Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2018-09-26 17:36:41+02:00

Contents

11 | Collections of Data

F# is tuned to work with collections of data, and there are several built-in types of collections with various properties making them useful for different tasks. Examples include strings, lists, arrays, and sequences. Strings were discussed in Chapter 5 and will be revisited here in more details. Sequences will not be discussed,[1] and we will concentrate on lists and one- and two-dimensional arrays.

The data structures discussed below all have operators, properties, methods, and modules to help you write elegant programs using them.

Properties and methods are common object-oriented terms used in conjunction with the discussed functionality. They are synonymous with values and functions and will be discussed in Chapter 20. Properties and methods for a value or variable are called using the *dot notation*, i.e., with the "."-lexeme. For example, `"abcdefg".Length` is a property and is equal to the length of the string, and `"abcdefg".ToUpper()` is a method and creates a new string where all characters have been converted to upper case.

· dot notation

The data structures also have accompanying modules with a wealth of functions and where some are mentioned here. Further, the data structures are all implemented as classes offering even further functionality. The modules are optimized for functional programming, see Chapters 13 to 17, while classes are designed to support object oriented programming, see Chapters 20 to 22.

11.1 Strings

Strings have been discussed in Chapter 5, the content of which will be briefly revisited here followed by a description of some of the many supporting built-in functions in F# on strings.

A *string* is a sequence of characters. Each character is represented using UTF-16, see Appendix C for further details on the unicode standard. The type `string` is an alias for *`System.string`*. String literals are delimited by double quotation marks "`""`" and inside the delimiters, character escape sequences are allowed (see Table 5.2), which are replaced by the corresponding character code. Examples are `"This is a string"`, `"\tTabulated string"`, `"A \"quoted\" string"`, and `""`. Strings may span several lines, and new lines inside strings are part of the string unless the line is ended with a backslash. Strings may be *verbatim* by preceding the string with "`@`", in which case escape sequences are not replaced, but two double quotation marks are an escape sequence which is replaced by a one double quotation mark. Examples of "`@`"-verbatim strings are: `@"This is a string"`,

· string

· `System.string`

· verbatim string

[1] Jon: **Should we discuss sequences?**

`@"\tNon-tabulated string"`, `@"A ""quoted"" string"`, and `@""`. Alternatively, a verbatim string may be delimited by three double quotation marks. Examples of """-verbatim strings are: `"""This is a string"""`, `"""\tNon-tabulated string"""`, `"""A "quoted" string"""`, and `"""""""`. Strings may be indexed using the `.[]` notation, as demonstrated in Listing 5.27.

11.1.1 String Properties and Methods

Strings have a few properties which are values attached to each string and accessed using the "." notation. The only to be mentioned here is:

`IndexOf(): string -> int`. Returns the index of the first occurence of the argument or −1, if the argument does not appear in the string.

Listing 11.1: `IndexOf()`

```
> "Hello World".IndexOf("World");;
val it : int = 6
```

`Length: int`. Returns the length of the string.

Listing 11.2: `Length`

```
val it : int = 4
```

`ToLower(): unit -> string`. Returns a copy of the string where each letter has been converted to lower case.

Listing 11.3: `ToLower()`

```
> "aBcD".ToLower();;
val it : string = "abcd"
```

`ToUpper(): unit -> string`. Returns a copy of the string where each letter has been converted to upper case.

Listing 11.4: `ToUpper()`

```
> "aBcD".ToUpper();;
val it : string = "ABCD"
```

`Trim(): unit -> string`. Returns a copy of the string where leading and trailing whitespaces have been removed.

Listing 11.5: `Trim()`

```
> "  Hello World  ".Trim();;
val it : string = "Hello World"
```

`Split(): unit -> string []`. Splits a string of words separated by spaces into an array of words. See Section 11.3 for more information about arrays.

Listing 11.6: `Split()`

```
> "Hello World".Split();;
val it : string [] = [|"Hello"; "World"|]
```

11.1.2 String Module

The `String` module offers many functions for working with strings. Some of the most powerful ones are listed below, and they are all higher order functions.

`String.collect: (char -> string) -> string -> string`. Creates a new string whose characters are the results of applying a specified function to each of the characters of the input string and concatenating the resulting strings.

Listing 11.7: `String.collect`

```
> String.collect (fun c -> (string c) + ", ") "abc";;
val it : string = "a, b, c, "
```

`String.exists: (char -> bool) -> string -> bool`. Tests if any character of the string satisfies the given predicate.

Listing 11.8: `String.exists`

```
> String.exists (fun c -> c = 'd') "abc";;
val it : bool = false
```

`String.forall: (char -> bool) -> string -> bool`. Tests if all characters in the string satisfy the given predicate.

Listing 11.9: `String.forall`

```
> String.forall (fun c -> c < 'd') "abc";;
val it : bool = true
```

`String.init: int -> (int -> string) -> string`. Creates a new string whose characters are the result of applying a specified function to each index and concatenating the resulting strings.

Listing 11.10: `String.init`

```
> String.init 5 (fun i -> (string i) + ", ");;
val it : string = "0, 1, 2, 3, 4, "
```

`String.iter: (char -> unit) -> string -> unit`. Applies a specified function to each character in the string.

Listing 11.11: `String.iter`

```
> String.iter (fun c -> printfn "%c" c) "abc";;
a
b
c
val it : unit = ()
```

`String.map: (char -> char) -> string -> string`. Creates a new string whose characters are the results of applying a specified function to each of the characters of the input string.

