免費電子書

學習

F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fsharp

It is an unofficial and free F# ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official F#.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

1: F

F“”。 。

F - - IL.NETC F;Mono .NET CoreWindows.NET Framework。

1.x	2005-05-01
2.0	2010-04-01
3.0	2012-08-01
3.1	2013101
4	201571

Examples

Visual StudioexpresscommunityF。F。http://fsharp.org/use/windows/。

OS X.

Xamarin StudioF。VS Code for OS X Microsoft。
VS Code`VS Code Quick Open Ctrl + P ext install Ionide-fsharp`

Visual Studio for Mac 。

。

Linux

AptYum`mono-completefsharp`。 Visual Studio Code`ionide-fsharp`Atom`ionide-installer`。
http://fsharp.org/use/linux/ 。

“HelloWorld”STDOUT0

```
[<EntryPoint>]
let main argv =
    printfn "Hello, World!"
    0
```

- `[<EntryPoint>]` - .net “”。
- `let main argv` - `mainargv`。 `argv`。 。
- `printfn "Hello, World!"` - `printfn`**。
- `0` - F。 `0`。

**。 TextWriterFormat 。 “hello world”。

F

FInteractiveREPLF。

FVisual Studio`"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"`FInteractive 。 LinuxOS X`fsharpi /usr/bin/usr/local/bin`F - `PATHfsharpi`。

F

```
> let i = 1 // fsi prompt, declare i
- let j = 2 // declare j
- i+j // compose expression
- ;; // execute commands

val i : int = 1 // fsi output started, this gives the value of i
val j : int = 2 // the value of j
val it : int = 3 // computed expression

> #quit;; //quit fsi
```

`#help;;`

`;;`REPL。

F https://riptutorial.com/zh-TW/fsharp/topic/817/f-

2: .NET CoreF

Examples

dotnet CLI

.NET CLI

```
dotnet new --lang f#
```

。

```
dotnet new -l f#
```

project.json

```
dotnet restore
```

project.lock.json。

```
dotnet run
```

。

```
dotnet new -lf#
```

```
Hello World!
[|||]
```

.NET CoreF https://riptutorial.com/zh-TW/fsharp/topic/4404/-net-coref-

3: “”

Examples

0

23。 2DRGB1int。

0 。 F。 0() 。 0。 Funit。

CvoidFunit

```
let printResult = printfn "Hello"
```

F

```
val printResult : unit = ()
```

printResultunit () unit。

unit。 F。 unit 。

```
let doMath() = 2 + 4
```

```
let doMath () = 2 + 4
```

unitint6.F

```
val doMath : unit -> int
```

()“”F。 unit。 。 unit ()0“”。

unit。 let-binding

```
module Time =
    let now = System.DateTime.Now   // value is set and fixed for duration of program
    let now() = System.DateTime.Now // value is calculated when function is called (each time)
```

“worker”2。 worker - 。 。

```
let startWorker value =
    let current = ref value
    let stop = ref false
    let nextValue () = current := !current + 1
    let stopOnNextTick () = stop := true
    let rec loop () = async {
```

```
        if !stop then
            printfn "Stopping work."
            return ()
        else
            printfn "Working on %d." !current
            do! Async.Sleep 2000
            return! loop () }
    Async.Start (loop ())
    nextValue, stopOnNextTick
```

```
let nextValue, stopOnNextTick = startWorker 12
```

- F。

```
nextValue ()
```

。

```
stopOnNextTick ()
```

。

unit“” - 。

“” https://riptutorial.com/zh-TW/fsharp/topic/2513/--

4: 1FsXamlFWPF

FWPFMVVMMVC“”。

。 。 。

。

Examples

FWPF。

F。

Windows 。

FsXaml NuGet。

。

MainWindow.xaml

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300">
    <Canvas>
        <Button Name="btnTest" Content="Test" Canvas.Left="10" Canvas.Top="10" Height="28"
Width="72"/>
    </Canvas>
</Window>
```

MainWindow.xaml.fs

```
namespace FirstDemo

type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">

type MainWindow() as this =
    inherit MainWindowXaml()

    let whenLoaded _ =
        ()

    let whenClosing _ =
        ()

    let whenClosed _ =
        ()

    let btnTestClick _ =
        this.Title <- "Yup, it works!"
```

```
    do
        this.Loaded.Add whenLoaded
        this.Closing.Add whenClosing
        this.Closed.Add whenClosed
        this.btnTest.Click.Add btnTestClick
```

App.xaml

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

App.xaml.fs

```
namespace FirstDemo

open System

type App = FsXaml.XAML<"App.xaml">

module Main =

    [<STAThread; EntryPoint>]
    let main _ =
        let app = App()
        let mainWindow = new MainWindow()
        app.Run(mainWindow) // Returns application's exit code.
```

Program.fs。

xaml 。

.NET**UIAutomationTypes**。

。

。 。

FsXaml。 XAMLCF。 。

3

。

FPower Tools / New FolderImages。

Images。

Visual Studio*Images* **Add / Existing Item** *All Files* 。 **。

Build Action*Resource* 。

MainWindow.xamlIcon。 。

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300"
    Icon="Images/MainWindow.ico">
<Canvas>
```

RebuildBuild。Visual Studio。

。 。Windows。

4

AppIcon.rc。

```
1 ICON "AppIcon.ico"
```

AppIcon.ico。

。

```
"C:\Program Files (x86)\Windows Kits\10\bin\x64\rc.exe" /v AppIcon.rc
```

rc.exe**C\ Program Filesx86\ Windows Kits**。MicrosoftWindows SDK。

AppIcon.res。

Visual Studio。 。

“ ”*AppIcon.res Images \ AppIcon.res* 。

“。 。 。

。 。 。

2

。

MyControl.xaml

```
<UserControl
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" Height="50" Width="150">
    <Canvas Background="LightGreen">
```

```
        <Button Name="btnMyTest" Content="My Test" Canvas.Left="10" Canvas.Top="10"
Height="28" Width="106"/>
    </Canvas>
</UserControl>
```

MyControl.xaml.fs

```
namespace FirstDemo

type MyControlXaml = FsXaml.XAML<"MyControl.xaml">

type MyControl() =
    inherit MyControlXaml()
```

*MyControl.xaml***Build Action***Resource* 。

MyControl“as this”。

MainWindow.xaml.fs MainWindow

```
let myControl = MyControl()
```

do -section。

```
    this.mainCanvas.Children.Add myControl |> ignore
    myControl.btnMyTest.Content <- "We're in business!"
```

do- section。

。

。 WPF。

CWPFXAML。

```
xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"
```

FsXaml。

XAML。

。

```
xmlns:xctk="clr-namespace:Xceed.Wpf.Toolkit;assembly=Xceed.Wpf.Toolkit"
```

。

```
<xctk:IntegerUpDown Name="tbInput" Increment="1" Maximum="10" Minimum="0" Canvas.Left="13"
Canvas.Top="27" Width="270"/>
```

Extended Wpf ToolkitNuGet。NuGetXAML“”。

1FsXamlFWPF https://riptutorial.com/zh-TW/fsharp/topic/9008/1-fsxamlf-wpf

5: FWPF

WPF ◦ MārisKrivtežsref ◦

1 \◦

2 \Gjallarhorn◦

@GitHub

- FSharp.ViewModule FsXaml
- Gjallarhorn

MārisKrivtežs

- FXAML - MVVMMVC ◦

  XAML◦ ◦ ◦

- FXAML - MVVM◦

Examples

FSharp.ViewModule

◦ ◦ ◦

```
namespace Score.Model

type Score = { ScoreA: int ; ScoreB: int }
type ScoringEvent = IncA | DecA | IncB | DecB | NewGame
```

◦ OOP◦

```
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update score event =
        match event with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

EventViewModelBase<'a> IObservable<'a>EventStream ◦ ScoringEvent◦

◦ Score -> ScoringEvent -> Score◦ ◦

eventHandler。 EventViewModelBase<'a>EventValueCommandEventValueCommandChecked。

```
namespace Score.ViewModel

open Score.Model
open FSharp.ViewModule

type MainViewModel(controller : Score -> ScoringEvent -> Score) as self =
    inherit EventViewModelBase<ScoringEvent>()

    let score = self.Factory.Backing(<@ self.Score @>, Score.zero)

    let eventHandler ev = score.Value <- controller score.Value ev

    do
        self.EventStream
        |> Observable.add eventHandler

    member this.IncA = this.Factory.EventValueCommand(IncA)
    member this.DecA = this.Factory.EventValueCommandChecked(DecA, (fun _ -> this.Score.ScoreA
> 0), [ <@@ this.Score @@> ])
    member this.IncB = this.Factory.EventValueCommand(IncB)
    member this.DecB = this.Factory.EventValueCommandChecked(DecB, (fun _ -> this.Score.ScoreB
> 0), [ <@@ this.Score @@> ])
    member this.NewGame = this.Factory.EventValueCommand(NewGame)

    member __.Score = score.Value
```

* .xaml.fs `controller` `MainViewModel`。

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

type CompositionRoot() =
    member __.ViewModel = Score.ViewModel.MainViewModel(Score.Model.Score.update)
```

`CompositionRoot`XAML。

```
<Window.Resources>
    <ResourceDictionary>
        <local:CompositionRoot x:Key="CompositionRoot"/>
    </ResourceDictionary>
</Window.Resources>
<Window.DataContext>
    <Binding Source="{StaticResource CompositionRoot}" Path="ViewModel" />
</Window.DataContext>
```

XAMLWPFGitHub 。

Gjallarhorn`IObservable<'a>` FSharp.ViewModule`EventStream`。 。 *MessageEvent* 。

```
namespace ScoreLogic.Model
```

```
type Score = { ScoreA: int ; ScoreB: int }
type ScoreMessage = IncA | DecA | IncB | DecB | NewGame

// Module showing allowed operations
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update msg score =
        match msg with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

GjallarhornUI`Component。 sourceBindingSource`Gjallarhorn.Bindable。

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        // Bind the score to the view
        model |> Binding.toView source "Score"

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessage "DecA" DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessage "DecB" DecB
        ]
```

FSharp.ViewModuleCanExecute。。

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        let aScored = Mutable.create false
        let bScored = Mutable.create false

        // Bind the score itself to the view
        model |> Binding.toView source "Score"

        // Subscribe to changes of the score
        model |> Signal.Subscription.create
            (fun currentValue ->
```

```
                aScored.Value <- currentValue.ScoreA > 0
                bScored.Value <- currentValue.ScoreB > 0)
            |> ignore

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessageChecked "DecA" aScored DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessageChecked "DecB" bScored DecB
        ]

    let application =
        // Create our score, wrapped in a mutable with an atomic update function
        let score = new AsyncMutable<_>(Score.zero)

        // Create our 3 functions for the application framework

        // Start with the function to create our model (as an ISignal<'a>)
        let createModel () : ISignal<_> = score :> _

        // Create a function that updates our state given a message
        // Note that we're just taking the message, passing it directly to our model's update
function,
        // then using that to update our core "Mutable" type.
        let update (msg : ScoreMessage) : unit = Score.update msg |> score.Update |> ignore

        // An init function that occurs once everything's created, but before it starts
        let init () : unit = ()

        // Start our application
        Framework.application createModel init update scoreComponent
```

MainWindow。

```
namespace ScoreBoard.Views

open System

open FsXaml
open ScoreLogic.Model

// Create our Window
type MainWindow = XAML<"MainWindow.xaml">

module Main =
    [<STAThread>]
    [<EntryPoint>]
    let main _ =
        // Run using the WPF wrappers around the basic application framework
        Gjallarhorn.Wpf.Framework.runApplication System.Windows.Application MainWindow
Program.application
```

Reed Copsey 。

- 。
- 。
-

-

FWPF https://riptutorial.com/zh-TW/fsharp/topic/8758/f-wpf

6: F

Examples

F

F# ◦

◦ ◦

```
type Sex =
  | Male
  | Female

type Customer =
  {
    Name    : string
    Born    : System.DateTime
    Sex     : Sex
  }
```

◦

```
//  If any row in this list matches the Customer, the customer isn't eligible for the car
insurance.
let exclusionList =
  let __          _   = true
  let olderThan   x y = x < y
  let youngerThan x y = x > y
  [|
//  Description                          Age               Sex
    "Not allowed for senior citizens" , olderThan   65  , __
    "Not allowed for children"        , youngerThan 16  , __
    "Not allowed for young males"     , youngerThan 25  , (=) Male
  |]
```

◦

◦

```
// Splits customers into two buckets: potential customers and denied customers.
// The denied customer bucket also includes the reason for denying them the car insurance
let splitCustomers (today : System.DateTime) (cs : Customer []) : Customer
[]*(string*Customer) [] =
  let potential = ResizeArray<_> 16 // ResizeArray is an alias for
System.Collections.Generic.List
  let denied    = ResizeArray<_> 16

  for c in cs do
    let age = today.Year - c.Born.Year
    let sex = c.Sex
    match exclusionList |> Array.tryFind (fun (_, testAge, testSex) -> testAge age && testSex
```

```
sex) with
    | Some (description, _, _)  -> denied.Add (description, c)
    | None                     -> potential.Add c

  potential.ToArray (), denied.ToArray ()
```

```
let customers =
  let c n s y m d: Customer = { Name = n; Born = System.DateTime (y, m, d); Sex = s }
  [|
//    Name                      Sex     Born
    c "Clint Eastwood Jr."      Male    1930 05 31
    c "Bill Gates"              Male    1955 10 28
    c "Melina Gates"            Female  1964 08 15
    c "Justin Drew Bieber"      Male    1994 03 01
    c "Sophie Turner"           Female  1996 02 21
    c "Isaac Hempstead Wright"  Male    1999 04 09
  |]

[<EntryPoint>]
let main argv =
  let potential, denied = splitCustomers (System.DateTime (2014, 06, 01)) customers
  printfn "Potential Customers (%d)\n%A" potential.Length potential
  printfn "Denied Customers (%d)\n%A"    denied.Length    denied
  0
```

```
Potential Customers (3)
[|{Name = "Bill Gates";
   Born = 1955-10-28 00:00:00;
   Sex = Male;}; {Name = "Melina Gates";
                  Born = 1964-08-15 00:00:00;
                  Sex = Female;}; {Name = "Sophie Turner";
                                   Born = 1996-02-21 00:00:00;
                                   Sex = Female;}|]
Denied Customers (3)
[|("Not allowed for senior citizens", {Name = "Clint Eastwood Jr.";
                                       Born = 1930-05-31 00:00:00;
                                       Sex = Male;});
  ("Not allowed for young males", {Name = "Justin Drew Bieber";
                                   Born = 1994-03-01 00:00:00;
                                   Sex = Male;});
  ("Not allowed for children", {Name = "Isaac Hempstead Wright";
                                Born = 1999-04-09 00:00:00;
                                Sex = Male;})|]
```

F https://riptutorial.com/zh-TW/fsharp/topic/3925/f-

7: F

Examples

for-loop F#break continuereturn。 F#。

ListtryFind。 F# returntryFind

```
let tryFind predicate vs =
  for v in vs do
    if predicate v then
      return Some v
  None
```

F#。 tail-recursion

```
let tryFind predicate vs =
  let rec loop = function
    | v::vs -> if predicate v then
                   Some v
               else
                   loop vs
    | _ -> None
  loop vs
```

F#F#while-loop。 ILSpyloop

```
internal static FSharpOption<a> loop@3-10<a>(FSharpFunc<a, bool> predicate, FSharpList<a>
_arg1)
{
  while (_arg1.TailOrNull != null)
  {
    FSharpList<a> fSharpList = _arg1;
    FSharpList<a> vs = fSharpList.TailOrNull;
    a v = fSharpList.HeadOrDefault;
    if (predicate.Invoke(v))
    {
      return FSharpOption<a>.Some(v);
    }
    FSharpFunc<a, bool> arg_2D_0 = predicate;
    _arg1 = vs;
    predicate = arg_2D_0;
  }
  return null;
}
```

JIT-er。

F#。 sumList。

