Troubleshooting F#

Why won't my code compile?

As the saying goes, “if it compiles, it’s correct”, but it can be extremely frustrating just trying to get the code to compile at all! So this page is devoted to helping you troubleshoot your F# code.

I will first present some general advice on troubleshooting and some of the most common errors that beginners make. After that, I will describe each of the common error messages in detail, and give examples of how they can occur and how to correct them.

(Jump to the error numbers)

General guidelines for troubleshooting

By far the most important thing you can do is to take the time and effort to understand exactly how F# works, especially the core concepts involving functions and the type system. So please read and reread the series “thinking functionally” and “understanding F# types”, play with the examples, and get comfortable with the ideas before you try to start doing serious coding. If you don’t understand how functions and types work, then the compiler errors will not make any sense.

If you are coming from an imperative language such as C#, you may have developed some bad habits by relying on the debugger to find and fix incorrect code. In F#, you will probably not get that far, because the compiler is so much stricter in many ways. And of course, there is no tool to “debug” the compiler and step through its processing. The best tool for debugging compiler errors is your brain, and F# forces you to use it!

Nevertheless, there are a number of extremely common errors that beginners make, and I will quickly go through them.

Don’t use parentheses when calling a function

In F#, whitespace is the standard separator for function parameters. You will rarely need to use parentheses, and in particular, do not use parentheses when calling a function.

```
let add x y = x + y
let result = add (1 2)  //wrong
    // error FS0003: This value is not a function and cannot be applied
let result = add 1 2    //correct
```

Don’t mix up tuples with multiple parameters

If it has a comma, it is a tuple. And a tuple is one object not two. So you will get errors about passing the wrong type of parameter, or too few parameters.

```
addTwoParams (1,2)  // trying to pass a single tuple rather than two args
   // error FS0001: This expression was expected to have type
   // int but here has type 'a * 'b
```

The compiler treats `(1,2)` as a generic tuple, which it attempts to pass to “`addTwoParams`”. Then it complains that the first parameter of `addTwoParams` is an int, and we’re trying to pass a tuple.

If you attempt to pass *two* arguments to a function expecting *one* tuple, you will get another obscure error.

```
addTuple 1 2   // trying to pass two args rather than one tuple
  // error FS0003: This value is not a function and cannot be applied
```

Watch out for too few or too many arguments

The F# compiler will not complain if you pass too few arguments to a function (in fact “partial application” is an important feature), but if you don’t understand what is going on, you will often get strange “type mismatch” errors later.

Similarly the error for having too many arguments is typically "This value is not a function" rather than a more straightforward error.

The "printf" family of functions is very strict in this respect. The argument count must be exact.

This is a very important topic – it is critical that you understand how partial application works. See the series "thinking functionally" for a more detailed discussion.

Use semicolons for list separators

In the few places where F# needs an explicit separator character, such as lists and records, the semicolon is used. Commas are never used. (Like a broken record, I will remind you that commas are for tuples).

```
let list1 = [1,2,3]    // wrong! This is a ONE-element list containing
                       // a three-element tuple
let list1 = [1;2;3]    // correct

type Customer = {Name:string, Address: string}  // wrong
type Customer = {Name:string; Address: string}  // correct
```

Don't use ! for not or != for not-equal

The exclamation point symbol is not the "NOT" operator. It is the deferencing operator for mutable references. If you use it by mistake, you will get the following error:

```
let y = true
let z = !y
// => error FS0001: This expression was expected to have
// type 'a ref but here has type bool
```

The correct construction is to use the "not" keyword. Think SQL or VB syntax rather than C syntax.

```
let y = true
let z = not y       //correct
```

And for "not equal", use "<>", again like SQL or VB.

```
let z = 1 <> 2      //correct
```

Don't use = for assignment

If you are using mutable values, the assignment operation is written "<-". If you use the equals symbol you might not even get an error, just an unexpected result.

```
let mutable x = 1
x = x + 1          // returns false. x is not equal to x+1
x <- x + 1         // assigns x+1 to x
```

Watch out for hidden tab characters

The indenting rules are very straightforward, and it is easy to get the hang of them. But you are not allowed to use tabs, only spaces.

```
let add x y =
{tab}x + y
// => error FS1161: TABs are not allowed in F# code
```

Be sure to set your editor to convert tabs to spaces. And watch out if you are pasting code in from elsewhere. If you do run into persistent problems with a bit of code, try removing the whitespace and re-adding it.

