1 Sequences and computation expressions

1.1 Sequences

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

Listing 1.1: Creating sequences by range explicitly stating elements, a range expressions, a computation expression, and an infinite computation expression

```
> #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>
```

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 60th degree angle in radians. F# warns against recursive values, as defined in the example, since it will check the soundness of the value at runtime 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.,

Listing 1.2: Index a sequence with `Seq.item` and `Seq.take`

```
> 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>
```

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,

Listing 1.3: Sequences elements are first evaluated, when needed.

```
$ fsharpc --nologo seqDelayedEval.fsx && mono
    seqDelayedEval.exe
(calculating item 0)0 (calculating item 2)2 (calculating item
    4)4 (calculating item 6)6 (calculating item 8)8 (calculating
    item 10)10
```

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

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. The `try`-keyword constructions will be discussed in ??, and the `use`-keyword is a variant of `let` but used in asynchronous computations, which will not be treated here.

Infinite 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,

Listing 1.4 seqInfinteValue2.fsx:
Recursive value definitions gives a warning. Compare with Listing1.5, 1.6, and 1.7.

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

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

```
$ fsharpc --nologo seqInfinteValue2.fsx && mono
   seqInfinteValue2.exe

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; ...]
```

F# warns against using recursive values, since it will check the soundness of the value at runtime 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

Listing 1.5 seqInfinteValue.fsx:
Infinite value definition without recursion nor warning. Compare with Listing 1.4, 1.6, and1.7.

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

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

```
$ fsharpc --nologo seqInfinteValue.fsx && mono
   seqInfinteValue.exe
seq [1; 2; 3; 1; ...]
```

or alternatively define a recursive function,

Listing 1.6 seqInfinteFunction.fsx:
Recursive function definitions gives no a warning. Compare with Listing 1.4, 1.5, and 1.7.

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

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

```
$ fsharpc --nologo seqInfinteFunction.fsx && mono
   seqInfinteFunction.exe
seq [1; 2; 3; 1; ...]
```

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

Listing 1.7 seqInitInfinite.fsx:
Using `Seq.initInfinte` and a function to generate an infinte sequence. Compare with Listing 1.4, 1.5, and 1.6.

```fsharp
let s = Seq.initInfinite (fun i -> (float i)**2.0)
printfn "%A"  s
```

```
$ fsharpc --nologo seqInitInfinite.fsx && mono
   seqInitInfinite.exe
seq [0.0; 1.0; 4.0; 9.0; ...]
```

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,

Listing 1.8 seqConversion.fsx:
Conversion between sequences and lists and arrays using the `List` module.

```fsharp
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
```

```
$ fsharpc --nologo seqConversion.fsx && mono seqConversion.exe
seq [1; 2; 3], [1; 2; 3], [|1; 2; 3|], [|1; 2; 3|], [1; 2; 3]
```

There are quite a number of built-in functions for sequences many which will be discussed in ??.

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