```
let sum vs =
  let mutable s = LanguagePrimitives.GenericZero
```

```
  for v in vs do
    s <- s + v
  s
```

```
let sum vs =
  let rec loop s = function
    | v::vs -> loop (s + v) vs
    | _ -> s
  loop LanguagePrimitives.GenericZero vs
```

F#while-loop ◦

F#

;◦

JIT◦ ◦

ILSpy◦ JITer◦ IL-code◦

;◦ ◦

F

F#1..n◦ ◦

```
// now () returns current time in milliseconds since start
let now : unit -> int64 =
  let sw = System.Diagnostics.Stopwatch ()
  sw.Start ()
  fun () -> sw.ElapsedMilliseconds

// time estimates the time 'action' repeated a number of times
let time repeat action : int64*'T =
  let v = action ()  // Warm-up and compute value

  let b = now ()
  for i = 1 to repeat do
    action () |> ignore
  let e = now ()

  e - b, v
```

time◦

1..n◦

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum

// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
```

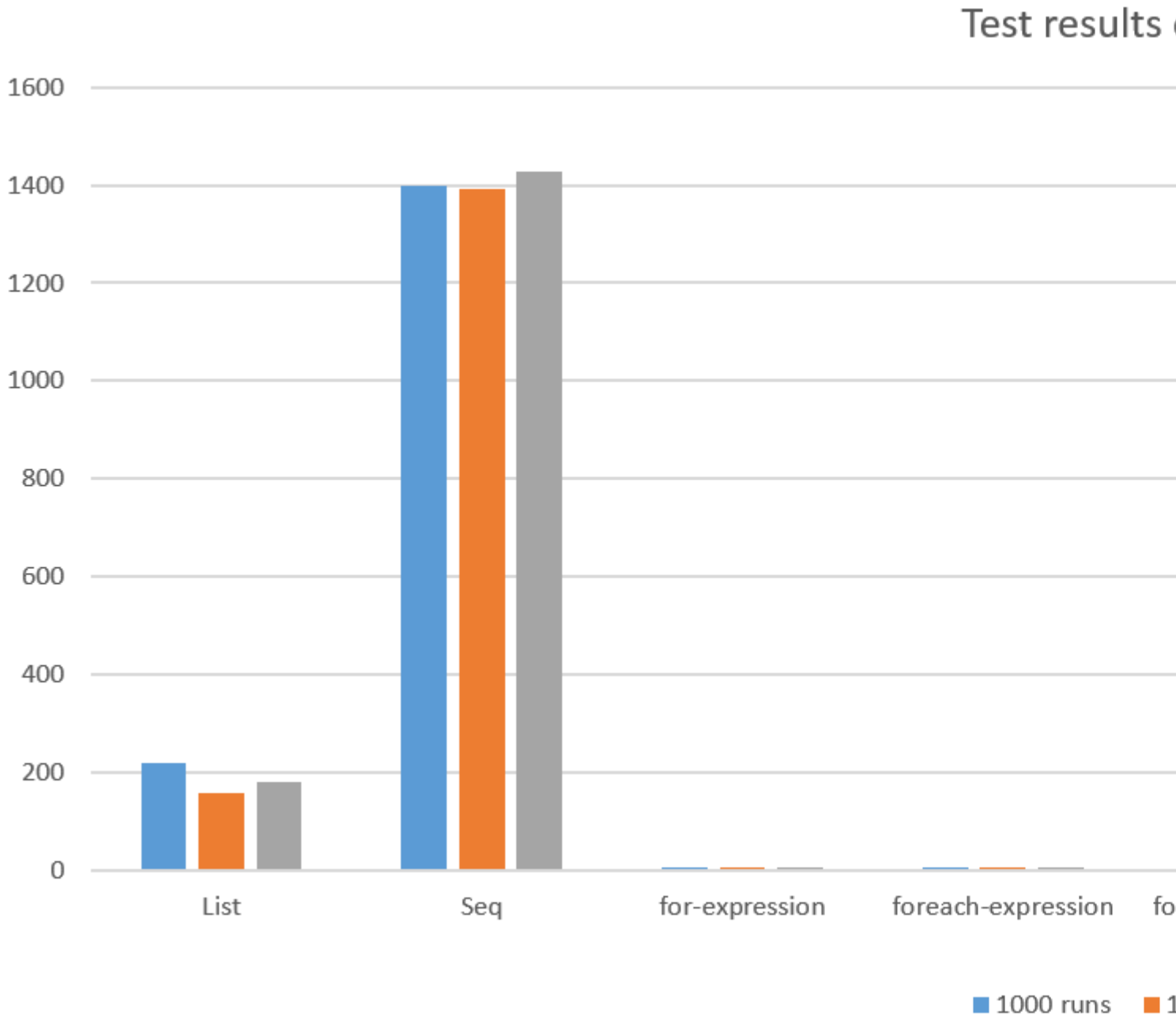
```
  Seq.init (n + 1) id
  |> Seq.sum

// Accumulates all integers 1..n using 'for-expression'
let accumulateUsingFor n =
  let mutable sum = 0
  for i = 1 to n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum

// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int

// Accumulates all integers n..1 using 'for-expression' in reverse order
let accumulateUsingReverseFor n =
  let mutable sum = 0
  for i = n downto 1 do
    sum <- sum + i
  sum

// Accumulates all 64 integers n..1 using 'tail-recursion' in reverse order
let accumulateUsingReverseTailRecursion n =
  let rec loop sum i =
    if i > 0 then
      loop (sum + i) (i - 1)
    else
      sum
  loop 0 n
```

2。

```
let testRun (path : string) =
  use testResult = new System.IO.StreamWriter (path)
  let write   (l : string)  = testResult.WriteLine l
  let writef  fmt           = FSharp.Core.Printf.kprintf write fmt
```

```
  write "Name\tTotal\tOuter\tInner\tCC0\tCC1\tCC2\tTime\tResult"

  // total is the total number of iterations being executed
  let total   = 10000000
  // outers let us variate the relation between the inner and outer loop
  //  this is often useful when the algorithm allocates different amount of memory
  //  depending on the input size. This can affect cache locality
  let outers  = [| 1000; 10000; 100000 |]
  for outer in outers do
    let inner = total / outer

    // multiplier is used to increase resolution of certain tests that are significantly
    //  faster than the slower ones

    let testCases =
      [|
    //   Name of test                          multiplier    action
        "List"                               , 1           , accumulateUsingList
        "Seq"                                , 1           , accumulateUsingSeq
        "for-expression"                     , 100         , accumulateUsingFor
        "foreach-expression"                 , 100         , accumulateUsingForEach
        "foreach-expression over List"       , 1           , accumulateUsingForEachOverList
        "foreach-expression increment of 2" , 1           , accumulateUsingForEachStep2
        "foreach-expression over 64 bit"     , 1           , accumulateUsingForEach64
        "reverse for-expression"             , 100         , accumulateUsingReverseFor
        "reverse tail-recursion"             , 100         ,
accumulateUsingReverseTailRecursion
      |]
    for name, multiplier, a in testCases do
      System.GC.Collect (2, System.GCCollectionMode.Forced, true)
      let cc g = System.GC.CollectionCount g

      printfn "Accumulate using %s with outer=%d and inner=%d ..." name outer inner

      // Collect collection counters before test run
      let pcc0, pcc1, pcc2 = cc 0, cc 1, cc 2

      let ms, result       = time (outer*multiplier) (fun () -> a inner)
      let ms               = (float ms / float multiplier)

      // Collect collection counters after test run
      let acc0, acc1, acc2 = cc 0, cc 1, cc 2
      let cc0, cc1, cc2    = acc0 - pcc0, acc1 - pcc1, acc1 - pcc1
      printfn "  ... took: %f ms, GC collection count %d,%d,%d and produced %A" ms cc0 cc1 cc2
result

      writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%f\t%d" name total outer inner cc0 cc1 cc2 ms result
```

.NET 4.5.2 x64



。

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum
```

`1..n`。 for42。

GC100。CPU。

SEQ

```
// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
  Seq.init (n + 1) id
```

```
  |> Seq.sum
```

SeqListfor270x。 GC661x。

Seq。

Seq 。 。

manofstick Seq.init。 { 0 .. n }Seq.init (n+1) id。 PR。 Seq.init ... |> Seq.sum Seq.init ... |> Seq.map id |> Seq.sum。 Seq.initCurrent Lazy - PR..... 。

foreach-expression over List

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
    sum <- sum + i
  sum
```

76。

```
public static int accumulateUsingForEach(int n)
{
  int sum = 0;
  int i = 1;
  if (n >= i)
  {
    do
    {
      sum += i;
      i++;
    }
    while (i != n + 1);
  }
  return sum;
}

public static int accumulateUsingForEachOverList(int n)
{
  int sum = 0;
  FSharpList<int> fSharpList =
SeqModule.ToList<int>(Operators.CreateSequence<int>(Operators.OperatorIntrinsics.RangeInt32(1,
1, n)));
  for (FSharpList<int> tailOrNull = fSharpList.TailOrNull; tailOrNull != null; tailOrNull =
fSharpList.TailOrNull)
  {
    int i = fSharpList.HeadOrDefault;
    sum += i;
```

```
    fSharpList = tailOrNull;
  }
  return sum;
}
```

accumulateUsingForEachwhilefor i in [1..n]

```
FSharpList<int> fSharpList =
  SeqModule.ToList<int>(
    Operators.CreateSequence<int>(
      Operators.OperatorIntrinsics.RangeInt32(1, 1, n)));
```

1..nSeq ToList ◦

◦

foreach-expression2

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum
```

~25

ILSpy

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  IEnumerable<int> enumerable = Operators.OperatorIntrinsics.RangeInt32(1, 2, n);
  foreach (int i in enumerable)
  {
    sum += i;
  }
  return sum;
}
```

1..2..nSeq Seq ◦

F#

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  for (int i = 1; i < n; i += 2)
```

```
  {
    sum += i;
  }
  return sum;
}
```

F#1int32for。 Operators.OperatorIntrinsics.RangeInt32 。

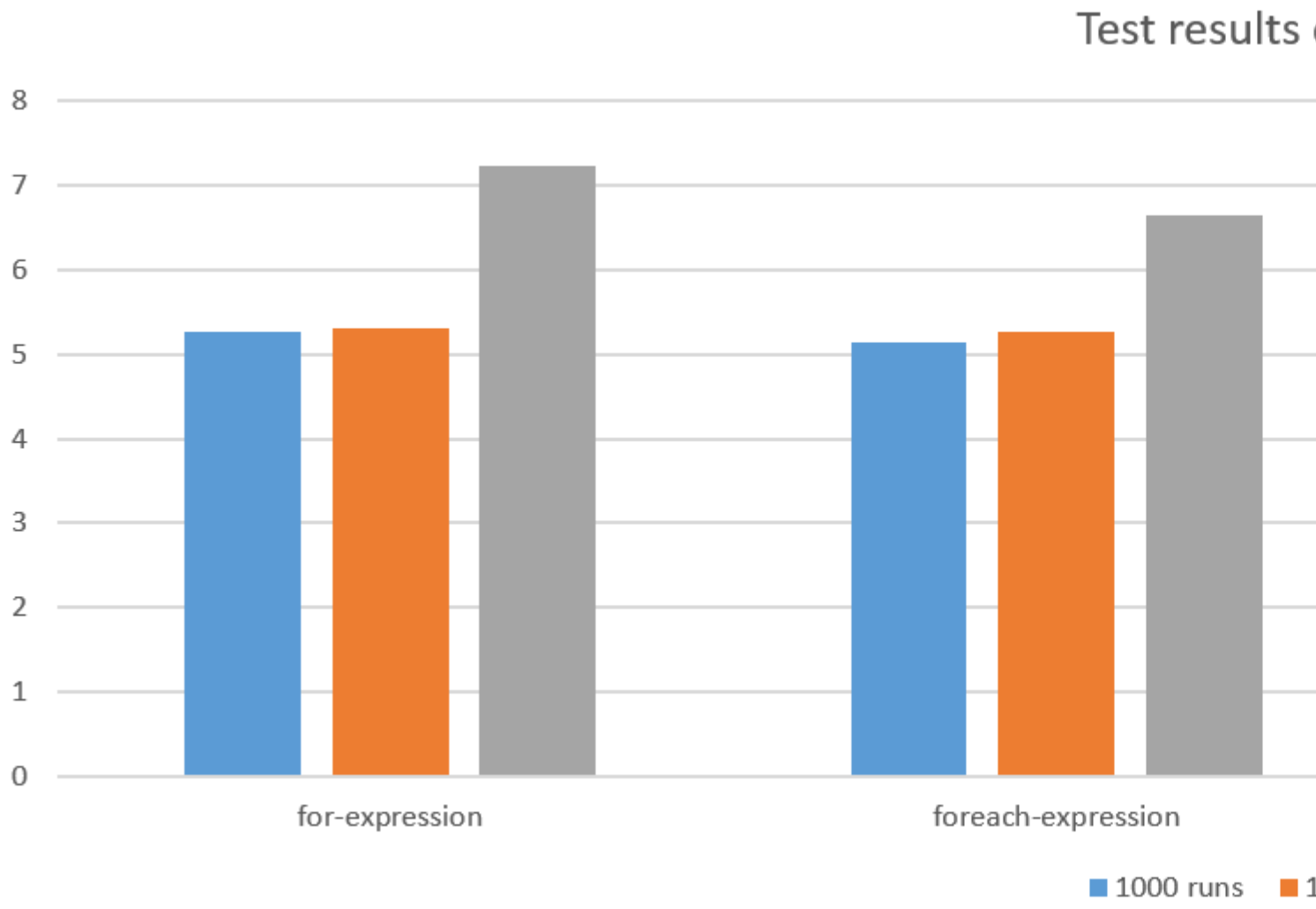
foreach-expression64

```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

for4764。 ILSpy

```
public static int accumulateUsingForEach64(int n)
{
  long sum = 0L;
  IEnumerable<long> enumerable = Operators.OperatorIntrinsics.RangeInt64(1L, 1L, (long)n);
  foreach (long i in enumerable)
  {
    sum += i;
  }
  return (int)sum;
}
```

F#int32forOperators.OperatorIntrinsics.RangeInt64 。



actionaction ◦

0CPUCPU◦

270◦

◦ ◦

◦

F

F# List SeqArray ◦

◦

cpu

```
let seqTest n =
  Seq.init (n + 1) id
  |> Seq.map    int64
  |> Seq.filter (fun v -> v % 2L = 0L)
```

```
   |> Seq.map    ((+) 1L)
   |> Seq.sum
```

