Functional Programming

tutorials point
SIMPLY EASY LEARNING

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

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com
# Table of Contents

## About the Tutorial

## Audience

## Prerequisites

## Copyright & Disclaimer

## Table of Contents

## 1. F# – OVERVIEW

### About F#

### Features of F#

### Uses of F#

## 2. F# – ENVIRONMENT SETUP

### Integrated Development Environment (IDE) for F#

### Writing F# Programs on Linux

## 3. F# – PROGRAM STRUCTURE

## 4. F# – BASIC SYNTAX

### Tokens in F#

### Comments in F#

### A Basic Program and Application Entry Point in F#

## 5. F# – DATA TYPES

### Integral Data Types

### Floating Point Data Types

### Text Data Types

### Other Data Types

## 6. F# – VARIABLES

### Variable Declaration in F#
Variable Definition with Type Declaration

Mutable Variables

7. F# – OPERATORS ................................................................. 22
   Arithmetic Operators .......................................................... 22
   Comparison Operators ....................................................... 23
   Boolean Operators .............................................................. 25
   Bitwise Operators ............................................................... 26
   Operators Precedence .......................................................... 28

8. F# – DECISION MAKING .................................................... 30
   F# - if /then Statement ....................................................... 31
   F# - if /then/else Statement ............................................... 32
   F# - if /then/elif/else Statement ......................................... 33
   F# - Nested if Statements .................................................. 34

9. F# – LOOPS ....................................................................... 35
   F# - for...to and for... downto Expressions ............................ 36
   F# - for... in Expressions ..................................................... 38
   F# - While...do Expressions .................................................. 39
   F# - Nested Loops .................................................................. 39

10. F# – FUNCTIONS ............................................................... 41
    Defining a Function ............................................................ 41
    Parameters of a Function ..................................................... 41
    Recursive Functions ............................................................ 43
    Arrow Notations in F# .......................................................... 44
    Lambda Expressions ............................................................ 45
    Function Composition and Pipelining ................................... 45

11. F# – STRINGS .................................................................... 47
12. F# – OPTIONS .................................................................51
   Using Options ..................................................................51
   Option Properties and Methods .......................................52

13. F# – TUPLES ..................................................................54
   Accessing Individual Tuple Members ..................................55

14. F# – RECORDS ..............................................................56
   Defining a Record ................................................................56
   Creating a Record .............................................................56

15. F# – LISTS ...................................................................59
   Creating and Initializing a List ...........................................59
   Properties of List Data Type ..............................................61
   Basic Operations on List ....................................................62

16. F# – SEQUENCES ..........................................................70
   Defining Sequences ..........................................................70
   Creating Sequences and Sequence Expressions .................70
   Basic Operations on Sequence ..........................................72

17. F# – SETS ..................................................................80
   Creating Sets ....................................................................80
   Basic Operations on Sets ................................................80

18. F# – MAPS ..................................................................84
   Creating Maps ....................................................................84
   Basic Operations on Maps ................................................85
19. F# – DISCRIMINATED UNIONS ................................................................. 88

20. F# – MUTABLE DATA ............................................................................ 91
    Mutable Variables ................................................................................... 91
    Uses of Mutable Data ............................................................................. 93

21. F# – ARRAYS .......................................................................................... 95
    Creating Arrays ....................................................................................... 95
    Basic Operations on Arrays ..................................................................... 96
    Creating Arrays Using Functions ............................................................ 101
    Searching Arrays ..................................................................................... 104

22. F# – MUTABLE LISTS ............................................................................ 106
    Creating a Mutable List .......................................................................... 106
    The List(T) Class ..................................................................................... 106

23. F# – MUTABLE DICTIONARY ................................................................. 114
    Creating of a Mutable Dictionary ............................................................ 114
    The Dictionary(TKey, TValue) Class ....................................................... 114

24. F# – BASIC IO ......................................................................................... 118
    Core.Printf Module .................................................................................. 118
    Format Specifications .............................................................................. 119
    The Console Class ................................................................................... 121
    The System.IO Namespace ..................................................................... 126

25. F# – GENERICS ....................................................................................... 131
    Generic Class .......................................................................................... 132

