Learning to program with F#

Jon Sporring

July 16, 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.

- statement
- `let`
- keyword
- binding
- expression
- format string
- string
- 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:

- 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" [ "mutable" ] ident [":" type] "=" expr ["in" 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 binding may be mutable, which will be discussed in Section 5.4, and the binding may only be for the last expression as indicated by the *in*. 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 6. Examples of let statements with *literals*

· let
· :
· in
· 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 specify the type of the name, and that F# interpreted the type from the literal form of the right-hand-side. Types matter, since the operations that can be performed on integers are quite different from those that can be performed on characters and strings. I.e.,

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	3y	Integer values from -128 to 127
	int16	s	3s	Integer values from -32768 to 32767
	uint16	us	3us	Integer values from 0 to 65535
	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	3I	Integer not limited to 64 bits
	nativeint	n	3n	A native pointer as a signed integer
	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	Synonymous 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 characters see Appendix B.

```
> 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 5.2: fsharpi

the variables `a`, `b`, `c`, and `d` all represent the number 3, but their types are different, and hence they are quite different values. When specifying the type, then the type and the literal form must match, i.e., mixing types and literals gives an error,

```
  > 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 5.3: 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 5.4: 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 5.5: fsharpi

Here `System.Convert.ToBoolean` is the name of a function `ToBoolean`, which is a *member* of the · member

class `Convert` that is included in the *namespace* `System`. Namespaces, classes, and members are all part of Structured programming to be discussed in Part IV. For more on functions see Section 6
Typecasting is often a destructive operation, e.g., typecasting a float to int removes the part after the decimal point without rounding,

· class
· namespace

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

```
357.6 -> 357
```

Listing 5.6: quickStartDownCast.fsx - Fractional part is removed by downcasting.

Here we typecasted to a lesser type, in the sense that integers is a subset of floats, which is called *downcasting*. The opposite is called *upcasting* is often non-destructive, as Listing 5.4 showed, where an integer was casted to a float while retaining its value. As a side note, *rounding* a number $y.x$, where y is the whole part and x is the fractional part, is the operation of mapping numbers in the interval $y.x \in [y.0, y.5)$ to y and $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting as follows,

· downcasting
· upcasting
· rounding

```
let a = 357.6
let b = int (a + 0.5)
printfn "%A -> %A" a b
```

```
357.6 -> 358
```

Listing 5.7: rounding.fsx - The rounding function may be obtained by downcasting.

since if $y.x \in [y.0, y.5)$, then $y.x + 0.5 \in [y.5, y + 1)$, from which downcasting removes the fractional part resulting in y. And if $y.x \in [y.5, y + 1)$, then $y.x + 0.5 \in [y + 1, y + 1.5)$, from which downcasting removes the fractional part resulting in $y + 1$. Hence, the result is rounding.
If parentheses are ommitted in Listing 5.7, then F# will interpret the expression as `(int a)+ 0.5`, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., typecasting takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression, whose result is bound to `a` by

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

val a : int = 23
```

Listing 5.8: fsharpi

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* tokens `+` 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 arithmetic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedences, and for some common built-in operators shown in Table 5.2, the precedence is shown by the order they are given in the table. Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, e.g., in

· infix notation
· operator
· precedence

```
> let a = 3.0*4.0*5.0
- let b = (3.0*4.0)*5.0
```

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. Rows are ordered from highest to lowest precedences, such that `op * op` has higher precedence than `op + op`. Operators in the same row has same precedence.

```
- let c = 3.0*(4.0*5.0);;

val a : float = 60.0
val b : float = 60.0
val c : float = 60.0

> let d = 4.0 ** 3.0 ** 2.0
- let e = (4.0 ** 3.0) ** 2.0
- let f = 4.0 ** (3.0 ** 2.0);;

val d : float = 262144.0
val e : float = 4096.0
val f : float = 262144.0
```

Listing 5.9: fsharpi

the expression for `a` is interpreted as `b` but gives the same results as `c` since association does not matter for multiplication of numbers, but the expression for `d` is interpreted as `f` which is quite different from `e`.

A less common notation is to define bindings for expressions using the *in* keyword, e.g., · `in`

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

```
9.0
```

Listing 5.10: numbersIn.fsx - The name `p` is only bound in the nested scope following the keyword `in`.

Here `p` is only bound in the *scope* of the expression following the `in` keyword, in this the `printfn` statement, and `p` is unbound in lines that follows. · scope

5.1 Integers and Reals

The set of integers and reals are infinitely large, and since all computers have limited resources, it is not possible to represent these sets in their entirety. The various integer and floating point types listed in Table 5.1 are finite subset where the integer types have been reduced by limiting their ranges and the floating point types have been reduced by sampling the space of reals. An in-depth description of integer and floating point implementations can be found in Appendix **??**. The `int` and `float` are the most common types.

For integers the following arithmetic operators are defined:

`+op, -op`: These are unary plus and minus operators, and plus has no effect, but minus changes the sign, e.g.,

