Learning to program with F#

Jon Sporring

July 15, 2016

Contents

Part I

F# basics

Chapter 4

Quick-start guide

Programming is the art of solving problems by writing a program to be executed by a computer. For example, to solve the following problem,

> What is the sum of 357 and 864?

we have written the following program in F#,

```
let a = 357
let b = 864
let c = a + b
printfn "%A" c
```

```
1221
```

Listing 4.1: quickStartSum.fsx - A script to add 2 numbers and print the result to the console.

In box the above, we see our program was saved as a script in a file called `quickStartSum.fsx`, and in the console we executed the program by typing the command `fsharpi quickStartSum.fsx`. The result is then printed in the console to be `1221`.

To solve the program we made program consisting of several lines, where each line was a *statement*. The first statement `let a = 357` used the *`let` keyword* to *bind* the value 357 to the name `a`. Likewise, we bound the value 864 to the name `b`, but to the name `c` we bound the result of evaluating the *expression* `a + b`. That is, first the value `a + b` was calculated by substituting the names of `a` and `b` with their values to give the expression, `357 + 864`, then this expression was evaluated by adding the values to give, `1221`, and this value was finally bound to the name `c`. The last line printed the value of `c` to the console followed by a LF (line feed, see Appendix B.1) with the `printfn` function. Here `printfn` is a function of 2 arguments: `"%A"` and `c`. Notice, that in contrast to many other languages, F# does not use parentheses to frame the list of arguments, nor does it use commas to separate them. In general, the `printfn` function always has 1 or more arguments, and the first is a *format string*. A *string* is a sequence of characters starting and ending with double quotation marks. E.g., `let s = "this is a string of characters"` binds the string `"this is..."` to the name `s`. For the `printfn` function, the format string may be any string, but if it contains format character sequences, such as `%A`, then the values following the format string are substituted. The format string must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches any type.

Types are a central concept in F#. In the script 4.1 we bound values of types `int` and `string` to names. The values were not *declared* to have these types, instead the types were *inferred* by F#. Had we typed these statements line by line in an interactive session, then we would have seen the inferred types:

- statement
- `let`
- keyword
- binding
- expression
- format string
- string
- type
- type declaration
- type inference

```
> let a = 357;;

val a : int = 357

> let b = 864;;

val b : int = 864

> let c = a + b;;

val c : int = 1221

> printfn "%A" c;;
1221
val it : unit = ()
```

Listing 4.2: fsharpi

The an interactive session displays the type using the *val* keyword. Since the value is also responded, then the last `printfn` statement is superfluous. However, it is ill adviced to design programs to be run in an interactive session, since the scripts needs to be manually copied every time it is to be run, and since the starting state may be unclear.

· `val`

Advice!

Were we to solve a slightly different problem,

> What is the sum of 357.6 and 863.4?

then we would have to use floating point arithmatic instead of integers, and the program would look like,

```
let a = 357.6
let b = 863.4
let c = a + b
printfn "%A" c
```

```
1221.0
```

Listing 4.3: quickStartSumFloat.fsx - Floating point types and arithmatic.

On the surface, this could appear as an almost negligible change, but the set of integers and the set of real numbers (floats) require quite different representations, in order to be effective on a computer, and as a consequence, the implementation of their operations such as addition are very different. Thus, although the response is an integer, it has type `float`, which is indicated by `1221.0` which is not the same as `1221`. F# is very picky about types, and generally does not allow types to be mixed. E.g., in an interactive session,

```
> let a = 357;;

val a : int = 357

> let b = 863.4;;

val b : float = 863.4

> let c = a + b;;
```

```
  let c = a + b;;
  ------------^

/Users/sporring/Desktop/fsharpNotes/stdin(3,13): error FS0001: The
   type 'float' does not match the type 'int'
```
Listing 4.4: fsharpi

we see that binding a name to a number without a decimal point is inferred to be integer, while when binding to a number with a decimal point, then the type is inferred to be a float, and when trying to add values of integer and floating point, then we get an error.
F# is a functional first programming language, and one implication is that names are constant and cannot be changed. If attempted, then F# will return an error as, e.g.,

```
let a = 357
let a = 864
```

```

/Users/sporring/repositories/fsharpNotes/quickStartRebindError.fsx
   (2,5): error FS0037: Duplicate definition of value 'a'
```
Listing 4.5: quickStartRebindError.fsx - A name cannot be rebound.

