Learning to program with F#

Jon Sporring

August 2, 2016

Contents

Chapter 9

Ordered series of data

[1] F# is tuned to work with ordered series, 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 9.1: tuplesQuadraticEq.fsx - Using tuples to gather values.

F# has 4 built-in list types: strings, tuples, lists, arrays, and sequences. Strings were discussed in Chapter 5, and tuples, lists, arrays, and sequences following this (simplified) syntax:

```
tupleList = expr | expr "," tupleList
listOrArrayList =  expr | expr ";" listOrArrayList
range-exp = expr ".." expr [".." expr]
comp-expr =
  "let" pat "=" expr "in" comp-expr
  | "use" pat = expr "in" comp-expr
  | ("yield" | "yield!") 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
short-comp-expr = "for" pat "in" (expr | range-expr) "->" expr
comp-or-range-expr = comp-expr| short-comp-expr | range-expr
```

[1] **possibly add maps and sets as well.**

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

[2]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 data structures in detail.

9.1 Tuples

Tuples are unions of immutable types, · tuple

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

and the they are identified by the `,` lexeme. Most often the tuple is enclosed in parentheses, 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 9.2: 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
```

[2]**Spec-4.0: grammar for list and array expressions are subsets of computation list and array expressions.**

```
val it : unit = ()
```

Listing 9.3: 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. Another example is `let solution a b c = ...`, which is the beginning of the function definition in Listing 9.1. It is a function of 3 arguments, while `let solution (a, b, c)= ...` would be a function of 1 argument, which is a 3-tuple. Functions of several arguments makes currying easy, i.e., we could define a new function which fixes the quadratic term to be 0 as `let solutionToLinear = solution 0.0`, that is, without needing to specify anything else. With tuples, we would need the slightly more complicated, `let solutionToLinear (b, c)= solution (0.0, b, c)`.

Advice

Tuples comparison are defined similarly as strings. Tuples of different lengths are different. For tuples of equal length, then they are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The `<>` operator is the boolean negation of the `=` operator. For the `<` , `<=`, `>`, and `>=` operators, the strings are ordered alphabetically like, such that `('a', 'b', 'c')< ('a', 'b', 's')&& ('a', 'b', 's')< ('c', 'o', 's')` is true, that is, the `<` operator on two tuples is true, if the left operand should come before the right, when sorting alphabetically like.

```
let lessThan (a, b, c) (d, e, f) =
  if a <> d then a < d
  elif b <> e then b < d
  elif c <> f then c < f
  else false

let printTest x y =
  printfn "%A < %A is %b" x y (lessThan x y)

let a = ('a', 'b', 'c');
let b = ('d', 'e', 'f');
let c = ('a', 'b', 'b');
let d = ('a', 'b', 'd');
printTest a b
printTest a c
printTest a d
```

```
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

Listing 9.4: tupleCompare.fsx - Tuples are compared as strings are compared alphabetically.

The algorithm for deciding the boolean value of `(a1, a2)< (b1, b2)` is as follows: we start by examining the first elements, and if `la1` and `b1` are different, then the `(a1, a2)< (b1, b2)` is equal to `a1 < b1`. If `la1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The `<=`, `>`, and `>=` operators are defined similarly.
Binding tuples to mutuals does not make the tuple mutable, e.g.,

```
let mutable a = 1
let mutable b = 2
let c = (a, b)
printfn "%A, %A, %A" a b c
a <- 3
printfn "%A, %A, %A" a b c
```

```
1, 2, (1, 2)
```

```
3, 2, (1, 2)
```

Listing 9.5: tupleOfMutables.fsx - A mutable change value, but the tuple defined by it does not refer to the new value.

