organisation: %s" org
```

When you compile and execute the program, it yields the following output:

```
Choice: y
Name: Zara Ali
Organisation: Tutorials Point
```

Other Data Types

The following table provides some other data types of F#.

F# Type	Size	Range	Example	Remarks
bool	1 byte	Only two possible values, true or false	true false	Stores boolean values

Example

```
let trueVal = true
let falseVal = false
printfn "True Value: %b" (trueVal)
printfn "False Value: %b" (falseVal)
```

When you compile and execute the program, it yields the following output:

```
True Value: true
False Value: false
```

6. F# – Variables

A variable is a name given to a storage area that our programs can manipulate. Each variable has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Variable Declaration in F#

The **let** keyword is used for variable declaration:

For example,

```
let x = 10
```

It declares a variable x and assigns the value 10 to it.

You can also assign an expression to a variable:

```
let x = 10
let y = 20
let z = x + y
```

The following example illustrates the concept:

Example

```
let x = 10
let y = 20
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

When you compile and execute the program, it yields the following output:

```
x: 10
```

```
y: 20

z: 30
```

Variables in F# are **immutable**, which means once a variable is bound to a value, it can't be changed. They are actually compiled as static read-only properties.

The following example demonstrates this.

Example

```
let x = 10

let y = 20

let z = x + y


printfn "x: %i" x

printfn "y: %i" y

printfn "z: %i" z


let x = 15

let y = 20

let z = x + y


printfn "x: %i" x

printfn "y: %i" y

printfn "z: %i" z
```

When you compile and execute the program, it shows the following error message:

```
Duplicate definition of value 'x'

Duplicate definition of value 'y'

Duplicate definition of value 'z'
```

Variable Definition with Type Declaration

A variable definition tells the compiler where and how much storage for the variable should be created. A variable definition may specify a data type and contains a list of one or more variables of that type as shown in the following example.

Example

```
let x:int32 = 10
let y:int32 = 20
let z:int32 = x + y

printfn "x: %d" x
printfn "y: %d" y
printfn "z: %d" z

let p:float = 15.99
let q:float = 20.78
let r:float = p + q

printfn "p: %g" p
printfn "q: %g" q
printfn "r: %g" r
```

When you compile and execute the program, it shows the following error message:

```
x: 10
y: 20
z: 30
p: 15.99
q: 20.78
r: 36.77
```

Mutable Variables

At times you need to change the values stored in a variable. To specify that there could be a change in the value of a declared and assigned variable, in later part of a program, F# provides the **mutable** keyword. You can declare and assign mutable variables using this keyword, whose values you will change.

The **mutable** keyword allows you to declare and assign values in a mutable variable.

You can assign some initial value to a mutable variable using the **let** keyword. However, to assign new subsequent value to it, you need to use the **<-** operator.

For example,

```
let mutable x = 10

x <- 15
```

The following example will clear the concept:

Example

```
let mutable x = 10

let y = 20

let mutable z = x + y

printfn "Original Values:"

printfn "x: %i" x

printfn "y: %i" y

printfn "z: %i" z


printfn "Let us change the value of x"

printfn "Value of z will change too."

x <- 15

z <- x + y

printfn "New Values:"

printfn "x: %i" x

printfn "y: %i" y

printfn "z: %i" z
```

When you compile and execute the program, it yields the following output:

```
Original Values:
x: 10
y: 20
z: 30
Let us change the value of x
Value of z will change too.
New Values:
x: 15
y: 20
z: 35
```

7. F# – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. F# is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Comparison Operators
- Boolean Operators
- Bitwise Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by F# language. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
**	Exponentiation Operator, raises an operand to the power of another	B**A will give 20^{10}

Example

```
let a : int32 = 21
let b : int32 = 10
let mutable c = a + b
printfn "Line 1 - Value of c is %d" c
c <- a - b;
printfn "Line 2 - Value of c is %d" c
```

```
c <- a * b;
printfn "Line 3 - Value of c is %d" c
c <- a / b;
printfn "Line 4 - Value of c is %d" c
c <- a % b;
printfn "Line 5 - Value of c is %d" c
```

When you compile and execute the program, it yields the following output:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
```

Comparison Operators

The following table shows all the comparison operators supported by F# language. These binary comparison operators are available for integral and floating-point types. These operators return values of type bool.

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.

>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example

```
let mutable a : int32 = 21
let mutable b : int32 = 10
if (a = b) then
   printfn "Line 1 - a is equal to b"
else
   printfn "Line 1 - a is not equal to b"

if (a < b) then
   printfn "Line 2 - a is less than b"
else
   printfn "Line 2 - a is not less than b"

if (a > b) then
   printfn "Line 3 - a is greater than b"
else
   printfn "Line 3 - a is not greater than b"

(* Lets change value of a and b *)
a <- 5
b <- 20
if  (a <= b) then
   printfn "Line 4 - a is either less than or equal to  b"
else
   printfn "Line4 - a is a is greater than b"
```

When you compile and execute the program, it yields the following output:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
```

Boolean Operators

The following table shows all the Boolean operators supported by F# language. Assume variable A holds **true** and variable B holds **false**, then:

Operator	Description	Example
&&	Called Boolean AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Boolean OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
not	Called Boolean NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not (A && B) is true.

Example

```
let mutable a : bool = true;
let mutable b : bool = true;
if ( a && b ) then
   printfn "Line 1 - Condition is true"
else
    printfn "Line 1 - Condition is not true"
if ( a || b ) then
   printfn "Line 2 - Condition is true"
else
    printfn "Line 2 - Condition is not true"
```

```
(* lets change the value of  a *)
a <- false
if ( a && b ) then
   printfn "Line 3 - Condition is true"
else
    printfn "Line 3 - Condition is not true"
if ( a || b ) then
   printfn "Line 4 - Condition is true"
else
    printfn "Line 4 - Condition is not true"
```

When you compile and execute the program, it yields the following output:

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. The truth tables for &&& (bitwise AND), ||| (bitwise OR), and ^^^ (bitwise exclusive OR) are as follows:

p	q	p &&& q	p ||| q	p ^^^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&&&B = 0000 1100

A|||B = 0011 1101

A^^^B = 0011 0001

~~~A = 1100 0011

The Bitwise operators supported by F# language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&&&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A &&& B) will give 12, which is 0000 1100
|||	Binary OR Operator copies a bit if it exists in either operand.	(A ||| B) will give 61, which is 0011 1101
^^^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^^^ B) will give 49, which is 0011 0001
~~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~~~A) will give -61, which is 1100 0011 in 2's complement form.
<<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A <<< 2 will give 240 which is 1111 0000
>>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >>> 2 will give 15 which is 0000 1111

Example

```
let a : int32 = 60 // 60 = 0011 1100
let b : int32 = 13 // 13 = 0000 1101
let mutable c : int32 = 0

c <- a &&& b // 12 = 0000 1100
printfn "Line 1 - Value of c is %d" c
```
~~~

```
c <- a ||| b // 61 = 0011 1101
printfn "Line 2 - Value of c is %d" c

c <- a ^^^ b // 49 = 0011 0001
printfn "Line 3 - Value of c is %d" c

c <- ~~~a // -61 = 1100 0011
printfn "Line 4 - Value of c is %d" c

c <- a <<< 2 // 240 = 1111 0000
printfn "Line 5 - Value of c is %d" c

c <- a >>> 2 // 15 = 0000 1111
printfn "Line 6 - Value of c is %d" c
```

When you compile and execute the program, it yields the following output:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is 49
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```