Learning to program with F#

Jon Sporring

July 26, 2016

Contents

Chapter 6

Values, functions, and variables

In the previous chapter, we saw how to use F# as a calculator working with literals, operators and built-in functions. To save time and make programs easier to read and debug, it is useful to bind expressions to identifiers either as new constants, functions or operators. For example, to solve for x, when

$$ax^2 + bx + c = 0 \tag{6.1}$$

we use the quadratic formula from elementary algebra,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{6.2}$$

and write a small program that defines functions calculating relevant values for any set of coefficients,

```
let determinant a b c = b ** 2.0 - 2.0 * a * c
let positiveSolution a b c = (-b + sqrt (determinant a b c)) / (2.0 * a)
let negativeSolution a b c = (-b - sqrt (determinant a b c)) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let d = determinant a b c
let xp = positiveSolution a b c
let xn = negativeSolution a b c
printfn "%A * x ** 2.0 + %A * x + %A" a b c
printfn "  has determinant %A and solutions %A and %A" d xn xp
```

```
1.0 * x ** 2.0 + 0.0 * x + -1.0
  has determinant 2.0 and solutions -0.7071067812 and 0.7071067812
```

Listing 6.1: identifiersExample.fsx - Finding roots for quadratic equations using function name binding.

Here 3 functions are defined as `determinant`, `postiveSolution`, and `negativeSolution` are defined, and applied to 3 values named `a`, `b`, and `c`, and the results are named `d`, `xn`, and `xp`. These names are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing amount of code, which needs to be debugged. The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific grammar: An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters. An identifier must not be a keyword or a reserved-keyword listed in Figure 6.1 and 6.2. For characters in the Basic Latin Block, i.e., the first 128 code points alias ASCII characters, an ident is,

```
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter = "A" | "B" | ... |  "Z" | "a" | "b" | ... | "z"
```

abstract, and, as, assert, base, begin, class, default, delegate, do, done, downcast, downto, elif, else, end, exception, extern, false, finally, for, fun, function, global, if, in, inherit, inline, interface, internal, lazy, let, match, member, module, mutable, namespace, new, null, of, open, or, override, private, public, rec, return, sig, static, struct, then, to, true, try, type, upcast, use, val, void, when, while, with, and yield.

Figure 6.1: List of keywords in F#.

atomic, break, checked, component, const, constraint, constructor, continue, eager, fixed, fori, functor, include, measure, method, mixin, object, parallel, params, process, protected, pure, recursive, sealed, tailcall, trait, virtual, and volatile.

Figure 6.2: List of reserved keywords for possible future use in F#.

```
specialChar = "_"
ident = (letter | "_") {letter | dDigit | specialChar}
```

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. The for the full definition, `letter = Lu | Ll | Lt | Lm | Lo | Nl` and `specialChar = Pc | Mn | Mc | Cf`, which referes to the Unicode general categories described in Appendix **??**, and there are currently 19.345 possible Unicode code points in the `letter` category and 2.245 possible Unicode code points in the `specialChar` category. Binding expressions to identifiers is done with the keyword `let`, using the following simplified syntax:

```
arg = ident | "(" ident ":" type ")"
argList = arg | arg argList
identOrOp = ident | ( operatorName )
expr = ...
  | "let " ["mutable "] ident  [":" type] "=" expr " in " expr (* binding a value
     *)
  | "let " ident-or-op argList [":" type] "=" expr " in " expr (* binding a
     function or operator *)
  | "let " rec function-or-value-defns (* recursive definition *)
  | "fun " argList "->" expr (* a function as value *)
  | expr ":" type (* type annotation *)
  | "begin " expr " end" (* alternative block expression *)
  | expr; expr (* sequence of expression *)
  | ...
```

which will be discussed in the following.[1]

6.1 Values (Constant bindings)

Binding identifiers to literals or expressions that are evaluated to be values is called value binding, and examples are `let a = 3.0` and `let b = cos 0.9`. On EBNF the simplified syntax,