However, tuples may be mutual such that new tuple values can be assigned to it, e.g., in the Fibonacci example, we can write a more compact script by using mutable tuples and the `fst` and `snd` functions as follows.

```
let fib n =
  if n < 1 then
    0
  else
    let mutable prev = (0, 1)
    for i = 2 to n do
      prev <- (snd prev, (fst prev) + (snd prev))
    snd prev

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

Listing 9.6: fibTuple.fsx - Calculating Fibonacci numbers using mutable tuple.

In this example, the central computation has been packed into a single line, `prev <- (snd prev, (fst prev)+ (snd prev))`, where both the calculation of $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$ and the rearrangement of memory to hold the new values $\text{fib}(n)$ and $\text{fib}(n-1)$ based on the old values $\text{fib}(n-2)+\text{fib}(n-1)$. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, and in this case, an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without an accompanying explanation. Hence, *always keep an eye out for compact and concise ways to write code, but never at the expense of readability.*

· obfuscation

Advice

9.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Its grammar follows *computation expressions*, which is very rich and shared with arrays and sequences, and we will delay a discussion on most computation expressions to Section 9.4, and here just consider a subset of the grammar:

· list
· computation expressions

```
listOrArrayList =  expr | expr ";" listOrArrayList
range-exp = expr ".." expr [".." expr]
expr = ...
  | "[" (listOrArrayList | ... | range-expr) "]" (* computation list expression *)
  | ...
```

Simple examples of a list grammars are, `[expr; expr; ... ; expr]`, `[expr ".."expr]`, `[ expr ".."expr ".."expr]`, e.g., an explicit list `let lst = [1; 2; 3; 4; 5]`, which may be written shortly as *range expression* as `let lst = [1 .. 5]`, and ranges may include a step size `let lst = [1 .. 2 .. 5]`, which is the same as `let lst = [1; 3; 5]`.

· range expression

Lists may be indexed and concatenated much like strings, e.g.,

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

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

let a = [1; 2;]
let b = [3; 4; 5]
let c = a @ b
let d = 0 :: c
printfn "%A, %A, %A, %A" a b c d
printList d
printListAlt d
```

```
[1; 2], [3; 4; 5], [1; 2; 3; 4; 5], [0; 1; 2; 3; 4; 5]
0 1 2 3 4 5
0 1 2 3 4 5
```

Listing 9.7: listIndexing.fsx - Examples of list concatenation, indexing.

A list type is identified with the `list` keyword, as here a list of integers is `int list`. Above, we used the *@* and *::* concatenation operators, the *.[]* index method, and the *Length* property. Notice, as strings, list elements are counted from 0, and thus the last element has `lst.Length - 1`. In `printList` the `for-in` is used, which runs loops through each element of the list and assigns it to the identifier `elm`. This is in contrast to `printListAlt`, which uses uses the `for-to` keyword and explicitly represents the index `i`. Explicit representation of the index makes more complicated programs, and thus increases the chances of programming errors. Hence, *`for-in` is to be preferred over `for-to`*. Lists support slicing identically to strings, e.g.,

· `@`
· `::`
· `.[]`
· `Length`

Advice

```
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.[*]
```

```
'a'
'd'
['d'; 'e'; 'f'; 'g']
['a'; 'b'; 'c'; 'd']
['b'; 'c'; 'd']
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

Listing 9.8: listSlicing.fsx - Examples of list slicing. Compare with Listing 5.27.

Lists are well suited for recursive functions and pattern matching with, e.g., `match-with` as illustrated in the next example:

```
let rec printListRec (lst : int list) =
  match lst with
```

```
    elm::rest ->
      printf "%A " elm
      printListRec rest
    | _ ->
      printfn ""

let a = [1; 2; 3; 4; 5]
printListRec a
```

```
1 2 3 4 5
```

Listing 9.9: listPatternMatching.fsx - Examples of list concatenation, indexing.

The pattern `l::rest` is the pattern for the first element followed by a list of the rest of the list. This pattern matches all lists except an empty list, hence `rest` may be empty. Thus the wildcard pattern matching anything including the empty list, will be used only when `lst` is empty.

Pattern matching with lists is quite powerful, consider the following problem:

· pattern matching

> Given a list of pairs of course names and course grades, calculate the average grade.

A list of course names and grades is `[("name1", grade1); ("name2", grade2); ...]`. Let's take a recursive solution. First problem will be to iterate through the list. For this we can use pattern matching similarly to Listing 9.9 with `(name, grade)::rest`. The second problem will be to calculate the average. The average grade is the sum all grades and divide by the number of grades. Assume that we already have made a function, which calculates the `sum` and `n`, the sum and number of elements, for `rest`, then all we need is to add `grade` to the `sum` and `1` to `n`. For an empty list, `sum` and `n` should be `0`. Thus we arrive at the following solution,