Listing 11.12: `String.map`

```
> let toUpper c = c + char (int 'A' - int 'a')
- String.map toUpper "abcd";;
val toUpper : c:char -> char
val it : string = "ABCD"
```

11.2 Lists

Lists are unions of immutable values of the same type. A list can be expressed as a *sequence expression*, · list · sequence expression

Listing 11.13: The syntax for a a list using the sequence expression.

```
[[<expr>{; <expr>}]]
```

Examples are a list of integers `[1; 2; 3]`, a list of strings `["This"; "is"; "a"; "list"]`, a list of anonymous functions `[(fun x -> x); (fun x -> x*x)]`, and an empty list `[]`. Lists may also be given as ranges,

Listing 11.14: The syntax for a list using the range expressions.

```
[<expr> .. <expr> [.. <expr>]]
```

where `<expr>` in *range expressions* must be of integers, floats, or characters. Examples are `[1 .. 5]`, `[-3.0 .. 2.0]`, and `['a' .. 'z']`. Range expressions may include a step size, thus, `[1 .. 2 .. 10]` evaluates to `[1; 3; 5; 7; 9]`. · range expressions

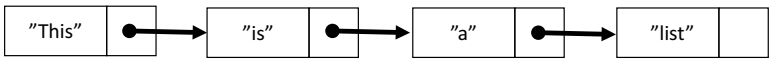

Figure 11.1: A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

A list type is identified with the *`list`* keyword, such that a list of integers has the type `int list`. Like strings, lists may be indexed using the *". []"* notation, the lengths of lists is retrieved using the *`Length`* property, and we may test whether a list is empty by using the *`isEmpty`* property. These features are demonstrated in Listing 11.15.

· `list`
· `.[]`
· `Length`
· `isEmpty`

Listing 11.15 listIndexing.fsx:
Lists are indexed as strings and has a `Length` property.

```
let printList (lst : int list) : unit =
  for i = 0 to lst.Length - 1 do
    printf "%A " lst.[i]
  printfn ""

let lst = [3; 4; 5]
printfn "lst = %A, lst.[1] = %A" lst lst.[1]
printfn "lst.Length = %A, lst.isEmpty = %A" lst.Length
  lst.IsEmpty
printList lst
```

```
$ fsharpc --nologo listIndexing.fsx && mono listIndexing.exe
lst = [3; 4; 5], lst.[1] = 4
lst.Length = 3, lst.isEmpty = false
3 4 5
```

F# implements lists as linked lists, as illustrated in Figure 11.1. As a consequence, indexing element i has *computational complexity* $\mathcal{O}(i)$. The computational complexity of an operation is a description of how long a computation will take without considering the hardware it is performed on. The notation is sometimes called *Big-O* notation. In the present case, the complexity is $\mathcal{O}(i)$, which means that the complexity is linear in i and indexing element $i+1$ takes 1 unit longer than indexing element i when i is very large. The size of the unit is on purpose unspecified and depends on implementation and hardware details. Nevertheless, Big-O notation is a useful tool for reasoning about the efficiency of an operation. F# has access to the list's elements only by traversing the list from its beginning. I.e., to obtain the value of element i, F# starts with element 0, follows the link to element 1 and so on, until element i is reached. To reach element $i+1$ instead, we would need to follow 1 more link, and assuming that following a single link takes some constant amount of time we find that the computational complexity is $\mathcal{O}(i)$. Compared to arrays, to be discussed below, this is slow, which is why **indexing lists should be avoided.**

· computational complexity
· Big-O
Advice

Notice especially that lists are zero-indexed, and thus, the last element in a list `lst` is `lst.Length -1`. This is a very common source of error! Therefore, indexing in lists using *for*-loops is supported using a special notation with the *`in`* keyword,

· for
· `in`

Listing 11.16: For-in loop with in expression.

```
for <ident> in <list> do <bodyExpr> [done]
```

In `for-in` loops, the loop runs through each element of the `<list>`, and assigns it to the identifier `<ident>`. This is demonstrated in Listing 11.17.

Listing 11.17 listFor.fsx:
The `for-in` loops are preferred over `for-to` loops.

```
let printList (lst : int list) : unit =
  for elm in lst do
    printf "%A " elm
  printfn ""

printList [3; 4; 5]
```

```
$ fsharpc --nologo listFor.fsx && mono listFor.exe
3 4 5
```

Using `for-in`-expressions remove the risk of off-by-one indexing errors, and thus, **`for-in` is to be preferred over `for-to`.** Advice

Lists support slicing identically to strings, as demonstrated in Listing 11.18.

Listing 11.18 listSlicing.fsx:
Examples of list slicing. Compare with Listing 5.27.

```
let lst = ['a' .. 'g']
printfn "%A" lst.[0]
printfn "%A" lst.[3]
printfn "%A" lst.[3..]
printfn "%A" lst.[..3]
printfn "%A" lst.[1..3]
printfn "%A" lst.[*]
```

```
$ fsharpc --nologo listSlicing.fsx && mono listSlicing.exe
'a'
'd'
['d'; 'e'; 'f'; 'g']
['a'; 'b'; 'c'; 'd']
['b'; 'c'; 'd']
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

Lists may be concatenated using either the *"`@`"*[2] *concatenation* operator or the *"`::`" cons* operators. The difference is that "`@`" concatenates two lists of identical types, while "`::`" concatenates an element and a list of identical types. This is demonstrated in Listing 11.19.

- `@`
- list concatenation
- `::`
- list cons

[2] Jon: **why does the at-symbol not appear in the index?**

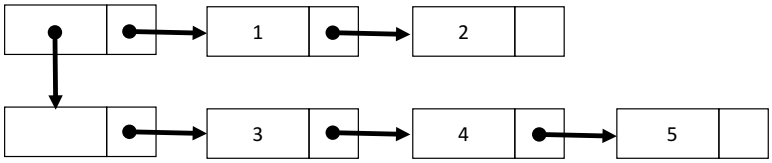

Figure 11.2: A list is a ragged linked list: Here is illustrated the linked list of `[[1;2];[3;4;5]]`.

Listing 11.19 listCon.fsx:
Examples of list concatenation.

