ME 2017.09.25

11 | Collections of ~~Ordered series of~~ data

Examples of collection types include

collections of data | types of collections

F# is tuned to work with ~~ordered series~~, and there are several built-in ~~lists~~ with various properties making them useful for different tasks. ~~F# has four built-in list types:~~ strings, ~~tuples~~, lists, arrays, and sequences. Strings were discussed in Chapter 5 and will be revisited here ~~with~~ more details; sequences will not be discussed.[1] ~~Here we~~ will concentrate on ~~tuples~~, lists, and arrays.

Besides strings we

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.

is it? / How the can the size operation be O(1)

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 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 preceeding the string with "`@`", in which case escape sequences are not replaced, but a double quotation marks is an escape sequence which is replaced by a single, e.g., `@"This is a string"`, `@"\tNon-tabulated string"`, `@"A ""quoted"" string"`, and `@""`. Alternatively, a verbatim string may be delimited by tripple quotes, e.g., `"""This is a string"""`, `"""\tNon-tabulated string"""`, `"""A "quoted" string"""`, and `""""""`. Strings

- string
- `System.string`
- verbatim string

variable containing a

For example, if a is a string, the i'th character in a can be accessed using the notation a.[i].

Strings may be indexed using the `.[]` notation. ~~such that aString.[]~~
In the `String` module, the following functions are available:

`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.

`String.concat: string -> seq<string> -> string`. Returns a new string made by concatenating the given strings with a separator.

are types such as seq(-) introduced?

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

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

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

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

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

`String.iteri: (int -> char -> unit) -> string -> unit`. Applies a specified function to the index of each character in the string and the character itself.

`String.length: string -> int`. Returns the length of the string.

`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.

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

`String.replicate: int -> string -> string`. Returns a string by concatenating a specified number of instances of a string.

11.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Lists can be stated as a *sequence expression*, · list · sequence expres

Listing 11.1 Lists with a sequence expression.

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

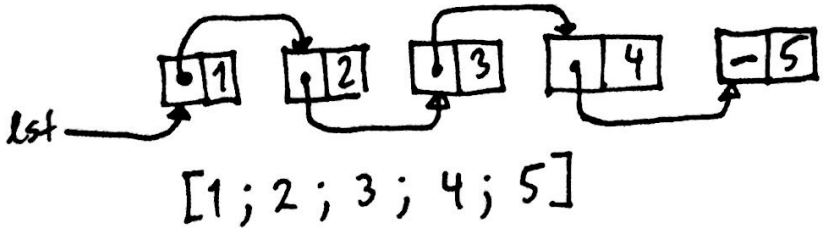
and examples are `[1; 2; 3; 4; 5]`, which is a list of integers, `["This"; "is"; "a"; "list"]`, which is a list of strings, `[(fun x -> x); (fun x -> x*x)]`, which is a list of anonymous functions, and `[]`, which is an empty list. Lists may also be given as ranges,

Listing 11.2 Lists with a range expressions.

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

but lists 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 expressio

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 *". []"* method, the lengths of lists is retrieved using the *`Length`* property, and we may test whether a list is empty using the *`isEmpty`* property. This is demonstrated in Listing 11.3. · `list` · `. []` · `Length` · `isEmpty`



[1; 2; 3; 4; 5]

Note:

Consing an element to *lst* is very efficient... Appending is expensive...

Listing 11.3 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
```

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 11.4 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 11.5.

Listing 11.5 listFor.fsx:
The - loops are preferred over - .

```
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-expression`s removes 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 11.6.

Listing 11.6 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 *"@"*[2] *concatenation* operator or the *"::" cons* operators. The differences is that "@" concatenates two lists of identical types, while "::" concatenates an element and a list of identical types. This is demonstrated in Listing 11.7.

