1 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, and arrays. Strings were discussed in **??** and will be revisited here in more details.

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 **??**. 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 **??**–**??**, while classes are designed to support object-oriented programming, see **??**–**??**.

In the following, a brief overview of many properties, methods, and functions is given by describing their name and type-definition, and by giving a short description and an example of their use. Several definitions are general and works with many different types. To describe this we will use the notation of generic types, see **??**. The name of a generic type starts with the "`'`" lexeme, such as `'T`. The implication of the appearance of a generic type in, e.g., a function's type-definition, is that the function may be used with any real type such as `int` or `char`. If the same generic type name is used in several places in the type-definition, then the function must use a real type consistently. For example, The `List.fromArray` function has type `arr:'T [] -> 'T list`, meaning that it takes an array of some type and returns a list of the same type.

See the F# Language Reference at `https://docs.microsoft.com/en-us/dotnet/fsharp/` for a full description of all available functionality including variants of those included here.

1.1 Strings

Strings have been discussed in **??**, 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 **??** 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 **??**), which are replaced by the corresponding char- · string · System.string@Syste acter 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"`, `@"\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 ??.

· verbatim string

1.1.1 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()`: `str:string -> int`. Returns the index of the first occurence of `s` or -1, if `str` does not appear in the string.

Listing 1.1: `IndexOf()`

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

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

Listing 1.2: `Length`

```
> "abcd".Length;;
val it : int = 4
```

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

Listing 1.3: `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 1.4: `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 1.5: `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 1.3 for more information about arrays.

Listing 1.6: `Split()`

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

1.1.2 The 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: f:(char -> string) -> str:string -> string`. Creates a new string whose characters are the results of applying `f` to each of the characters of `str` and concatenating the resulting strings.

Listing 1.7: `String.collect`

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

`String.exists: f:(char -> bool) -> str:string -> bool`. Returns true if any character in `str` evaluates to true when using `f`.

Listing 1.8: `String.exists`

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

`String.forall: f:(char -> bool) -> str:string -> bool`. Returns true if all characters in `str` evalutes to true when using `f`.

Listing 1.9: `String.forall`

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

`String.init: n:int -> f:(int -> string) -> string`. Creates a new string with length `n` and whose characters are the result of applying `f` to each index of that string.

Listing 1.10: `String.init`

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

`String.iter: f:(char -> unit) -> str:string -> unit`. Applies `f` to each character in `str`.

Listing 1.11: `String.iter`

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

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

Listing 1.12: `String.map`

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

1.2 Lists

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

· list
· sequence expression

Listing 1.13: The syntax for 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 1.14: 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

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 1.15.

· list@`list`
· .[]@`.[]`
· Length@`Length`
· IsEmpty@`IsEmpty`

Listing 1.15 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 1.1. As a consequence, indexing

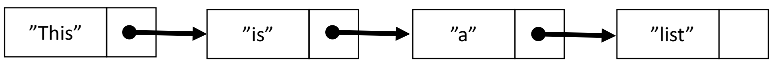

Figure 1.1: A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

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 or *Landau 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
· Landau notation

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@`in`

Listing 1.16: 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 1.17.

Listing 1.17 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 1.18.

Listing 1.18: Examples of list slicing. Compare with ??.

```
> let lst = ['a' .. 'g'];;
val lst : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']

> lst.[0];;
val it : char = 'a'

> lst.[3];;
val it : char = 'd'

> lst.[3..];;
val it : char list = ['d'; 'e'; 'f'; 'g']

> lst.[..3];;
val it : char list = ['a'; 'b'; 'c'; 'd']

> lst.[1..3];;
val it : char list = ['b'; 'c'; 'd']

> lst.[*];;
val it : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

Lists may be concatenated using either the *"@" 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 1.19.

· @@@
· list concatenation
· ::@::
· list cons

Listing 1.19: Examples of list concatenation.

```
> ([1] @ [2; 3]);;
val it : int list = [1; 2; 3]

> ([1; 2] @ [3; 4]);;
val it : int list = [1; 2; 3; 4]

> (1 :: [2; 3]);;
val it : int list = [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 1.20.