。

n。

1.
2.
3.
4. LINQ
5. Seq
6. Nessos/
7. PullStream
8. PushStream

Imperative CPU。 。

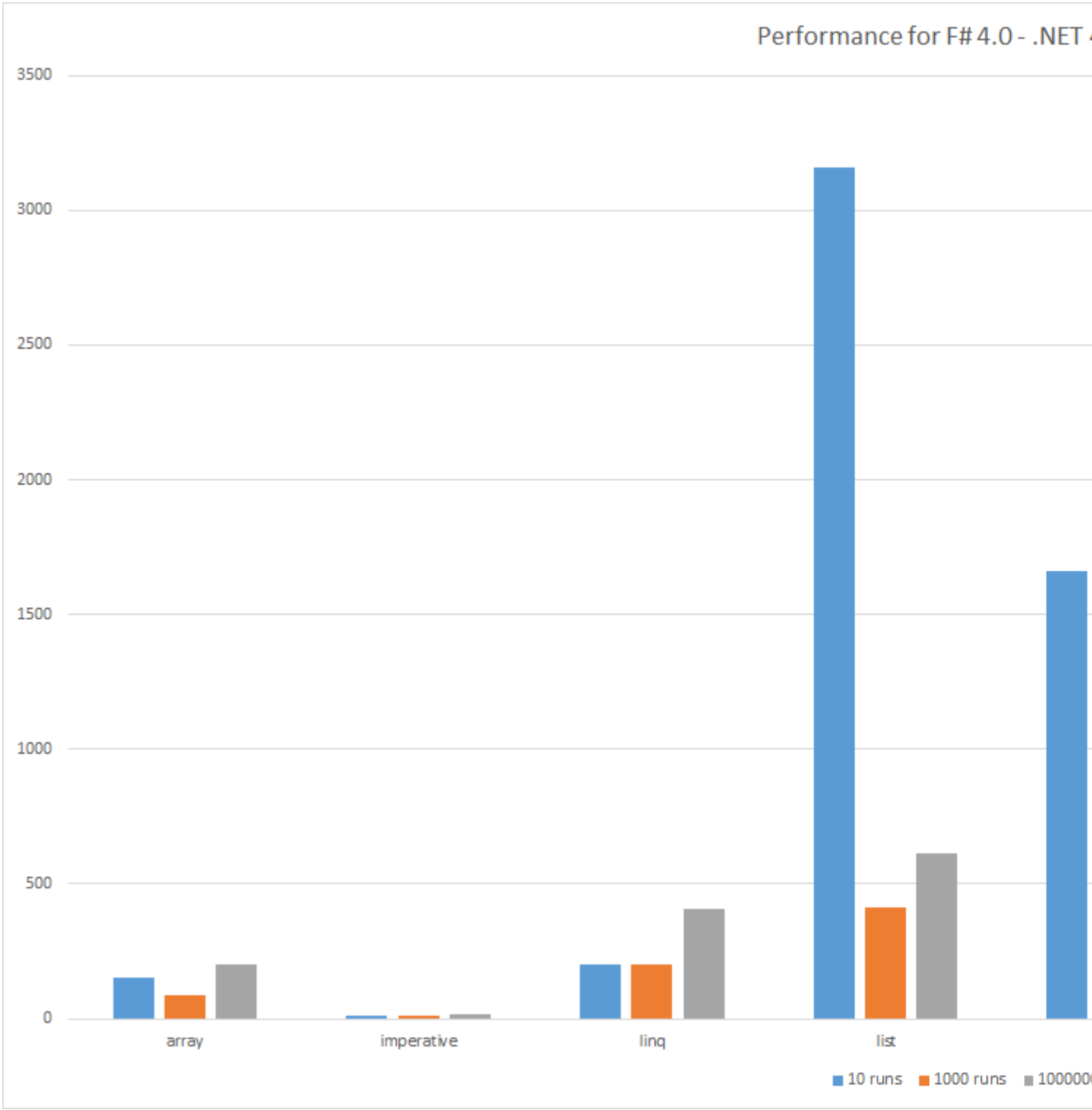
ArrayListArray / List 。

LINQSeqIEnumerable<'T> pull。 。

NessosJava Stream 。

PullStreamPushStreamPull Push。

F4.0 - .NET 4.6.1 - x64



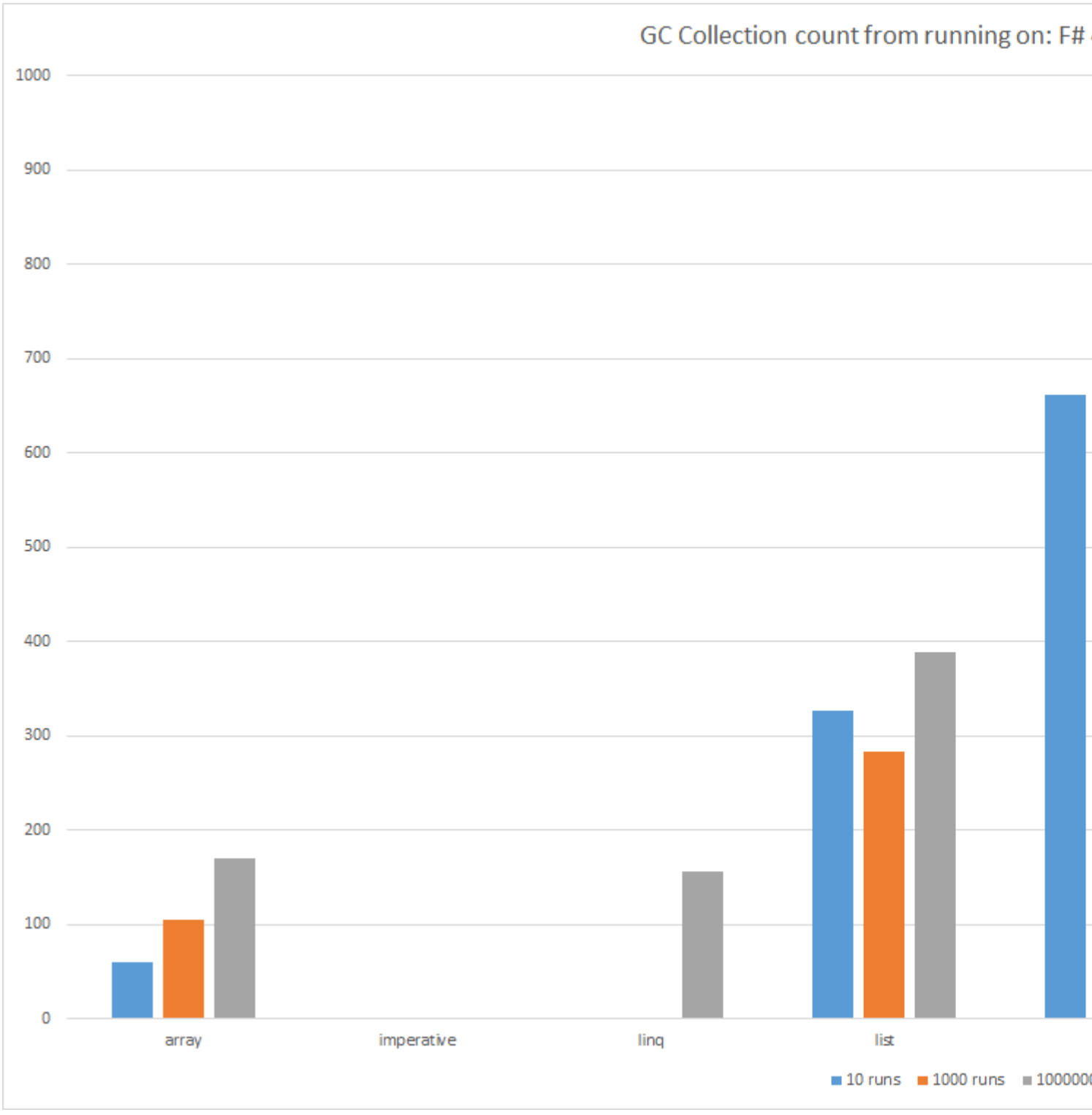
。 。 。

。

1. List。 GC。
2. Array。
3. LINQSeq IEnumerable<'T> 。 SeqLINQ。
4. PushPull 。
5. PushNessos。 Nessos。
6. Nessos。
7.

Imperative

GCF4.0 - .NET 4.6.1 - x64



GC◦ ◦

◦

1. ListArrayList◦ ◦
2. List Array2◦ ◦
3. Seq◦ List◦

4. LINQ Nessos PushPull◦ GC◦
5. ImperativeGC◦

◦

◦ Array ◦

◦ “” Seq◦

◦ GCGC◦

F#Nessos Streams ◦

Imperative◦

◦ ◦

```
module PushStream =
  type Receiver<'T> = 'T -> bool
  type Stream<'T>   = Receiver<'T> -> unit

  let inline filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
    fun r -> s (fun v -> if f v then r v else true)

  let inline map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
    fun r -> s (fun v -> r (m v))

  let inline range b e : Stream<int> =
    fun r ->
      let rec loop i = if i <= e && r i then loop (i + 1)
      loop b

  let inline sum (s : Stream<'T>) : 'T =
    let mutable state = LanguagePrimitives.GenericZero<'T>
    s (fun v -> state <- state + v; true)
    state

module PullStream =

  [<Struct>]
  [<NoComparison>]
  [<NoEqualityAttribute>]
  type Maybe<'T>(v : 'T, hasValue : bool) =
    member    x.Value        = v
    member    x.HasValue     = hasValue
    override  x.ToString ()  =
      if hasValue then
        sprintf "Just %A" v
      else
        "Nothing"

  let Nothing<'T>     = Maybe<'T> (Unchecked.defaultof<'T>, false)
  let inline Just v   = Maybe<'T> (v, true)

  type Iterator<'T> = unit -> Maybe<'T>
  type Stream<'T>   = unit -> Iterator<'T>
```

```
  let filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
    fun () ->
      let i = s ()
      let rec pop () =
        let mv = i ()
        if mv.HasValue then
          let v = mv.Value
          if f v then Just v else pop ()
        else
          Nothing
      pop

  let map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
    fun () ->
      let i = s ()
      let pop () =
        let mv = i ()
        if mv.HasValue then
          Just (m mv.Value)
        else
          Nothing
      pop

  let range b e : Stream<int> =
    fun () ->
      let mutable i = b
      fun () ->
        if i <= e then
          let p = i
          i <- i + 1
          Just p
        else
          Nothing

  let inline sum (s : Stream<'T>) : 'T =
    let i = s ()
    let rec loop state =
      let mv = i ()
      if mv.HasValue then
        loop (state + mv.Value)
      else
        state
    loop LanguagePrimitives.GenericZero<'T>

module PerfTest =

  open System.Linq
#if USE_NESSOS
  open Nessos.Streams
#endif

  let now =
    let sw = System.Diagnostics.Stopwatch ()
    sw.Start ()
    fun () -> sw.ElapsedMilliseconds

  let time n a =
    let inline cc i       = System.GC.CollectionCount i

    let v                 = a ()
```

```
    System.GC.Collect (2, System.GCCollectionMode.Forced, true)

    let bcc0, bcc1, bcc2  = cc 0, cc 1, cc 2
    let b                 = now ()

    for i in 1..n do
      a () |> ignore

    let e = now ()
    let ecc0, ecc1, ecc2  = cc 0, cc 1, cc 2

    v, (e - b), ecc0 - bcc0, ecc1 - bcc1, ecc2 - bcc2

  let arrayTest n =
    Array.init (n + 1) id
    |> Array.map    int64
    |> Array.filter (fun v -> v % 2L = 0L)
    |> Array.map    ((+) 1L)
    |> Array.sum

  let imperativeTest n =
    let rec loop s i =
      if i >= 0L then
        if i % 2L = 0L then
          loop (s + i + 1L) (i - 1L)
        else
          loop s (i - 1L)
      else
        s
    loop 0L (int64 n)

  let linqTest n =
    (((Enumerable.Range(0, n + 1)).Select int64).Where(fun v -> v % 2L = 0L)).Select((+)
1L).Sum()

  let listTest n =
    List.init (n + 1) id
    |> List.map     int64
    |> List.filter  (fun v -> v % 2L = 0L)
    |> List.map     ((+) 1L)
    |> List.sum

#if USE_NESSOS
  let nessosTest n =
    Stream.initInfinite id
    |> Stream.take    (n + 1)
    |> Stream.map     int64
    |> Stream.filter  (fun v -> v % 2L = 0L)
    |> Stream.map     ((+) 1L)
    |> Stream.sum
#endif

  let pullTest n =
    PullStream.range 0 n
    |> PullStream.map     int64
    |> PullStream.filter  (fun v -> v % 2L = 0L)
    |> PullStream.map     ((+) 1L)
    |> PullStream.sum

  let pushTest n =
    PushStream.range 0 n
```

```
    |> PushStream.map     int64
    |> PushStream.filter  (fun v -> v % 2L = 0L)
    |> PushStream.map     ((+) 1L)
    |> PushStream.sum

  let seqTest n =
    Seq.init (n + 1) id
    |> Seq.map    int64
    |> Seq.filter (fun v -> v % 2L = 0L)
    |> Seq.map    ((+) 1L)
    |> Seq.sum

  let perfTest (path : string) =
    let testCases =
      [|
        "array"       , arrayTest
        "imperative"  , imperativeTest
        "linq"        , linqTest
        "list"        , listTest
        "seq"         , seqTest
#if USE_NESSOS
        "nessos"      , nessosTest
#endif
        "pull"        , pullTest
        "push"        , pushTest
      |]
    use out                   = new System.IO.StreamWriter (path)
    let write (msg : string)  = out.WriteLine msg
    let writef fmt            = FSharp.Core.Printf.kprintf write fmt

    write "Name\tTotal\tOuter\tInner\tElapsed\tCC0\tCC1\tCC2\tResult"

    let total   = 10000000
    let outers = [| 10; 1000; 1000000 |]
    for outer in outers do
      let inner = total / outer
      for name, a in testCases do
        printfn "Running %s with total=%d, outer=%d, inner=%d ..." name total outer inner
        let v, ms, cc0, cc1, cc2 = time outer (fun () -> a inner)
        printfn "  ... %d ms, cc0=%d, cc1=%d, cc2=%d, result=%A" ms cc0 cc1 cc2 v
        writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d" name total outer inner ms cc0 cc1 cc2 v

[<EntryPoint>]
let main argv =
  System.Environment.CurrentDirectory <- System.AppDomain.CurrentDomain.BaseDirectory
  PerfTest.perfTest "perf.tsv"
  0
```

F https://riptutorial.com/zh-TW/fsharp/topic/3562/f-

8: FWPFFsXaml

Windows Presentation FoundationWPFF。FWPF。WPF。。

Examples

Visual Studio 2015VS 2015。VS。|Windows。

NuGetFsXaml.Wpf;Reed CopseyJr。FWPF。WPF。 FsXaml。

UIAutomationTypes;.NET。

“”

。Program.fs。Window Canvas。Spirograph.fs。

```
namespace Spirograph

// open System.Windows does not automatically open all its sub-modules, so we
// have to open them explicitly as below, to get the resources noted for each.
open System                             // for Math.PI
open System.Windows                     // for Point
open System.Windows.Controls            // for Canvas
open System.Windows.Shapes              // for Ellipse
open System.Windows.Media               // for Brushes

// ----------------------------------------------------------------------------
// This file is first in the build sequence, so types should be defined here
type DialogBoxXaml  = FsXaml.XAML<"DialogBox.xaml">
type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">
type App            = FsXaml.XAML<"App.xaml">

// ----------------------------------------------------------------------------
// Model: This draws the Spirograph
type MColor = | MBlue   | MRed | MRandom

type Model() =
  let mutable myCanvas: Canvas = null
  let mutable myR              = 220    // outer circle radius
  let mutable myr              = 65     // inner circle radius
  let mutable myl              = 0.8    // pen position relative to inner circle
  let mutable myColor          = MBlue  // pen color

  let rng                      = new Random()
  let mutable myRandomColor    = Color.FromRgb(rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte)

  member this.MyCanvas
    with get() = myCanvas
    and  set(newCanvas) = myCanvas <- newCanvas

  member this.MyR
    with get() = myR
```

```
  and  set(newR) = myR <- newR

member this.Myr
  with get() = myr
  and  set(newr) = myr <- newr

member this.Myl
  with get() = myl
  and  set(newl) = myl <- newl

member this.MyColor
  with get() = myColor
  and  set(newColor) = myColor <- newColor

member this.Randomize =
  // Here we randomize the parameters. You can play with the possible ranges of
  // the parameters to find randomized spirographs that are pleasing to you.
  this.MyR      <- rng.Next(100, 500)
  this.Myr      <- rng.Next(this.MyR / 10, (9 * this.MyR) / 10)
  this.Myl      <- 0.1 + 0.8 * rng.NextDouble()
  this.MyColor  <- MRandom
  myRandomColor <- Color.FromRgb(rng.Next(0, 255) |> byte,
                                 rng.Next(0, 255) |> byte,
                                 rng.Next(0, 255) |> byte)

member this.DrawSpirograph =
  // Draw a spirograph. Note there is some fussing with ints and floats; this
  // is required because the outer and inner circle radii are integers. This is
  // necessary in order for the spirograph to return to its starting point
  // after a certain number of revolutions of the outer circle.

  // Start with usual recursive gcd function and determine the gcd of the inner
  // and outer circle radii. Everything here should be in integers.
  let rec gcd x y =
      if y = 0 then x
      else gcd y (x % y)

  let g = gcd this.MyR this.Myr             // find greatest common divisor
  let maxRev = this.Myr / g                 // maximum revs to repeat

  // Determine width and height of window, location of center point, scaling
  // factor so that spirograph fits within the window, ratio of inner and outer
  // radii.

  // Everything from this point down should be float.
  let width, height = myCanvas.ActualWidth, myCanvas.ActualHeight
  let cx, cy = width / 2.0, height / 2.0    // coordinates of center point
  let maxR   = min cx cy                    // maximum radius of outer circle
  let scale  = maxR / float(this.MyR)             // scaling factor
  let rRatio = float(this.Myr) / float(this.MyR)  // ratio of the radii

  // Build the collection of spirograph points, scaled to the window.
  let points = new PointCollection()
  for degrees in [0 .. 5 .. 360 * maxRev] do
    let angle = float(degrees) * Math.PI / 180.0
    let x, y = cx + scale * float(this.MyR) *
               ((1.0-rRatio)*Math.Cos(angle) +
                this.Myl*rRatio*Math.Cos((1.0-rRatio)*angle/rRatio)),
               cy + scale * float(this.MyR) *
               ((1.0-rRatio)*Math.Sin(angle) -
                this.Myl*rRatio*Math.Sin((1.0-rRatio)*angle/rRatio))
```

```
      points.Add(new Point(x, y))

    // Create the Polyline with the above PointCollection, erase the Canvas, and
    // add the Polyline to the Canvas Children
    let brush = match this.MyColor with
                | MBlue   -> Brushes.Blue
                | MRed    -> Brushes.Red
                | MRandom -> new SolidColorBrush(myRandomColor)

    let mySpirograph = new Polyline()
    mySpirograph.Points <- points
    mySpirograph.Stroke <- brush

    myCanvas.Children.Clear()
    this.MyCanvas.Children.Add(mySpirograph) |> ignore
```

