Learning to program with F#

Jon Sporring

July 29, 2016

Contents

Chapter 8

Tuples, Lists, Arrays, and Sequences

F# is tuned to work with lists, and there are several built-in lists with various properties making them useful for different tasks. E.g.,

```
let solution a b c =
  let d = b ** 2.0 - 2.0 * a * c
  if d < 0.0 then
    (nan, nan)
  else
    let xp = (-b + sqrt d) / (2.0 * a)
    let xn = (-b - sqrt d) / (2.0 * a)
    (xp,xn)

let (a, b, c) = (1.0, 0.0, -1.0)
let (xp, xn) = solution a b c
printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
printfn "  has solutions %A and %A" xn xp
```

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

Listing 8.1: tuplesQuadraticEq.fsx - Using tuples to gather values.

F# has 4 built-in list types: tuples, lists, arrays, and sequences following this syntax:

```
tupleList = expr | expr "," tupleList
listList = expr | expr ";" listList
comp-or-range-expr = comp-expr | short-comp-expr | range-expr
short-comp-expr = "for " pat " in " expr-or-range-expr "->" expr
range-exp = expr ".." expr ["..." expr]
comp-expr =
  ("let " | "let! ") pat "=" expr " in " comp-expr
  | ("do " | "do! ") expr " in " comp-expr
  | ("use " | "use! ") pat = expr " in " comp-expr
  | ("yield " | "yield! ") expr
  | ("return " | "return! ") expr
  | "if " expr " then " comp-expr [" else " comp-expr]
  | "match " expr " with " comp-rules
  | "try " comp-expr " with " comp-rules
  | "try " comp-expr " finally " expr
  | "while " expr " do " expr [" done "]
  | "for " ident "=" expr " to " expr " do " comp-expr [" done "]
  | "for " pat " in " expr-or-range-expr " do " comp-expr [" done "]
  | comp-expr ";" comp-expr
  | expr
```

```
comp-rule = pat pattern-guardopt "->" comp-expr
comp-rules = comp-rule | comp-rule '|' comp-rules
expr = ...
  | tupleList
  | "[" listList "]" (* list *)
  | "[|" listList "|]" (* array *)
  | expr "{" comp-or-range-expr "}" (* computation expression *)
  | "[" comp-or-range-expr "]" (* computed list expression *)
  | "[|" comp-or-range-expr "|]" (* computed array expression *)
  | ...
```

Tuples are a direct extension of constants. They are immutable and do not have concatenations nor indexing operations. This is in contrast to lists. Lists are also immutable, but have a simple syntax for concatenation and indexing. Arrays are mutable lists, and support higher order structures such as tables and 3 dimensional arrays. Sequences are like lists, but with the added advantage of a very flexible construction mechanism, and the option of representing infinite long sequences. In the following, we will present these datastructures in detail.

8.1 Tuples

Tuples are unions of types, · tuple

```
tupleList = expr | expr "," tupleList
expr = ...
  | tupleList
  | ...
```

and the they are identified by the `,` token. Most often the tuple is enclosed in parantheses, but that is not required. Consider the tripel, also known as a 3-tuple, `(2,true,"hello")` in interactive mode,

```
> let tp = (2, true, "hello")
- printfn "%A" tp;;
(2, true, "hello")

val tp : int * bool * string = (2, true, "hello")
val it : unit = ()
```

Listing 8.2: fsharpi, Definition of a tuple.

The values 2, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F# we see that the tuple is inferred to have the type `int * bool * string`, where the `*` is cartesian product between the three sets. Notice, that tuples can be products of any types and have lexical scope like value and function bindings. Notice also, that a tuple may be printed as a single entity by the `%A` placeholder. In the example, we bound `tp` to the tuple, the opposite is also possible, · member · length

```
> let deconstructNPrint tp =
-   let (a, b, c) = tp
-   printfn "tp = (%A, %A, %A)" a b c
-
- deconstructNPrint (2, true, "hello")
- deconstructNPrint (3.14, "Pi", 'p');;
tp = (2, true, "hello")
tp = (3.14, "Pi", 'p')

val deconstructNPrint : 'a * 'b * 'c -> unit
val it : unit = ()
```

Listing 8.3: fsharpi, Definition of a tuple.