26. F# – DELEGATES ..................................................................................... 133
    Declaring Delegates .............................................................................. 133
27. F# – ENUMERATIONS ............................................................................................................. 136
   Declaring Enumerations ......................................................................................................... 136

28. F# – PATTERN MATCHING .................................................................................................. 138
   Pattern Matching Functions ..................................................................................................... 140
   Adding Filters or Guards to Patterns ....................................................................................... 140
   Pattern Matching with Tuples .................................................................................................. 141
   Pattern Matching with Records .............................................................................................. 142

29. F# – EXCEPTION HANDLING .............................................................................................. 143
   Examples of Exception Handling ............................................................................................ 145

30. F# – CLASSES ..................................................................................................................... 149
   Constructor of a Class ............................................................................................................. 150
   Let Bindings ............................................................................................................................ 151

31. F# – STRUCTURES ............................................................................................................. 152

32. F# – OPERATOR OVERLOADING ...................................................................................... 154
   Implementation of Operator Overloading ................................................................................ 154

33. F# – INHERITANCE ............................................................................................................ 156
   Base Class and Sub Class ........................................................................................................ 156
   Overriding Methods ................................................................................................................ 158
   Abstract Classes ..................................................................................................................... 159

34. F# – INTERFACES ............................................................................................................. 162
   Calling Interface Methods ........................................................................................................ 163
   Interface Inheritance ............................................................................................................... 164

35. F# – EVENTS ..................................................................................................................... 166
   The Event Class and Event Module ....................................................................................... 166
Creating Events .................................................................................................................. 167

36. F# – MODULES ........................................................................................................ 171

37. F# – NAMESPACES .................................................................................................... 174

Declaring a Namespace ..................................................................................................... 174
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.
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/](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/](http://www.compileonline.com/).

```fsharp
(* 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.
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:

```fsharp
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.
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.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abstract</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>and</code></td>
<td>Used in mutually recursive bindings, in property declarations, and with multiple constraints on generic parameters.</td>
</tr>
<tr>
<td><code>as</code></td>
<td>Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.</td>
</tr>
<tr>
<td><code>assert</code></td>
<td>Used to verify code during debugging.</td>
</tr>
<tr>
<td><code>base</code></td>
<td>Used as the name of the base class object.</td>
</tr>
<tr>
<td><code>begin</code></td>
<td>In verbose syntax, indicates the start of a code block.</td>
</tr>
<tr>
<td>keyword</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>class</strong></td>
<td>In verbose syntax, indicates the start of a class definition.</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.</td>
</tr>
<tr>
<td><strong>delegate</strong></td>
<td>Used to declare a delegate.</td>
</tr>
<tr>
<td><strong>do</strong></td>
<td>Used in looping constructs or to execute imperative code.</td>
</tr>
<tr>
<td><strong>done</strong></td>
<td>In verbose syntax, indicates the end of a block of code in a looping expression.</td>
</tr>
<tr>
<td><strong>downcast</strong></td>
<td>Used to convert to a type that is lower in the inheritance chain.</td>
</tr>
<tr>
<td><strong>downto</strong></td>
<td>In a <code>for</code> expression, used when counting in reverse.</td>
</tr>
<tr>
<td><strong>elif</strong></td>
<td>Used in conditional branching. A short form of <code>else if</code>.</td>
</tr>
<tr>
<td><strong>else</strong></td>
<td>Used in conditional branching.</td>
</tr>
</tbody>
</table>
| **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. |
<p>| <strong>exception</strong> | Used to declare an exception type. |
| <strong>extern</strong> | Indicates that a declared program element is defined in another binary or assembly. |
| <strong>false</strong> | Used as a Boolean literal. |</p>
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finally</td>
<td>Used together with try to introduce a block of code that executes regardless of whether an exception occurs.</td>
</tr>
<tr>
<td>for</td>
<td>Used in looping constructs.</td>
</tr>
<tr>
<td>fun</td>
<td>Used in lambda expressions, also known as anonymous functions.</td>
</tr>
<tr>
<td>function</td>
<td>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.</td>
</tr>
<tr>
<td>global</td>
<td>Used to reference the top-level .NET namespace.</td>
</tr>
<tr>
<td>if</td>
<td>Used in conditional branching constructs.</td>
</tr>
<tr>
<td>in</td>
<td>Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.</td>
</tr>
<tr>
<td>inherit</td>
<td>Used to specify a base class or base interface.</td>
</tr>
<tr>
<td>inline</td>
<td>Used to indicate a function that should be integrated directly into the caller's code.</td>
</tr>
<tr>
<td>interface</td>
<td>Used to declare and implement interfaces.</td>
</tr>
<tr>
<td>internal</td>
<td>Used to specify that a member is visible inside an assembly but not outside it.</td>
</tr>
<tr>
<td>lazy</td>
<td>Used to specify a computation that is to be performed only when a result is needed.</td>
</tr>
<tr>
<td>let</td>
<td>Used to associate, or bind, a name to a value or function.</td>
</tr>
<tr>
<td>Keyword</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>let!</td>
<td>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.</td>
</tr>
<tr>
<td>match</td>
<td>Used to branch by comparing a value to a pattern.</td>
</tr>
<tr>
<td>member</td>
<td>Used to declare a property or method in an object type.</td>
</tr>
<tr>
<td>module</td>
<td>Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.</td>
</tr>
<tr>
<td>mutable</td>
<td>Used to declare a variable, that is, a value that can be changed.</td>
</tr>
<tr>
<td>namespace</td>
<td>Used to associate a name with a group of related types and modules, to logically separate it from other code.</td>
</tr>
</tbody>
</table>
| 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. |
<p>| 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. |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| **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. |
<p>| <strong>then</strong> | Used in conditional expressions. |</p>
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also used to perform side effects after object construction.</td>
<td></td>
</tr>
<tr>
<td><strong>to</strong></td>
<td>Used in for loops to indicate a range.</td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>Used as a Boolean literal.</td>
</tr>
<tr>
<td><strong>try</strong></td>
<td>Used to introduce a block of code that might generate an exception. Used together with <code>with</code> or <code>finally</code>.</td>
</tr>
<tr>
<td><strong>type</strong></td>
<td>Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.</td>
</tr>
<tr>
<td><strong>upcast</strong></td>
<td>Used to convert to a type that is higher in the inheritance chain.</td>
</tr>
<tr>
<td><strong>use</strong></td>
<td>Used instead of <code>let</code> for values that require <code>Dispose</code> to be called to free resources.</td>
</tr>
<tr>
<td><strong>use!</strong></td>
<td>Used instead of <code>let!</code> in asynchronous workflows and other computation expressions for values that require <code>Dispose</code> to be called to free resources.</td>
</tr>
<tr>
<td><strong>val</strong></td>
<td>Used in a signature to indicate a value, or in a type to declare a member, in limited situations.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td>Indicates the .NET <code>void</code> type. Used when interoperating with other .NET languages.</td>
</tr>
<tr>
<td><strong>when</strong></td>
<td>Used for Boolean conditions (when guards) on pattern matches and to introduce a constraint clause for a generic type parameter.</td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>Introduces a looping construct.</td>
</tr>
<tr>
<td><strong>with</strong></td>
<td>Used together with the match keyword in pattern matching expressions. Also used in object expressions, record copying</td>
</tr>
</tbody>
</table>
expressions, and type extensions to introduce member definitions, and to introduce exception handlers.

<table>
<thead>
<tr>
<th><em>yield</em></th>
<th>Used in a sequence expression to produce a value for a sequence.</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>yield!</em></td>
<td>Used in a computation expression to append the result of a given computation expression to a collection of results for the containing computation expression.</td>
</tr>
</tbody>
</table>

Some reserved keywords came from the OCaml language:

<table>
<thead>
<tr>
<th>asr</th>
<th>land</th>
<th>lor</th>
<th>lsl</th>
<th>lsr</th>
<th>lxor</th>
<th>mod</th>
<th>sig</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>atomic</th>
<th>break</th>
<th>checked</th>
<th>component</th>
<th>const</th>
<th>constraint</th>
<th>constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>eager</td>
<td>event</td>
<td>external</td>
<td>fixed</td>
<td>functor</td>
<td>include</td>
</tr>
<tr>
<td>method</td>
<td>mixin</td>
<td>object</td>
<td>parallel</td>
<td>process</td>
<td>protected</td>
<td>pure</td>
</tr>
<tr>
<td>sealed</td>
<td>tailcall</td>
<td>trait</td>
<td>virtual</td>
<td>volatile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:

```fsharp
(* 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!
```
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.