Listing 1.20 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 1.2. · ragged multidimensional list

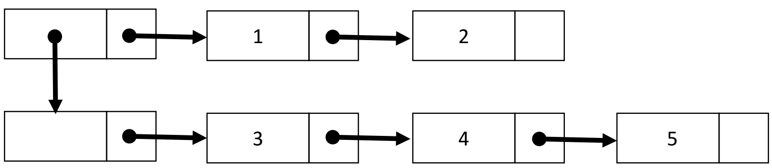

Figure 1.2: A list is a ragged linked list: Here is illustrated the linked list of `[[1;2];[3;4;5]]`.

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 1.3.

1.2.1 List Properties

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

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

Listing 1.21: `Head`

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

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

Listing 1.22: `IsEmpty`

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

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

Listing 1.23: `Length`

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

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

Listing 1.24: `Tail`

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

1.2.2 The List Module

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

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

Listing 1.25: `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: elm:'T -> lst:'T list -> bool`. Returns true or false depending on whether or not `elm` is contained in `lst`.

Listing 1.26: `List.contains`

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

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

Listing 1.27: `List.filter`

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

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

Listing 1.28: `List.find`

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

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

Listing 1.29: `List.findIndex`

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

`List.fold: f:('State -> 'T -> 'State) -> elm:'State -> lst:'T list -> 'State`. Updates an accumulator iteratively by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For example, when `lst` consists of `n+1` elements `List.fold` calculates:

`f (... (f (f elm lst.[0]) lst.[1]) ...) lst.[n]`.

Listing 1.30: `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: f:('T -> 'State -> 'State) -> lst:'T list -> elm:'State -> 'State`. Updates an accumulator iteratively backwards by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For exampel, when `lst` consists of `n+1` elements `List.foldBack` calculates:

`f lst.[0] (f lst.[1] (...(f lst.[n] elm) ...))`.

Listing 1.31: `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: f:('T -> bool) -> lst:'T list -> bool`. Returns true if all elements in `lst` are true when `f` is applied to them.

Listing 1.32: `List.forall`

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

`List.head: lst:'T list -> int`. Returns the first element in `lst`. An exception is raised if `lst` is empty. See **??** for more on exceptions.

Listing 1.33: `List.head`

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

`List.init: m:int -> f:(int -> 'T) -> 'T list`. Create a list with `m` elements and whose value is the result of applying `f` to the index of the element.

Listing 1.34: `List.init`

```
> List.init 10 (fun i -> i * i);;
val it : int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

`List.isEmpty: lst:'T list -> bool`. Returns true if `lst` is empty.

Listing 1.35: `List.isEmpty`

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

`List.iter: f:('T -> unit) -> lst:'T list -> unit`. Applies `f` to every element in `lst`.

Listing 1.36: `List.iter`

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

`List.map: f:('T -> 'U) -> lst:'T list -> 'U list`. Returns a list as a concatenation of applying `f` to every element of `lst`.

Listing 1.37: `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: arr:'T [] -> 'T list`. Returns a list whose elements are the same as `arr`. See Section 1.3 for more on arrays.

Listing 1.38: `List.ofArray`

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

`List.rev: lst:'T list -> 'T list`. Returns a new list with the same elements as in `lst` but in reversed order.

Listing 1.39: `List.rev`

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

`List.sort: lst:'T list -> 'T list`. Returns a new list with the same elements as in `lst` but where the elements are sorted.

Listing 1.40: `List.sort`

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

`List.tail: 'T list -> 'T list`. Returns a new list identical to `lst` but without its first element. An Exception is raised if `lst` is empty. See **??** for more on exceptions.

Listing 1.41: `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: lst:'T list -> 'T []`. Returns an array whose elements are the same as `lst`. See Section 1.3 for more on arrays.

Listing 1.42: `List.toArray`

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

`List.unzip: lst:('T1 * 'T2) list -> 'T1 list * 'T2 list`. Returns a pair of lists of all the first elements and all the second elements of `lst`, respectively.

Listing 1.43: `List.unzip`

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

>
```

`List.zip: lst1:'T1 list -> lst2:'T2 list -> ('T1 * 'T2) list`. Returns a list of pairs, where elements in `lst1` and `lst2` are iteratively paired.