```
> let a = 5
- let b = -a;;

val a : int = 5
val b : int = -5
```

Listing 5.11: fsharpi

`op + op, op - op, op * op`: These are binary operators, where addition, subtraction and multiplication performs the usual operations,

```
> let a = 7 + 3
- let b = 7 - 3
- let c = 7 * 3;;

val a : int = 10
val b : int = 4
val c : int = 21
```

Listing 5.12: fsharpi

`op / op, op % op`: These are binary operators, and division performs integer division, where the fractional part is discarded after division, and the `\%` is the remainder operator, which calculates the remainder after integer division,

```
> let a = 7 / 3
- let b = 7 % 3;;

val a : int = 2
val b : int = 1
```

Listing 5.13: fsharpi

Together integer division and remainder is a lossless representation of the original number as,

```
> let x = 7
- let whole = x / 3
- let remainder = x % 3
- let y = whole * 3 + remainder;;

val x : int = 7
val whole : int = 2
val remainder : int = 1
val y : int = 7
```

Listing 5.14: fsharpi

And we see that `x` and `y` is bound to the same value.

Integer exponentiation is not defined as an operator, but this is available the built-in function `pown`, e.g.,

```
> pown 2 5;;
val it : int = 32
```
Listing 5.15: fsharpi

which is equal to 2^5. Note that when no let statement is used in conjunction with an expression then F# automatically binds the result to the *it* name, i.e., the above is equal to · it

```
  > let it = pown 2 5;;

val it : int = 32
```
Listing 5.16: fsharpi

Rumor has it, that the name `it` is an abreviation for 'irrelevant'.
Performing arithmetic operations on `int` types requires extra care, since the result may cause *overflow*, · overflow
underflow, and even exceptions, e.g., the range of the integer type `sbyte` is $[-128 \ldots 127]$, which causes · underflow
problems in the following example,

```
  > let a = 100y - let b = 30y - let c = a+b;;

val a : sbyte = 100y
val b : sbyte = 30y
val c : sbyte = -126y
```
Listing 5.17: fsharpi

Here $100+30 = 130$, which is larger than the biggest `sbyte`, and the result is an overflow. Similarly, we get an underflow, when the arithmetic result falls below the smallest value storable in an `sbyte`,

```
> let a = -100y
- let b = -30y
- let c = a+b;;

val a : sbyte = -100y
val b : sbyte = -30y
val c : sbyte = 126y
```
Listing 5.18: fsharpi

Notice that neither overflow nor underflow error gave rise to an error message, which is why such bugs are difficult to find. Dividing any non-zero number with 0 is infinite, which is also outside the domain of any of the integer types, but in this case, F# casts an *exception*, · exception

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0007>.$FSI_0007.main@ () <0x6b78180 + 0x0000e
     > in <filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
     InternalInvoke (System.Reflection.MonoMethod,object,object[],
     System.Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj,
     BindingFlags invokeAttr, System.Reflection.Binder binder,
     System.Object[] parameters, System.Globalization.CultureInfo
     culture) <0x1a55ba0 + 0x000a1> in <filename unknown>:0
```

```
Stopped due to error
```
Listing 5.19: fsharpi

The output looks daunting at first sight, but the first and last line of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle are technical details concerning which part of the program caused this, and can be ignored for the time being. Exceptions are a type of *run-time error*, and are treated in Chapter **??** · run-time error
Integers 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 5.20: fsharpi

For a description of binary representations see Appendix A.1. The overflow error in Listing 5.17 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`,

```
> let a = 0b10000010uy
- let b = 0b10000010y;;

val a : byte = 130uy
val b : sbyte = -126y
```
Listing 5.21: fsharpi

That is, for signed bytes, the left-most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_b$ is $130 = 10000010_2$ causes the left-most bit to be used, then this is wrongly interpreted as a negative number, when stored in an `sbyte`.
For floating point numbers the following arithmetic operators are defined:

`+op`, `-op`: These are unary plus and minus operators, and plus has no effect, but minus changes the sign, e.g.,

```
> let a = 5.0
- let b = -a;;

val a : float = 5.0
val b : float = -5.0
```
Listing 5.22: fsharpi

