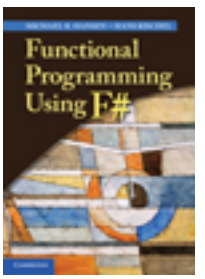


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

11 - Sequences pp. 251-278

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.012

Cambridge University Press

11

Sequences

A *sequence* is a possibly infinite, ordered collection of elements `seq` $[e_0; e_1; \ldots]$. The elements of a sequence are computed on demand only, as it would make no sense to actually compute an infinite sequence. Thus, at any stage in a computation, just a finite portion of the sequence has been computed.

The notion of a sequence provides a useful abstraction in a variety of applications where you are dealing with elements that should be processed one after the other. Sequences are supported by the collection library of F# and many of the library functions on lists presented in Chapter 5 have similar sequence variants. Furthermore, sequences can be defined in F# using *sequence expressions*, defining a process for generating the elements.

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>` and any .NET framework type that implements this interface can be used as a sequence. One consequence of this is, for the F# language, that lists and arrays (that are specializations of sequences) can be used as sequence arguments for the functions in the `Seq` library. Another consequence is that results from the *Language-Integrated Query* or *LINQ* component of the .NET framework can be viewed as F# sequences. LINQ gives query support for different kinds of sources like SQL databases and XML repositories. We shall exploit this in connection with a database for a simple product-register application, where we shall introduce the concept *type provider* that makes it possible to work with values from external data sources (like SQL databases) in a type safe manner, and the concept of *query expressions* that gives support for expressing queries on SQL databases in F#.

Sequence expressions and query expressions are special kinds of computation expressions (also called workflows), a concept of F# that will be addressed in Chapter 12.

11.1 The sequence concept in F#

A finite sequence can be formed like a list using the "sequence builder" *seq* (see also Section 11.7):

```
seq [10; 7; -25];;
val it : seq<int> = [10; 7; -25]
```

and we obtain a value of type `seq<int>`. This simple example works just like a list.

To study the special features of sequences, we will consider infinite sequences. An infinite sequence can be obtained by using the library function:

```
Seq.initInfinite: (int -> 'a) -> seq<'a>
```

The value of `Seq.initInfinite` f, denotes the infinite sequence:

$$\texttt{seq}\,[f(0); f(1); f(2); \ldots]$$

where the elements are computed on demand.

The sequence of natural numbers $0, 1, 2, \ldots$ is obtained as follows:

```
let nat = Seq.initInfinite (fun i -> i);;
val nat : seq<int>
```

and the i'th element in the sequence `nat`, when numbering starts with 0, is obtained using the function `Seq.nth: int -> seq<'a> -> 'a`. For example:

```
Seq.nth 5 nat;;
val it : int = 5
```

So far just the fifth element of `nat` is computed – this is the only element demanded.

To study the consequences of such *on demand* computation we modify the example so that the number is printed whenever it is demanded, using the `printfn` function (Section 10.7). Evaluation of the expression `printfn "%d"` i has the side-effect that the integer i is printed on the console, for example:

```
printfn "%d" 10;;
10
val it : unit = ()
```

The natural number sequence with print of demanded elements is declared as follows:

```
let idWithPrint i = printfn "%d" i
                    i;;
val idWithPrint : int -> int

let natWithPrint = Seq.initInfinite idWithPrint;;
val natWithPrint : seq<int>
```

The function `idWithPrint` is the identity function on integers that has the side-effect of printing the returned value, for example:

```
idWithPrint 5;;
5
val it : int = 5
```

Extracting the third and fifth elements of `natWithPrint` will print just those elements:

```
Seq.nth 3 natWithPrint;;
3
val it : int = 3
```

```
Seq.nth 5 natWithPrint;;
5
val it : int = 5
```

In particular, the elements 0, 1, 2, and 4 are not computed at all.

Extracting the third element again will result in a reprint of that element:

```
Seq.nth 5 natWithPrint;;
5
val it : int = 5
```

Thus the n'th element of a sequence is recomputed each time it is demanded. If elements are needed once only or if the re-computation is cheap or giving a desired side-effect, then this is fine.

Cached sequences

A *cached sequence* can be used if a re-computation of elements (such as 5 above) is undesirable. A cached sequence remembers the initial portion of the sequence that has already been computed. This initial portion, also called a *prefix*, comprises the elements $e_0, \ldots, e_n$, when e_n is the demanded element with the highest index n.