However, if the same was performed in an interactive session,

```
> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864
```
Listing 4.6: fsharpi

then apparently rebinding is legal. The difference is that the *;; token* defines a new nested *scope*. A token is a letter or a word, which the F# considers as an atomic unit. A scope is an area in a program, where a binding is valid. Scopes can be *nested*, and in F# a binding may reuse names in a nested scope, in which case the previous value is *overshadowed*. I.e., attempting the same without `;;` between the two let statements results in an error, e.g.,

- ;;
- token
- scope
- nested scope
- overshadow

```
> let a = 357
- let a = 864;;

  let a = 864;;
  ----^

/Users/sporring/Desktop/fsharpNotes/stdin(2,5): error FS0037:
   Duplicate definition of value 'a'
```
Listing 4.7: fsharpi

Scopes can be visualized as nested squares as shown in Figure 4.1.
In F# *functions* are also values, and defining a function `sum` as part of the solution to the above program gives,

- function

```
let a = 357
let a = 864;;
```
(a) Illegal

```
let a = 357;;
  let a = 864;;
```
(b) Legal

Figure 4.1: Binding of the the same name in the same scope is illegal in F# 2, but legal in a different scopes. In (a) the two bindings are in the same scope, which is illegal, while in (b) the bindings are in separate scopes by the extra `;;` token, which is legal.

```
let sum x y = x + y
let c = sum 357 864
printfn "%A" c
```

```
1221
```

Listing 4.8: quickStartSumFct.fsx - A script to add 2 numbers using a user defined function.

Entering the function into an interactive session will illustrate the inferred type, the function `sum` has: `val sum : x:int * y:int -> int`, by which is meant that `sum` is a mapping from the set product of integers with integers into integers. Type inference in F# may cause problems, since the type of a function is inferred in the context, in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#'s favorite type, which will give an error, if it in a nested scope is to be used for floats,

```
> let sum x y = x + y;;

val sum : x:int -> y:int -> int

> let c = sum 357.6 863.4;;

  let c = sum 357.6 863.4;;
  ------------^^^^^

/Users/sporring/Desktop/fsharpNotes/stdin(2,13): error FS0001: This
    expression was expected to have type
    int
but here has type
    float
```

Listing 4.9: fsharpi

A remedy is to either define the function in the same scope as its use,

```
> let sum x y = x + y
- let c = sum 357.6 863.4;;

val sum : x:float -> y:float -> float
val c : float = 1221.0
```

Listing 4.10: fsharpi

In this chapter, we have scratched the surface of learning how to program by concentrating on a number of key programming concepts and how they are expressed in the F# language. In the following chapters, we will expand the description of F# with features used in all programming approaches.

Chapter 5

Numbers, Characters, and Strings

All programs rely on processing of data, and an essential property of data is its *type*. F# contains a number of built-in types, and it designed such that it is easy to define new types. The simplest types are called *primitive types*, and a table of some of the most commonly used primitive types are shown in Table 5.1. A *literal* is a fixed value such as "3", and F# supports *literal types* which are indicated by a suffix in most cases and as shown in the table.
A name is bound to a value by the syntax,

· type
· primitive types
· literal
· literal type

```
"let" name [ ":" type ] "="  expr
```

That is, the *let* keyword indicates that the following is a binding of a name with an expression, and that the type may be specified with the *:* token. The simplest example of an expression is a *literal*, i.e., a constant such as the number 3 or a function. Functions will be discussed in detail in Chapter **??**.
Examples of let statements with *literals*

· let
· :
· literal
· literals

```
> let a = 3
- let b = 4u
- let c = 5.6
- let d = 7.9f
- let e = 'A'
- let f = 'B'B
- let g = "ABC"
- let h = ();;

val a : int = 3
val b : uint32 = 4u
val c : float = 5.6
val d : float32 = 7.9000001f
val e : char = 'A'
val f : byte = 66uy
val g : string = "ABC"
val h : unit = ()
```

Listing 5.1: fsharpi

Here `a`, `b`, ..., `h` are names that we have chosen, and which by the binding operation are made equivalent to the corresponding values. Note that we did not specity the type of the name, and that F# interpreted the type from the literal form of the right-hand-side. When specifying the type, then the type and the literal form must match, i.e., mixing types and literals gives an error,