In this a function is defined that takes 1 argument, a 3-tuple, and which is bound to a tuple with 3 named members. Since we used the `%A` placeholder in the `printfn` function, then the function is generic and can be called with 3-tuples of different types. Note, *don't confuse a function of n arguments with a function of an n-tuple.* The later has only 1 argument, and the difference is the `,`'s.

Advice

Tuples may be compared, and are equal if their type and value are equal. Pairs, 2-tuples, are so common that two built-in functions have been supplied, ***fst*** and ***snd***, to retrieve the first and second element of the tuple. [1]

- `fst`
- `snd`

8.2 Lists

However, an elegant alternative is available as

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let mutable sum = 0;
let mutable n = 0;
for (title, grade) in courseGrades do
  printfn "Course: %s, Grade: %d" title grade
  sum <- sum + grade;
  n <- n + 1;
let avg =  (float sum) / (float n)
printfn "Average grade: %g" avg
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Listing 8.4: flowForLists.fsx -

This to be preferred, since we completely can ignore list boundary conditions and hence avoid out of range indexing. For comparison see a recursive implementation of the same,

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let rec printAndSum lst =
  match lst with
    | (title, grade)::rest ->
      printfn "Course: %s, Grade: %d" title grade
      let (sum, n) = printAndSum rest
      (sum + grade, n + 1)
    | _ -> (0, 0)
let (sum, n) = printAndSum courseGrades
let avg = (float sum) / (float n)
printfn "Average grade: %g" avg
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

[1] **possibly more on comparison.**

Listing 8.5: flowForListsRecursive.fsx -

Note how this implementation avoids the use of variables in contrast to the previous examples.

8.3 Arrays

8.3.1 1 dimensional arrays

Roughly speaking, arrays are mutable lists, and may be created and indexed in the same manner, e.g.,

```
let A = [| 1; 2; 3; 4; 5 |]
let B = [| 1 .. 5 |]
let C = [| for a in 1 ..5 do yield a |]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

printArray A
printArray B
printArray C
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Listing 8.6: arrayCreation.fsx -

Notice that as for lists, arrays are indexed starting with 0, and that in this particular case it was necessary to specify the type of the argument for `printArray` as an array of integers with the `array` keyword. The `array` keyword is synonymous with '`[]`'. Arrays do not support pattern matching, cannot be resized, but are mutable,

```
let A = [| 1; 2; 3; 4; 5 |]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

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

printArray A
square A
printArray A
```

```
1 2 3 4 5
1 4 9 16 25
```

Listing 8.7: arrayReassign.fsx -

Notice that in spite the missing `mutable` keyword, the function `square` still had the side-effect of squaring alle entries in `A`. Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range, e.g., · slicing

```
let A = [| 1; 2; 3; 4; 5 |]
let B = A.[1..3]
let C = A.[..2]
let D = A.[3..]
let E = A.[*]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

printArray A
printArray B
printArray C
printArray D
printArray E
```

```
1 2 3 4 5
2 3 4
1 2 3
4 5
1 2 3 4 5
```

Listing 8.8: arraySlicing.fsx -

As illustrated, the missing start or end index implies from the first or to the last element.
There are quite a number of built-in procedures for all arrays some of which we summarize in Table 8.1.
Thus, the `arrayReassign.fsx` program can be written using arrays as,

```
let A = [| 1 .. 5 |]

let printArray (a : int array) =
  Array.iter (fun x -> printf "%d " x) a
  printf "\n"

let square a = a * a

printArray A
let B = Array.map square A
printArray A
printArray B
```

```
1 2 3 4 5
1 2 3 4 5
1 4 9 16 25
```

Listing 8.9: arrayReassignModule.fsx -

and the `flowForListsIndex.fsx` program can be written using arrays as,

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let A = Array.ofList courseGrades
let printCourseNGrade (title, grade) =
  printfn "Course: %s, Grade: %d" title grade
Array.iter printCourseNGrade A
```