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

I.e., the *`let`* keyword dictates that the identifier `ident` is an alias of the expression `expr`. The type may be especified with the *:* token to type `type`. The binding may be mutable as indicated by the keyword *`mutable`*, which will be discussed in Section **??**, and the binding holds *lexically* for the last expression as indicated by the *`in`* keyword.[2] For example, letting the identifier `p` be bound to the value `2.0` and using it in an expression is done as follows,

· let
· :
· mutable
· lexically
· in

[1] **Spec-4.0 Section 6.6, function-or-value-defns, 1. makes little sense to have values definitions recursive, and 2. possible Mono deviation from specification: let rec function-defn and function-defn requires newline before and.**

[2] https://coders-corner.net/2013/11/12/lexical-scope-vs-dynamic-scope/

```
let p = 2.0 in printfn "%A" (p ** 3.0)
```

```
8.0
```

Listing 6.2: letValue.fsx - The identifier `p` is used in the expression following the `in` keyword.

In the interactive mode used in the example above, we see that F# infers the type... F# will ignore most newlines between tokens, i.e., the above is equivalent to writting,

```
let p = 2.0 in
printfn "%A" (3.0 ** p)
```

```
9.0
```

Listing 6.3: letValueLF.fsx - Newlines after `in` make the program easier to read.

F# also allows for an alternative notation called *lightweight syntax*, where e.g., the `in` keyword is replaced with a newline, and the expression starts on the next line at the same column as `let` starts in, i.e., the above is equivalent to

· lightweight syntax

```
let p = 2.0
printfn "%A" (3.0 ** p)
```

```
9.0
```

Listing 6.4: letValueLightWeight.fsx - Lightweight syntax does not require the `in` keyword, but expression must be aligned with the `let` keyword.

The same expression in interactive mode will also respond the infered types, e.g.,

```
> let p = 2.0
- printfn "%A" (3.0 ** p);;
9.0

val p : float = 2.0
val it : unit = ()
```

Listing 6.5: fsharpi, Interactive mode also responds inferred types.

By the `val` keyword in the line `val p : float = 2.0` we see that `p` is inferred to be of type `float` and bound to the value `2.0`. The inference is based on the type of the right-hand-side, which is of type `float`. Identifiers may be defined to have a type using the `:` token, but the types on the left-hand-side and right-hand-side of the `=` token must be identical. I.e., mixing types gives an error,

```
let p : float = 3
printfn "%A" (3.0 ** p)
```

```


/Users/sporring/repositories/fsharpNotes/src/letValueTypeError.fsx(1,17):
    error FS0001: This expression was expected to have type
     float
but here has type
     int
```

Listing 6.6: letValueTypeError.fsx - Binding error due to type mismatch.

Here, the left-hand-side is defined to be an identifier of type float, while the right-hand-side is a literal of type integer.
An expression can be a sequence of expressions separated by the token `;`, e.g.,

```
let p = 2.0 in printfn "%A" p; printfn "%A" (3.0 ** p)
```

```
2.0
9.0
```

Listing 6.7: letValueSequence.fsx - A value binding for a sequence of expressions.

The lightweight syntax automatically inserts the ; token at newlines, hence using the lightweight syntax the above is the same as,

```
let p = 2.0
printfn "%A" p
printfn "%A" (3.0 ** p)
```

```
2.0
9.0
```

Listing 6.8: letValueSequenceLightWeight.fsx - A value binding for a sequence using lightweight syntax.

A key concept of programming is *scope*. In F#, the scope of a value binding is lexically meaning that the binding is constant from the `let` statement defining it, untill it is redefined, e.g.,

· scope

```
let p = 3 in let p = 4 in printfn " %A" p;
```

```
 4
```

Listing 6.9: letValueScopeLower.fsx - Redefining identifiers is allowed in lower scopes.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent. F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, e.g.,

```
let p = 3
let p = 4
printfn "%A" p;
```

```


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

Listing 6.10: letValueScopeLowerError.fsx - Redefining identifiers is not allowed in lightweight syntax at top level.

But using `begin` and `end` keywords, we create a *block* which acts as a *nested scope*, and then redefining is allowed, e.g.,

· block
· nested scope