```
printfn "[1] @ [2; 3] = %A" ([1] @ [2; 3])
printfn "[1; 2] @ [3; 4] = %A" ([1; 2] @ [3; 4])
printfn "1 :: [2; 3] = %A" (1 :: [2; 3])
```

```
$ fsharpc --nologo listCon.fsx && mono listCon.exe
[1] @ [2; 3] = [1; 2; 3]
[1; 2] @ [3; 4] = [1; 2; 3; 4]
1 :: [2; 3] = [1; 2; 3]
```

Since lists are represented as linked lists, the cons operator is very efficient and has computational complexity $\mathcal{O}(1)$, while concatenation has computational complexity $\mathcal{O}(n)$, where n is the length of the first list.

It is possible to make multidimensional lists as lists of lists, as shown in Listing 11.20.

Listing 11.20 listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

```
let a = [[1;2];[3;4;5]]
let row = a.Item 0 in printfn "%A" row
let elm = row.Item 1 in printfn "%A" elm
let elm = (a.Item 0).Item 1 in printfn "%A" elm
```

```
$ fsharpc --nologo listMultidimensional.fsx
$ mono listMultidimensional.exe
[1; 2]
2
2
```

The example shows a *ragged multidimensional list*, since each row has a different number of elements. This is also illustrated in Figure 11.2.

· ragged multidimensional list

The indexing of a particular element is slow due to the linked list implementation of lists, which is why arrays are often preferred for two- and higher-dimensional data structures, see Section 11.3.

11.2.1 List Properties

Lists support a number of properties, some of which are listed below.

`Head`: Returns the first element of a list.

Listing 11.21: `Head`

```
> [1; 2; 3].Head;;
val it : int = 1
```

`IsEmpty`: Returns true if the list is empty.

Listing 11.22: `Head`

```
> [1; 2; 3].IsEmpty;;
val it : bool = false
```

`Length`: Returns the number of elements in the list.

Listing 11.23: `Length`

```
> [1; 2; 3].Length;;
val it : int = 3
```

`Tail`: Returns the list, except for its first element.

Listing 11.24: `Tail`

```
> [1; 2; 3].Tail;;
val it : int list = [2; 3]
```

11.2.2 List Module

The built-in `List` module contains a wealth of functions for lists, some of which are briefly summarized below:

`List.collect: ('T -> 'U list) -> 'T list -> 'U list`. Applies the supplied function to each element in a list and return a concatenated list of the results.

Listing 11.25: `List.collect`

```
> List.collect (fun elm -> [elm; elm; elm]) [1; 2; 3];;
val it : int list = [1; 1; 1; 2; 2; 2; 3; 3; 3]
```

`List.contains: 'T -> 'T list -> bool`. Returns true or false depending on whether or not an element is contained in the list.

Listing 11.26: `List.contains`

```
> List.contains 3 [1; 2; 3];;
val it : bool = true
```

`List.empty`: `'T list`. An empty list of inferred type.

Listing 11.27: `List.empty`

```
> let a : int list = List.empty;;
val a : int list = []
```

`List.filter`: `('T -> bool) -> 'T list -> 'T list`. Returns a new list with all the elements of the original list for which the supplied function evaluates to true.

Listing 11.28: `List.filter`

```
> List.filter (fun x -> x % 2 = 1) [0 .. 9];;
val it : int list = [1; 3; 5; 7; 9]
```

`List.find`: `('T -> bool) -> 'T list -> 'T`. Returns the first element for which the given function is true.

Listing 11.29: `List.find`

```
> List.find (fun x -> x % 2 = 1) [0 .. 9];;
val it : int = 1
```

`List.findIndex`: `('T -> bool) -> 'T list -> int`. Returns the index of the first element for which the given function is true.

Listing 11.30: `List.findIndex`

```
> List.findIndex (fun x -> x = 'k') ['a' .. 'z'];;
val it : int = 10
```

`List.fold`: `('State -> 'T -> 'State) -> 'State -> 'T list -> 'State`. Updates an accumulator iteratively by applying the supplied function to each element in a list, e.g., for a list consisting of $x_0, x_1, x_2, \ldots, x_n$, a supplied function f, and an initial value for the accumulator s, `List.fold` calculates $f(\ldots f(f(f(s, x_0), x_1), x_2), \ldots, x_n)$.

Listing 11.31: `List.fold`

```
> let addSquares acc elm = acc + elm*elm
- List.fold addSquares 0 [0 .. 9];;
val addSquares : acc:int -> elm:int -> int
val it : int = 285
```

`List.foldBack: ('T -> 'State -> 'State) -> 'T list -> 'State -> 'State`. Updates an accumulator iteratively by applying function to each element in a list, e.g., for a list consisting of $x_0, x_1, x_2, \ldots, x_n$, a supplied function f, and an initial value for the accumulator s, `List.foldBack` calculates $f(x_0, f(x_1, f(x_2, \ldots, f(x_n, s))))$.

Listing 11.32: `List.foldBack`