```
let averageGrade courseGrades =
  let rec sumNCount lst =
    match lst with
      | (title, grade)::rest ->
        let (sum, n) = sumNCount rest
        (sum + grade, n + 1)
      | _ -> (0, 0)

  let (sum, n) = sumNCount courseGrades
  (float sum) / (float n)

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

printfn "Course and grades:\n%A" courseGrades
printfn "Average grade: %.1f" (averageGrade courseGrades)
```

```
Course and grades:
[("Introduction to programming", 95); ("Linear algebra", 80);
 ("User Interaction", 85)]
Average grade: 86.7
```

Listing 9.10: avgGradesRec.fsx - Calculating a list of average grades using recursion and pattern matching.

Pattern matching and appending is a useful combination, if we wish to produce new from old lists. E.g., a function returning a list of squared entries of its argument can be programmed as,

```
let rec square a =
```

```
  match a with
    elm :: rest -> elm*elm :: (square rest)
    | _ -> []

let a = [1 .. 10]
printfn "%A" (square a)
```

```
[1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

Listing 9.11: listSquare.fsx - Using pattern matching and list appending elements to lists.

This is a prototypical functional programming style solution, and which uses the `::` for 2 different purposes: First the list `[1 .. 10]` is first matched with `1 :: [2 .. 10]`, and then we assume that we have solved the problem for `square rest`, such that all we need to do is append `1*1` to the beginning output from `square rest`. Hence we get, `square [1 .. 10]` $\curvearrowright$ `1 * 1 :: square [2 .. 10]` $\curvearrowright$ `1 * 1 :: (2 * 2 :: square [3 .. 10])` $\curvearrowright \ldots$ `1 * 1 :: (2 * 2 :: ... 10 * 10 :: [])`, where the stopping criterium is reached, when the `elm :: rest` does not match with a, hence it is empty, which does match the wildcard pattern `_`. More on functional programming in Section 17
The basic properties and members of lists are summarized in Table 9.1. In addition, lists have many other built-in functions, such as functions for converting lists to arrays and sequences,

```
let lst = ['a' .. 'c']
let arr = List.toArray lst
let sq = List.toSeq lst
printfn "%A, %A, %A" lst arr sq
```

```
['a'; 'b'; 'c'], [|'a'; 'b'; 'c'|], ['a'; 'b'; 'c']
```

Listing 9.12: listConversion.fsx - The `List` module contains functions for conversion to arrays and sequences.

These and more will be discussed in Chapter 14 and Part III.
It is possible to make multidimensional lists as lists of lists, e.g.,

```
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
```

```
[1; 2]
2
2
```

Listing 9.13: listMultidimensional.fsx - A ragged multidimensional list, built as lists of lists, and its indexing.

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 in F#.

· ragged multidimensional list

9.3 Arrays

One dimensional arrays or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Its grammar follows *computation expressions*, which will be discussed in Section 9.4. Here we consider a subset of the grammar:

· computation expressions

```
listOrArrayList =  expr | expr ";" listOrArrayList
range–exp = expr ".." expr [".." expr]
expr = ...
```

Function name	Example	Description

`Length`	`> [1; 2; 3].Length;;`
	`val it : int = 3`
	`> let a = [1; 2; 3] in a.Length;;`
	`val it : int = 3`
		The number of elements in a list

`List.Empty`	`> let a : int list = List.Empty;;`
	
	`val a : int list = []`
	
	`> let b = List<int>.Empty;;`
	
	`val b : int list = []`
		An empty list of specified type

`IsEmpty`	`> [1; 2; 3].IsEmpty;;`
	`val it : bool = false`
	`> let a = [1; 2; 3] in a.IsEmpty;;`
	`val it : bool = false`
		Compare with the empty list

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

`Head`	`> [1; 2; 3].Head;;`
	`val it : int = 1`
	`> let a = [1; 2; 3] in a.Head;;`
	`val it : int = 1`
		The first element in the list. Exception if empty.

`Tail`	`> [1; 2; 3].Tail;;`
	`val it : int list = [2; 3]`
	`> let a = [1; 2; 3] in a.Tail;;`
	`val it : int list = [2; 3]`
		The list except its first element. Exception if empty.

