Learning to program with F#

Jon Sporring

September 7, 2016

Contents

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,

Problem 4.1:

What is the sum of 357 and 864?

we have written the following program in F#,

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

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

```
1221
```

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 problem, 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 newline (LF possibly preceded by CR, 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 many types.

· statement
· `let`
· keyword
· binding
· expression
· format string
· string
· type

Types are a central concept in F#. In the script 4 we bound values of integer type to names. There are several different integer types in F#, here we used the one called `int`. 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:

· type declaration
· type inference

Program 4.2, typeInference.fsx:
Inferred types are given as part of the response from the interpreter.

```
> 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 = ()
```

The an interactive session displays the type using the *val* keyword followed by the name used in the binding, its type, and its value. Since the value is also responded, then the last `printfn` statement is superfluous. However, **it is ill advised 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,

Problem 4.2:

What is the sum of 357.6 and 863.4?

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

Program 4.3, quickStartSumFloat.fsx:
Floating point types and arithmetic.

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

```
1221.0
```

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,

Program 4.4, typeInferenceError.fsx:
Mixing types is often not allowed.

```
> 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/repositories/fsharpNotes/src/stdin(4,13): error FS0001:
    The type 'float' does not match the type 'int'
```

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 have a *lexical scope*. A scope is an area in a program, where a binding is valid, and lexical scope means that when a binding is used, then its value is substituted at the place of binding regardless of whether its value is rebound later in the text. Further, at the outer most level, rebinding is not allowed. If attempted, then F# will return an error as, e.g., [1]

· lexical scope

Program 4.5, quickStartRebindError.fsx:
A name cannot be rebound.

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

```



/Users/sporring/repositories/fsharpNotes/src/quickStartRebindError.fsx
    (2,5): error FS0037: Duplicate definition of value 'a'
```

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

Program 4.6, blocksNNames.fsx:
Names may be reused when separated by the lexeme ;;.

```
> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864
```

then rebinding did not cause an error. The difference is that the *;; lexeme*, which specifies the end of a *script-fragment*. A lexeme is a letter or a word, which the F# considers as an atomic unit. Script-fragments may be defined both in scripts and in interactive mode, and rebinding is not allowed at the outermost level in script-fragments.

· ;;
· lexeme
· script-fragment

In F# *functions* are also values, and defining a function `sum` as part of the solution to the above

· function

[1] Todo: **When command is omitted, then error messages have unwanted blank lines.**

program gives,

Program 4.7, quickStartSumFct.fsx:
A script to add 2 numbers using a user defined function.

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

```
1221
```

Entering the function into an interactive session will illustrate the inferred type, the function `sum` has: `val sum : x:int -> y:int -> int`. The `->` is the mapping operator in the sense that functions are mappings between sets. The type of the function `sum`, should be read as `val sum : x:int -> (y: int -> int)`, that is, `sum` takes an integer and returns a function, which takes an integer and returns an integer. 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,

Program 4.8, typesNBlockInferenceError.fsx:
Types are inferred in blocks, and F# tends to prefer integers.

```
> 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/repositories/fsharpNotes/src/stdin(3,13): error FS0001:
    This expression was expected to have type
     int
but here has type
     float
```

A remedy is to define the function in the same script-fragment as it is used, i.e,

Program 4.9, typesNBlockInference.fsx:
Defining a function together with its use, makes F# infer the appropriate types.

```
> 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
```

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.

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