Spirograph.fsF。Canvas。。

XAML

XAML。MainWindow.xamlXAML

```
<!-- This defines the main window, with a menu and a canvas. Note that the Height
     and Width are overridden in code to be 2/3 the dimensions of the screen -->
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Spirograph" Height="200" Width="300">
    <!-- Define a grid with 3 rows: Title bar, menu bar, and canvas. By default
         there is only one column -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <!-- Define the menu entries -->
        <Menu  Grid.Row="0">
            <MenuItem Header="File">
                <MenuItem Header="Exit"
                          Name="menuExit"/>
            </MenuItem>
            <MenuItem Header="Spirograph">
                <MenuItem Header="Parameters..."
                          Name="menuParameters"/>
                <MenuItem Header="Draw"
                          Name="menuDraw"/>
            </MenuItem>
            <MenuItem Header="Help">
                <MenuItem Header="About"
                          Name="menuAbout"/>
            </MenuItem>
        </Menu>
        <!-- This is a canvas for drawing on. If you don't specify the coordinates
             for Left and Top you will get NaN for those values -->
        <Canvas Grid.Row="1" Name="myCanvas" Left="0" Top="0">
        </Canvas>
    </Grid>
```

```
</Window>
```

XAML。XAML。。XAML。

XAMLF

XAML。。 - - WPF/。

```
<!-- This first part is boilerplate, except for the title, height and width.
     Note that some fussing with alignment and margins may be required to get
     the box looking the way you want it. -->
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Parameters" Height="200" Width="250">
    <!-- Here we define a layout of 3 rows and 2 columns below the title bar -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <!-- Define a label and a text box for the first three rows. Top row is
             the integer radius of the outer circle -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="0"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="R: Outer"
                   Height="24" Width='65'/>
            <TextBox x:Name="radiusR"  Margin="0,0,0,0.5" Width="120"
                     VerticalAlignment="Bottom" Height="20">Integer</TextBox>
        </StackPanel>
        <!-- This defines a label and text box for the integer radius of the
             inner circle -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="1"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="r: Inner"
                   Height="24" Width='65'/>
            <TextBox x:Name="radiusr"  Margin="0,0,0,0.5" Width="120"
                     VerticalAlignment="Bottom" Height="20" Text="Integer"/>
        </StackPanel>
        <!-- This defines a label and text box for the float ratio of the inner
             circle radius at which the pen is positioned -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="2"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="l: Ratio"
                   Height="24" Width='65'/>
            <TextBox x:Name="ratiol"  Margin="0,0,0,1" Width="120"
                     VerticalAlignment="Bottom" Height="20" Text="Float"/>
        </StackPanel>
        <!-- This defines a radio button group to select color -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="3"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,4,5.333" Content="Color"
```

```
                  Height="24"/>
            <RadioButton x:Name="buttonBlue" Content="Blue" GroupName="Color"
                         HorizontalAlignment="Left"  VerticalAlignment="Top"
                         Click="buttonBlueClick"
                         Margin="5,13,11,3.5" Height="17"/>
            <RadioButton x:Name="buttonRed"  Content="Red"  GroupName="Color"
                         HorizontalAlignment="Left" VerticalAlignment="Top"
                         Click="buttonRedClick"
                         Margin="5,13,5,3.5" Height="17" />
            <RadioButton x:Name="buttonRandom"  Content="Random"
                         GroupName="Color" Click="buttonRandomClick"
                         HorizontalAlignment="Left" VerticalAlignment="Top"
                         Margin="5,13,5,3.5" Height="17" />
        </StackPanel>
        <!-- These are the standard OK/Cancel buttons -->
        <Button Grid.Row="4" Grid.Column="0" Name="okButton"
                Click="okButton_Click" IsDefault="True">OK</Button>
        <Button Grid.Row="4" Grid.Column="1" Name="cancelButton"
                IsCancel="True">Cancel</Button>
    </Grid>
</Window>
```

Dialog.Box。XXX.xaml.fsXAMLXXX.xaml。

```
namespace Spirograph

open System.Windows.Controls

type DialogBox(app: App, model: Model, win: MainWindowXaml) as this =
  inherit DialogBoxXaml()

  let myApp   = app
  let myModel = model
  let myWin   = win

  // These are the default parameters for the spirograph, changed by this dialog
  // box
  let mutable myR = 220                // outer circle radius
  let mutable myr = 65                 // inner circle radius
  let mutable myl = 0.8                // pen position relative to inner circle
  let mutable myColor = MBlue          // pen color

  // These are the dialog box controls. They are initialized when the dialog box
  // is loaded in the whenLoaded function below.
  let mutable RBox: TextBox = null
  let mutable rBox: TextBox = null
  let mutable lBox: TextBox = null

  let mutable blueButton: RadioButton   = null
  let mutable redButton: RadioButton    = null
  let mutable randomButton: RadioButton = null

  // Call this functions to enable or disable parameter input depending on the
  // state of the randomButton. This is a () -> () function to keep it from
  // being executed before we have loaded the dialog box below and found the
  // values of TextBoxes and RadioButtons.
  let enableParameterFields(b: bool) =
    RBox.IsEnabled <- b
    rBox.IsEnabled <- b
    lBox.IsEnabled <- b
```

```
let whenLoaded _ =
  // Load and initialize text boxes and radio buttons to the current values in
  // the model. These are changed only if the OK button is clicked, which is
  // handled below. Also, if the color is Random, we disable the parameter
  // fields.
  RBox <- this.FindName("radiusR") :?> TextBox
  rBox <- this.FindName("radiusr") :?> TextBox
  lBox <- this.FindName("ratiol")  :?> TextBox

  blueButton   <- this.FindName("buttonBlue")   :?> RadioButton
  redButton    <- this.FindName("buttonRed")    :?> RadioButton
  randomButton <- this.FindName("buttonRandom") :?> RadioButton

  RBox.Text <- myModel.MyR.ToString()
  rBox.Text <- myModel.Myr.ToString()
  lBox.Text <- myModel.Myl.ToString()

  myR <- myModel.MyR
  myr <- myModel.Myr
  myl <- myModel.Myl

  blueButton.IsChecked   <- new System.Nullable<bool>(myModel.MyColor = MBlue)
  redButton.IsChecked    <- new System.Nullable<bool>(myModel.MyColor = MRed)
  randomButton.IsChecked <- new System.Nullable<bool>(myModel.MyColor = MRandom)

  myColor <- myModel.MyColor
  enableParameterFields(not (myColor = MRandom))

let whenClosing _ =
  // Show the actual spirograph parameters in a message box at close. Note the
  // \n in the sprintf gives us a linebreak in the MessageBox. This is mainly
  // for debugging, and it can be deleted.
  let s = sprintf "R = %A\nr = %A\nl = %A\nColor = %A"
                  myModel.MyR myModel.Myr myModel.Myl myModel.MyColor
  System.Windows.MessageBox.Show(s, "Spirograph") |> ignore
  ()

let whenClosed _ =
  ()

do
  this.Loaded.Add whenLoaded
  this.Closing.Add whenClosing
  this.Closed.Add whenClosed

override this.buttonBlueClick(sender: obj,
                              eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MBlue
  enableParameterFields(true)
  ()

override this.buttonRedClick(sender: obj,
                             eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRed
  enableParameterFields(true)
  ()

override this.buttonRandomClick(sender: obj,
                                eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRandom
```

```
    enableParameterFields(false)
    ()

  override this.okButton_Click(sender: obj,
                               eArgs: System.Windows.RoutedEventArgs) =
    // Only change the spirograph parameters in the model if we hit OK in the
    // dialog box.
    if myColor = MRandom
    then myModel.Randomize
    else myR <- RBox.Text |> int
         myr <- rBox.Text |> int
         myl <- lBox.Text |> float

         myModel.MyR   <- myR
         myModel.Myr   <- myr
         myModel.Myl   <- myl
         model.MyColor <- myColor

    // Note that setting the DialogResult to nullable true is essential to get
    // the OK button to work.
    this.DialogResult <- new System.Nullable<bool> true
    ()
```

Spirograph.fsspirograph。。。

。。

System.Windows.Element.FindName。。。。StackOverflowNuGet。

MainWindow.xaml

```
namespace Spirograph

type MainWindow(app: App, model: Model) as this =
  inherit MainWindowXaml()

  let myApp   = app
  let myModel = model

  let whenLoaded _ =
    ()

  let whenClosing _ =
    ()

  let whenClosed _ =
    ()

  let menuExitHandler _ =
    System.Windows.MessageBox.Show("Good-bye", "Spirograph") |> ignore
    myApp.Shutdown()
    ()

  let menuParametersHandler _ =
    let myParametersDialog = new DialogBox(myApp, myModel, this)
    myParametersDialog.Topmost <- true
    let bResult = myParametersDialog.ShowDialog()
    myModel.DrawSpirograph
```

```
    ()

  let menuDrawHandler _ =
    if myModel.MyColor = MRandom then myModel.Randomize
    myModel.DrawSpirograph
    ()

  let menuAboutHandler _ =
    System.Windows.MessageBox.Show("F#/WPF Menus & Dialogs", "Spirograph")
    |> ignore
    ()

  do
    this.Loaded.Add whenLoaded
    this.Closing.Add whenClosing
    this.Closed.Add whenClosed
    this.menuExit.Click.Add menuExitHandler
    this.menuParameters.Click.Add menuParametersHandler
    this.menuDraw.Click.Add menuDrawHandler
    this.menuAbout.Click.Add menuAboutHandler
```

“”。

App.xamlApp.xaml.fs

```
<!-- All boilerplate for now -->
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

```
namespace Spirograph

open System
open System.Windows
open System.Windows.Controls

module Main =
  [<STAThread; EntryPoint>]
  let main _ =
    // Create the app and the model with the "business logic", then create the
    // main window and link its Canvas to the model so the model can access it.
    // The main window is linked to the app in the Run() command in the last line.
    let app = App()
    let model = new Model()
    let mainWindow = new MainWindow(app, model)
    model.MyCanvas <- (mainWindow.FindName("myCanvas") :?> Canvas)

    // Make sure the window is on top, and set its size to 2/3 of the dimensions
    // of the screen.
    mainWindow.Topmost <- true
    mainWindow.Height  <-
      (System.Windows.SystemParameters.PrimaryScreenHeight * 0.67)
    mainWindow.Width   <-
      (System.Windows.SystemParameters.PrimaryScreenWidth * 0.67)
```

```
    app.Run(mainWindow) // Returns application's exit code.
```

App.xaml。App.xaml.fsModelMainWindowMainWindow。

xamlBuildResource。exe。FFsXaml。

。。。

FWPFFsXaml https://riptutorial.com/zh-TW/fsharp/topic/9145/f--wpf-fsxaml-

9:

Examples

F 。

```
let add x y = x + y
```

F

```
val add : x:int -> y:int -> int
```

`int -> int -> int` 。 `->int -> (int -> int)` 。 addint **intint** 。

```
let addTwo = add 2
// val addTwo : (int -> int)
let five = addTwo 3
// val five : int = 5
```

“”`add`

```
let three = add 1 2
// val three : int = 3
```

```
let addFiveNumbers a b c d e = a + b + c + d + e
// val addFiveNumbers : a:int -> b:int -> c:int -> d:int -> e:int -> int
let addFourNumbers = addFiveNumbers 0
// val addFourNumbers : (int -> int -> int -> int -> int)
let addThreeNumbers = addFourNumbers 0
// val addThreeNumbers : (int -> int -> int -> int)
let addTwoNumbers = addThreeNumbers 0
// val addTwoNumbers : (int -> int -> int)
let six = addThreeNumbers 1 2 3  // This will calculate 0 + 0 + 1 + 2 + 3
// val six : int = 6
```

。 。

F。

F`let`

```
let timesTwo x = x * 2
```

`timesTwox` 。 F `fsharpi`OS XLinux `fsi.exe`Windows`;;fsharpi`

```
val timesTwo : x:int -> int
```

timesTwointx int ◦ int -> int ◦

Fx ◦ x2 x2◦ F;◦

```
let hello =  // This is a value, not a function
    printfn "Hello world"
```

```
let hello () =
    printfn "Hello world"
```

hellounit -> unit “unit” ◦

Curried vs Tupled Functions

FCurriedTupled◦

```
let curriedAdd x y = x + y // Signature: x:int -> y:int -> int
let tupledAdd (x, y) = x + y // Signature:  x:int * y:int -> int
```

F.NETFTupled◦ FCurried◦

CurriedF◦ Tupled◦

```
let addTen = curriedAdd 10 // Signature: int -> int

// Or with the Piping operator
3 |> curriedAdd 7 // Evaluates to 10
```

Curried◦ !!