`Cons`	`> list.Cons (1, [2; 3]);;`
	`val it : int list = [1; 2; 3]`
	`> 1 :: [2; 3];;`
	`val it : int list = [1; 2; 3]`
		Append an element to the front of the list

`@`	`> [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]`
		Concatenate two lists

Table 9.1: Basic properties and members of lists. The syntax used in `List<int>.Empty` ensures that the empty list is of type `int`.

```
  | "[|" (listOrArrayList | ... | range−expr) "|]" (* computation array expression
      *)
  | ...
```

Thus the creation of arrays is identical to lists, but there is no explicit operator support for appending and concatenation, e.g.,

```
let printArray (arr : int array) =
  for elm in arr do
    printf "%d " elm
  printf "\n"

let printArrayAlt (arr : int array) =
  for i = 0 to arr.Length - 1 do
    printf "%A " arr.[i]
  printfn ""

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

```
[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
1 2 3 4 5
1 2 3 4 5
```

Listing 9.14: arrayCreation.fsx - Creating arrays with a syntax similarly to lists.

The array type is defined using the `array` keyword or alternatively the `[]` lexeme. Arrays cannot be resized, but are mutable,

```
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]

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

printArray A
square A
printArray A
```

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

Listing 9.15: arrayReassign.fsx - Arrays are mutable in spite the missing `mutable` keyword.

Notice that in spite the missing `mutable` keyword, the function `square` still had the *side-effect* of squaring alle entries in `A`. Arrays only support direct pattern matching, e.g., · side-effect

```
let name2String (arr : string array) =
  match arr with
    [| first; last|] -> last + ", " + first
    | _ -> ""
```

```
let listNames (arr :string array array) =
  let mutable str = ""
  for a in arr do
    str <- str + name2String a + "\n"
  str

let A = [|[|"Jon"; "Sporring"|]; [|"Alonzo"; "Church"|]; [|"John"; "McCarthy"
    |]|]
printf "%s" (listNames A)
```

```
Sporring, Jon
Church, Alonzo
McCarthy, John
```

Listing 9.16: arrayPatternMatching.fsx - Only simple pattern matching is allowed for arrays.

The given example is the first example of a 2-dimensional array, which can be implemented as arrays of arrays and here written as `string array array`. Below further discussion of on 2 and higher dimensional arrays be discussed. 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 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.[*]
```

```
'a'
'd'
[|'d'; 'e'; 'f'; 'g'|]
[|'a'; 'b'; 'c'; 'd'|]
[|'b'; 'c'; 'd'|]
[|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

Listing 9.17: arraySlicing.fsx - Examples of array slicing. Compare with Listing 9.8 and Listing 5.27.

As illustrated, the missing start or end index implies from the first or to the last element.
Arrays can be converted to lists and sequences by,

```
let arr = [|'a' .. 'c'|]
let lst = Array.toList arr
let sq = Array.toSeq arr
printfn "%A, %A, %A" arr lst sq
```

```
[|'a'; 'b'; 'c'|], ['a'; 'b'; 'c'], seq ['a'; 'b'; 'c']
```

Listing 9.18: arrayConversion.fsx - The `Array` module contains functions for conversion to lists and sequences.

There are quite a number of built-in procedures for all arrays many which will be discussed in Chapter 14.
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 arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

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

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

Listing 9.19: arrayJagged.fsx - An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

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,

```
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
```

```
  1   2   3   4
  1   9  25  49
  1  16  49 100
```

Listing 9.20: arrayJaggedSquare.fsx - A rectangular array.

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 \lstinlineelm!` 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 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 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
```

```
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Listing 9.21: array2D.fsx - Creating a 3 by 4 rectangular arrays of intigers.

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, e.g.,

```
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,*]
```

```
11
[[10]
 [11]]
[[0; 3; 6; 9]
 [1; 4; 7; 10]]
[|1; 4; 7; 10|]
[[1; 4; 7; 10]]
```

Listing 9.22: array2DSlicing.fsx - Examples of Array2D slicing. Compare with Listing 9.21.

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]
Array2D and higher have a number of built-in functions that will be discussed in Chapter 14.

9.4 Sequences

Sequences are lists, where the elements are build as needed. Examples are[4]