`op + op`, `op - op`, `op * op`, `op / op`: These are binary operators, where addition, subtraction, multiplication, and division performs the usual operations,

```
> let a = 7.0 + 3.0
- let b = 7.0 - 3.0
- let c = 7.0 * 3.0
- let d = 7.0 / 3.0;;
```

```
val a : float = 10.0
val b : float = 4.0
val c : float = 21.0
val d : float = 2.333333333
```
Listing 5.23: fsharpi

`op % op:` The binary remainder operator, and division performs integer division, where the fractional part is discarded after division, and the `\%` is the remainder operator, which calculates the remainder after integer division,

```
> let a = 7.0 / 3.0
- let b = 7.0 % 3.0;;

val a : int = 2.0
val b : int = 1.0
```
Listing 5.24: fsharpi

The remainder for floating point numbers can be fractional, but division, rounding, and remainder is still a lossless representation of the original number as,

```
> let x = 7.0
- let division = x / 3.2
- let whole = float (int (division + 0.5))
- let remainder = x % 3.2
- let y = whole * 3.2 + remainder;;

val x : float = 7.0
val division : float = 2.1875
val whole : float = 2.0
val remainder : float = 0.6
val y : float = 7.0
```
Listing 5.25: fsharpi

And we see that `x` and `y` is bound to the same value.

`op ** op:` In spite of an unusual notation, the binary exponentiation operator performs the usual calculation,

```
> let a = 2.0 ** 5.0;;

val a : float = 32.0
```
Listing 5.26: fsharpi

which is equal to 2^5.

Arithmetic using `float` will not cause over- and underflow problems, since the IEEE 754 standard includes the special numbers $\pm\infty$ and NaN. E.g.,

```
> let a = 1.0/0.0
- let b = 0.0/0.0;;

val a : float = infinity
```

a	b	$a \cdot b$	$a + b$	$\bar{a}$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Table 5.3: Truth table for boolean 'and', 'or', and 'not' operators. Value 0 is false and 1 is true.

```
val b : float = nan
```

Listing 5.27: fsharpi

However, the `float` type has limite precision, since there is only a finite number of numbers that can be stored in a float. E.g.,

```
> let a = 357.8
- let b = a+0.1
- let c = b+0.1
- let d = c - 358.0;;

val a : float = 357.8
val b : float = 357.9
val c : float = 358.0
val d : float = 5.684341886e-14
```

Listing 5.28: fsharpi

Hence, although `c` appears to be correctly calculated, by the subtraction we see, that the value bound in `c` is not exactly the same as `358.0`, and the reason is that the neither `357.8` nor `0.1` are exactly representable as a `float`, which is why the repeated addition accumulates a small representation error.

5.2 Booleans

Boolean arithmatic is the basis of almost all computers and particularly important for controling program flow, which will be discussed in Chapter **??**. Boolean values are one of 2 possible values, true or false, which is also sometimes written as 1 and 0. Two basic operations on boolean values are '*and*' often also written as multiplication, and '*or*' often written as addition, and '*not*' often written as a bar above the value. All possible combination of input on these values can be written on tabular form, known as a *truth table*, shown in Table 5.3. That is, the multiplication and addition are good mnemonics for remembering the result of the 'and' and 'or' operators. In F# the values `true` and `false` are used, and the operators `\&\&` for 'and', `||` for 'or', and the function `not` for 'not', such that the above table is reproduced by,

- and
- or
- not
- truth table

```
> let t = true
- let f = false
- printfn "a     b     a*b   a+b   not a"
- printfn "%A %A %A %A %A" f f (f && f) (f || f) (not f)
- printfn "%A %A  %A %A  %A" f t (f && t) (f || t) (not f)
- printfn "%A  %A %A %A  %A" t f (t && f) (t || f) (not t)
- printfn "%A  %A  %A  %A  %A" t t (t && t) (t || t) (not t);;
a     b     a*b   a+b   not a
false false false false true
false true  false true  true
true  false false true  false
```

```
true  true  true  true  false

val t : bool = true
val f : bool = false
val it : unit = ()
```
Listing 5.29: fsharpi

Careful spacing in the format string of the `printfn` function was used to align columns. Next section will discuss more elegant formatting options.

5.3 Chars and Strings

5.4 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 5.30: 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 5.31: 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 5.32: 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 5.33: 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 5.34: 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 ())
```

```
0
```

Listing 5.35: 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/
   mutableAssignReturnVariable.fsx(3,3): error FS0039: The value
   or constructor 'x' is not defined
```

Listing 5.36: 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
```

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

```
printfn "%d" !y
```

```
0
3
```

Listing 5.37: 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

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

Listing 5.38: 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 5.39: 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.

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