This topic covers a number of useful features of F# and its libraries sequences: generic functions, sequences, sets, and maps.

Recommended reading: to the end of Chapter 3 of Expert F#.
F# libraries

- F# comes with a small set of libraries.
  - These provide immutable data structures and other support.
  - They are in the namespace Microsoft.FSharp
  - For everything else, F# is designed to depend on .NET

- See start menu or C:\Program Files\Fsharp for documentation.
  - Also on the web – there’s a link from the schedule page.

- Microsoft.Fsharp.Core includes the basic types: int float ...
  - FSharp.Core.Operators includes: + - = < cos fst char ... (much more)

- FSharp.Collections includes:
  - Lists
  - Sequences
  - Immutable sets and maps based on binary trees
  - Arrays, including resizable ones (later – these are imperative).

- We will see some more parts of Microsoft.Fsharp.Core later.

- FSharp.PowerPack is a dll assembly with modules that are less central or may change in future versions of F#.
  - To use this you need to add it as a reference for your Visual Studio project.
Generic Functions

- The libraries include the following predefined *generic functions* that apply to any type of value, but which act differently depending on the type.

  ```ml
  val compare : 'a -> 'a -> int
  val (=) : 'a -> 'a -> bool
  val (<) : 'a -> 'a -> bool
  val (<=) : 'a -> 'a -> bool
  val (>) : 'a -> 'a -> bool
  val (>=) : 'a -> 'a -> bool
  val min : 'a -> 'a -> 'a
  val max : 'a -> 'a -> 'a
  val hash : 'a -> int
  ```

- Comparisons work as expected for basic types like int, float, string, ....

- Tuples and lists: components are compared in order, recursively.
  - This is called *lexicographic ordering* (it’s like the dictionary order for words).

- For unions: follow the order of constructors in the type definition, and recursively compare if the constructor is the same.

- For functions: comparison isn’t meaningful.
  - In generally, comparing functions is not computable.

- Hashing works similarly via recursion.
  - Defining the same function twice will yield different hash codes.

- You can override these defaults with your own functions (later).
Sequences

- Sequences are ordered-collections, like lists.
- However, unlike lists, sequences are evaluated as needed
  - This is can be more efficient, and sequences can even be infinite.
- Sequences are defined just like list comprehensions.
  
  ```
  seq { for a in 1 .. 10 do yield (a, a*a) };
  val it : seq<int * int> = seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
  ```
- In fact, a list comprehension is just an abbreviation for a sequence comprehension followed by Seq.toList
- Sequences essentially allow generating a head element and a tail sequence, like the recursive function type:
  
  ```
  type 'a seq = Seq of unit -> ('a * 'a seq) // PARENS not needed
  ```
- Sequences are important: all other collection types are subtypes.
  - This includes lists, sets, and arrays (which we’ll see later).
  - Also, 'a seq is actually an abbreviation for the .NET type IEnumerable<'a> which is the core .NET interface for collections.
Working with sequences

- Comprehensions are the main way to produce sequences.
- Also, most of the list functions have versions for sequences: `zip`, `map`, `length`, `nth`, `min`, `sort`, ...
- Other collection types can also be converted to and from sequences.
  - Lists, sets, maps (as a seq of pairs), arrays.
  - Any .NET collection type.
- A common pattern is to convert from another collection type, do some comprehensions, and then convert back.
- Functions that just scan a collection should generally accept a seq.
- If the an element is requested more than once, it is evaluated again.
  - This can be inefficient – so be careful.
  - Using lists avoids this.
  - You can also create a sequence that stores calculated elements via:
    ```
    val Seq.cache : seq<'a> -> CachedSeq<'a>
    ```
Sets

- Sets are a useful alternative to lists in some situations.
- They are implemented with binary trees, so allow fast checking for membership, and avoid duplicates.
- They can be created by starting from an empty list, and adding elements.
- There are functions in Fsharp.Collections.Set for just about everything:
  - Union, difference, ...
  - Member, subset
  - Conversion to and from lists (in sorted order)
- Example:
  ```fsharp
  let a = Set.ofSeq [ 1 .. 10 ]
  let b = Set.ofSeq [ 5 .. 15 ]
  > Set.subset c a;;
  val it : bool = true
  ```
Maps

- Maps are similar to sets, except that along with each “key” is an associated “value”.

```
let holidays =
    [  ("Christmas", "Dec. 25");
        ("Halloween", "Oct. 31");
        ("Darwin Day", "Feb. 12")]
|> Map.ofList;;  // Convert list to Map
```

```
val holidays : Map<string,string>
```

```
holidays.["Christmas"];;
```

```
val it : string = "Dec. 25"
```

- See FSharp.Collections.Map for details of the functions available, including add, exists, filter, find, remove, tryfind.