```
> #nowarn "40"
- let a = { 1 .. 10 };;

val a : seq<int>

> let b = seq { 1 .. 10 };;

val b : seq<int>

> let c = seq {for i = 1 to 10 do yield i*i done};;

val c : seq<int>

> let rec d =
-   seq {
-     for i = 0 to 59 do yield (float i)*2.0*3.1415/60.0 done;
-     yield! d
-     };;

val d : seq<float>
```

[3] **Array2D.ToString produces `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion.**
[4] **Mono does not support specification Spec-4.0 Section 6.3.11, seq comp-expr, in the form seq 3 or seq 3; 4.**

Listing 9.23: fsharpi, Creating sequences by range explicitely stating elements, a range expressions, a computation expression, and an infinite computation expression

Sequences are built using the following subset of the general syntax,

```
range—exp = expr ".." expr [".." expr]
comp—expr =
  "let" pat "=" expr "in" comp—expr
  | "use" pat = expr "in" comp—expr
  | ("yield" | "yield!") 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
short—comp—expr = "for" pat "in" (expr | range—expr) "—>" expr
comp—or—range—expr = comp—expr| short—comp—expr | range—expr
comp—rule = pat pattern—guardopt "—>" comp—expr
comp—rules = comp—rule | comp—rule '|' comp—rules
expr = ...
  | "seq" "{" comp—or—range—expr "}" (* computation expression *)
  | ...
```

Sequence may be defined using simple range expressions but most often are defined as a small program, that generates values with the `yield` keyword or `yield!` keyword. The `yield!` is called *yield bang*, and appends a sequence instead of adding a sequence as an element. Thus, `seq {3; 5}` is not permitted, but `seq {yield 3; yield 5}` and `seq {yield!(seq {yield 3; yield 5})}` are, both creating `seq <int> = seq [3; 5]`, i.e., a sequence of integers. Most often computation expressions are used to produced members that are not just ranges, but more complicated expressions of ranges, e.g., `c` in the example. Sequences may even in principle be infinitely long, e.g., `d`. Calculating the complete sequence at the point of definition is impossible due to lack of memory, as is accessing all its elements due to lack of time. But infinite sequences are still very useful, e.g., identifier `d` illustrates the parametrization of a circle, which is an infinite domain, and any index will be converted to the equivalent 60'th degree angle in radians. Fsharp warns against recursive values, as defined in the example, since it will check the soundness of the value at run-time rather than at compile-time. The warning can be removed by adding `#nowarn "40"` in the script or `--nowarn:40` as argument to `fsharpi` or `fsharpc`. · yield bang

Sequences are generalisations of lists and arrays, and functions taking sequences as argument may equally take lists and arrays as argument. Sequences do not have many built-in operators, but a rich collection of functions in the `Collections.Seq`. E.g.,

```
> let sq = seq { 1 .. 10 };; (* make a sequence *)

val sq : seq<int>

> let itm = Seq.item 0 sq;; (* take firste element *)

val itm : int = 1

> let sbsq = Seq.take 3 sq;; (* make new sequence of first 3 elements *)

val sbsq : seq<int>
```

Listing 9.24: fsharpi, Index a sequence with `Seq.item` and `Seq.take`

which as usual index from 0 and will cast an exception, if indexing is out of range for the sequence.

That sequences really are programs rather than values can be seen by the following example,

```
1
That was 0
2
That was 1
The sequence was evaluated to this point.
3
That was 2
```

Listing 9.25: fsharpi, Sequences elements are first evaluated, when needed.

In the example, we see that the `printfn` function embedded in the definition is first executed, when the 3rd item is requested.
[5]

The only difference between computation expression's programming constructs and the similar regular expressions constructs is that they must return a value with the `yield` or `yield!` keywords.[6] The `try`-keyword constructions will be discussed in Chapter 10, and the `use`-keyword is a variant of `let` but used in asynchronous computations, which will not be treated here.
Intinfite sequences is a useful concept in many programs and may be generated in a number of ways. E.g., to generate a repeated sequence, we could use recursive value definition, a computation expression, a recursive function, or the `Seq` module. Using a recursive value definition,