```
begin
  let p = 3
  let p = 4
  printfn "%A" p
end
```

```
4
```

Listing 6.11: letValueScopeBlockAlternative2.fsx - A block has lower scope level, and rebinding is allowed.

It is said that the second binding *overshadows* the first. Alternatively we may use parentheses to create a block, e.g.,

· overshadows

```
(
  let p = 3
  let p = 4
  printfn "%A" p
)
```

```
4
```

Listing 6.12: letValueScopeBlockAlternative3.fsx - A block may be created using parentheses.

In both cases we used indentation, which is good practice, but not required here. Lowering level is a natural part of function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter **??**.
Defining blocks is useful for controling the extend of a lexical scope of bindings. For example, adding a second `printfn` statement,

```
let p = 3 in let p = 4 in printfn "%A" p; printfn "%A" p
```

```
4
4
```

Listing 6.13: letValueScopeBlockProblem.fsx - Overshadowing hides the first binding.

will print the value 4 last bound to the identifier `p`, since token `;` associates to the right, i.e., the above is interpreted as `let p = 3 in let p = 4 in (printfn "%A"p; printfn "%A"p)`. Instead we may create a block as,[3]

```
let p = 3 in (let p = 4 in printfn " %A" p); printfn " %A" p;
```

```
 4
 3
```

Listing 6.14: letValueScopeBlock.fsx - Blocks allow for the return to the previous scope.

Here the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at `)`, returning to the lexical scope of `let p = 3 in ....` Alternatively, the `begin` and `end` keywords could equally have been used.
[4]

6.2 Functions (function bindings)

A function is a mapping between an input and output domain. F# is a functional first programming language, and offers a number of alternative methods for specifying parameters, which will be discussed in this section. Binding identifieres to functions follows a syntax similar to value binding,

```
arg = ident | "(" ident ":" type ")"
argList = arg | arg argList
identOrOp = ident | ( operatorName )
expr = ...
  | "let " ident—or—op argList [":" type] "=" expr " in " expr (* binding a
     function or operator *)
  | ...
```

An example in interactive mode is,

[3]**spacing in lstinline mode after double quotation mark is weird.**
[4]**Remember to say something about interactive scripts and the `;;` token and scope**

```
> let sum (x : float) (y : float) : float = x + y in
- let c = sum 357.6 863.4 in
- printfn "%A" c;;
1221.0

val sum : x:float -> y:float -> float
val c : float = 1221.0
val it : unit = ()
```

Listing 6.15: fsharpi, An example of a binding of an identifier and a function.

and we see that the function is interpreted to have the type `val sum : x:float -> y:float -> float`. The `->` token means a mapping between sets, in this case floats. The function is also a higher order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as argument and returns a float.
Not all types need to be declared, just sufficent for F# to be able to infer the types for the full statement. In the example, one sufficent specification is, and we could just have specified the type of the result,

```
let sum x y : float = x + y
```

Listing 6.16: All types need most often not be specified.

or even just one of the arguments,

```
let sum (x : float) y = x + y
```

Listing 6.17: Just one type is often enough for F# to infer the rest.

In both cases, since the `+` *operator* is only defined for *operands* of the same type, then when the type of either the result, any or both operands are declared, then the type of the remaining follows directly. As for values, lightweight syntax automatically inserts the keyword `in` and the token `;;`,

· operator
· operand

```
let sum x y : float = x + y
let c = sum 357.6 863.4
printfn "%A" c
```

```
1221.0
```

Listing 6.18: letFunctionLightWeight.fsx - Lightweight syntax for function definitions.

Arguments need not always be inferred to types, but may be of generic type, which F# prefers, when *type safety* is ensured, e.g.,

· type safety

```
> let second x y = y
- let a = second 3 5
- printfn "%A" a
- let b = second "horse" 5.0
- printfn "%A" b;;
5
5.0

val second : x:'a -> y:'b -> 'b
val a : int = 5
val b : float = 5.0
val it : unit = ()
```

Listing 6.19: fsharpi, Typesafety implies that a function will work for any type, and hence it is generic.

Here the function `second` does not use the first argument, `x` which is any type called `'a`, and the type of the second element, `y`, is also any type an not nessecarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*. · generic function

A function may contain a sequence of expressions, but must return a value. E.g., the quadratic formula may written as,