Don't mistake simple values for function values

If you are trying to create a function pointer or delegate, watch out that you don't accidentally create a simple value that has already been evaluated.

If you want a parameterless function that you can reuse, you will need to explicitly pass a unit parameter, or define it as a lambda.

```
let reader = new System.IO.StringReader("hello")
let nextLineFn   =  reader.ReadLine()  //wrong
let nextLineFn() =  reader.ReadLine()  //correct
let nextLineFn   =  fun() -> reader.ReadLine()  //correct

let r = new System.Random()
let randomFn   =  r.Next()  //wrong
let randomFn() =  r.Next()  //correct
let randomFn   =  fun () -> r.Next()  //correct
```

See the series "thinking functionally" for more discussion of parameterless functions.

Tips for troubleshooting "not enough information" errors

The F# compiler is currently a one-pass left-to-right compiler, and so type information later in the program is unavailable to the compiler if it hasn't been parsed yet.

A number of errors can be caused by this, such as "FS0072: Lookup on object of indeterminate type" and "FS0041: A unique overload for could not be determined". The suggested fixes for each of these specific cases are described below, but there are some general principles that can help if the compiler is complaining about missing types or not enough information. These guidelines are:

- Define things before they are used (this includes making sure the files are compiled in the right order)
- Put the things that have "known types" earlier than things that have "unknown types". In particular, you might be able reorder pipes and similar chained functions so that the typed objects come first.
- Annotate as needed. One common trick is to add annotations until everything works, and then take them away one by one until you have the minimum needed.

Do try to avoid annotating if possible. Not only is it not aesthetically pleasing, but it makes the code more brittle. It is a lot easier to change types if there are no explicit dependencies on them.

F# compiler errors

A listing of common errors, ordered by error number

Here is a list of the major errors that seem to me worth documenting. I have not documented any errors that are self explanatory, only those that seem obscure to beginners.

I will continue to add to the list in the future, and I welcome any suggestions for additions.

- FS0001: The type 'X' does not match the type 'Y'
- FS0003: This value is not a function and cannot be applied
- FS0008: This runtime coercion or type test involves an indeterminate type
- FS0010: Unexpected identifier in binding
- FS0010: Incomplete structured construct
- FS0013: The static coercion from type X to Y involves an indeterminate type
- FS0020: This expression should have type 'unit'

FS0001: The type ‘X’ does not match the type ‘Y’

This is probably the most common error you will run into. It can manifest itself in a wide variety of contexts, so I have grouped the most common problems together with examples and fixes. Do pay attention to the error message, as it is normally quite explicit about what the problem is.

Error message	**Possible causes**
The type 'float' does not match the type 'int'	A. Can't mix floats and ints
The type 'int' does not support any operators named 'DivideByInt'	A. Can't mix floats and ints.
The type 'X' is not compatible with any of the types	B. Using the wrong numeric type.
This type (function type) does not match the type (simple type). Note: function types have a arrow in them, like `'a -> 'b`.	C. Passing too many arguments to a function.
This expression was expected to have (function type) but here has (simple type)	C. Passing too many arguments to a function.
This expression was expected to have (N part function) but here has (N-1 part function)	C. Passing too many arguments to a function.
This expression was expected to have (simple type) but here has (function type)	D. Passing too few arguments to a function.
This expression was expected to have (type) but here has (other type)	E. Straightforward type mismatch. F. Inconsistent returns in branches or matches. G. Watch out for type inference effects buried in a function.
Type mismatch. Expecting a (simple type) but given a (tuple type). Note: tuple types have a star in them, like `'a * 'b`.	H. Have you used a comma instead of space or semicolon?
Type mismatch. Expecting a (tuple type) but given a (different tuple type).	I. Tuples must be the same type to be compared.
This expression was expected to have type 'a ref but here has type X	J. Don't use ! as the "not" operator.