```
let explicitCurriedAdd x = (fun y -> x + y) // Signature: x:int -> y:int -> int
let veryExplicitCurriedAdd = (fun x -> (fun y -> x + y)) // Same signature
```

◦

.NETTupled◦

◦

◦ ◦ ◦

inline

```
// inline indicate that we want to replace each call to sayHello with the body
// of the function
let inline sayHello s1 =
    sprintf "Hello %s" s1

// Call to sayHello will be replaced with the body of the function
let s = sayHello "Foo"
```

```
// After inlining ->
// let s = sprintf "Hello %s" "Foo"

printfn "%s" s
// Output
// "Hello Foo"
```

```
let inline addAndMulti num1 num2 =
    let add = num1 + num2
    add * 2

let i = addAndMulti 2 2
// After inlining ->
// let add = 2 + 2
// let i = add * 2

printfn "%i" i
// Output
// 8
```

。 。

F

- `|>`

  ```
  let print message =
      printf "%s" message

  // "Hello World" will be passed as a parameter to the print function
  "Hello World" |> print
  ```

- `<|`

  ```
  let print message =
      printf "%s" message

  // "Hello World" will be passed as a parameter to the print function
  print <| "Hello World"
  ```

```
// We want to create a sequence from 0 to 10 then:
// 1 Keep only even numbers
// 2 Multiply them by 2
// 3 Print the number

let mySeq = seq { 0..10 }
let filteredSeq = Seq.filter (fun c -> (c % 2) = 0) mySeq
let mappedSeq = Seq.map ((*) 2) filteredSeq
let finalSeq = Seq.map (sprintf "The value is %i.") mappedSeq

printfn "%A" finalSeq
```

```
// Using forward pipe, we can eliminate intermediate let binding
let finalSeq =
    seq { 0..10 }
```

```
    |> Seq.filter (fun c -> (c % 2) = 0)
    |> Seq.map ((*) 2)
    |> Seq.map (sprintf "The value is %i.")

printfn "%A" finalSeq
```

。

|。F|||><|||。

```
let printPerson name age =
    printf "My name is %s, I'm %i years old" name age

("Foo", 20) ||> printPerson
```

https://riptutorial.com/zh-TW/fsharp/topic/2525/

10:

Examples

F

◦

```
let mi  = typeof<System.String>.GetMethod "StartsWith"
```

1. String.StartsWith
2.
3. Rename methods◦

◦ ◦

F#◦

```
open FSharp.Quotations
open System.Reflection

let getConstructorInfo (e : Expr<'T>) : ConstructorInfo =
  match e with
  | Patterns.NewObject (ci, _) -> ci
  | _ -> failwithf "Expression has the wrong shape, expected NewObject (_, _) instead got: %A"
e


let getMethodInfo (e : Expr<'T>) : MethodInfo =
  match e with
  | Patterns.Call (_, mi, _) -> mi
  | _ -> failwithf "Expression has the wrong shape, expected Call (_, _, _) instead got: %A" e
```

```
printfn "%A" <| getMethodInfo <@ "".StartsWith "" @>
printfn "%A" <| getMethodInfo <@ List.singleton 1 @>
printfn "%A" <| getConstructorInfo <@ System.String [||] @>
```

```
Boolean StartsWith(System.String)
Void .ctor(Char[])
Microsoft.FSharp.Collections.FSharpList`1[System.Int32] Singleton[Int32](Int32)
```

<@ ... @>F#◦ <@ "".StartsWith "" @> Call (Some (Value ("")), StartsWith, [Value ("")]) ◦

getMethodInfo◦

◦

https://riptutorial.com/zh-TW/fsharp/topic/4124/

11:

Examples

。

```
type Shape =
    | Circle of diameter:int
    | Rectangle of width:int * height:int

let shapeIsTenWide = function
    | Circle(diameter=10)
    | Rectangle(width=10) -> true
    | _ -> false
```

C -。“Item”“Item1”“Item2”.....

F。C ++VB。

```
// define a discriminated union that can hold either a float or a string
type numOrString =
    | F of float
    | S of string

let str = S "hi" // use the S constructor to create a string
let fl = F 3.5 // use the F constructor to create a float

// you can use pattern matching to deconstruct each type
let whatType x =
    match x with
        | F f -> printfn "%f is a float" f
        | S s -> printfn "%s is a string" s

whatType str // hi is a string
whatType fl // 3.500000 is a float
```

。。

```
// This union can represent any one day of the week but none of
// them are tied to a specific underlying F# type
type DayOfWeek = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Reflection

Discriminated Union

```
module UnionConversion
    open Microsoft.FSharp.Reflection

    let toString (x: 'a) =
        match FSharpValue.GetUnionFields(x, typeof<'a>) with
```

```
        | case, _ -> case.Name

    let fromString<'a> (s : string) =
        match FSharpType.GetUnionCases typeof<'a> |> Array.filter (fun case -> case.Name = s)
with
        | [|case|] -> Some(FSharpValue.MakeUnion(case, [||])) :?> 'a)
        | _ -> None
```

。

```
// Define single-case discriminated union type.
type OrderId = OrderId of int
// Construct OrderId type.
let order = OrderId 123
// Deconstruct using pattern matching.
// Parentheses used so compiler doesn't think it is a function definition.
let (OrderId id) = order
```

FCJava。

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

let point2 = Point(-2.5, -4.0)
```

```
let (Point(x1, y1)) = point1
// val x1 : float = 0.0
// val y1 : float = 3.0

let distance (Point(x1,y1)) (Point(x2,y2)) =
    pown (x2-x1) 2 + pown (y2-y1) 2 |> sqrt
// val distance : Point -> Point -> float

distance point1 point2
// val it : float = 7.433034374
```

RequireQualifiedAccess

`RequireQualifiedAccess`union`MyUnion.MyCaseMyCase`。

```
type [<RequireQualifiedAccess>] Requirements =
    None | Single | All

// Uses the DU with qualified access
let noRequirements = Requirements.None

// Here, None still refers to the standard F# option case
let getNothing () = None
```

```
// Compiler error unless All has been defined elsewhere
let invalid = All
```

System SingleSystem.Single◦ Requirements.Single◦

◦

```
type Tree =
    | Branch of int * Tree list
    | Leaf of int
```

```
    1
  2   5
3   4
```

```
let leaf1 = Leaf 3
let leaf2 = Leaf 4
let leaf3 = Leaf 5

let branch1 = Branch (2, [leaf1; leaf2])
let tip = Branch (1, [branch1; leaf3])
```

```
let rec toList tree =
    match tree with
    | Leaf x -> [x]
    | Branch (x, xs) -> x :: (List.collect toList xs)

let treeAsList = toList tip // [1; 2; 3; 4; 5]
```

◦

```
// BAD
type Arithmetic = {left: Expression; op:string; right: Expression}
// illegal because until this point, Expression is undefined
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

```
// BAD
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of {left: Expression; op:string; right: Expression}
// illegal in recent F# versions
```

and

```
// GOOD
type Arithmetic = {left: Expression; op:string; right: Expression}
```

```
and Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

https://riptutorial.com/zh-TW/fsharp/topic/1025/

12:

Examples

Monads

F

“Monads”。Hitchhikers。

Monads。Liskov。。Monads。

MonadsMonadic Parser Combinator。Monadic。

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)
```

Parser

```
// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)

// Different ways to create parser result
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
let eos : Parser<unit> = Parser <| fun (s, p) ->
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter
```

`satisfysatEOSsat。satisfy。`

FSI

```
> run digit "";;
val it : char option * int = (null, 0)
> run digit "123";;
val it : char option * int = (Some '1', 1)
> run digit "hello";;
val it : char option * int = (null, 0)
```

。

```
// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)
```

bind

```
> run (bind digit (fun v -> digit)) "123";;
val it : char option * int = (Some '2', 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "123";;
val it : (char * char) option * int = (Some ('1', '2'), 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "1";;
val it : (char * char) option * int = (null, 1)
```

bind◦ bind◦

```
> run (bind digit (fun v -> bind digit (fun w -> bind digit (fun u -> return_ (v,w,u)))))
"123";;
val it : (char * char * char) option * int = (Some ('1', '2', '3'), 3)
```

bind◦

。

```
type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()
```

FSI

```
let p = parser {
          let! v = digit
          let! u = digit
          return v,u
        }
run p "123"
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

```
> let p = bind digit (fun v -> bind digit (fun u -> return_ (v,u)))
run p "123";;
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

orElse

```
// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp
```

letterOrDigit

```
> let letterOrDigit = orElse letter digit;;
val letterOrDigit : Parser<char> = Parser <fun:orElse@70-1>
> run letterOrDigit "123";;
val it : char option * int = (Some '1', 1)
> run letterOrDigit "hello";;
val it : char option * int = (Some 'h', 1)
> run letterOrDigit "!!!";;
val it : char option * int = (null, 0)
```

Infix

FP>>= >=> <-◦ + - * /% ◦ FP◦ FP>>= >=> <-◦

```
let (>>=)   t   uf  = bind t uf
let (<|>)   t   u   = orElse t u
```

>>=bind <|>orElse ◦

```
let letterOrDigit = letter <|> digit
let p = digit >>= fun v -> digit >>= fun u -> return_ (v,u)
```

```
// 'map' runs parser 't' and maps the result using 'm'
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
```

```
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }
```

manysepBy ◦ manysepBy

```
// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop
```

1+2*3

pint

```
// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }
```

char list ◦ fail ◦

FSI`pint`

```
> run pint "123";;
val it : int option * int = (Some 123, 3)
```

```
// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
```

```
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)
```

FSI

```
> run padd "+";;
val it : (int -> int -> int) option * int = (Some <fun:padd@121-1>, 1)
```

```
// 'pmullike' parsers integers separated by operators with same precedence as multiply
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

FSI

```
> run pexpr "2+123*2-3";;
val it : int option * int = (Some 245, 9)
```

`Parser<'T> return_ bindreturn_` Monadic Parser Combinator。

MonadsParsersParsers。 Monads。

。 FParsec。

Monads。 Monads。

Monads

1. State Monad -
2. Tracer Monad - 。 State Monad
3. Turtle Monad - TurtleLogosMonad。 State Monad
4. Monad - Monad。 F`async`

Monads。 。

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)

// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)
```

```
// Different ways to create parser result
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
let eos : Parser<unit> = Parser <| fun (s, p) ->
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter

// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)

type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()

// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp

let (>>=) t uf    = bind t uf
let (<|>) t u     = orElse t u

// 'map' runs parser 't' and maps the result using 'm'
```

```
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }

// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop

// A simplistic integer expression parser

// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }

// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)

// 'pmullike' parsers integers separated by operators with same precedence as multiply
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

Computation ExpressionsMonad

MonadsF# CE。 CEMonads

```
let v = m >>= fun x -> n >>= fun y -> return_ (x, y)
```

```
let v = ce {
    let! x = m
    let! y = n
    return x, y
  }
```

。

CEID。 ID。 。

id。 Monads Log Monadid。

```
type Context =
  {
    CorrelationId : Guid
  }
  static member New () : Context = { CorrelationId = Guid.NewGuid () }

type Function<'T> = Context -> 'T

// Runs a Function<'T> with a new log context
let run t = t (Context.New ())
```

ID

```
let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId v
let tracef fmt              = kprintf trace fmt
```

traceFunction<unit>。 IDv

bindreturn_ Monad LawsLog Monad。

```
let bind t uf : Function<_> = fun ctx ->
  let tv = t ctx  // Invoke t with the log context
  let u  = uf tv  // Create u function using result of t
  u ctx           // Invoke u with the log context

// >>= is the common infix operator for bind
let inline (>>=) (t, uf) = bind t uf

let return_ v : Function<_> = fun ctx -> v
```

LogBuilder CELog Monads。

```
type LogBuilder() =
  member x.Bind   (t, uf) = bind t uf
```

```
  member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()
```

```
let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }
```

g

```
printfn "g produced %A" (Log.run g)
```

```
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: starting..."
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "f: called with: x = 1, y = 2"
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: f produced 3"
g produced 3
```

CorrelationId`rungf` ◦

CECE S◦

```
module Log =
  open System
  open FSharp.Core.Printf

  type Context =
    {
      CorrelationId : Guid
    }
    static member New () : Context = { CorrelationId = Guid.NewGuid () }

  type Function<'T> = Context -> 'T

  // Runs a Function<'T> with a new log context
  let run t = t (Context.New ())

  let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId
v
  let tracef fmt              = kprintf trace fmt

  let bind t uf : Function<_> = fun ctx ->
    let tv = t ctx  // Invoke t with the log context
    let u  = uf tv  // Create u function using result of t
    u ctx           // Invoke u with the log context
```

```
  // >>= is the common infix operator for bind
  let inline (>>=) (t, uf) = bind t uf

  let return_ v : Function<_> = fun ctx -> v

  type LogBuilder() =
    member x.Bind   (t, uf) = bind t uf
    member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()

let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }

[<EntryPoint>]
let main argv =
  printfn "g produced %A" (Log.run g)
  0
```

https://riptutorial.com/zh-TW/fsharp/topic/3320/

13:

Examples

```
let string1 = "Hello" //simple string

let string2 = "Line\nNewLine" //string with newline escape sequence

let string3 = @"Line\nSameLine" //use @ to create a verbatim string literal

let string4 = @"Line""with""quoutes inside" //double quoute to indicate a single quoute inside
@ string

let string5 = """single "quote" is ok""" //triple-quote string literal, all symbol including
quote are verbatim

let string6 = "ab
cd"// same as "ab\ncd"

let string7 = "xx\
    yy" //same as "xxyy", backslash at the end contunies the string without new line, leading
whitespace on the next line is ignored
```

。

.NET`String.Format``StringBuilder.AppendFormat`

```
open System
open System.Text

let hello = String.Format ("Hello {0}", "World")
// return a string with "Hello World"

let builder = StringBuilder()
let helloAgain = builder.AppendFormat ("Hello {0} again!", "World")
// return a StringBuilder with "Hello World again!"
```

FC。.NET

- `sprintf` String.Format

```
open System

let hello = sprintf "Hello %s" "World"
// "Hello World", "%s" is for string

let helloInt = sprintf "Hello %i" 42
// "Hello 42", "%i" is for int

let helloFloat = sprintf "Hello %f" 4.2
// "Hello 4.2000", "%f" is for float

let helloBool = sprintf "Hello %b" true
// "Hello true", "%b" is for bool
```

```
let helloNativeType = sprintf "Hello %A again!" ("World", DateTime.Now)
// "Hello {formatted date}", "%A" is for native type

let helloObject = sprintf "Hello %O again!" DateTime.Now
// "Hello {formatted date}", "%O" is for calling ToString
```

- `bprintf` StringBuilder.AppendFormat

```
open System
open System.Text

let builder = StringBuilder()

// Attach the StringBuilder to the format function with partial application
let append format = Printf.bprintf builder format

// Same behavior as sprintf but strings are appended to a StringBuilder
append "Hello %s again!\n" "World"
append "Hello %i again!\n" 42
append "Hello %f again!\n" 4.2
append "Hello %b again!\n" true
append "Hello %A again!\n" ("World", DateTime.Now)
append "Hello %O again!\n" DateTime.Now

builder.ToString() // Get the result string
```

.NET

-
-
- F

https://riptutorial.com/zh-TW/fsharp/topic/1397/

14: CF

Examples

POCO。

```
// C#
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
}
```

F3.0Cauto-properties

```
// F#
type Person() =
    member val FirstName = "" with get, set
    member val LastName = "" with get, set
    member val BirthDay = System.DateTime.Today with get, set
```

```
// C#
var person = new Person { FirstName = "Bob", LastName = "Smith", Birthday = DateTime.Today };
// F#
let person = new Person(FirstName = "Bob", LastName = "Smith")
```

F。

```
type Person = {
    FirstName:string;
    LastName:string;
    Birthday:System.DateTime
}
```

```
let person = { FirstName = "Bob"; LastName = "Smith"; Birthday = System.DateTime.Today }
```

specifiying`with`

```
let formal = { person with FirstName = "Robert" }
```

◦ C`IDisposable`

```
public class Resource : IDisposable
{
    private MustBeDisposed internalResource;

    public Resource()
```

```
    {
        internalResource = new MustBeDisposed();
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing){
        if (disposing){
            if (resource != null) internalResource.Dispose();
        }
    }
}
```

F`interface interface`

```
type Resource() =
    let internalResource = new MustBeDisposed()

    interface IDisposable with
        member this.Dispose(): unit =
            this.Dispose(true)
            GC.SuppressFinalize(this)

    member __.Dispose disposing =
        match disposing with
        | true  -> if (not << isNull) internalResource then internalResource.Dispose()
        | false -> ()
```

CF https://riptutorial.com/zh-TW/fsharp/topic/6828/c-f-

15:

Examples

。

Seq

```
// Create an empty generic sequence
let emptySeq = Seq.empty

// Create an empty int sequence
let emptyIntSeq = Seq.empty<int>

// Create a sequence with one element
let singletonSeq = Seq.singleton 10

// Create a sequence of n elements with the specified init function
let initSeq = Seq.init 10 (fun c -> c * 2)

// Combine two sequence to create a new one
let combinedSeq = emptySeq |> Seq.append singletonSeq

// Create an infinite sequence using unfold with generator based on state
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0
```

```
// Create a sequence with element from 0 to 10
let intSeq = seq { 0..10 }

// Create a sequence with an increment of 5 from 0 to 50
let intIncrementSeq = seq{ 0..5..50 }

// Create a sequence of strings, yield allow to define each element of the sequence
let stringSeq = seq {
    yield "Hello"
    yield "World"
}

// Create a sequence from multiple sequence, yield! allow to flatten sequences
let flattenSeq = seq {
    yield! seq { 0..10 }
    yield! seq { 11..20 }
}
```

。System.Collections.Generic.IEnumerablelazy。。Seq.module。

```
let mySeq = { 0..20 } // Create a sequence of int from 0 to 20
mySeq
|> Seq.iter (printf "%i ") // Enumerate each element of the sequence and print it
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Seq.map

```
let seq = seq {0..10}

s |> Seq.map (fun x -> x * 2)

> val it : seq<int> = seq [2; 4; 6; 8; ...]
```