```
> let addSquares elm acc = acc + elm*elm
- List.foldBack addSquares [0 .. 9] 0;;
val addSquares : elm:int -> acc:int -> int
val it : int = 285
```

`List.forall: ('T -> bool) -> 'T list -> bool`. Tests if all elements in a list satisfy the given predicate.

Listing 11.33: `List.forall`

```
> List.forall (fun x -> x % 2 = 1) [0 .. 9];;
val it : bool = false
```

`List.head: 'T list -> int`. The first element in the list. Exception if empty.

Listing 11.34: `List.head`

```
> List.head [1; -2; 0];;
val it : int = 1
```

`List.isEmpty: 'T list -> int`. Compare with the empty list

Listing 11.35: `List.isEmpty`

```
> List.isEmpty [1; 2; 3];;
val it : bool = false

> let a = [1; 2; 3] in List.isEmpty a;;
val it : bool = false
```

`List.iter: ('T -> unit) -> 'T list -> unit`. Applies a procedure to every element in the list.

Listing 11.36: `List.iter`

```
> List.iter (fun x -> printfn "%A " x) [0; 1; 2];;
0
1
2
val it : unit = ()
```

`List.map: ('T -> 'U) -> 'T list -> 'U list`. Returns a list where the supplied function has been applied to every element.

Listing 11.37: `List.map`

```
> List.map (fun x -> x*x) [0 .. 9];;
val it : int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

`List.ofArray: 'T list -> int`. Returns a list whose elements are the same as the supplied array.

Listing 11.38: `List.ofArray`

```
> List.ofArray [|1; 2; 3|];;
val it : int list = [1; 2; 3]
```

`List.rev: 'T list -> 'T list`. Returns a list whose elements have been reversed.

Listing 11.39: `List.rev`

```
> List.rev [1; 2; 3];;
val it : int list = [3; 2; 1]
```

`List.sort: 'T list -> 'T list`. Returns a list whose elements have been sorted.

Listing 11.40: `List.sort`

```
> List.sort [3; 1; 2];;
val it : int list = [1; 2; 3]
```

`List.tail: 'T list -> 'T list`. The list except for its first element. Exception if empty.

Listing 11.41: `List.tail`

```
> List.tail [1; 2; 3];;
val it : int list = [2; 3]

> let a = [1; 2; 3] in List.tail a;;
val it : int list = [2; 3]
```

`List.toArray: 'T list -> 'T []`. Returns an array whose elements are the same as the supplied list.

Listing 11.42: `List.toArray`

```
> List.toArray [1; 2; 3];;
val it : int [] = [|1; 2; 3|]
```

`List.unzip: ('T1 * 'T2) list -> 'T1 list * 'T2 list`. Returns a pair of lists, whose elements are taken from pairs of a list.

Listing 11.43: `List.unzip`

```
> List.unzip [(1, 'a'); (2, 'b'); (3, 'c')];;
val it : int list * char list = ([1; 2; 3], ['a'; 'b';
  'c'])

>
```

`List.zip: 'T1 list -> 'T2 list -> ('T1 * 'T2) list`. Returns a list of pairs, whose elements are taken iteratively from two lists.

Listing 11.44: `List.zip`

```
> List.zip [1; 2; 3] ['a'; 'b'; 'c'];;
val it : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
```

11.3 Arrays

One dimensional *arrays* or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Arrays can be stated as *sequence expressions*,

· arrays
· sequence expressions

Listing 11.45: Arrays with a sequence expression.

```
[|[<expr>{; <expr>}]|]
```

Examples are arrays of integers `[|1; 2; 3|]`, of strings `[|"This"; "is"; "an"; "array"|]`, of functions `[|(fun x -> x); (fun x -> x*x)|]`, and an empty array `[||]`. Arrays may also be given as ranges,

Listing 11.46: Arrays with a range expressions.

```
[|<expr> .. <expr> [.. <expr>]|]
```

but arrays of *range expressions* must be of `<expr>` integers, floats, or characters. Examples are `[|1 .. 5|]`, `[|-3.0 .. 2.0|]`, and `[|'a' .. 'z'|]`. Range expressions may include a step size, thus, `[|1 .. 2 .. 10|]` evaluates to `[|1; 3; 5; 7; 9|]`.

· range expressions

The array type is defined using the `array` keyword or alternatively the "`[]`" lexeme. Like strings and lists, arrays may be indexed using the *". []"* notation. Arrays cannot be resized, but are mutable as shown in Listing 11.47.

· `. []`

Listing 11.47 arrayReassign.fsx:
Arrays are mutable in spite the missing `mutable` keyword.

```
let square (a : int array) =
  for i = 0 to a.Length - 1 do
    a.[i] <- a.[i] * a.[i]

let A = [| 1; 2; 3; 4; 5 |]
printfn "%A" A
square A
printfn "%A" A
```

```
$ fsharpc --nologo arrayReassign.fsx && mono arrayReassign.exe
[|1; 2; 3; 4; 5|]
[|1; 4; 9; 16; 25|]
```

Notice that in spite the missing `mutable` keyword, the function `square` still had the *side-effect* of squaring all entries in `A`. F# implements arrays as chunks of memory and indexes arrays via address arithmetic. I.e., element i in an array, whose first element is in memory address α and whose elements fill β addresses each is found at address $\alpha + i\beta$.[3] Hence, indexing has computational complexity of $\mathcal{O}(1)$, but appending and prepending values to arrays and array concatenation requires copying the new and existing values to a fresh area in memory and thus has computational complexity $\mathcal{O}(n)$, where n is the total number of elements. Thus, **indexing arrays is fast, but cons and concatenation is slow and should be avoided.**

· side-effect

Advice

Arrays support *slicing*, that is, indexing an array with a range results in a copy of the array with values corresponding to the range. This is demonstrated in Listing 11.48.