Metatype	Type name	Suffix	Literal	Description
Boolean	bool	none	true	Boolean values true or false
Integer	**int**	none or l	3	Integer values from -2,147,483,648 to 2,147,483,647
	byte	uy	3uy	Integer values from 0 to 255
	sbyte	y 7	3y	Integer values from -128 to 12
	int16	s	3s	Integer values from -32768 to 32767
	uint16	us 5	3us	Integer values from 0 to 6553
	int32	none or l	3	Synonymous with int
	uint32	u or ul	3u	Integer values from 0 to 4,294,967,295
	int64	L	3L	Integer values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	uint64	UL	3UL	Integer values from 0 to 18,446,744,073,709,551,615
	bigint	I s	3I	Integer not limited to 64 bit
	nativeint	n r	3n	A native pointer as a signed intege
	unativeint	un	3un	A native pointer as an unsigned integer
Real	**float**	none	3.0	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	double	none	3.0	Synonumous with float
	single	F or f	3.0f	A 32-bit floating point type
	float32	F or f	3.0f	Synonymous with single
	decimal	M or m	3.0m	A floating point data type that has at least 28 significant digits
Character	**char**	none	'c'	Unicode character
	byte	B	'c'B	ASCII character
	string	none	"abc"	Unicode sequence of characters
	byte[]	B	"abc"B	Unicode sequence of characters
	string or byte[]	@	@"\n"	Verbatim string
None	**unit**	none	()	No value denoted

Table 5.1: List of primitive types and the corresponding literal. The most commonly used types are highlighted in bold. Note that string verbatim uses a prefix instead of suffix notation. For at description of floating point numbers see Appendix A.2 and for ASCII and Unicode charaters see Appendix B.

```
> let a : float = 3;;

  let a : float = 3;;
  ----------------^

/Users/sporring/repositories/fsharpNotes/stdin(50,17): error FS0001
    : This expression was expected to have type
     float
but here has type
     int
```
Listing 5.2: fsharpi

since the left-hand-side is a name of type float while the right-hand-side is a literal of type integer. Many primitive types are compatible and may changed to each other by *type casting*. E.g., · type casting

```
> let a = float 3;;

val a : float = 3.0
```
Listing 5.3: fsharpi

where the left-hand-side is inferred to by of type float, since the integer number `3` is casted to float resulting in a similar floating point value, in this case the float point number `3.0`. As a particular note, the boolean values are often treated as the integer values 0 and 1, however casting can only be performed with built-in functions, e.g.,

```
> let a = System.Convert.ToBoolean 1
- let b = System.Convert.ToBoolean 0
- let c = System.Convert.ToInt32 true
- let d = System.Convert.ToInt32 false;;

val a : bool = true
val b : bool = false
val c : int = 1
val d : int = 0
```
Listing 5.4: fsharpi

Here `System.Convert.ToBoolean` is the name of a function `ToBoolean`, which is a *member* of the *class* `Convert` that is included in the *namespace* `System`. Namespaces, classes and members are all part of Structure programming to be discussed in Part IV. For more on functions see Section **??** Typecasting is often a destructive operation, e.g., typecasting a float to int removes the part after the decimal point without rounding, · member · class · namespace

```
let a = 357.6
let b = int a
printfn "%A -> %A" a b
```

```
357.6 -> 357
```
Listing 5.5: quickStartDownCast.fsx - Fractional part is removed by downcasting.

The typecasting in the example is called *downcasting*, since the floating time is casted to a lesser type, in the sense that integers is a subset of floats. The opposite is called *upcasting* is often non-destructive, as the previous example showed, where an integer was casted to a float while retaining its value. · downcasting · upcasting

Operator	Associativity	Example	Description
`f x`	Left	`f 3`	Function evaluation
`+op, -op`	Left	`-3`	Unary operator
`&&`	Left	`true && true`	Boolean and
`||`	Left	`true || true`	Boolean or
`op**op,`	Right	`3.0 ** 2.0`	Exponent
`op*op, op/op, op%op,`	Left	`3.0 / 2.0`	Multiplication, division and remainder
`op+op, op-op`	Left	`3.0 + 2.0`	Addition and subtraction binary operators

Table 5.2: Some common operators, their precedence, and their associativity.