Seq.map

Seq.filter

◦ Seq`filter`◦ `filter('a -> bool) -> seq<'a> -> seq<'a>`；'a'a'a ◦

```
// Function that tests if an integer is even
let isEven x = (x % 2) = 0

// Generates an infinite sequence that contains the natural numbers
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0

// Can be used to filter the naturals sequence to get only the even numbers
let evens = Seq.filter isEven naturals
```

```
let data = [1; 2; 3; 4; 5;]
let repeating = seq {while true do yield! data}
```

`seq {}`

https://riptutorial.com/zh-TW/fsharp/topic/2354/

16:

Examples

F。Lazy Evaluation。 F`lazy``sequences`。

```
// define a lazy computation
let comp = lazy(10 + 20)

// we need to force the result
let ans = comp.Force()
```

Lazy Evaluation

```
let rec factorial n =
  if n = 0 then
    1
  else
    (factorial (n - 1)) * n


let computation = lazy(printfn "Hello World\n"; factorial 10)

// Hello World will be printed
let ans = computation.Force()

// Hello World will not be printed here
let ansAgain = computation.Force()
```

F

F。 。Lazy Evaluation。 。

`lazy``Sequences`FLazy

```
// 23 * 23 is not evaluated here
// lazy keyword creates lazy computation whose evaluation is deferred
let x = lazy(23 * 23)

// we need to force the result
let y = x.Force()

// Hello World not printed here
let z = lazy(printfn "Hello World\n"; 23424)

// Hello World printed and 23424 returned
let ans1 = z.Force()

// Hello World not printed here as z as already been evaluated, but 23424 is
// returned
let ans2 = z.Force()
```

https://riptutorial.com/zh-TW/fsharp/topic/3682/

17:

`if` / `else if` / `else`。 “” 。

F。

Examples

```
let result = Some("Hello World")
match result with
| Some(message) -> printfn message
| None -> printfn "Not feeling talkative huh?"
```

```
let x = true
match x with
| true -> printfn "x is true"
```

C\ Program Filesx86\ Microsoft VS Code \ Untitled-12,7FS0025。 “false”。

bool。

boolsint

```
let x = 5
match x with
| 1 -> printfn "x is 1"
| 2 -> printfn "x is 2"
| _ -> printfn "x is something else"
```

_° _°

_

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
```

```
match sobriety with
| Sober -> printfn "drive home"
| _ -> printfn "call an uber"
```

° _

```
type Sobriety =
```

```
    | Sober
    | Tipsy
    | Drunk
    | Unconscious
```

F。 。 。

```
let x = 4
match x with
| 1 -> printfn "x is 1"
| _ -> printfn "x is anything that wasn't listed above"
| 4 -> printfn "x is 4"
```

x

x = 1x = 4_

```
let x = 4
match x with
| 1 -> printfn "x is 1"
| 4 -> printfn "x is 4"
| _ -> printfn "x is anything that wasn't listed above"
```

x4

```
type Person = {
    Age : int
    PassedDriversTest : bool }

let someone = { Age = 19; PassedDriversTest = true }

match someone.PassedDriversTest with
| true when someone.Age >= 16 -> printfn "congrats"
| true -> printfn "wait until you are 16"
| false -> printfn "you need to pass the test"
```

https://riptutorial.com/zh-TW/fsharp/topic/1335/

18:

Examples

◦ ◦

```
let rev list =
    let rec loop acc = function
        | []           -> acc
        | head :: tail -> loop (head :: acc) tail
    loop [] list
```

◦ Finteractive `'T list -> 'T list ◦ 'T◦ 'a'T - ◦ int liststring list int liststring list ◦`

Finteractive

```
> let rev list = ...
val it : 'T list -> 'T list
> rev [1; 2; 3; 4];;
val it : int list = [4; 3; 2; 1]
> rev ["one", "two", "three"];;
val it : string list = ["three", "two", "one"]
```

```
let map f list =
    let rec loop acc = function
        | []           -> List.rev acc
        | head :: tail -> loop (f head :: acc) tail
    loop [] list
```

`('a -> 'b) -> 'a list -> 'b list ◦ 'a'b◦ 'aflist◦ - ◦`

```
> let map f list = ...
val it : ('a -> 'b) -> 'a list -> 'b list
> map (fun x -> float x * 1.5) [1; 2; 3; 4];;
val it : float list = [1.5; 3.0; 4.5; 6.0]
> map (sprintf "abc%.1f") [1.5; 3.0; 4.5; 6.0];;
val it : string list = ["abc1.5"; "abc3.0"; "abc4.5"; "abc6.0"]
> map (fun x -> x + 1) [1.0; 2.0; 3.0];;
error FS0001: The type 'float' does not match the type 'int'
```

https://riptutorial.com/zh-TW/fsharp/topic/7731/

19:

Examples

match。

```
let (|Positive|Negative|Zero|) num =
    if num > 0 then Positive
    elif num < 0 then Negative
    else Zero
```

```
let Sign value =
    match value with
    | Positive -> printf "%d is positive" value
    | Negative -> printf "%d is negative" value
    | Zero -> printf "The value is zero"

Sign -19 // -19 is negative
Sign 2 // 2 is positive
Sign 0 // The value is zero
```

。

```
let (|HasExtension|_|) expected (uri : string) =
    let result = uri.EndsWith (expected, StringComparison.CurrentCultureIgnoreCase)
    match result with
    | true -> Some true
    | _ -> None
```

```
    let isXMLFile uri =
        match uri with
        | HasExtension ".xml" _ -> true
        | _ -> false
```

Active Patterns

F#Active Patterns。

```
// val f : string option -> string option -> string
let f v u =
  let v = defaultArg v "Hello"
  let u = defaultArg u "There"
  v + " " + u

// val g : 'T -> 'T (requires 'T null)
let g v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v.ToString ()
```

◦ F#Active Patternsintent◦

```
let inline (|DefaultArg|) dv ov = defaultArg ov dv

let inline (|NotNull|) v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v

// val f : string option -> string option -> string
let f (DefaultArg "Hello" v) (DefaultArg "There" u) = v + " " + u

// val g : 'T -> string (requires 'T null)
let g (NotNull v) = v.ToString ()
```

fg◦

```
printfn "%A" <| f (Some "Test") None  // Prints "Test There"
printfn "%A" <| g "Test"              // Prints "Test"
printfn "%A" <| g null                // Will throw
```

Active Patterns◦ ILSpyfg◦

```
public static string f(FSharpOption<string> _arg2, FSharpOption<string> _arg1)
{
  return Operators.DefaultArg<string>(_arg2, "Hello") + " " +
Operators.DefaultArg<string>(_arg1, "There");
}

public static string g<a>(a _arg1) where a : class
{
  if (_arg1 != null)
  {
    a a = _arg1;
    return a.ToString();
  }
  throw new NullReferenceException();
}
```

inline Active Patterns◦

.NET API

Active Patterns.NET API◦

System.Int32.TryParse

```
let couldParse, parsedInt = System.Int32.TryParse("1")
if couldParse then printfn "Successfully parsed int: %i" parsedInt
else printfn "Could not parse int"
```

```
match System.Int32.TryParse("1") with
| (true, parsedInt) -> printfn "Successfully parsed int: %i" parsedInt
| (false, _) -> printfn "Could not parse int"
```

System.Int32.TryParse

```
let (|Int|_|) str =
    match System.Int32.TryParse(str) with
    | (true, parsedInt) -> Some parsedInt
    | _ -> None
```

```
match "1" with
| Int parsedInt -> printfn "Successfully parsed int: %i" parsedInt
| _ -> printfn "Could not parse int"
```

API

```
let (|MatchRegex|_|) pattern input =
    let m = System.Text.RegularExpressions.Regex.Match(input, pattern)
    if m.Success then Some m.Groups.[1].Value
    else None

match "bad" with
| MatchRegex "(good|great)" mood ->
    printfn "Wow, it's a %s day!" mood
| MatchRegex "(bad|terrible)" mood ->
    printfn "Unfortunately, it's a %s day." mood
| _ ->
    printfn "Just a normal day"
```

- 。

“”

```
let (|Odd|Even|) number =
  if number % 2 = 0
  then Even
  else Odd
```

。

```
let n = 13
match n with
| Odd -> printf "%i is odd" n
| Even -> printf "%i is even" n
```

。

option。 _。

```
let (|Integer|_|) str =
  match Int32.TryParse(str) with
  | (true, i) -> Some i
  | _ -> None
```

。

```
let s = "13"
match s with
| Integer i -> "%i was successfully parsed!" i
| _ -> "%s is not an int" s
```

。

https://riptutorial.com/zh-TW/fsharp/topic/962/

20:

- [] //。

  head :: tail //。 ::Cons。

  let list1 = [1; 2; 3] //。

  let list2 = 0 :: list1 //[0; 1; 2; 3]

  let list3 = list1 @ list2 //[1; 2; 3; 0; 1; 2; 3]。 @。

  list4 = [1..3] //[1; 2; 3]

  list5 = [1..2..10] //[1; 3; 5; 7; 9]

  let list6 = [for i in 1..10 do if i2 = 1 then yield i] // result is [1; 3; 5; 7; 9]

Examples

```
let list1 = [ 1; 2 ]
let list2 = [ 1 .. 100 ]

// Accessing an element
printfn "%A" list1.[0]

// Pattern matching
let rec patternMatch aList =
    match aList with
    | [] -> printfn "This is an empty list"
    | head::tail -> printfn "This list consists of a head element %A and a tail list %A" head
tail
                    patternMatch tail

patternMatch list1

// Mapping elements
let square x = x*x
let list2squared = list2
                   |> List.map square
printfn "%A" list2squared
```

```
let rec sumTotal list =
    match list with
    | [] -> 0 // empty list -> return 0
    | head :: tail -> head + sumTotal tail
```

“`list` 0.。 `[1]` `[1; 2]` [1; 2; 3]。 `list`[1]`head1 tail[]head + sumTotal tail` 。

```
sumTotal [1; 2; 3]
// head -> 1, tail -> [2; 3]
1 + sumTotal [2; 3]
```

```
1 + (2 + sumTotal [3])
1 + (2 + (3 + sumTotal [])) // sumTotal [] is defined to be 0, recursion stops here
1 + (2 + (3 + 0)))
1 + (2 + 3)
1 + 5
6
```

sumTotal

```
let sumTotal list = List.fold (+) 0 list
```

◦ ◦

```
> let integers = [1; 2; 45; -1];;
val integers : int list = [1; 2; 45; -1]


> let floats = [10.7; 2.0; 45.3; -1.05];;
val floats : float list = [10.7; 2.0; 45.3; -1.05]
```

◦

```
> let emptyList = [];;
val emptyList : 'a list
```

byte

```
> let bytes = [byte(55); byte(10); byte(100)];;
val bytes : byte list = [55uy; 10uy; 100uy]
```

◦

```
> type number = | Real of float | Integer of int;;

type number =
  | Real of float
  | Integer of int

> let numbers = [Integer(45); Real(0.0); Integer(127)];;
val numbers : number list = [Integer 45; Real 0.0; Integer 127]
```

intfloatchar...startend

[start..end]

```
> let c=['a' .. 'f'];;
val c : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']

let f=[45 .. 60];;
val f : int list =
  [45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60]
```

[start..step..end]

```
> let i=[4 .. 2 .. 11];;
val i : int list = [4; 6; 8; 10]

> let r=[0.2 .. 0.05 .. 0.28];;
val r : float list = [0.2; 0.25]
```

。

```
[for <identifier> in range -> expr]

[for <identifier> in range do ... yield expr]
```

```
> let oddNumbers = [for i in 0..10 -> 2 * i + 1];; // odd numbers from 1 to 21
val oddNumbers : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19; 21]

> let multiples3Sqrt = [for i in 1..27 do if i % 3 = 0 then yield sqrt(float(i))];; //sqrt of
multiples of 3 from 3 to 27
val multiples3Sqrt : float list =
  [1.732050808; 2.449489743; 3.0; 3.464101615; 3.872983346; 4.242640687;    4.582575695;
4.898979486; 5.196152423]
```

Cons::

::head

```
> let l=12::[] ;;
val l : int list = [12]

> let l1=7::[14; 78; 0] ;;
val l1 : int list = [7; 14; 78; 0]

> let l2 = 2::3::5::7::11::[13;17] ;;
val l2 : int list = [2; 3; 5; 7; 11; 13; 17]
```

@。

```
> let l1 = [12.5;89.2];;
val l1 : float list = [12.5; 89.2]

> let l2 = [1.8;7.2] @ l1;;
val l2 : float list = [1.8; 7.2; 12.5; 89.2]
```

https://riptutorial.com/zh-TW/fsharp/topic/1268/

21:

Examples

◦ seq<'a>'b'b'a ◦ ◦

'aint ◦ ◦ [1; 2; 3]

```
List.fold (fun x y -> x + y) 0 [1; 2; 3] // val it : int = 6
```

Listfold List.fold ◦ fold take ◦ ◦ fold◦

```
let add x y = x + y // this is our folder (a binary function, takes two arguments)
List.fold add 0 [1; 2; 3;]
=> List.fold add (add 0 1) [2; 3]
// the binary function is passed again as the folder in the first argument
// the initial value is updated to add 0 1 = 1 in the second argument
// the tail of the list (all elements except the first one) is passed in the third argument
// it becomes this:
List.fold add 1 [2; 3]
// Repeat untill the list is empty -> then return the "inital" value
List.fold add (add 1 2) [3]
List.fold add 3 [3] // add 1 2 = 3
List.fold add (add 3 3) []
List.fold add 6 [] // the list is empty now -> return 6
6
```

List.sumList.sumList.fold add LanguagePrimitives.GenericZero ◦

elemetscount

1◦

```
List.fold (fun x y -> x + 1) 0 [1; 2; 3] // val it : int = 3
```

```
[1; 2; 3]
|> List.map (fun x -> 1) // turn every elemet into 1, [1; 2; 3] becomes [1; 1; 1]
|> List.sum // sum [1; 1; 1] is 3
```

count

```
let count xs =
    xs
    |> List.map (fun x -> 1)
    |> List.fold (+) 0 // or List.sum
```

List.reduceList.fold compairing

```
let max x y = if x > y then x else y
// val max : x:'a -> y: 'a -> 'a when 'a : comparison, so only for types that we can compare
List.reduce max [1; 2; 3; 4; 5] // 5
List.reduce max ["a"; "b"; "c"] // "c", because "c" > "b" > "a"
List.reduce max [true; false] // true, because true > false
```

```
let min x y = if x < y then x else y
List.reduce min [1; 2; 3; 4; 5] // 1
List.reduce min ["a"; "b"; "c"] // "a"
List.reduce min [true; false] // false
```

@

```
// [1;2] @ [3; 4] = [1; 2; 3; 4]
let merge xs ys = xs @ ys
List.fold merge [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
```

```
List.fold (@) [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
List.fold (+) 0 [1; 2; 3] // 6
List.fold (||) false [true; false] // true, more on this below
List.fold (&&) true [true; false] // false, more on this below
// etc...
```

◦ n[1 .. n]◦

```
// the folder
let times x y = x * y
let factorial n = List.fold times 1 [1 .. n]
// Or maybe for big integers
let factorial n = List.fold times 1I [1I .. n]
```

---

forall existscontains

forall◦ exists ◦ bool◦ ◦

true&&true◦

```
List.fold (&&) true [true; true; true] // true
List.fold (&&) true [] // true, empty list -> return inital value
List.fold (&&) true [false; true] // false
```

true ||false

```
List.fold (||) false [true; false] // true
```

```
List.fold (||) false [false; false] // false, all are false, no element is true
List.fold (||) false [] // false, empty list -> return inital value
```

forallexists。

```
let forall condition elements =
    elements
    |> List.map condition // condition : 'a -> bool
    |> List.fold (&&) true

let exists condition elements =
    elements
    |> List.map condition
    |> List.fold (||) false
```

[1; 2; 3; 4]5