· slicing

Listing 11.48 arraySlicing.fsx:
Examples of array slicing. Compare with Listing 11.18 and Listing 5.27.

```
let arr = [|'a' .. 'g'|]
printfn "%A" arr.[0]
printfn "%A" arr.[3]
printfn "%A" arr.[3..]
printfn "%A" arr.[..3]
printfn "%A" arr.[1..3]
printfn "%A" arr.[*]
```

```
$ fsharpc --nologo arraySlicing.fsx && mono arraySlicing.exe
'a'
'd'
[|'d'; 'e'; 'f'; 'g'|]
[|'a'; 'b'; 'c'; 'd'|]
[|'b'; 'c'; 'd'|]
[|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

As illustrated, the missing start or end index implies from the first or to the last element.

Arrays have explicit operator support for appending and concatenation, instead the `Array`

[3] Jon: **Add a figure illustrating address indexing.**

namespace includes an `Array.append` function, as shown in Listing 11.49.

Listing 11.49 arrayAppend.fsx: Two arrays are appended with `Array.append`.

```
let a = [|1; 2;|]
let b = [|3; 4; 5|]
let c = Array.append a b
printfn "%A, %A, %A" a b c
```

```
$ fsharpc --nologo arrayAppend.fsx && mono arrayAppend.exe
[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
```

Arrays are *reference types*, meaning that identifiers are references and thus suffers from aliasing, as illustrated in Listing 11.50.

· reference types

Listing 11.50 arrayAliasing.fsx: Arrays are reference types and suffer from aliasing.

```
let a = [|1; 2; 3|];
let b = a
a.[0] <- 0
printfn "a = %A, b = %A" a b;;
```

```
$ fsharpc --nologo arrayAliasing.fsx && mono arrayAliasing.exe
a = [|0; 2; 3|], b = [|0; 2; 3|]
```

11.3.1 Array properties and methods

Arrays support a number of properties and methods, i.e., values and functions that are attached to each array and access using the "." notation, some of which are:

`Clone()`: Returns a copy of the array.

Listing 11.51: `Clone`

```
> let a = [|1; 2; 3|];
- let b = a.Clone()
- a.[0] <- 0
- printfn "a = %A, b = %A" a b;;
a = [|0; 2; 3|], b = [|1; 2; 3|]
val a : int [] = [|0; 2; 3|]
val b : obj = [|1; 2; 3|]
val it : unit = ()
```

`Length`: Returns the number of elements in the array.

Listing 11.52: `Length`

```
> [|1; 2; 3|].Length;;
val it : int = 3
```

11.3.2 Array module

There are quite a number of built-in procedures for arrays in the `Array` module, some of which are summarized below.[4]

`Array.append: 'T [] -> 'T [] -> 'T []`. Creates an array that contains the elements of one array followed by the elements of another array.

Listing 11.53: `Array.append`

```
> Array.append [|1; 2;|] [|3; 4; 5|];;
val it : int [] = [|1; 2; 3; 4; 5|]
```

`Array.compareWith: ('T -> 'T -> int) -> 'T [] -> 'T [] -> int`. Compares two arrays using the given comparison function, element by element.

Listing 11.54: `Array.compareWith`

```
> let compArr elm1 elm2 =
-     if elm1 > elm2 then 1
-     elif elm1 < elm2 then -1
-     else 0
- Array.compareWith compArr [|1; 2; 4|] [|1; 2; 3|];;
val compArr : elm1:'a -> elm2:'a -> int when 'a :
   comparison
val it : int = 1
```

`Array.concat: seq<'T []> -> 'T []`. Creates an array that contains the elements of each of the supplied sequence of arrays.

Listing 11.55: `Array.concat`

```
> Array.concat [[|1; 2; 3|]; [|4; 5|]; [|6; 7; 8|]];;
val it : int [] = [|1; 2; 3; 4; 5; 6; 7; 8|]
```

`Array.contains: .` Evaluates to true if the given element is in the input array.

Listing 11.56: `Array.contains`

```
> Array.contains 3 [|1; 2; 3|];;
val it : bool = true
```

[4] Jon: **rewrite description**

`Array.copy: 'T [] -> 'T []`. Creates an array that contains the elements of the supplied array.

Listing 11.57: `Array.copy`

```
> let a = [|1; 2; 3|]
- let b = Array.copy a;;
val a : int [] = [|1; 2; 3|]
val b : int [] = [|1; 2; 3|]
```

`Array.create: int -> 'T -> 'T []`. Creates an array whose elements are initiallized the supplied value.

Listing 11.58: `Array.create`

```
> Array.create 4 3.14;;
val it : float [] = [|3.14; 3.14; 3.14; 3.14|]
```

`Array.empty: 'T []`. Returns an empty array of the given type.

Listing 11.59: `Array.empty`

```
> let a : int [] = Array.empty;;
val a : int [] = [||]
```

`Array.exists: ('T -> bool) -> 'T [] -> bool`. Tests whether any element of an array satisfies the supplied predicate.

Listing 11.60: `Array.exists`

```
> let odd x = (x % 2 = 1) in Array.exists odd [|0 .. 2 ..
  4|];;
val it : bool = false
```

`Array.fill: 'T [] -> int -> int -> 'T -> unit`. Fills a range of elements of an array with the supplied value.

Listing 11.61: `Array.fill`