Listing 1.44: `List.zip`

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

1.3 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 a *sequence expression*,

· arrays
· sequence expression

Listing 1.45: The syntax for an array using the 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 1.46: The syntax for an array using the range expression.

```
[|<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 1.47.

Listing 1.47 arrayReassign.fsx:
Arrays are mutable in spite of 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 of the missing `mutable` keyword, the function `square` still has 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 $\alpha + i\beta$. 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 result in a copy of the array with values corresponding to the range. This is demonstrated in Listing 1.48.

· slicing

Listing 1.48: Examples of array slicing. Compare with Listing 1.18 and ??.

```
> let arr = [|'a' .. 'g'|];;
val arr : char [] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]

> arr.[0];;
val it : char = 'a'

> arr.[3];;
val it : char = 'd'

> arr.[3..];;
val it : char [] = [|'d'; 'e'; 'f'; 'g'|]

> arr.[..3];;
val it : char [] = [|'a'; 'b'; 'c'; 'd'|]

> arr.[1..3];;
val it : char [] = [|'b'; 'c'; 'd'|]

> arr.[*];;
val it : char [] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

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

Arrays do not have explicit operator support for appending and concatenation, instead the `Array` namespace includes an `Array.append` function, as shown in Listing 1.49.

Listing 1.49 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 suffer from aliasing, as illustrated in Listing 1.50. · reference types

Listing 1.50 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|]
```

1.3.1 Array Properties and Methods

Some important properties and methods for arrays are:

`Clone(): 'T []`. Returns a copy of the array.

Listing 1.51: `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: int`. Returns the number of elements in the array.

Listing 1.52: Length

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

1.3.2 The Array Module

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

`Array.append: arr1:'T [] -> arr2:'T [] -> 'T []`. Creates an new array whose elements are a concatenated copy of `arr1` and `arr2`.

Listing 1.53: Array.append

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

`Array.contains: elm:'T -> arr:'T [] -> bool`. Returns true if `arr` contains `elm`.

Listing 1.54: Array.contains

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

`Array.exists: f:('T -> bool) -> arr:'T [] -> bool`. Returns true if any application of `f` evaluates to true when applied to the elements of `arr`.

Listing 1.55: Array.exists

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

`Array.filter: f:('T -> bool) -> arr:'T [] -> 'T []`. Returns an array of elements from `arr` who evaluate to true when `f` is applied to them.

Listing 1.56: Array.filter

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

`Array.find: f:('T -> bool) -> arr:'T [] -> 'T`. Returns the first element in `arr` for which `f` evaluates to true. The `System.Collections.Generic.KeyNotFoundException` exception is raised if no element is found. See **??** for more on exceptions.

Listing 1.57: `Array.find`

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

`Array.findIndex: f:('T -> bool) -> arr:'T [] -> int`. Returns the index of the first element in in `arr` for which `f` evaluates to true. If none are found, then the `System.Collections.Generic.KeyNotFoundException` exception is raised. See ?? for more on exceptions.

Listing 1.58: `Array.findIndex`

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

`Array.fold: f:('State -> 'T -> 'State) -> elm:'State -> arr:'T [] -> 'State`. Updates an accumulator iteratively by applying `f` to each element in `arr`. The initial value of the accumulator is `elm`. For example, when `arr` consists of `n+1` elements `Array.fold` calculates:

`f (... (f (f elm arr.[0]) arr.[1]) ...) arr.[n]`.

Listing 1.59: `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: f:('T -> 'State -> 'State) -> arr:'T [] -> elm:'State -> 'State`. Updates an accumulator iteratively backwards by applying `f` to each element in `arr`. The initial value of the accumulator is `elm`. For exampel, when `arr` consists of `n+1` elements `List.foldBack` calculates:

`f arr.[0] (f arr.[1] (...(f arr.[n] elm) ...))`.

Listing 1.60: `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: f:('T -> bool) -> arr:'T [] -> bool`. Returns true if `f` evaluates to true for every element in `arr`.

Listing 1.61: `Array.forall`

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

`Array.init: m:int -> f:(int -> 'T) -> 'T []`. Create an array with `m` elements and whose value is the result of applying `f` to the index of the element.

Listing 1.62: `Array.init`

```
> Array.init 10 (fun i -> i * i);;
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.isEmpty: arr:'T [] -> bool`. Returns true if `arr` is empty.