Types matter, since the operations that can be performed on integers are quite different from those that can be performed on characters and strings. Hence, in the following example,

```
> let a = 3
- let b = 3.0
- let c = '3'
- let d = "3";;

val a : int = 3
val b : float = 3.0
val c : char = '3'
val d : string = "3"
```

Listing 5.6: fsharpi

the variables `a`, `b`, `c`, and `d` all represent the number 3, their types are different, and hence they are quite different values.
To perform calculations values, names, and other programming constructs are often combined into expressions, e.g., in

```
> let a = 3 + 4 * 5;;

val a : int = 23
```

Listing 5.7: fsharpi

the arithmetic expression `3 + 4 * 5` is evaluated and the result is bound to the name `a`. Here, the addition and multiplication functions are shown in *infix notation* with the *operator* symbols `+` and `*`. To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. There are 2 possible orders, `3 + (4 * 5)` or `(3 + 4) * 5`, which gives different results. For integer arithmatic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Some built-in operators are shown in Table 5.2. In the table, the rows are shown from highest to lowest precedencens, and operators in the same row has same precedence. Associativity implies the order in which calculations are performed for operators of same precendence. E.g., the multiplication operator associates to the left which means that the expression `3 * 4 * 5` is evaluated as `(3 * 4) * 5` while the exponentation operator associates to the right implying that `3 ** 4 ** 5` is evaluated as `3 ** (4 ** 5)`.

· infix notation
· operator
· precedence

5.1 Integers

Integer types can also be written in binary, octal, or hexadecimal format using the prefixes `0b`, `0o`, and `0x`, e.g.,

```
>    let a = 0b1011
-    let b = 0o13
-    let c = 0xb;;

val a : int = 11
val b : int = 11
val c : int = 11
```
Listing 5.8: fsharpi

For a description of binary representations see Appendix A.1.
Using Extended Backus-Naur form (see Appendix C), the response from the the interpreter follows the syntax:

```
"val" name ":" type "=" value
```

A subtle point here is that the above problem is stated in using base 10 numbers, while almost all computers perform their calculations in base 2 numbers. E.g., the number 357.6 in base 10 can be considere the sum,

$$357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}, \tag{5.1}$$

and in general any base 10 number, also known as *decimal numbers*, has the representation $a_n a_{n-1} a_{n-2} \ldots a_{\overline{m}}$ a decimal number for $n+1$ digits to the left and m digits to the right of the period, and where $0 \leq a_i \leq 9$, and which translate to the equation

$$v = \sum_{i=-m}^{n} a_i 10^i. \tag{5.2}$$

Base 2 numbers, also known as *binary numbers*, has the general form · binary number

$$v = \sum_{i=-p}^{q} b_i 2^i, \tag{5.3}$$

where $0 \leq b_i \leq 1$ are binary digits, and where $357.6_{10} = 101100101.\overline{10011}_2$, using subscript to denote base. and where the bar notation means that the digits are repeated infinitely many times. I.e.,

$$357.6 = 1 \cdot 10^8 + 1 \cdot 10^6 + 1 \cdot 10^5 + 1 \cdot 10^2 + 1 \cdot 10^0 + 1 \cdot 10^{-1} + 1 \cdot 10^{-4} + 1 \cdot 10^{-5} \ldots \tag{5.4}$$

where alle terms with coefficient 0 have been removed for brevity. The example illustrates that while the number 357.6 is short to write in decimal, it has infinte number of digits in binary, and since any computer only has finite memory, then we must use an approximation. Floating point numbers is a popular approximation, whose type is `float` which is synonymous with `double` in F#, and which implements the IEEE 754 floating point standard for doubles.
...

5.2 Mutable Data

The most common syntax for a value definition is

```
"let'' [ "mutable" ] ident [":" type] "=" expr "in" expr
```

or alternatively

```
"let" ["mutable"] ident [":" type] "=" expr ["in"]
  expr
```

In the above, `ident` may be replaced with a more complicated pattern, but this is outside the scope of this text. If a value has been defined as mutable, then it's value may be changed using the following syntax,

```
expr "<-" expr
```

Mutable data is synonymous with the term *variable*. A variable an area in the computers working memory associated with a name and a type, and this area may be read from and written to during program execution. For example,

· Mutable data
· variable

```
let mutable x = Unchecked.defaultof<int> // Declare a variable x of
    type int and assign the corresponding default value to it.
printfn "%d" x
x <- 5 // Assign a new value 5 to x
printfn "%d" x
x <- -3 // Assign a new value -3 to x
printfn "%d" x
```

```
0
5
-3
```