Error message	**Possible causes**
The type (type) does not match the type (other type)	K. Operator precedence (especially functions and pipes).
This expression was expected to have type (monadic type) but here has type 'b * 'c	L. let! error in computation expressions.

A. Can’t mix ints and floats

Unlike C# and most imperative languages, ints and floats cannot be mixed in expressions. You will get a type error if you attempt this:

```
1 + 2.0  //wrong
   // => error FS0001: The type 'float' does not match the type 'int'
```

The fix is to cast the int into a `float` first:

```
float 1 + 2.0  //correct
```

This issue can also manifest itself in library functions and other places. For example, you cannot do “`average`” on a list of ints.

```
[1..10] |> List.average   // wrong
   // => error FS0001: The type 'int' does not support any
   // operators named 'DivideByInt'
```

You must cast each int to a float first, as shown below:

```
[1..10] |> List.map float |> List.average  //correct
[1..10] |> List.averageBy float  //correct (uses averageBy)
```

B. Using the wrong numeric type

You will get a “not compatible” error when a numeric cast failed.

```
printfn "hello %i" 1.0  // should be a int not a float
  // error FS0001: The type 'float' is not compatible
  // with any of the types byte,int16,int32...
```

One possible fix is to cast it if appropriate.

```
printfn "hello %i" (int 1.0)
```

C. Passing too many arguments to a function

```
let add x y = x + y
let result = add 1 2 3
// ==> error FS0001: The type ''a -> 'b' does not match the type 'int'
```

The clue is in the error.

The fix is to remove one of the arguments!

Similar errors are caused by passing too many arguments to `printf`.

```
printfn "hello" 42
// ==> error FS0001: This expression was expected to have type 'a -> 'b
// but here has type unit

printfn "hello %i" 42 43
// ==> Error FS0001: Type mismatch. Expecting a 'a -> 'b -> 'c
// but given a 'a -> unit

printfn "hello %i %i" 42 43 44
// ==> Error FS0001: Type mismatch. Expecting a 'a -> 'b -> 'c -> 'd
// but given a 'a -> 'b -> unit
```

D. Passing too few arguments to a function

If you do not pass enough arguments to a function, you will get a partial application. When you later use it, you get an error because it is not a simple type.

```
let reader = new System.IO.StringReader("hello");

let line = reader.ReadLine        //wrong but compiler doesn't complain
printfn "The line is %s" line     //compiler error here!
// ==> error FS0001: This expression was expected to have type string
// but here has type unit -> string
```

This is particularly common for some .NET library functions that expect a unit parameter, such as `ReadLine` above.

The fix is to pass the correct number of parameters. Check the type of the result value to make sure that it is indeed a simple type. In the `ReadLine` case, the fix is to pass a `()` argument.

```
let line = reader.ReadLine()      //correct
printfn "The line is %s" line     //no compiler error
```

E. Straightforward type mismatch

The simplest case is that you have the wrong type, or you are using the wrong type in a print format string.

```
printfn "hello %s" 1.0
// => error FS0001: This expression was expected to have type string
// but here has type float
```

F. Inconsistent return types in branches or matches

A common mistake is that if you have a branch or match expression, then every branch MUST return the same type. If not, you will get a type error.

```
let f x =
  if x > 1 then "hello"
  else 42
// => error FS0001: This expression was expected to have type string
// but here has type int
let g x =
  match x with
  | 1 -> "hello"
  | _ -> 42
// error FS0001: This expression was expected to have type
// string but here has type int
```