```
forall (fun n -> n < 5) [1 .. 4] // true
```

existscontains

```
let contains x xs = exists (fun y -> y = x) xs
```

```
let contains x xs = exists ((=) x) xs
```

[1 .. 5]2

```
contains 2 [1..5] // true
```

---

reverse

```
let reverse xs = List.fold (fun acc x -> x :: acc) [] xs
reverse [1 .. 5] // [5; 4; 3; 2; 1]
```

---

mapfilter

```
let map f = List.fold (fun acc x -> List.append acc [f x]) List.empty
// map (fun x -> x * x) [1..5] -> [1; 4; 9; 16; 25]

let filter p = Seq.fold (fun acc x -> seq { yield! acc; if p(x) then yield x }) Seq.empty
// filter (fun x -> x % 2 = 0) [1..10] -> [2; 4; 6; 8; 10]
```

fold

floatintbig integerList.sum。List.fold。

1.

。

System.Numerics

System.Numerics

0。

```
let clist = [new Complex(1.0, 52.0); new Complex(2.0, -2.0); new Complex(0.0, 1.0)]

let sum = List.fold (+) (new Complex(0.0, 0.0)) clist
```

```
(3, 51)
```

2.

unionfloatint。

```
type number =
| Float of float
| Int of int
```

```
let list = [Float(1.3); Int(2); Float(10.2)]

let sum = List.fold (
                        fun acc elem ->
                                        match elem with
                                        | Float(elem) -> acc + elem
                                        | Int(elem) -> acc + float(elem)
                       ) 0.0 list
```

```
13.5
```

floatnumber。 elemIntfloat。

https://riptutorial.com/zh-TW/fsharp/topic/2250/

22:

◦ ToString◦

CfFfloat<m>double ◦

Examples

◦ ◦

```
[<Measure>] type m // meters
[<Measure>] type s // seconds
[<Measure>] type accel = m/s^2 // acceleration defined as meters per second squared
```

◦

```
// Compile-time checking that this function will return meters, since (m/s^2) * (s^2) -> m
// Therefore we know units were used properly in the calculation.
let freeFallDistance (time:float<s>) : float<m> =
    0.5 * 9.8<accel> * (time*time)

// It is also made explicit at the call site, so we know that the parameter passed should be
in seconds
let dist:float<m> = freeFallDistance 3.0<s>
printfn "%f" dist
```

```
[<Measure>] type m // meters
[<Measure>] type cm // centimeters

// Conversion factor
let cmInM = 100<cm/m>

let distanceInM = 1<m>
let distanceInCM = distanceInM * cmInM // 100<cm>

// Conversion function
let cmToM (x : int<cm>) = x / 100<cm/m>
let mToCm (x : int<m>) = x * 100<cm/m>

cmToM 100<cm> // 1<m>
mToCm 1<m> // 100<cm>
```

F1<m>100<cm> ◦ ◦ ◦

```
[<Measure>] type kg

// Valid code, invalid physics
let kgToM x = x / 100<kg/m>
```

```
// Invalid code
[<Measure>] type m = 100<cm>
```

“”“”。

```
// Valid code
[<Measure>] type s
[<Measure>] type Hz = /s

1 / 1<s> = 1 <Hz> // Evaluates to true

[<Measure>] type N = kg m/s // Newtons, measuring force. Note lack of multiplication sign.

// Usage
let mass = 1<kg>
let distance = 1<m>
let time = 1<s>

let force = mass * distance / time // Evaluates to 1<kg m/s>
force = 1<N> // Evaluates to true
```

LanguagePrimitives

`LanguagePrimitives`

```
/// This cast preserves units, while changing the underlying type
let inline castDoubleToSingle (x : float<'u>) : float32<'u> =
    LanguagePrimitives.Float32WithMeasure (float32 x)
```

1

```
[<Measure>]
type USD

let toMoneyImprecise (amount : float) =
   amount * 1.<USD>
```

System.Double

```
open LanguagePrimitives

let toMoney amount =
   amount |> DecimalWithMeasure<'u>
```

Finteractive

```
val toMoney : amount:decimal -> decimal<'u>
val toMoneyImprecise : amount:float -> float<USD>
```

`[<Measure>]`

```
type CylinderSize<[<Measure>] 'u> =
    { Radius : float<'u>
```

```
      Height : float<'u> }
```

```
open Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols

/// This has type CylinderSize<m>.
let testCylinder =
    { Radius = 14.<m>
      Height =  1.<m> }
```

SIF`Microsoft.FSharp.Data.UnitSystems.SI` ◦ `UnitNamesUnitSymbols` ◦ SI

```
/// Seconds, the SI unit of time. Type abbreviation for the Microsoft standardized type.
type [<Measure>] s = Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols.s
```

```
/// Seconds, the SI unit of time
type [<Measure>] s // DO NOT DO THIS! THIS IS AN EXAMPLE TO EXPLAIN A PROBLEM.
```

SI◦ ◦

SI◦ `UnitNamesUnitSymbols`

```
open Microsoft.FSharp.Data.UnitSystems.SI

/// This is valid, since both versions refer to the same authoritative type.
let validSubtraction = 1.<UnitSymbols.s> - 0.5<UnitNames.second>
```

https://riptutorial.com/zh-TW/fsharp/topic/1055/

23:

Examples

Memoization。 。

```
let factorial index =
    let rec innerLoop i acc =
        match i with
        | 0 | 1 -> acc
        | _ -> innerLoop (i - 1) (acc * i)

    innerLoop index 1
```

。

Memoization。

memoization

```
// memoization takes a function as a parameter
// This function will be called every time a value is not in the cache
let memoization f =
    // The dictionary is used to store values for every parameter that has been seen
    let cache = Dictionary<_,_>()
    fun c ->
        let exist, value = cache.TryGetValue (c)
        match exist with
        | true ->
            // Return the cached result directly, no method call
            printfn "%O -> In cache" c
            value
        | _ ->
            // Function call is required first followed by caching the result for next call
with the same parameters
            printfn "%O -> Not in cache, calling function..." c
            let value = f c
            cache.Add (c, value)
            value
```

`memoization。 f:('a -> 'b) -> ('a -> 'b)。` memoization。

`printfn;。`

memoization

```
// Pass the function we want to cache as a parameter via partial application
let factorialMem = memoization factorial

// Then call the memoized function
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
```

```
printfn "%i" (factorialMem 4)
printfn "%i" (factorialMem 4)

// Prints
// 10 -> Not in cache, calling function...
// 3628800
// 10 -> In cache
// 3628800
// 10 -> In cache
// 3628800
// 4 -> Not in cache, calling function...
// 24
// 4 -> In cache
// 24
```

。

memoization`f fact。 ffact(n-1)n。`

`memorecmemorecfactfact(n-1)。`

```
let memorec f =
   let cache = Dictionary<_,_>()
   let rec frec n =
       let value = ref 0
       let exist = cache.TryGetValue(n, value)
       match exist with
       | true ->
           printfn "%O -> In cache" n
       | false ->
           printfn "%O -> Not in cache, calling function..." n
           value := f frec n
           cache.Add(n, !value)
       !value
   in frec

let f = fun fact n -> if n<2 then 1 else n*fact(n-1)

[<EntryPoint>]
let main argv =
    let g = memorec(f)
    printfn "%A" (g 10)
    printfn "%A" "----------------------"
    printfn "%A" (g 5)
    Console.ReadLine()
    0
```

```
10 -> Not in cache, calling function...
9 -> Not in cache, calling function...
8 -> Not in cache, calling function...
7 -> Not in cache, calling function...
6 -> Not in cache, calling function...
5 -> Not in cache, calling function...
4 -> Not in cache, calling function...
3 -> Not in cache, calling function...
2 -> Not in cache, calling function...
1 -> Not in cache, calling function...
3628800
```

```
"----------------------"
5 -> In cache
120
```

https://riptutorial.com/zh-TW/fsharp/topic/2698/

24:

Examples

```
type person = {Name: string; Age: int} with   // Defines person record
    member this.print() =
        printfn "%s, %i" this.Name this.Age

let user = {Name = "John Doe"; Age = 27} // creates a new person

user.print() // John Doe, 27
```

```
type person = {Name: string; Age: int}   // Defines person record

let user1 = {Name = "John Doe"; Age = 27} // creates a new person
let user2 = {user1 with Age = 28}         // creates a copy, with different Age
let user3 = {user1 with Name = "Jane Doe"; Age = 29} //creates a copy with different Age and
Name

let printUser user =
    printfn "Name: %s, Age: %i" user.Name user.Age

printUser user1 // Name: John Doe, Age: 27
printUser user2 // Name: John Doe, Age: 28
printUser user3 // Name: Jane Doe, Age: 29
```

https://riptutorial.com/zh-TW/fsharp/topic/1136/

25:

Examples

。 。

+ - * /。

```
let x = 1 + 2 + 3 * 2
```

>> << |><| 。

```
// val f : int -> int
let f v   = v + 1
// val g : int -> int
let g v   = v * 2

// Different ways to compose f and g
// val h : int -> int
let h1 v  = g (f v)
let h2 v  = v |> f |> g   // Forward piping of 'v'
let h3 v  = g <| (f <| v) // Reverse piping of 'v' (because <| has left associcativity we need
())
let h4    = f >> g        // Forward functional composition
let h5    = g << f        // Reverse functional composition (closer to math notation of 'g o
f')
```

F#

1.
2. <|<<F#
3. 。

Monad

MonadsOption<'T>List<'T> Monads>>= >=> <|><*> 。

```
let (>>=) t uf  = Option.bind uf t
let (>=>) tf uf = fun v -> tf v >>= uf
// val oinc   : int -> int option
let oinc   v    = Some (v + 1)    // Increment v
// val ofloat : int -> float option
let ofloat v    = Some (float v)  // Map v to float

// Different ways to compose functions working with Option Monad
// val m : int option -> float option
let m1 v  = Option.bind (fun v -> Some (float (v + 1))) v
let m2 v  = v |> Option.bind oinc |> Option.bind ofloat
let m3 v  = v >>= oinc >>= ofloat
let m4    = oinc >=> ofloat

// Other common operators are <|> (orElse) and <*> (andAlso)
```

```
// If 't' has Some value then return t otherwise return u
let (<|>) t u =
  match t with
  | Some _  -> t
  | None    -> u

// If 't' and 'u' has Some values then return Some (tv*uv) otherwise return None
let (<*>) t u =
  match t, u with
  | Some tv, Some tu  -> Some (tv, tu)
  | _                 -> None

// val pickOne : 'a option -> 'a option -> 'a option
let pickOne t u v = t <|> u <|> v

// val combine : 'a option -> 'b option  -> 'c option -> (('a*'b)*'c) option
let combine t u v = t <*> u <*> v
```

◦ |> >> >>=>=>+ - */◦

F

F#◦ ◦

C# dynamic ◦ JSON◦

F#??<- ◦

```
// (?) allows us to lookup values in a map like this: map?MyKey
let inline (?)   m k   = Map.tryFind k m
// (?<-) allows us to update values in a map like this: map?MyKey <- 123
let inline (?<-) m k v = Map.add k v m

let getAndUpdate (map : Map<string, int>) : int option*Map<string, int> =
  let i = map?Hello       // Equivalent to map |> Map.tryFind "Hello"
  let m = map?Hello <- 3  // Equivalent to map |> Map.add "Hello" 3
  i, m
```

F#◦

https://riptutorial.com/zh-TW/fsharp/topic/4641/

26:

Examples

Option NoneSome ◦

```
type Option<'T> = Some of 'T | None
```

Option <'T>

F# null◦

C#

```
string x = SomeFunction ();
int    l = x.Length;
```

xnull x.Length

```
string x = SomeFunction ();
int    l = x != null ? x.Length : 0;
```

```
string x = SomeFunction () ?? "";
int    l = x.Length;
```

```
string x = SomeFunction ();
int    l = x?.Length;
```

F#null

```
let x = SomeFunction ()
let l = x.Length
```

◦ Option<'T>

```
let SomeFunction () : string option = ...
```

SomeFunctionSome stringNone ◦ string

```
let v =
  match SomeFunction () with
  | Some x  -> x.Length
  | None    -> 0
```

```
string x = SomeFunction ();
int    l = x.Length;
```

string optionLength ◦ stringstring◦

◦ ROP ◦

f

```
let tryParse s =
  let b, v = System.Int32.TryParse s
  if b then Some v else None

let f (g : string option) : float option =
  match g with
  | None    -> None
  | Some s  ->
    match tryParse s with           // Parses string to int
    | None              -> None
    | Some v when v < 0 -> None     // Checks that int is greater than 0
    | Some v -> v |> float |> Some  // Maps int to float
```

fstringSome int ◦ int0float ◦ None ◦

match◦

ROP

1. - Some
2. - None

◦ ◦

ROPg

```
let g (v : string option) : float option =
  v
  |> Option.bind    tryParse  // Parses string to int
  |> Option.filter  ((<) 0)   // Checks that int is greater than 0
  |> Option.map     float     // Maps int to float
```

F#◦

Option<'T>List<'T> 01Option.bindList.pick Option.bindList.collectList.pick◦

bind filtermap g◦

Option<_>Option<_>|>>>◦

ROP◦

C

OptionCC◦ FSharp.CoreCCFNonenull ◦

F4.0

```
let OptionToObject opt =
    match opt with
    | Some x -> x
    | None -> null
```

`System.Nullable` ◦

```
let OptionToNullable x =
    match x with
    | Some i -> System.Nullable i
    | None -> System.Nullable ()
```

F4.0

F4.0`ofObj toObj ofNullabletoNullableOption`◦ Finteractive

```
let l1 = [ Some 1 ; None ; Some 2]
let l2 = l1 |> List.map Option.toNullable;;

// val l1 : int option list = [Some 1; null; Some 2]
// val l2 : System.Nullable<int> list = [1; null; 2]

let l3 = l2 |> List.map Option.ofNullable;;
// val l3 : int option list = [Some 1; null; Some 2]

// Equality
l1 = l2 // fails to compile: different types
l1 = l3 // true
```

`Nonenull` ◦ F`None` ◦

```
let lA = [Some "a"; None; Some "b"]
let lB = lA |> List.map Option.toObj

// val lA : string option list = [Some "a"; null; Some "b"]
// val lB : string list = ["a"; null; "b"]

let lC = lB |> List.map Option.ofObj
// val lC : string option list = [Some "a"; null; Some "b"]

// Equality
lA = lB // fails to compile: different types
lA = lC // true
```

https://riptutorial.com/zh-TW/fsharp/topic/3175/

27:

MailboxProcessorPost◦ RetrieveScan◦ ◦

◦ ◦ ◦

Examples

Hello World

“Hello world” MailboxProcessor◦

◦ Discriminated Unions◦

```
// In this example there is only a single case, it could technically be just a string
type GreeterMessage = SayHelloTo of string
```

◦ MailboxProcessor<'message>.Start◦ ◦

```
let processor = MailboxProcessor<GreeterMessage>.Start(fun inbox ->
    let rec innerLoop () = async {
        // This way you retrieve message from the mailbox queue
        // or await them in case the queue empty.
        // You can think of the `inbox` parameter as a reference to self.
        let! message = inbox.Receive()
        // Now you can process the retrieved message.
        match message with
        | SayHelloTo name ->
            printfn "Hi, %s! This is mailbox processor's inner loop!" name
        // After that's done, don't forget to recurse so you can process the next messages!
        innerLoop()
    }
    innerLoop ())
```

StartMailboxProcessor◦ ReceiveScan◦ ◦

◦ Post - ◦

```
processor.Post(SayHelloTo "Alice")
```

processor◦ ◦

"Hi, Alice! This is mailbox processor's inner loop!"◦

◦ ◦

```
// Increment or decrement by one.
type CounterMessage =
    | Increment
```

```
    | Decrement

let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
        // You can represent the processor's internal mutable state
        // as an immutable parameter to the inner loop function
        let rec innerLoop state = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            // In each call you use the current state to produce a new
            // value, which will be passed to the next call, so that
            // next message sees only the new value as its local state
            match message with
            | Increment ->
                let state' = state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop state'
            | Decrement ->
                let state' = state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop state'
        }
        // We pass the initialState to the first call to innerLoop
        innerLoop initialState)