Listing 5.9: mutableAssignReassing.fsx -

Here a area in memory was denoted `x`, declared as type integer and assigned a default value. Later, this value of of `x` was replaced with another integer and yet another integer. The operator '<-' is used to distinguish the statement from the mathematical concept of equality. A short-hand for the above is available as,

```
let mutable x = 5 // Declare a variable x and assign the value 5 to
    it
printfn "%d" x
x <- -3 // Assign a new value -3 to x
printfn "%d" x
```

```
5
-3
```

Listing 5.10: mutableAssignReassingShort.fsx -

where the assignment of the default value was skipped, and the type was inferred from the assignment operation. However, it's important to note, that when the variable is declared, then the '=' operator must be used, while later reassignments must use the '<-' operator. Type mismatches will result in an error,

```
let mutable x = 5 // Declare a variable x and assign the value 5 to
    it
printfn "%d" x
x <- -3.0 // This is illegal, since -3.0 is a float, while x is of
   type int
printfn "%d" x
```

```
/Users/sporring/repositories/fsharpNotes/
   mutableAssignReassingTypeError.fsx(3,6): error FS0001: This
   expression was expected to have type
    int
but here has type
    float
```

Listing 5.11: mutableAssignReassingTypeError.fsx -

A typical variable is a counter of type integer, and a typical use of counters is to increment them, i.e., erasing a new value to be one more that its previous value. For example,

```
let mutable x = 5 // Declare a variable x and assign the value 5 to
    it
printfn "%d" x
x <- x + 1 // Assign a new value -3 to x
printfn "%d" x
```

```
5
6
```

Listing 5.12: mutableAssignIncrement.fsx -

An function that elegantly implements the incrementation operation may be constructed as,

```
let incr =
  let mutable counter = 0
  fun () ->
    counter <- counter + 1
    counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

```
1
2
3
```

Listing 5.13: mutableAssignIncrementEncapsulation.fsx -

[1] Here the output of `incr` is an anonymous function, that takes no argument, increments the variable of `incr` and returns the new value of the counter. This construction is called *encapsulation*, since the variable `counter` is hidden by the function `incr` from the user, i.e., the user need not be concerned with how the increment operator is implemented and the variable name used by `incr` does not clutter the scope where it is used.

· encapsulation

Variables cannot be returned from functions, that is,

```
let g () =
  let x = 0
  x
printfn "%d" (g ())
```

---

[1] **Explain why this works!**

```
0
```

Listing 5.14: mutableAssignReturnValue.fsx -

declares a function that has no arguments and returns the value 0, while the same for a variable is illegal,

```
let g () =
  let mutual x = 0
  x
printfn "%d" (g ())
```

```
/Users/sporring/repositories/fsharpNotes/
   mutableAssignReturnVariable.fsx(3,3): error FS0039: The value
   or constructor 'x' is not defined
```

Listing 5.15: mutableAssignReturnVariable.fsx -

There is a workaround for this by using *reference cells* by the build-in function `ref` and operators `!` and `:=`, · reference cells

```
let g () =
  let x = ref 0
  x
let y = g ()
printfn "%d" !y
y := 3
printfn "%d" !y
```

```
0
3
```

Listing 5.16: mutableAssignReturnRefCell.fsx -

That is, the `ref` function creates a reference variable, the '`!`' and the '`:=`' operators reads and writes its value. Reference cells are in some language called pointers, and their use is strongly discouraged, since they may cause *side-effects*, which is the effect that one function changes the state of another, such as the following example demonstrates, · side-effects

```
let updateFactor factor =
  factor := 2

let multiplyWithFactor x =
  let a = ref 1
  updateFactor a
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
6
```

Listing 5.17: mutableAssignReturnSideEffect.fsx -

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the style of programming does not follow the usual assignment syntax. Better style of programming is,

```
let updateFactor () =
  2

let multiplyWithFactor x =
  let a = ref 1
  a := updateFactor ()
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
6
```

Listing 5.18: mutableAssignReturnWithoutSideEffect.fsx -

Here there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to refrain from using ref cells.

Part II

Imperative programming

Part III

Declarative programming

Part IV

Structured programming

Part V

Appendix

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index