```
> let arr = Array.zeroCreate 10;
- Array.fill arr 2 5 2;;
val arr : int [] = [|0; 0; 2; 2; 2; 2; 2; 0; 0; 0|]
val it : unit = ()
```

`Array.filter: ('T -> bool) -> 'T [] -> 'T []`. Returns a collection that contains only the elements of the supplied array for which the supplied condition returns true.

Listing 11.62: `Array.filter`

```
> let odd x = (x % 2 = 1) in Array.filter odd [|0 .. 9|];;
val it : int [] = [|1; 3; 5; 7; 9|]
```

`Array.find: ('T -> bool) -> 'T [] -> 'T`. Returns the first element for which the supplied function returns true. Raises `System.Collections.Generic.KeyNotFoundException`

Listing 11.63: `Array.find`

```
> let odd x = (x % 2 = 1) in Array.find odd [|0 .. 9|];;
val it : int = 1
```

`Array.findIndex: ('T -> bool) -> 'T [] -> int`. Returns the index of the first element in an array that satisfies the supplied condition. Raises `System.Collections.Generic.KeyNotFoundExc` if none of the elements satisfy the condition.

Listing 11.64: `Array.findIndex`

```
> let isK x = (x = 'k') in Array.findIndex isK [|'a' ..
   'z'|];;
val it : int = 10
```

`Array.fold: ('State -> 'T -> 'State) -> 'State -> 'T [] -> 'State`. Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f (...(f s i0)...) iN.

Listing 11.65: `Array.fold`

```
> let addSquares acc elm = acc + elm*elm
- Array.fold addSquares 0 [|0 .. 9|];;
val addSquares : acc:int -> elm:int -> int
val it : int = 285
```

`Array.foldBack: ('T -> 'State -> 'State) -> 'T [] -> 'State -> 'State`. Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f i0 (...(f iN s)).

Listing 11.66: `Array.foldBack`

```
> let addSquares elm acc = acc + elm*elm
- Array.foldBack addSquares [|0 .. 9|] 0;;
val addSquares : elm:int -> acc:int -> int
val it : int = 285
```

`Array.forall: ('T -> bool) -> 'T [] -> bool`. Tests whether all elements of an array

satisfy the supplied condition.

Listing 11.67: `Array.forall`

```
> let odd x = (x % 2 = 1) in Array.forall odd [|0 .. 9|];;
val it : bool = false
```

`Array.get: 'T [] -> int -> 'T`. Gets an element from an array.

Listing 11.68: `Array.get`

```
> Array.get [|1; 2; 3|] 2;;
val it : int = 3
```

`Array.init: int -> (int -> 'T) -> 'T []`. Uses a supplied function to create an array of the supplied dimension.

Listing 11.69: `Array.init`

```
> let squareIndex ind = ind*ind
- Array.init 10 squareIndex;;
val squareIndex : ind:int -> int
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.isEmpty: 'T [] -> bool`. Tests whether an array has any elements.

Listing 11.70: `Array.isEmpty`

```
> Array.isEmpty [||];;
val it : bool = true
```

`Array.iter: ('T -> unit) -> 'T [] -> unit`. Applies the supplied function to each element of an array.

Listing 11.71: `Array.iter`

```
> let prt x = printfn "%A " x in Array.iter prt [|0; 1;
  2|];;
0
1
2
val it : unit = ()
```

`Array.length: 'T [] -> int`. Returns the length of an array. The `System.Array.Length` property does the same thing.

Listing 11.72: `Array.length`

```
> let a = [|1; 2; 3|] in a.Length;;
val it : int = 3
```

`Array.map: ('T -> 'U) -> 'T [] -> 'U []`. Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.

Listing 11.73: `Array.map`

```
> let square x = x*x in Array.map square [|0 .. 9|];;
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.ofList: 'T list -> 'T []`. Creates an array from the supplied list.

Listing 11.74: `Array.ofList`

```
> Array.ofList [1; 2; 3];;
val it : int [] = [|1; 2; 3|]
```

`Array.rev: 'T [] -> 'T []`. Reverses the order of the elements in a supplied array.

Listing 11.75: `Array.rev`

```
> Array.rev [|1; 2; 3|];;
val it : int [] = [|3; 2; 1|]
```

`Array.set: 'T [] -> int -> 'T -> unit`. Sets an element of an array.

Listing 11.76: `Array.set`

```
> let arr = [|1; 2; 3|]
- Array.set arr 2 10
- printfn "%A" arr;;
[|1; 2; 10|]
val arr : int [] = [|1; 2; 10|]
val it : unit = ()
```

`Array.sort: 'T[] -> 'T []`. Sorts the elements of an array and returns a new array. `Operators.compare` is used to compare the elements.

Listing 11.77: `Array.sort`

```
> Array.sort [|3; 1; 2|];;
val it : int [] = [|1; 2; 3|]
```

`Array.sub: 'T [] -> int -> int -> 'T []`. Creates an array that contains the supplied subrange, which is specified by starting index and length.

Listing 11.78: `Array.sub`

```
> Array.sub [|0..9|] 2 5;;
val it : int [] = [|2; 3; 4; 5; 6|]
```

`Array.toList: 'T [] -> 'T list`. Converts the supplied array to a list.

Listing 11.79: `Array.toList`

```
> Array.toList [|1; 2; 3|];;
val it : int list = [1; 2; 3]
```

`Array.unzip: ('T1 * 'T2) [] -> 'T1 [] * 'T2 []`. Splits an array of tuple pairs into a tuple of two arrays.

Listing 11.80: `Array.unzip`