Obviously, the straightforward fix is to make each branch return the same type.

```
let f x =
  if x > 1 then "hello"
  else "42"

let g x =
  match x with
  | 1 -> "hello"
  | _ -> "42"
```

Remember that if an “else” branch is missing, it is assumed to return unit, so the “true” branch must also return unit.

```
let f x =
  if x > 1 then "hello"
// error FS0001: This expression was expected to have type
// unit but here has type string
```

If both branches cannot return the same type, you may need to create a new union type that can contain both types.

```
type StringOrInt = | S of string | I of int  // new union type
let f x =
  if x > 1 then S "hello"
  else I 42
```

G. Watch out for type inference effects buried in a function

A function may cause an unexpected type inference that ripples around your code. For example, in the following, the innocent print format string accidentally causes `doSomething` to expect a string.

```
let doSomething x =
   // do something
   printfn "x is %s" x
   // do something more

doSomething 1
// => error FS0001: This expression was expected to have type string
// but here has type int
```

The fix is to check the function signatures and drill down until you find the guilty party. Also, use the most generic types possible, and avoid type annotations if possible.

H. Have you used a comma instead of space or semicolon?

If you are new to F#, you might accidentally use a comma instead of spaces to separate function arguments:

```
// define a two parameter function
```

```
let add x y = x + 1

add(x,y)   // FS0001: This expression was expected to have
           // type int but here has type 'a * 'b
```

The fix is: don't use a comma!

```
add x y    // OK
```

One area where commas *are* used is when calling .NET library functions. These all take tuples as arguments, so the comma form is correct. In fact, these calls look just the same as they would from C#:

```
// correct
System.String.Compare("a","b")

// incorrect
System.String.Compare "a" "b"
```

I. Tuples must be the same type to be compared or pattern matched

Tuples with different types cannot be compared. Trying to compare a tuple of type `int * int`, with a tuple of type `int * string` results in an error:

```
let  t1 = (0, 1)
let  t2 = (0, "hello")
t1 = t2
// => error FS0001: Type mismatch. Expecting a int * int
// but given a int * string
// The type 'int' does not match the type 'string'
```

And the length must be the same:

```
let  t1 = (0, 1)
let  t2 = (0, 1, "hello")
t1 = t2
// => error FS0001: Type mismatch. Expecting a int * int
// but given a int * int * string
// The tuples have differing lengths of 2 and 3
```

You can get the same issue when pattern matching tuples during binding:

```
let x,y = 1,2,3
// => error FS0001: Type mismatch. Expecting a 'a * 'b
// but given a 'a * 'b * 'c
// The tuples have differing lengths of 2 and 3

let f (x,y) = x + y
let z = (1,"hello")
let result = f z
// => error FS0001: Type mismatch. Expecting a int * int
// but given a int * string
// The type 'int' does not match the type 'string'
```

J. Don’t use ! as the “not” operator

If you use ! as a “not” operator, you will get a type error mentioning the word “ref”.

```
let y = true
let z = !y     //wrong
// => error FS0001: This expression was expected to have
// type 'a ref but here has type bool
```

The fix is to use the “not” keyword instead.

```
let y = true
let z = not y   //correct
```

K. Operator precedence (especially functions and pipes)

If you mix up operator precedence, you may get type errors. Generally, function application is highest precedence compared to other operators, so you get an error in the case below:

```
String.length "hello" + "world"
   // => error FS0001: The type 'string' does not match the type 'int'

// what is really happening
(String.length "hello") + "world"
```

The fix is to use parentheses.

```
String.length ("hello" + "world")  // corrected
```

Conversely, the pipe operator is low precedence compared to other operators.

```
let result = 42 + [1..10] |> List.sum
 // => => error FS0001: The type ''a list' does not match the type 'int'

// what is really happening
let result = (42 + [1..10]) |> List.sum
```