- @
- list concatenatic
- ::
- list cons

Listing 11.7 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]
```

It is possible to make multidimensional lists as lists of lists as shown in Listing 11.8.

[2] Jon: why does the at-symbol not appear in the index? No idea ?!

Listing 11.8 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 different number of elements. The indexing of a particular element is not elegant, which is why arrays are often preferred for two- and higher-dimensional data structures, see Section 11.3.

· ragged multidimensiona

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

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

Listing 11.9: List.contains

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

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

Listing 11.10: List.empty

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

`List.exists: ('T -> bool) -> 'T list -> bool`. Returns true or false depending on whether any element is true for a given function.

Listing 11.11: List.exists

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

`List.filter: ('T -> bool) -> 'T list -> 'T list`. Returns a list, where all elements evaluate to true for a given function.

Listing 11.12: List.filter

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

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

Listing 11.13: List.find

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

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

Listing 11.14: List.findIndex

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

`List.fold: ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State`. Update 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, calculate $f(\ldots f(f(f(s, x_0), x_1), x_2), \ldots, x_n)$.

Listing 11.15: 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`. Update 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, calculate $f(x_0, f(x_1, f(x_2, \ldots, f(x_n, s))))$.

Listing 11.16: 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`. Apply a function to all element and logically and the result.

Listing 11.17: List.forall

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

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

Listing 11.18: List.head

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

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

Listing 11.19: 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.item: 'T list -> int`. Retrieve an element of a list by its index.

Listing 11.20: List.item

```
> let a = [1; -3; 0] in List.item 1 a;;
val it : int = -3
```

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

Listing 11.21: List.iter

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

`List.Length: 'T list -> int`. The number of elements in a list

Listing 11.22: List.Length

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

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

Listing 11.23: List.map

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

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

Listing 11.24: List.ofArray

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

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

Listing 11.25: List.rev

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

`List.sort: 'T list -> 'T list`. Return a list whos elements have been sorted.

Listing 11.26: List.sort

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

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

Listing 11.27: 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 []`. Return an array whos elements are the same as the supplied list.

Listing 11.28: List.toArray

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

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

Listing 11.29: 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`. Return a list of pairs, whos elements are taken iteratively from two lists.

Listing 11.30: 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 expres

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 ~~control of~~ guarantee that all sub-arrays are of ~~similar~~ the same size. E.g., the example in Listing 11.31 is a jagged array of increasing width.

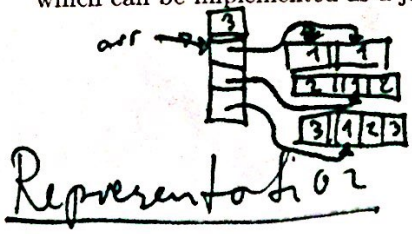
- multidimensiona arrays
- jagged arrays

Listing 11.31 arrayJagged.fsx:
An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

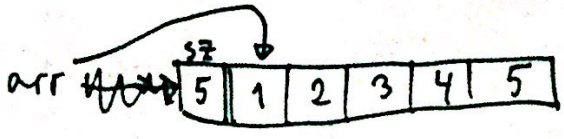
```
let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

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 11.32.



Representation



Efficient indexing, Inefficient to cons values in front of an array (total copy needed).

Listing 11.32 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 similar. An example of creating the same 2-dimensional array as above but as an `Array2D` is shown in Listing 11.33.

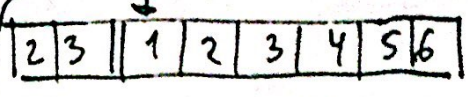
- `Array2D`
- `Array3D`
- `Array4D`

Listing 11.33 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 11.34.

Split section: multidimensional arrays have a different representation, $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ arr → 

I would guess...

Listing 11.34 array2DSlicing.fsx: Examples of Array2D slicing. Compare with Listing 11.33.

```
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. [3]

There are quite a number of built-in procedures for arrays in the `Array2D` namespace, some of which are summarized below.[4]

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

Listing 11.35: 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 initialized to the given value.

Listing 11.36: Array2D.create

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

[3] Jon: Array2D.ToString produces [[...]] and not [|[| ... |]|], which can cause confusion.
[4] Jon: rewrite description

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

Listing 11.37: 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.

Listing 11.38: 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 11.39: 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 11.40: 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 11.41: 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 11.42: 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 11.43: 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 = ()
```

11.5 Comparision