```
> Array.unzip [|(1, 'a'); (2, 'b'); (3, 'c')|];;
val it : int [] * char [] = ([|1; 2; 3|], [|'a'; 'b';
  'c'|])
```

`Array.zip: 'T1 [] -> 'T2 [] -> ('T1 * 'T2) []`. Combines three arrays into an array of tuples that have three elements. The three arrays must have equal lengths; otherwise, `System.ArgumentException` is raised.

Listing 11.81: `Array.zip`

```
> Array.zip [|1; 2; 3|] [|'a'; 'b'; 'c'|];;
val it : (int * char) [] = [|(1, 'a'); (2, 'b'); (3, 'c')|]
```

11.4 Multidimensional arrays

Multidimensional arrays can be created as arrays of arrays (of arrays ...). These are known as *jagged arrays* since there is no inherent guarantee that all sub-arrays are of the same size. E.g., the example in Listing 11.82 is a jagged array of increasing width.

- multidimensional arrays
- jagged arrays

Listing 11.82 arrayJagged.fsx:
An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

```
let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

for row in arr do
  for elm in row do
    printf "%A " elm
  printf "\n"
```

```
$ fsharpc --nologo arrayJagged.fsx && mono arrayJagged.exe
1
1 2
1 2 3
```

Indexing arrays of arrays is done sequentially, in the sense that in the above example, the number of outer arrays is `a.Length`, `a.[i]` is the i'th array, the length of the i'th array is `a.[i].Length`, and the j'th element of the i'th array is thus `a.[i].[j]`. Often 2-dimensional rectangular arrays are used, which can be implemented as a jagged array as shown in Listing 11.83.

Listing 11.83 arrayJaggedSquare.fsx:
A rectangular array.

```
let pownArray  (arr : int array array) p =
  for i = 1 to arr.Length - 1 do
    for j = 1 to arr.[i].Length - 1 do
      arr.[i].[j] <- pown arr.[i].[j] p

let printArrayOfArrays (arr : int array array) =
  for row in arr do
    for elm in row do
      printf "%3d " elm
    printf "\n"

let A = [|[|1 .. 4|]; [|1 .. 2 .. 7|]; [|1 .. 3 .. 10|]|]
pownArray A 2
printArrayOfArrays A
```

```
$ fsharpc --nologo arrayJaggedSquare.fsx && mono
  arrayJaggedSquare.exe
  1   2   3   4
  1   9  25  49
  1  16  49 100
```

Notice, the `for-in` cannot be used in `pownArray`, e.g.,

```
for row in arr do for elm in row do elm <- pown elm p done done,
```

since the iterator value `elm` is not mutable even though `arr` is an array. In fact, square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. In the following, we describe *`Array2D`*. The workings of *`Array3D`* and *`Array4D`* are very

· `Array2D`
· `Array3D`
· `Array4D`

similar. An example of creating the same 2-dimensional array as above but as an `Array2D` is shown in Listing 11.84.

Listing 11.84 array2D.fsx:
Creating a 3 by 4 rectangular arrays of intigers.

```
let arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
  for j = 0 to (Array2D.length2 arr) - 1 do
    arr.[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr
```

```
$ fsharpc --nologo array2D.fsx && mono array2D.exe
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Notice that the indexing uses a slightly different notation `[,]` and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case, 12. As can be seen, the `printf` supports direct printing of the 2-dimensional array. Higher dimensional arrays support slicing as shown in Listing 11.85.

Listing 11.85 array2DSlicing.fsx:
Examples of Array2D slicing. Compare with Listing 11.84.

```
let arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
  for j = 0 to (Array2D.length2 arr) - 1 do
    arr.[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr.[2,3]
printfn "%A" arr.[1..,3..]
printfn "%A" arr.[..1,*]
printfn "%A" arr.[1,*]
printfn "%A" arr.[1..1,*]
```

```
$ fsharpc --nologo array2DSlicing.fsx && mono
   array2DSlicing.exe
11
[[10]
 [11]]
[[0; 3; 6; 9]
 [1; 4; 7; 10]]
[|1; 4; 7; 10|]
[[1; 4; 7; 10]]
```

Note that in almost all cases, slicing produces a sub rectangular 2 dimensional array except for `arr.[1,*]`, which is an array, as can be seen by the single "`[`". In contrast, `A.[1..1,*]` is an Array2D. Note also, that `printfn` typesets 2 dimensional arrays as `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion with lists of lists. [5]

[5] Jon: **Array2D.ToString produces `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion.**

Multidimensional arrays have the same properties and methods as arrays, see Section 11.3.1.

11.4.1 Array2D module

There are quite a number of built-in procedures for arrays in the `Array2D` namespace, some of which are summarized below.[6]

`copy: 'T [,] -> 'T [,]`. Creates a new array whose elements are the same as the input array.

Listing 11.86: `Array2D.copy`

```
> let a = Array2D.init 3 4 (fun i j -> i + 10 * j)
- let b = Array2D.copy a;;
val a : int [,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
val b : int [,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
```

`create: int -> int -> 'T -> 'T [,]`. Creates an array whose elements are all initially the given value.

Listing 11.87: `Array2D.create`

```
> Array2D.create 2 3 3.14;;
val it : float [,] = [[3.14; 3.14; 3.14]
                      [3.14; 3.14; 3.14]]
```

`get: 'T [,] -> int -> int -> 'T`. Fetches an element from a 2D array. You can also use the syntax `array.[index1,index2]`.