Again, the fix is to use parentheses.

```
let result = 42 + ([1..10] |> List.sum)
```

L. let! error in computation expressions (monads)

Here is a simple computation expression:

```
type Wrapper<'a> = Wrapped of 'a

type wrapBuilder() =
    member this.Bind (wrapper:Wrapper<'a>) (func:'a->Wrapper<'b>) =
        match wrapper with
        | Wrapped(innerThing) -> func innerThing

    member this.Return innerThing =
        Wrapped(innerThing)

let wrap = new wrapBuilder()
```

However, if you try to use it, you get an error.

```
wrap {
    let! x1 = Wrapped(1)   // <== error here
    let! y1 = Wrapped(2)
    let z1 = x + y
    return z
    }
// error FS0001: This expression was expected to have type Wrapper<'a>
// but here has type 'b * 'c
```

The reason is that "`Bind`" expects a tuple `(wrapper,func)`, not two parameters. (Check the signature for bind in the F# documentation).

The fix is to change the bind function to accept a tuple as its (single) parameter.

```
type wrapBuilder() =
    member this.Bind (wrapper:Wrapper<'a>, func:'a->Wrapper<'b>) =
        match wrapper with
        | Wrapped(innerThing) -> func innerThing
```

FS0003: This value is not a function and cannot be applied

This error typically occurs when passing too many arguments to a function.

```
let add1 x = x + 1
let x = add1 2 3
// ==> error FS0003: This value is not a function and cannot be applied
```

It can also occur when you do operator overloading, but the operators cannot be used as prefix or infix.

```
let (!!) x y = x + y
(!!) 1 2              // ok
1 !! 2                // failed !! cannot be used as an infix operator
// error FS0003: This value is not a function and cannot be applied
```

FS0008: This runtime coercion or type test involves an indeterminate type

You will often see this when attempting to use "`:?`" operator to match on a type.

```
let detectType v =
    match v with
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"
// error FS0008: This runtime coercion or type test from type 'a to int
// involves an indeterminate type based on information prior to this program
point.
```

```
// Runtime type tests are not allowed on some types. Further type annotations are
needed.
```

The message tells you the problem: “runtime type tests are not allowed on some types”.

The answer is to “box” the value which forces it into a reference type, and then you can type check it:

```
let detectTypeBoxed v =
    match box v with      // used "box v"
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"

//test
detectTypeBoxed 1
detectTypeBoxed 3.14
```

FS0010: Unexpected identifier in binding

Typically caused by breaking the “offside” rule for aligning expressions in a block.

```
//3456789
let f =
  let x=1     // offside line is at column 3
   x+1        // oops! don't start at column 4
              // error FS0010: Unexpected identifier in binding
```

The fix is to align the code correctly!

See also FS0588: Block following this ‘let’ is unfinished for another issue caused by alignment.

FS0010: Incomplete structured construct

Often occurs if you are missing parentheses from a class constructor:

```
type Something() =
   let field = ()

let x1 = new Something     // Error FS0010
let x2 = new Something()   // OK!
```

Can also occur if you forgot to put parentheses around an operator:

```
// define new operator
let (|+) a = -a

|+ 1    // error FS0010:
        // Unexpected infix operator

(|+) 1  // with parentheses -- OK!
```

Can also occur if you are missing one side of an infix operator:

```
|| true  // error FS0010: Unexpected symbol '||'
false || true  // OK
```

Can also occur if you attempt to send a namespace definition to F# interactive. The interactive console does not allow namespaces.

```
namespace Customer  // FS0010: Incomplete structured construct

// declare a type
type Person= {First:string; Last:string}
```

FS0013: The static coercion from type X to Y involves an indeterminate type

This is generally caused by implic

FS0020: This expression should have type ‘unit’

This error is commonly found in two situations:

- Expressions that are not the last expression in the block
- Using wrong assignment operator