We will illustrate this notion of cached sequence using the previous example. A cached sequence of natural numbers is obtained using the library function

```
Seq.cache: seq<'a> -> seq<'a>
```

and it is used as follows:

```
let natWithPrintCached = Seq.cache natWithPrint;;
val natWithPrintCached : seq<int>
```

Demanding the third element of this sequence will lead to a computation of the prefix $0, 1, 2, 3$ as we can see from the output from the system:

```
Seq.nth 3 natWithPrintCached;;
0
1
2
3
val it : int = 3
```

Demanding the fifth element will extend the prefix to $0, 1, 2, 3, 4, 5$ but just two elements are computed and hence printed:

```
Seq.nth 5 natWithPrintCached;;
4
5
val it : int = 5
```

and there is no re-computation of cached elements:

```
Seq.nth 5 natWithPrintCached;;
val it : int = 5
```

Operation Meaning
`empty: seq<'a>`, where `empty` denotes the empty sequence
`init: int -> (int -> 'a) -> seq<'a>`, where `init` n f $=$ `seq` $[f(0); \ldots; f(n-1)]$
`initInfinite: (int -> 'a) -> seq<'a>`, where `initInfinite` f $=$ `seq` $[f(0); f(1); \ldots]$
`nth: int -> seq<'a> -> 'a`, where `nth` i $s = e_{i-1}$
`cache: seq<'a> -> seq<'a>`, where `cache` sq gives a cached sequence
`append: seq<'a> -> seq<'a> -> seq<'a>`, where `append` sq_1 sq_2 appends two sequences
`skip: int -> seq<'a> -> seq<'a>`, where `skip` i $s =$ `seq` $[e_i; e_{i+1}; \ldots]$
`ofList: 'a list -> seq<'a>`, where `ofList` $[a_0; \ldots; a_{n-1}] =$ `seq` $[a_0; \ldots; a_{n-1}]$
`toList: seq<'a> -> 'a list`, where `toList` `seq` $s = [a_0; \ldots; a_{n-1}]$ – just defined for finite sequences
`take: int -> seq<'a> -> seq<'a>`, where `take` n $s =$ `seq` $[e_0; \ldots; e_{n-1}]$ – undefined when s has fewer than n elements
`map: ('a -> 'b) -> seq<'a> -> seq<'>`, where `map` f $s =$ `seq` $[f(e_0); f(e_1); \ldots]$
`filter: ('a -> bool) -> seq<'a> -> seq<'a>`, where `filter` p $s = s'$ where s' is obtained from s by deletion of elements $e_i : p(e_i) =$ false
`collect: ('a -> seq<'c>) -> seq<'a> -> seq<'c>`, where `collect` f s is obtained by concatenation of the sequences f e_i for $i = 1, 2, 3 \ldots$

In this table s is a possibly infinite sequence $s =$ `seq` $[e_0; e_1; \ldots]$.

Table 11.1 *Selected functions from the sequence library* `Seq`

11.2 Some operations on sequences

The `Seq` library contains a rich collection of functions. Some of those are described in Table 11.1. In the remaining part of this chapter we illustrate the use of these functions.

Creating sequences

The empty sequence is denoted `Seq.empty`. A one element sequence, that is, a *singleton* sequence, is created using `Seq.singleton`, and a finite sequence can be generated from a list using `Seq.ofList`. For example:

```
Seq.empty;;
val it : seq<'a> = seq []

Seq.singleton "abc";;
val it : seq<string> = seq ["abc"]
```

```
Seq.ofList [(true,"a"); (false,"b"); (false,"ab")];;
val it : seq<bool * string>
       = [(true, "a"); (false, "b"); (false, "ab")]
```

The function `Seq.init` is used to generate a finite sequence. The value of `Seq.init` n f is the sequence

$$\texttt{seq}\,[f(0); f(1); f(2); \ldots; f(n-1)]$$

having n elements. For example:

```
Seq.init 3 (fun i -> 2*i);;
val it : seq<int> = seq [0; 2; 4]
```

Appending sequences

The operation `Seq.append` appending sequences works for finite sequences like `@` on lists, except for the "on demand" property:

```
let s1 = Seq.append (seq [1;2;3;4]) (seq [5;6]);;
val s1 : seq<int>

Seq.toList s1;;
val it : int list = [1; 2; 3; 4; 5; 6]
```

where the computation of the elements of `s1` is delayed until they are demanded by the conversion of the sequence to a list using `Seq.toList`.