Listing 11.88: `Array2D.get`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.get arr 1 2;;
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 21; 31]
                     [2; 12; 22; 32]]
val it : int = 21
```

`init: int -> int -> (int -> int -> 'T) -> 'T [,]`. Creates an array given the dimensions and a generator function to compute the elements.

[6] Jon: **rewrite description**

Listing 11.89: `Array2D.init`

```
> let idxFct i j = i + 10 * j
- Array2D.init 3 4 idxFct;;
val idxFct : i:int -> j:int -> int
val it : int [,] = [[0; 10; 20; 30]
                    [1; 11; 21; 31]
                    [2; 12; 22; 32]]
```

`iter: ('T -> unit) -> 'T [,] -> unit`. Applies the given function to each element of the array.

Listing 11.90: `Array2D.iter`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.iter (fun elm -> printf "%A " elm) arr
- printfn "";;
0 10 20 30 1 11 21 31 2 12 22 32
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 21; 31]
                     [2; 12; 22; 32]]
val it : unit = ()
```

`length1: 'T [,] -> int`. Returns the length of an array in the first dimension.

Listing 11.91: `Array2D.length1`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length1 arr;;
val it : int = 2
```

`length2: 'T [,] -> int`. Returns the length of an array in the second dimension.

Listing 11.92: `Array2D.forall length2`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length2 arr;;
val it : int = 3
```

`map: ('T -> 'U) -> 'T [,] -> 'U [,]`. Creates a new array whose elements are the results of applying the given function to each of the elements of the array.

Listing 11.93: `Array2D.map`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- let square x = x*x in Array2D.map square arr;;
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 21; 31]
                     [2; 12; 22; 32]]
val it : int [,] = [[0; 100; 400; 900]
                    [1; 121; 441; 961]
                    [4; 144; 484; 1024]]
```

`set: 'T [,] -> int -> int -> 'T -> unit`. Sets the value of an element in an array. You can also use the syntax `array.[index1,index2] <- value`.

Listing 11.94: `Array2D.set`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.set arr 1 2 100
- printfn "%A" arr;;
[[0; 10; 20; 30]
 [1; 11; 100; 31]
 [2; 12; 22; 32]]
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 100; 31]
                     [2; 12; 22; 32]]
val it : unit = ()
```

12 | The imperative programming paradigm

Imperative programming is a paradigm for programming states. In imperative programming, the focus is on how a problem is to be solved as a list of *statements* that affects *states*. In F# states are mutable and immutable values, and they are affected by functions and procedures. An imperative program is typically identified as using

- Imperative programming
- statements
- states
- mutable values

mutable values
: Mutable values are holders of state, they may change over time, and thus have a dynamic scope.

- procedures

Procedures
: Procedures are functions that returns "`()`", instead of functions that transform data. They are the embodiment of side-effects.

- side-effects

Side-effects
: Side-effects are changes of state that are not reflected in the arguments and return values of a function. The `printf` is an example of a procedure that uses side-effects to communicate with the terminal.

- `for`
- `while`

Loops
: The `for`- and `while`-loops typically uses an iteration value to update some state, e.g., `for`-loops are often used to iterate through a list and summarize its content.

In contrast, mono state or stateless programs as *functional programming* can be seen as a subset of imperative programming and is discussed in Chapter 17. *Object oriented programming* is an extension of imperative programming, where statements and states are grouped into classes. For a discussion on object-oriented programming, see Chapter 22.

- functional programming
- Object oriented programming

Imperative programs are like Turing machines, a theoretical machine introduced by Alan Turing in 1936 [10]. Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming languages, and almost all machine languages follow the imperative programming paradigm.

- machine code

A prototypical example is a baking recipe, e.g., to make a loaf of bread do the following:

1. Mix yeast with water
2. Stir in salt, oil, and flour
3. Knead until the dough has a smooth surface
4. Let the dough rise until it has double size
5. Shape dough into a loaf

6. Let the loaf rise until double size

7. Bake in the oven until the bread is golden brown

Each line in this example consists of one or more statements that are to be executed, and while executing them, states such as the size of the dough and the color of the bread changes. Some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

12.1 Imperative design

Programming is the act of solving a problem by writing a program to be executed on a computer. And imperative programming focusses on states. To solve a problem, you could work through the following list of actions

1. Understand the problem. As Pólya described it, see Chapter 2, the first step in any solution is to understand the problem. A good trick to check, whether you understand the problem is to briefly describe it in your own words.

2. Identify the main values, variables, functions, and procedures needed. If the list of procedures is large, then you most likely should organize them in modules. It is also useful to start in a coarse to fine manner.

3. For each function and procedure, write a precise description of what it should do. This can conveniently be performed as an in-code comment for the procedure using the F# XML documentation standard.

4. Make mockup functions and procedures using the intended types, but don't necessarily compute anything sensible. Run through examples in your mind, using this mockup program to identify any obvious oversights.

5. Write a suite of unit-tests that tests the basic requirements for your code. The unit tests should be runnable with your mockup code. Writing unit-tests will also allow you to evaluate the usefulness of the code pieces as seen from an application point of view.

6. Replace the mockup functions in a prioritized order, i.e., write the must-have code before you write the nice-to-have code, while regularly running your unit-tests to keep track on your progress.

7. Evaluate the code in relation to the desired goal and reiterate earlier actions as needed until the task has been sufficiently completed.

8. Complete your documentation both in-code and outside to ensure that the intended user has sufficient knowledge to effectively use your program and to ensure that you or a fellow programmer will be able to maintain and extend the program in the future.

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

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

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

[8] 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/`.

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

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index