Listing 1.63: `Array.isEmpty`

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

`Array.iter: f:('T -> unit) -> arr:'T [] -> unit`. Applies `f` to each element of `arr`.

Listing 1.64: `Array.iter`

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

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

Listing 1.65: `Array.map`

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

`Array.ofList: lst:'T list -> 'T []`. Creates an array whose elements are copied from `lst`.

Listing 1.66: `Array.ofList`

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

`Array.rev: arr:'T [] -> 'T []`. Creates a new array whose elements are identical to `arr` but in reverse order.

Listing 1.67: `Array.rev`

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

`Array.sort: arr:'T[] -> 'T []`. Creates a new array with the same elements as in `arr` but in sorted order

Listing 1.68: `Array.sort`

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

`Array.toList: arr:'T [] -> 'T list`. Creates a new list whose elements are copied from `arr`.

Listing 1.69: `Array.toList`

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

`Array.unzip: arr:('T1 * 'T2) [] -> 'T1 [] * 'T2 []`. Returns a pair of arrays of all the first elements and all the second elements of `arr`, respectively.

Listing 1.70: `Array.unzip`

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

`Array.zip: arr1:'T1 [] -> arr2:'T2 [] -> ('T1 * 'T2) []`. Returns a list of pairs, where elements in `arr1` and `arr2` are iteratively paired.

Listing 1.71: `Array.zip`

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

1.4 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. The example in Listing 1.72 is a jagged array of increasing width.

· multidimensional arrays

· jagged arrays

Listing 1.72 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 1.73.

Listing 1.73 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
```

Note that 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.

Square arrays of dimensions 2 to 4 are so common that F# has built-in modules for

· Array2D@`Array2D`
· Array3D@`Array3D`
· Array4D@`Array4D`

their support. Here, we will describe *`Array2D`*. The workings of *`Array3D`* and *`Array4D`* are very similar. A generic `Array2D` has type `'T [,]`, and it is indexed also using the `[,]` notation. The `Array2D.length1` and `Array2D.length2` functions are supplied by the `Array2D` module for obtaining the size of an array along the first and second dimension. Rewriting the with jagged array example in Listing 1.73 to use `Array2D` gives a slightly simpler program, which is shown in Listing 1.74.

Listing 1.74 array2D.fsx:
Creating a 3 by 4 rectangular array of integers.

```
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]]
```

Note that the `printf` supports direct printing of the 2-dimensional array. `Array2D` arrays support slicing. The "`*`" lexeme is particularly useful to obtain all values along a dimension. This is demonstrated in Listing 1.75.

Listing 1.75: Examples of Array2D slicing. Compare with Listing 1.74.

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

> arr.[2,3];;
val it : int = 32

> arr.[1..,3..];;
val it : int [,] = [[31]
                    [32]]

> arr.[..1,*];;
val it : int [,] = [[0; 10; 20; 30]
                    [1; 11; 21; 31]]

> arr.[1,*];;
val it : int [] = [|1; 11; 21; 31|]

> arr.[1..1,*];;
val it : int [,] = [[1; 11; 21; 31]]
```

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.

Multidimensional arrays have the same properties and methods as arrays, see Section 1.3.1.

1.4.1 The Array2D Module

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

`copy: arr:'T [,] -> 'T [,]`. Creates a new array whose elements are copied from `arr`.

Listing 1.76: `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: m:int -> n:int -> v:'T -> 'T [,]`. Creates an `m` by `n` array whose elements are set to `v`.

Listing 1.77: `Array2D.create`

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

`init: m:int -> n:int -> f:(int -> int -> 'T) -> 'T [,]`. Creates an `m` by `n` array whose elements are the result of applying `f` to the index of an element.

Listing 1.78: `Array2D.init`

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

`iter: f:('T -> unit) -> arr:'T [,] -> unit`. Applies `f` to each element of `arr`.

Listing 1.79: `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`: `arr:'T [,] -> int`. Returns the length the first dimension of `arr`.

Listing 1.80: `Array2D.length1`

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

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

Listing 1.81: `Array2D.forall length2`

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

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

Listing 1.82: `Array2D.map`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.map (fun x -> x * x) 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]]
```