// Let's pick an initial value and create the processor
let processor = createProcessor 10
```

```
processor.Post(Increment)
processor.Post(Increment)
processor.Post(Decrement)
processor.Post(Increment)
```

```
Waiting for message, the current state is: 10
Counter incremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
Counter decremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
```

。 。

```
let processor = createProcessor 0

[ async { processor.Post(Increment) }
  async { processor.Post(Increment) }
  async { processor.Post(Decrement) }
  async { processor.Post(Decrement) } ]
|> Async.Parallel
|> Async.RunSynchronously
```

。 0。

◦ ◦

```
let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
        // In this case we represent the state as a mutable binding
        // local to this function. innerLoop will close over it and
        // change its value in each iteration instead of passing it around
        let mutable state = initialState

        let rec innerLoop () = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            match message with
            | Increment ->
                let state <- state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop ()
            | Decrement ->
                let state <- state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop ()
        }
        innerLoop ())
```

Posts◦

AsyncReplyChannel<'a>◦

```
type MessageWithResponse = Message of InputData * AsyncReplyChannel<OutputData>
```

◦

```
let! message = inbox.Receive()
match message with
| MessageWithResponse (data, r) ->
    // ...process the data
    let output = ...
    r.Reply(output)
```

AsyncReplyChannel<'a> - MailboxProcessorAsync<'a>◦ PostAndAsynReply AsyncReplyChannel<OutputData> -> MessageWithResponse

```
let! output = processor.PostAndAsyncReply(r -> MessageWithResponse(input, r))
```

◦

PostAndReply◦

ScanTryScan◦ ◦ TryScanNoneScan◦

◦ RegularOperations PriorityOperation RegularOperations ◦

```
type Message =
    | RegularOperation of string
    | PriorityOperation of string

let processor = MailboxProcessor<Message>.Start(fun inbox ->
    let rec innerLoop () = async {
        let! priorityData = inbox.TryScan(fun msg ->
            // If there is a PriorityOperation, retrieve its data.
            match msg with
            | PriorityOperation data -> Some data
            | _ -> None)

        match priorityData with
        | Some data ->
            // Process the data from PriorityOperation.
        | None ->
            // No PriorityOperation was in the queue at the time, so
            // let's fall back to processing all possible messages
            let! message = inbox.Receive()
            match message with
            | RegularOperation data ->
                // We have free time, let's process the RegularOperation.
            | PriorityOperation data ->
                // We did scan the queue, but it might have been empty
                // so it is possible that in the meantime a producer
                // posted a new message and it is a PriorityOperation.
        // And never forget to process next messages.
        innerLoop ()
    }
    innerLoop())
```

https://riptutorial.com/zh-TW/fsharp/topic/9409/

28:

- `s^a◦`
- `^a'a 'A'T ◦`

Examples

Length

```
let inline getLength s = (^a: (member Length: _) s)
//usage:
getLength "Hello World" // or "Hello World" |> getLength
// returns 11
```

```
// Record
type Ribbon = {Length:int}
// Class
type Line(len:int) =
    member x.Length = len
type IHaveALength =
    abstract Length:int

let inline getLength s = (^a: (member Length: _) s)
let ribbon = {Length=1}
let line = Line(3)
let someLengthImplementer =
    { new IHaveALength with
        member __.Length = 5}
printfn "Our ribbon length is %i" (getLength ribbon)
printfn "Our Line length is %i" (getLength line)
printfn "Our Object expression length is %i" (getLength someLengthImplementer)
```

`GetLength`int

```
((^a : (static member GetLength : int) ()))
```

https://riptutorial.com/zh-TW/fsharp/topic/7228/

29:

Examples

`yield`◦ monadic`return`◦

```
> seq { yield 1; yield 2; yield 3 }
val it: seq<int> = seq [1; 2; 3]

> let homogenousTup2ToSeq (a, b) = seq { yield a; yield b }
> tup2Seq ("foo", "bar")
val homogenousTup2ToSeq: 'a * 'a -> seq<'a>
val it: seq<string> = seq ["foo"; "bar"]
```

`yield!` *yield bang* ◦ ◦ monads`bind`◦

```
> seq { yield 1; yield! [10;11;12]; yield 20 }
val it: seq<int> = seq [1; 10; 11; 12; 20]

// Creates a sequence containing the items of seq1 and seq2 in order
> let concat seq1 seq2 = seq { yield! seq1; yield! seq2 }
> concat ['a'..'c'] ['x'..'z']
val concat: seq<'a> -> seq<'a> -> seq<'a>
val it: seq<int> = seq ['a'; 'b'; 'c'; 'x'; 'y'; 'z']
```

◦ `Seq.take` n `Seq.iter` `Seq.toList` ◦ `yield!`◦

```
> let rec numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
> let naturals = numbersFrom 0
val numbersFrom: int -> seq<int>
val naturals: seq<int> = seq [0; 1; 2; ...]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let rec map f seq1 =
      if Seq.isEmpty seq1 then Seq.empty
      else seq { yield f (Seq.head seq1); yield! map f (Seq.tail seq1) }
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> 'b
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

`for`◦ “”◦ ◦

```
> let oneToTen = seq { for x in 1..10 -> x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
// Or, equivalently:
> let oneToTen = seq { for x in 1..10 do yield x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let map mapping seq1 = seq { for x in seq1 do yield mapping x }
```

```
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> seq<'b>
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]

// An infinite sequence of consecutive integers starting at 0
> let naturals =
      let numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
      numbersFrom 0
// Just like Seq.filter: returns a sequence consisting only of items from the input sequence
that satisfy the predicate
> let filter predicate seq1 = seq { for x in seq1 do if predicate x then yield x }
> let evenNaturals = naturals |> filter (fun x -> x % 2 = 0)
val naturals: seq<int> = seq [1; 2; 3; ...]
val filter: ('a -> bool) -> seq<'a> -> seq<'a>
val evenNaturals: seq<int> = seq [2; 4; 6; ...]

// Just like Seq.concat: concatenates a collection of sequences together
> let concat seqSeq = seq { for seq in seqSeq do yield! seq }
> concat [[1;2;3];[10;20;30]]
val concat: seq<#seq<'b>> -> seq<'b>
val it: seq<int> = seq [1; 2; 3; 10; 20; 30]
```

https://riptutorial.com/zh-TW/fsharp/topic/2785/

30:

Examples

```
// class declaration with a list of parameters for a primary constructor
type Car (model: string, plates: string, miles: int) =

    // secondary constructor (it must call primary constructor)
    new (model, plates) =
        let miles = 0
        new Car(model, plates, miles)

    // fields
    member this.model = model
    member this.plates = plates
    member this.miles = miles
```

```
let myCar = Car ("Honda Civic", "blue", 23)
// or more explicitly
let (myCar : Car) = new Car ("Honda Civic", "blue", 23)
```

https://riptutorial.com/zh-TW/fsharp/topic/3003/

31:

Examples

。 。

F

- F

```
// Functions
let a = fun c -> c

// Tuples
let b = (0, "Foo")

// Unit type
let c = ignore

// Records
type r = { Name : string; Age : int }
let d = { Name = "Foo"; Age = 10 }

// Discriminated Unions
type du = | Foo | Bar
let e = Bar

// List and seq
let f = [ 0..10 ]
let g = seq { 0..10 }

// Aliases
type MyAlias = string
```

- .NET
  - intboolstring...
  -
  -
  -

。

```
// Name is an alias for a string
type Name = string

// PhoneNumber is an alias for a string
type PhoneNumber = string
```

```
// Create a record type with the alias
type Contact = {
    Name : Name
    Phone : PhoneNumber }
```

```
// Create a record instance
// We can assign a string since Name and PhoneNumber are just aliases on string type
let c = {
    Name = "Foo"
    Phone = "00 000 000" }

printfn "%A" c

// Output
// {Name = "Foo";
// Phone = "00 000 000";}
```

。

```
let c = {
    Name = "Foo"
    Phone = "00 000 000" }
let d = {
    Name = c.Phone
    Phone = c.Name }

printfn "%A" d

// Output
// {Name = "00 000 000";
// Phone = "Foo";}
```

typeF

F#type◦

1.
2.
3.
4.
5.
6.

C#

```
// Equivalent C#:
//  using IntAliasType = System.Int32;
type IntAliasType = int // As in C# this doesn't create a new type, merely an alias

type DiscriminatedUnionType =
  | FirstCase
  | SecondCase  of int*string

  member x.SomeProperty = // We can add members to DU:s
    match x with
    | FirstCase         -> 0
    | SecondCase (i, _) -> i

type RecordType =
  {
    Id    : int
```

```
    Name  : string
  }
  static member New id name : RecordType = // We can add members to records
    { Id = id; Name = name } // { ... } syntax used to create records

// Equivalent C#:
//  interface InterfaceType
//  {
//    int     Id    { get; }
//    string  Name  { get; }
//    int Increment (int i);
//  }
type InterfaceType =
  interface // In order to create an interface type, can also use [<Interface>] attribute
    abstract member Id        : int
    abstract member Name      : string
    abstract member Increment : int -> int
  end

// Equivalent C#:
//  class ClassType : InterfaceType
//  {
//    static int increment (int i)
//    {
//      return i + 1;
//    }
//
//    public ClassType (int id, string name)
//    {
//      Id    = id    ;
//      Name  = name  ;
//    }
//
//    public int    Id    { get; private set; }
//    public string Name  { get; private set; }
//    public int   Increment (int i)
//    {
//      return increment (i);
//    }
//  }
type ClassType (id : int, name : string) = // a class type requires a primary constructor
  let increment i = i + 1 // Private helper functions

  interface InterfaceType with // Implements InterfaceType
    member x.Id           = id
    member x.Name         = name
    member x.Increment i  = increment i


// Equivalent C#:
//  class SubClassType : ClassType
//  {
//    public SubClassType (int id, string name) : base(id, name)
//    {
//    }
//  }
type SubClassType (id : int, name : string) =
  inherit ClassType (id, name) // Inherits ClassType

// Equivalent C#:
//  struct StructType
```

```
//  {
//    public StructType (int id)
//    {
//      Id = id;
//    }
//
//    public int Id { get; private set; }
//  }
type StructType (id : int) =
  struct  // In order create a struct type, can also use [<Struct>] attribute
    member x.Id = id
  end
```

◦ “Hindley-Milner”“HM”◦

-
-
-
-

◦ int“x”“x”int◦ “x”“x”◦

```
let inferInt x = x + 1
let inferFloat x = x + 1.0
let inferDecimal x = x + 1m     // m suffix means decimal
let inferSByte x = x + 1y       // y suffix means signed byte
let inferChar x = x + 'a'       // a char
let inferString x = x + "my string"
```

◦

```
let inferInt x = x + 1
let inferIndirectInt x = inferInt x       //deduce that x is an int

let inferFloat x = x + 1.0
let inferIndirectFloat x = inferFloat x   //deduce that x is a float

let x = 1
let y = x     //deduce that y is also an int
```

◦

```
let inferInt2 (x:int) = x                // Take int as parameter
let inferIndirectInt2 x = inferInt2 x    // Deduce from previous that x is int

let inferFloat2 (x:float) = x                // Take float as parameter
let inferIndirectFloat2 x = inferFloat2 x    // Deduce from previous that x is float
```

◦

```
let inferGeneric x = x
let inferIndirectGeneric x = inferGeneric x
let inferIndirectGenericAgain x = (inferIndirectGeneric x).ToString()
```

。 。 。

-
-
-

。

```
let square2 x = square x   // fails: square not defined
let square x = x * x
```

```
let square x = x * x
let square2 x = square x   // square already defined earlier
```

“”。 - 。

。 。 “rec”。

```
// the compiler does not know what "fib" means
let fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
   // error FS0039: The value or constructor 'fib' is not defined
```

“rec fib”

```
let rec fib n =              // LET REC rather than LET
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

。 Length。 。

```
let stringLength s = s.Length
  // error FS0072: Lookup on object of indeterminate type
  // based on information prior to this program point.
  // A type annotation may be needed ...
```

。

```
let stringLength (s:string) = s.Length
```

.NET。

concat。

```
let concat x = System.String.Concat(x)           //fails
let concat (x:string) = System.String.Concat(x)  //works
let concat x = System.String.Concat(x:string)    //works
```

。 StreamReader。

```
let makeStreamReader x = new System.IO.StreamReader(x)        //fails
let makeStreamReader x = new System.IO.StreamReader(path=x)   //works
```

https://riptutorial.com/zh-TW/fsharp/topic/3559/

32:

/。 /。

Examples

/

F“” 。 F - .net。

StringDuplicate 。

```
type System.String with
    member this.Duplicate times =
        Array.init times (fun _ -> this)
```

this - x。

。

```
// F#-style call
let result1 = "Hi there!".Duplicate 3

// C#-style call
let result2 = "Hi there!".Duplicate(3)

// Both result in three "Hi there!" strings in an array
```

C。

。 。

```
type System.String with
    member this.WordCount =
        ' ' // Space character
        |> Array.singleton
        |> fun xs -> this.Split(xs, StringSplitOptions.RemoveEmptyEntries)
        |> Array.length

let result = "This is an example".WordCount
// result is 4
```

F。

```
type System.String with
    static member EqualsCaseInsensitive (a, b) = String.Equals(a, b,
StringComparison.OrdinalIgnoreCase)
```

```
let x = String.EqualsCaseInsensitive("abc", "aBc")
```

```
// result is True
```

“”。 Fcurrying。

```
type System.String with
    static member AreEqual comparer a b = System.String.Equals(a, b, comparer)

let caseInsensitiveEquals = String.AreEqual StringComparison.OrdinalIgnoreCase

let result = caseInsensitiveEquals "abc" "aBc"
// result is True
```

。

```
namespace FSharp.Collections

module List =
    let pair item1 item2 = [ item1; item2 ]
```

List。

```
open FSharp.Collections

module Testing =
    let result = List.pair "a" "b"
    // result is a list containing "a" and "b"
```

https://riptutorial.com/zh-TW/fsharp/topic/2977/

33:

Examples

CSV

CSV

```
Id,Name
1,"Joel"
2,"Adam"
3,"Ryan"
4,"Matt"
```

```
#r "FSharp.Data.dll"
open FSharp.Data

type PeopleDB = CsvProvider<"people.csv">

let people = PeopleDB.Load("people.csv") // this can be a URL

let joel = people.Rows |> Seq.head

printfn "Name: %s, Id: %i" joel.Name joel.Id
```

WMI

WMIWMI。

WMIJSON

```
open FSharp.Management
open Newtonsoft.Json

// `Local` is based off of the WMI available at localhost.
type Local = WmiProvider<"localhost">

let data =
    [for d in Local.GetDataContext().Win32_DiskDrive -> d.Name, d.Size]

printfn "%A" (JsonConvert.SerializeObject data)
```

https://riptutorial.com/zh-TW/fsharp/topic/1631/

S. No		Contributors
1	F	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	.NET CoreF	Boggin, Joel Martinez
3	“”	4444, Abel, Jake Lishman, rmunn
4	1FsXamlFWPF	Bent Tranberg
5	FWPF	Funk
6	F	FuleSnabel, Ringil
7	F	FuleSnabel, Paul Westcott, Ringil, s952163
8	FWPFFsXaml	Bob McCrory, Goswin
9		asibahi, Julien Pires, rmunn, ronilk
10		FuleSnabel
11		chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
12		FuleSnabel
13		FireAlkazar, Julien Pires
14	CF	jdphenix, marklam, RamenChef
15		Foggy Finder, inzi, James McCalden, Julien Pires, s952163
16		inzi
17		asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
18		Jake Lishman
19		Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
20		asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj

21		Jean-Claude Colette, Zaid Ajaj
22		asibahi, goric, GregC, Vandroiy
23		Jean-Claude Colette, Julien Pires, Ringil
24		eirik, goric, Ringil
25		FuleSnabel
26		asibahi, chillitom, FuleSnabel
27		Honza Brestan
28		Maslow
29		Jwosty
30		asibahi, inzi, RamenChef, Tomasz Maczyński
31		Cedric Royer-Bertrand, FuleSnabel, Julien Pires
32		Jono Job
33		GregC, jdphenix, Joel Martinez