<table>
<thead>
<tr>
<th>F# Type</th>
<th>Size</th>
<th>Range</th>
<th>Example</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbyte</td>
<td>1 byte</td>
<td>-128 to 127</td>
<td>42y</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>0 to 255</td>
<td>42uy</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>int16</td>
<td>2 bytes</td>
<td>-32768 to 32767</td>
<td>42s</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>uint16</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
<td>42us</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>int/132</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>42</td>
<td>32-bit signed integer</td>
</tr>
</tbody>
</table>

21
<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
<td>42u, 200u</td>
</tr>
<tr>
<td>int64</td>
<td>8 bytes</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
<td>42L, -11L</td>
</tr>
<tr>
<td>uint64</td>
<td>8 bytes</td>
<td>0 to 18,446,744,073,709,551,615</td>
<td>42UL, 200UL</td>
</tr>
<tr>
<td>bigint</td>
<td>At least 4 bytes</td>
<td>any integer</td>
<td>42I, 1499999, 9999999, 9999999, 9999999, 9999I</td>
</tr>
</tbody>
</table>

**Example**

(* single byte integer *)

```fsharp
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 *)

```fsharp
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
### Floating Point Data Types

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

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

### Example

(* 32-bit signed floating point number *)

(* 7 significant digits *)

```fsharp
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#.

<table>
<thead>
<tr>
<th>F# Type</th>
<th>Size</th>
<th>Range</th>
<th>Example</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>2 bytes</td>
<td>U+0000 to U+ffff</td>
<td>'x' 't'</td>
<td>Single unicode characters</td>
</tr>
<tr>
<td>string</td>
<td>20 + (2 * string's length) bytes</td>
<td>0 to about 2 billion characters</td>
<td>&quot;Hello&quot; &quot;World&quot;</td>
<td>Unicode text</td>
</tr>
</tbody>
</table>

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#.

<table>
<thead>
<tr>
<th>F# Type</th>
<th>Size</th>
<th>Range</th>
<th>Example</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 byte</td>
<td>Only two possible values, true or false</td>
<td>true, false</td>
<td>Stores boolean values</td>
</tr>
</tbody>
</table>

**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
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,

```fsharp
let x = 10
```

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

You can also assign an expression to a variable:

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

The following example illustrates the concept:

**Example**

```fsharp
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
```
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**

```fsharp
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

```fsharp
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,

```fsharp
let mutable x = 10
x <- 15
```

The following example will clear the concept:

Example

```fsharp
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
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:

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

### Example

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

let 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Checks if the values of two operands are equal or not, if yes then condition becomes true.</td>
<td>(A == B) is not true.</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.</td>
<td>(A &lt;&gt; B) is true.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt; B) is not true.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt; B) is true.</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Condition</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt;= B) is not true.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt;= B) is true.</td>
</tr>
</tbody>
</table>

**Example**

```fsharp
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 "Line 4 - a is a is greater than b"
```
When you compile and execute the program, it yields the following output:

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1</td>
<td>a is not equal to b</td>
</tr>
<tr>
<td>Line 2</td>
<td>a is not less than b</td>
</tr>
<tr>
<td>Line 3</td>
<td>a is greater than b</td>
</tr>
<tr>
<td>Line 4</td>
<td>a is either less than or equal to b</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Called Boolean AND operator. If both the operands are non-zero, then condition becomes true.</td>
<td>(A &amp;&amp; B) is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>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.</td>
<td>not (A &amp;&amp; B) is true.</td>
</tr>
</tbody>
</table>

**Example**

```fsharp
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 | 0 |   1     |   0  |   1  |
| 1 | 1 |   1     |   1  |   0  |

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

A = 0011 1100
B = 0000 1101

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

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

**Example**

```fsharp
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
```


```fsharp
let c = a || b
printfn "Line 1 - Value of c is %d" c

let c = a ^^^ b
printfn "Line 2 - Value of c is %d" c

let c = ~~~a
printfn "Line 3 - Value of c is %d" c

let c = a <<< 2
printfn "Line 4 - Value of c is %d" c

let c = a >>> 2
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 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
```
End of ebook preview
If you liked what you saw...
Buy it from our store @ https://store.tutorialspoint.com