```
let repeat items =
  let rec ret =
    seq { yield! items
          yield! ret }
  ret

printfn "%A" (repeat [1;2;3])
```

```
/Users/sporring/repositories/fsharpNotes/src/seqInfinteValue2.fsx(4,18):
    warning FS0040: This and other recursive references to the object(s) being
     defined will be checked for initialization-soundness at runtime through
    the use of a delayed reference. This is because you are defining one or
    more recursive objects, rather than recursive functions. This warning may
    be suppressed by using '#nowarn "40"' or '--nowarn:40'.
seq [1; 2; 3; 1; ...]
```

Listing 9.26: seqInfinteValue2.fsx - Recursive value definitions gives a warning. Compare with Listing9.27, 9.28, and 9.29.

Fsharp warns against using recursive values, since it will check the soundness of the value at run-time rather than at compile-time. The warning can be removed by adding `#nowarn "40"` in the script or `--nowarn:40` as argument to `fsharpi` or `fsharpc`, but *warnings are messages from the designers of F# that your program is non-optimal, and you should avoid structures that throw warnings instead of relying on* `#nowarn` *and similar constructions.* Instead we may create an infinite loop using the `while-do` computation expression, as Advice

```
let repeat items =
  seq { while true do yield! items }

printfn "%A" (repeat [1;2;3])
```

[5]**Mono, missing support for Spec-4.0 Chapter 6, `do-in` in sequences. E.g., `seq let _ = printfn "hej"in yield 3` is ok, but `seq do printfn "hej"in yield 3` not. One could argue, that computation expression is the framework and that it is the seq implementation, which does not provide full access to the framework. but this is confusing, since seq gets special attention in the specification.**
[6]**Mono implements if-elif-else, but this is not in the specification.**

```
seq [1; 2; 3; 1; ...]
```

Listing 9.27: seqInfinteValue.fsx - Infinite value definition without recursion nor warning. Compare with Listing 9.26, 9.28, and9.29.

or alternatively define a recursive function,

```
let rec repeat items =
  seq { yield! items
        yield! repeat items }

printfn "%A" (repeat [1;2;3])
```

```
seq [1; 2; 3; 1; ...]
```

Listing 9.28: seqInfinteFunction.fsx - Recursive function definitions gives no a warning. Compare with Listing 9.26, 9.27, and 9.29.

Infinite expressions have built-in support through the `Seq` module using ,

```
let repeat items =
  let get items x = Seq.item (x % (Seq. length items)) items
  Seq.initInfinite (get items)

printfn "%A"  (repeat [1;2;3])
```

```
seq [1; 2; 3; 1; ...]
```

Listing 9.29: seqInitInfinite.fsx - Using `Seq.initInfinte` and a function to generate an infinte sequence. Compare with Listing 9.26, 9.27, and 9.28.

which takes a function as argument. Here we have used currying, i.e., `get items` is a function that takes on variable and returns a value. The use of the remainder operator makes the example rather contrived, since it might have been simpler to use the `get` indexing function directly.
Sequences are easily converted to and from lists and arrays as,

```
let sq = seq { 1 .. 3 }
let lst = Seq.toList sq (* convert sequence to list *)
let arr = Seq.toArray sq (* convert sequence to array *)
let sqFromArr = seq [| 1 .. 3|] (* convert an array to sequence *)
let sqFromLst = seq [1 .. 3] (* convert a list to sequence *)
printfn "%A, %A, %A, %A, %A" sq lst arr sqFromArr sqFromLst
```

```
seq [1; 2; 3], [1; 2; 3], [|1; 2; 3|], [|1; 2; 3|], [1; 2; 3]
```

Listing 9.30: seqConversion.fsx - Conversion between sequences and lists and arrays using the `List` module.

There are quite a number of built-in functions for sequences many which will be discussed in Chapter 14. Lists and arrays may be created from sequences through the short-hand notation called *list and array sequence expressions*, · list sequence expression

```
expr = ...
  | "[" (... | comp-expr | short-comp-expr | ...) "]" (* list sequence expression
      *)
  | "[|" (.. | comp-expr | short-comp-expr | ...) "|]" (* array sequence
      expression *)
  | ...
```

which implicitely creates the corresponding expression and return the result as a list or array.

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