```
let solution a b c sgn =
  let determinant a b c =
    b ** 2.0 - 2.0 * a * c
  let d = determinant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = solution a b c +1.0
let xn = solution a b c -1.0
printfn "%A * x ** 2.0 + %A * x + %A" a b c
printfn "  has solutions %A and %A" xn xp
```

```
1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

Listing 6.20: identifiersExampleAdvance.fsx - A function may contain sequences of expressions.

Here we used the lightweight syntax, where the `=` identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values. Note also that since the function `determinant` is defined in the nested scope of `solution`, then `determinant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

Lexical scope and function definitions can be a cause of confusion as the following example shows, · lexical scope

```
let testScope x =
  let a = 3.0
  let f z = a * x
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
6.0
```

Listing 6.21: lexicalScopeNFunction.fsx - Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

Here the value binding for `a` is redefined, after it has been used to define a helper function `f`. So which value of `a` is used when we later apply `f` to an argument? To resolve the confusion, remember that value binding is lexically defined, i.e., the binding `let f z = a * x` uses the value of `a`, it has by the ordering of the lines in the script, not dynamically by when `f` was called. Hence, think of lexical scope as substitution of an identifier with its value or function immediately at the place of definition. I.e., since `a` and `3.0` are synonymous in the first lines of the program, then the function `f` is really defined as, `let f z = 3.0 * x`. [5] Advice!

Functions do not need a name, but may be declared as an *anonymous function* using the `fun` keyword and the `->` token, · anonymous function

[5] **comment on dynamic scope and mutable variables.**

```
let first = fun x y -> x
printfn "%d" (first 5 3)
```

```
5
```

Listing 6.22: functionDeclarationAnonymous.fsx - Anonymous functions are functions as values.

Here a name is bound to an anonymous function, which returns the first of two arguments. The difference to `let first x y = x` is that anonymous functions may be treated as values, meaning that they may be used as arguments to other functions, and new values may be reassigned to their identifiers, when mutable, as will be discussed in Section . A common use of anonymous functions is as as arguments to other functions, e.g.,

```
let apply f x y  = f x y
let mul = fun a b -> a * b
printfn "%d" (apply mul 3 6)
```

```
18
```

Listing 6.23: functionDeclarationAnonymousAdvanced.fsx - Anonymous functions are often used as arguments for other functions.

Note that here `apply` is given 3 arguments, the function `mul` and 2 integers. It is not given the result of `mul 3 6`, since that would not match the definition of `apply`. Anonymous functions and functions as arguments are powerfull concepts, but tend to make programs harder to read, and their use should be limited. Advice!
Functions may be declared from other functions

```
let mul (x, y) = x*y
let double y = mul (2.0, y)
printfn "%g" (mul (5.0, 3.0))
printfn "%g" (double 3.0)
```

```
15
6
```

Listing 6.24: functionDeclarationTupleCurrying.fsx -

For functions of more than 1 argument, there exists a short notation, which is called *currying* in tribute of Haskell Curry, · currying

```
let mul x y = x*y
let double = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (double 3.0)
```

```
15
6
```

Listing 6.25: functionDeclarationCurrying.fsx -

Here `mul 2.0` is a partial specification of the function `mul x y`, where the first argument is fixed, and hence, `double` is a function of 1 argument being the second argument of `mul`. Currying is often used in functional programming, but generally currying should be used carefully, since currying may seriously reduce readability of code. Advice!
A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values, · procedure

```
let printIt a = printfn "This is '%A'" a
printIt 3
printIt 3.0
```

```
This is '3'
This is '3.0'
```

Listing 6.26: procedure.fsx - A procedure is a function that has no return value, which in F# implies `()` as return value.

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section ??. which also does not have a return value. Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.