FS0020 with expressions that are not the last expression in the block

Only the last expression in a block can return a value. All others must return unit. So this typically occurs when you have a function in a place that is not the last function.

```
let something =
  2+2               // => FS0020: This expression should have type 'unit'
  "hello"
```

The easy fix is use `ignore`. But ask yourself why you are using a function and then throwing away the answer – it might be a bug.

```
let something =
  2+2 |> ignore     // ok
  "hello"
```

This also occurs if you think you writing C# and you accidentally use semicolons to separate expressions:

```
// wrong
let result = 2+2; "hello";

// fixed
let result = 2+2 |> ignore; "hello";
```

FS0020 with assignment

Another variant of this error occurs when assigning to a property.

```
This expression should have type 'unit', but has type 'Y'.
```

With this error, chances are you have confused the assignment operator “`<-`” for mutable values, with the equality comparison operator “=”.

```
// '=' versus '<-'
let add() =
    let mutable x = 1
    x = x + 1          // warning FS0020
    printfn "%d" x
```

The fix is to use the proper assignment operator.

```
// fixed
let add() =
    let mutable x = 1
    x <- x + 1
    printfn "%d" x
```

FS0030: Value restriction

This is related to F#'s automatic generalization to generic types whenever possible.

For example, given :

```
let id x = x
let compose f g x = g (f x)
let opt = None
```

F#'s type inference will cleverly figure out the generic types.

```
val id : 'a -> 'a
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
val opt : 'a option
```

However in some cases, the F# compiler feels that the code is ambiguous, and, even though it looks like it is guessing the type correctly, it needs you to be more specific:

```
let idMap = List.map id             // error FS0030
let blankConcat = String.concat ""  // error FS0030
```

Almost always this will be caused by trying to define a partially applied function, and almost always, the easiest fix is to explicitly add the missing parameter:

```
let idMap list = List.map id list             // OK
let blankConcat list = String.concat "" list  // OK
```

For more details see the MSDN article on "automatic generalization".

FS0035: This construct is deprecated

F# syntax has been cleaned up over the last few years, so if you are using examples from an older F# book or webpage, you may run into this. See the MSDN documentation for the correct syntax.

```
let x = 10
let rnd1 = System.Random x          // Good
let rnd2 = new System.Random(x)     // Good
let rnd3 = new System.Random x      // error FS0035
```

FS0039: The field, constructor or member X is not defined

This error is commonly found in four situations:

- The obvious case where something really isn't defined! And make sure that you don't have a typo or case mismatch either.
- Interfaces
- Recursion
- Extension methods

FS0039 with interfaces

In F# all interfaces are “explicit” implementations rather than “implicit”. (Read the C# documentation on “explicit interface implementation” for an explanation of the difference).

The key point is that when a interface member is explicitly implemented, it cannot be accessed through a normal class instance, but only through an instance of the interface, so you have to cast to the interface type by using the `:>` operator.

Here’s an example of a class that implements an interface:

```
type MyResource() =
   interface System.IDisposable with
       member this.Dispose() = printfn "disposed"
```

This doesn’t work:

```
let x = new MyResource()
x.Dispose()  // error FS0039: The field, constructor
             // or member 'Dispose' is not defined
```

The fix is to cast the object to the interface, as below:

```
// fixed by casting to System.IDisposable
(x :> System.IDisposable).Dispose()   // OK

let y =  new MyResource() :> System.IDisposable
y.Dispose()   // OK
```

FS0039 with recursion

Here’s a standard Fibonacci implementation:

```
let fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Unfortunately, this will not compile:

```
Error FS0039: The value or constructor 'fib' is not defined
```

The reason is that when the compiler sees ‘fib’ in the body, it doesn’t know about the function because it hasn’t finished compiling it yet!

The fix is to use the “`rec`” keyword.