append	Creates an array that contains the elements of one array followed by the elements of another array.
average	Returns the average of the elements in an array.
blit	Reads a range of elements from one array and writes them into another.
choose	Applies a supplied function to each element of an array. Returns an array that contains the results x for each element for which the function returns Some(x).
collect	Applies the supplied function to each element of an array, concatenates the results, and returns the combined array.
concat	Creates an array that contains the elements of each of the supplied sequence of arrays.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
empty	Returns an empty array of the given type.
exists	Tests whether any element of an array satisfies the supplied predicate.
fill	Fills a range of elements of an array with the supplied value.
filter	Returns a collection that contains only the elements of the supplied array for which the supplied condition returns true.
find	Returns the first element for which the supplied function returns true. Raises System.Collections.Generic.KeyNotFoundException if no such element exists.
findIndex	Returns the index of the first element in an array that satisfies the supplied condition. Raises System.Collections.Generic.KeyNotFoundException if none of the elements satisfy the condition.
fold	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.
foldBack	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)).
forall	Tests whether all elements of an array satisfy the supplied condition.
isEmpty	Tests whether an array has any elements.
iter	Applies the supplied function to each element of an array.
init	...
length	Returns the length of an array. The System.Array.Length property does the same thing.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
mapi	
max	Returns the largest of all elements of an array. Operators.max is used to compare the elements.
min	Returns the smallest of all elements of an array. Operators.min is used to compare the elements.
ofList	Creates an array from the supplied list.
ofSeq	Creates an array from the supplied enumerable object.
partition	Splits an array into two arrays, one containing the elements for which the supplied condition returns true, and the other containing those for which it returns false.
rev	Reverses the order of the elements in a supplied array.
sort	Sorts the elements of an array and returns a new array. Operators.compare is used to compare the elements.
sub	Creates an array that contains the sup<plied subrange, which is specified by starting index and length.
sum	Returns the sum of the elements in the array.
toList	Converts the supplied array to a list.
toSeq	Views the supplied array as a sequence.
unzip	Splits an array of tuple pairs into a tuple of two arrays.
zeroCreate	Creates an array whose elements are all initially zero.
zip	Combines two arrays into an array of tuples that have two elements. The two arrays must have equal lengths; otherwise, System.ArgumentException is raised.

Table 8.1: Some built-in procedures in the Array module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

```
let (titles,grades) = Array.unzip A
let avg =  (float (Array.sum grades)) / (float grades.Length)
printfn "Average grade: %g" avg
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Listing 8.10: flowForListsIndexModule.fsx -

Both cases avoid the use of variables and side-effects which is a big advantage for code safety.

8.3.2 Multidimensional Arrays

Higher dimensional arrays can be created as arrays of arrays (of arrays . . .). These are known as *jagged arrays*, since there is no inherent control of that all sub-arrays are of similar size. E.g., the following is a jagged array of increasing width,

· jagged arrays

```
let A = [| for n in 1..3 do yield [| 1 .. n |] |]

let printArrayOfArrays (a : int array array) =
  for i = 0 to a.Length - 1 do
    for j = 0 to a.[i].Length - 1 do
      printf "%d " a.[i].[j]
    printf "\n"

printArrayOfArrays A
```

```
1
1 2
1 2 3
```

Listing 8.11: arrayJagged.fsx -

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 square arrays are used, which can be implemented as a jagged array as,

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

let A = [| for n in 1..3 do yield (pownArray [| 1 .. 4 |] n ) |]

let printArrayOfArrays (a : int array array) =
  for i = 0 to a.Length - 1 do
    for j = 0 to a.[i].Length - 1 do
      printf "%2d " a.[i].[j]
    printf "\n"

printArrayOfArrays A
```

```
 1  2  3  4
 1  4  9 16
 1  8 27 64
```

Listing 8.12: arrayJaggedSquare.fsx -

blit	Reads a range of elements from one array and writes them into another.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
iter	Applies the supplied function to each element of an array.
length1	Returns the length of an array in the first dimension.
length2	Returns the length of an array in the second dimension.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
mapi	
zeroCreate	Creates an array whose elements are all initially zero.

Table 8.2: Some built-in procedures in the Array2D module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

In fact, square arrays of dimensions 2 to 4 are so common that fsharp has built-in modules for their support. In the following 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,

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

let printArray2D (a : int [,]) =
  for i = 0 to (Array2D.length1 a) - 1 do
    for j = 0 to (Array2D.length2 a) - 1 do
      printf "%2d " a.[i, j]
    printf "\n"

printArray2D A
```

```
 1   2   3   4
 1   4   9  16
 1   8  27  64
```

Listing 8.13: array2D.fsx -

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.
[2]

There are a bit few built-in procedures for 2 dimensional array types, some of which are summarized in Table 8.2

8.4 Sequences

[2] **note that `A.[1,*]` is a Array but `A.[1..1,*]` is an Array2D.**

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

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

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

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

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

Index