· encapsulation
· side-effects

6.3 User-defined operators

Operators are functions, e.g., the infix multiplication operator `+` is equivalent to the function `(+)`, e.g.,

```
let a = 3.0
let b = 4.0
let c = a + b
let d = (+) a b
printfn "%A plus %A is %A and %A" a b c d
```

```
3.0 plus 4.0 is 7.0 and 7.0
```

Listing 6.27: addOperatorNFunction.fsx -

All operator has this option, and you may redefine them and define your own operators, who has names specified by the following simplified EBNF:

```
infixOrPrefixOp := "+" | "-" | "+. " | "-. " | "%" | "&" | "&&"
tildes = "~" | "~" tildes
prefixOp = infixOrPrefixOp | tildes | (! {opChar} - "!=")
dots = "." | "." dots
infixOp =
  {dots} (
    infixOrPrefixOp
    | "-" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | "  |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":=" | "::" | "$" | "?"
```

Listing 6.28: Grammar for infix and prefix tokens

The precedence rules and associtivity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table **??**. E.g., `.*`, `+++`, and `<+` are legal operator names for infix operators, they have precedence as ordered, and their associativity are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are,

```
let (.*) x y = x * y + 1
printfn "%A" (3 .* 4)
let (+++) x y = x * y + y
printfn "%A" (3 +++ 4)
let (<+) x y = x < y + 2.0
printfn "%A" (3.0 <+ 4.0)
let (~+.) x = x+1
printfn "%A" (+.1)
```

```
13
16
true
2
```

Listing 6.29: operatorDefinitions.fsx -

Beware, redefining existing operators lexically redefines all future uses of operator for all types, hence it is not a good idea to redefine operators, but better to define new.[6] In Chapter /refchap:oop we will discuss how to define type specific operators including prefix operators.

Advice!

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment. `(*`. [7]

6.4 Printf

[8]

6.5 Variables (Mutable bindings)

The `mutable` in `let` bindings means that the identifier may be rebound to a new value using the following syntax,

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

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

· Mutable data
· variable

```
let mutable x = 5
printfn "%d" x
x <- -3
printfn "%d" x
```

```
5
-3
```

Listing 6.30: mutableAssignReassingShort.fsx - A variable is defined and later reassigned a new value.

Here a area in memory was denoted `x`, initially assigned the integer value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the `<-` token. The `<-` token

· <-

[6]**It seems there is a bug in mono:** `let ( +) x = x+1 in printfn "%A" +1;;` **prints 1 and not 2.**
[7]**this requires comments to be describe previously!**
[8]**Maybe explain the printf function, Spec-4.0 Section 6.3.16 'printf' Formats, but also max and min comparison functions and math functions Section 18.2.2 and 18.2.4?**

is used to distinguish the assignment from the comparison operator, i.e., if we by mistake had written,

```
> let mutable a = 0
- a = 3;;

val mutable a : int = 0
val it : bool = false
```

Listing 6.31: fsharpi, example of changing the content of a variable.

then we instead would have obtained the default assignment of the result of the comparision of the content of a with the integer 3, which is false. However, it's important to note, that when the variable is initially defined, then the '|=|' operator must be used, while later reassignments must use the |<-| operator.
Assignment type mismatches will result in an error,

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

```



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

Listing 6.32: mutableAssignReassingTypeError.fsx - Assignmetn type mismatching causes a compile time error.

I.e., once the type of an identifier has been declared or infered, then it cannot be changed.
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 6.33: mutableAssignIncrement.fsx - Variable increment is a common use of variables.

which is an example we will return to many times later in this text.
****
A 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 6.34: mutableAssignIncrementEncapsulation.fsx -

[9] 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 ())
```

```
0
```

Listing 6.35: 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/src/mutableAssignReturnVariable.fsx
    (3,3): error FS0039: The value or constructor 'x' is not defined
```

Listing 6.36: 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 6.37: 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

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

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

Listing 6.38: 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 6.39: 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.

0
10
11
...

[10] **Somewhere I should talk about whitespaces and newlines Spec-4.0 Chapter 3.1**
[11] **Somewhere I should possibly talk about Lightweight Syntax, Spec-4.0 Chapter 15.1**

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