```
let rec fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Note that this only applies to “`let`” functions. Member functions do not need this, because the scope rules are slightly different.

```
type FibHelper() =
    member this.fib i =
       match i with
       | 1 -> 1
       | 2 -> 1
       | n -> fib(n-1) + fib(n-2)
```

FS0039 with extension methods

If you have defined an extension method, you won't be able to use it unless the module is in scope.

Here's a simple extension to demonstrate:

```
module IntExtensions =
    type System.Int32 with
        member this.IsEven = this % 2 = 0
```

If you try to use it the extension, you get the FS0039 error:

```
let i = 2
let result = i.IsEven
    // FS0039: The field, constructor or
    // member 'IsEven' is not defined
```

The fix is just to open the `IntExtensions` module.

```
open IntExtensions // bring module into scope
let i = 2
let result = i.IsEven  // fixed!
```

FS0041: A unique overload for could not be determined

This can be caused when calling a .NET library function that has multiple overloads:

```
let streamReader filename = new System.IO.StreamReader(filename) // FS0041
```

There a number of ways to fix this. One way is to use an explicit type annotation:

```
let streamReader filename = new System.IO.StreamReader(filename:string) // OK
```

You can sometimes use a named parameter to avoid the type annotation:

```
let streamReader filename = new System.IO.StreamReader(path=filename) // OK
```

Or you can try to create intermediate objects that help the type inference, again without needing type annotations:

```
let streamReader filename =
    let fileInfo = System.IO.FileInfo(filename)
    new System.IO.StreamReader(fileInfo.FullName) // OK
```

FS0049: Uppercase variable identifiers should not generally be used in patterns

When pattern matching, be aware of a subtle difference between the pure F# union types which consist of a tag only, and a .NET Enum type.

Pure F# union type:

```
type ColorUnion = Red | Yellow
let redUnion = Red
```

```
match redUnion with
| Red -> printfn "red"      // no problem
| _ -> printfn "something else"
```

But with .NET enums you must fully qualify them:

```
type ColorEnum = Green=0 | Blue=1      // enum
let blueEnum = ColorEnum.Blue

match blueEnum with
| Blue -> printfn "blue"      // warning FS0049
| _ -> printfn "something else"
```

The fixed version:

```
match blueEnum with
| ColorEnum.Blue -> printfn "blue"
| _ -> printfn "something else"
```

FS0072: Lookup on object of indeterminate type

This occurs when "dotting into" an object whose type is unknown.

Consider the following example:

```
let stringLength x = x.Length // Error FS0072
```

The compiler does not know what type "x" is, and therefore does not know if "`Length`" is a valid method.

There a number of ways to fix this. The crudest way is to provide an explicit type annotation:

```
let stringLength (x:string) = x.Length  // OK
```

In some cases though, judicious rearrangement of the code can help. For example, the example below looks like it should work. It's obvious to a human that the `List.map` function is being applied to a list of strings, so why does `x.Length` cause an error?

```
List.map (fun x -> x.Length) ["hello"; "world"] // Error FS0072
```

The reason is that the F# compiler is currently a one-pass compiler, and so type information present later in the program cannot be used if it hasn't been parsed yet.

Yes, you can always explicitly annotate:

```
List.map (fun x:string -> x.Length) ["hello"; "world"] // OK
```

But another, more elegant way that will often fix the problem is to rearrange things so the known types come first, and the compiler can digest them before it moves to the next clause.

```
["hello"; "world"] |> List.map (fun x -> x.Length)   // OK
```

It's good practice to avoid explicit type annotations, so this approach is best, if it is feasible.

FS0588: Block following this 'let' is unfinished

Caused by outdenting an expression in a block, and thus breaking the "offside rule".

```
//3456789
let f =
  let x=1    // offside line is at column 3
```

```
x+1         // offside! You are ahead of the ball!
            // error FS0588: Block following this
            // 'let' is unfinished
```

The fix is to align the code correctly.

See also FS0010: Unexpected identifier in binding for another issue caused by alignment.