If s denotes an infinite sequence, then `Seq.append` s s' is equal to s for any sequence s'. For example:

```
let s2 = Seq.append nat s1;;
val s2 : seq<int>

Seq.nth 1000 s2;;
val it : int = 1000
```

If $s = \texttt{seq}\,[e_0; \ldots; e_{n-1}]$ and $s' = \texttt{seq}\,[e'_0; e'_1; \ldots]$, then

$$s'' = \texttt{Seq.append}\; s\; s' = \texttt{seq}\,[e_0; \ldots; e_{n-1}; e'_0; e'_1; \ldots]$$

that is, s'' is the sequence, where

$$\texttt{Seq.nth}\; i\; s'' = \begin{cases} e_i & \text{for } 0 \leq i < n \\ e'_{i-n} & \text{for } n \geq i, \text{ provided that } s' \text{ has at least } i-n \text{ elements} \end{cases}$$

For example:

```
let s3 = Seq.append s1 nat;;
(Seq.nth 2 s3 = Seq.nth 2 s1)
&& (Seq.nth 10 s3 = Seq.nth (10-6) nat);;
val it : bool = true
```

since `s1` has six elements.

There is no function in the `Seq` library that directly corresponds to the function for cons'ing an element x to a list xs, that is, $x :: xs$. But a cons function for sequence is easily defined using `Seq.singleton` and `Seq.append`:

```
let cons x sq = Seq.append (Seq.singleton x) sq;;
val cons : 'a -> seq<'a> -> seq<'a>

cons 5 (seq [6; 7; 8]);;
val it : seq<int> = seq [5; 6; 7; 8]
```

11.3 Delays, recursion and side-effects

The functions above yield sequences that are lazy, that is, elements of the resulting sequence are not computed unless they are needed. The same is not necessarily true when sequences are defined by recursive functions. This is illustrated by the following *failing attempt* to declare a function that generates the sequence of integers starting from i:

```
let rec from i = cons i (from(i+1));;
val from : int -> seq<int>
```

Applying this function to any integer i yields a non-terminating evaluation:

$$\begin{aligned} & \texttt{from}\, i \\ \rightsquigarrow\ & \texttt{cons}\, i\, (\texttt{from}\, (i+1)) \\ \rightsquigarrow\ & \texttt{cons}\, i\, (\overline{\texttt{cons}\, (i+1)}\, (\underline{\texttt{from}\, (i+2)})) \\ \rightsquigarrow\ & \ldots \end{aligned}$$

The problem is that the arguments to `cons` are evaluated before the application of `cons` can create a "lazy" sequence using `Seq.append`. This is caused by the eager evaluation strategy of F#, and it results in an infinite recursive unfolding of `from`.

This problem can be avoided as described below by using the function

```
Seq.delay: (unit -> seq<'a>) -> seq<'a>
```

that suspends the computation of a sequence. We first motivate this function:

Suppose that e is an expression of type `seq<'a>`. The closure `fun () ->` e is then a value of type `unit -> seq<'a>` and this value can be viewed as a *lazy* representation of the value of e. A sequence generated by `seq` is for instance not lazy:

```
seq [idWithPrint 1];;
1
val it : seq<int> = [1]
```

but "packing" this expression into a closure

```
let sf = fun () -> seq [idWithPrint 1];;
val sf : unit -> seq<int>
```

gives a representation of the sequence where the element remain unevaluated. The element is evaluated when the function is applied:

```
sf();;
1
val it : seq<int> = [1]
```

The function `Seq.delay` converts a closure like `sf` into a lazy sequence:

```
let s1 = Seq.delay sf;;
val s1 : seq<int>

Seq.nth 0 s1;;
1
val it : int = 1
```

There are two ways of using `Seq.delay` to modify the above recursive declaration of `from` in order to avoid the problem with non-terminating evaluations. Either the recursive call of `from` can be delayed:

```
let rec from1 i = cons i (Seq.delay (fun () -> from1(i+1)));;
val from1 : int -> seq<int>
```

or the computation of the function value can be delayed:

```
let rec from2 i = Seq.delay (fun () -> cons i (from2(i+1)));;
val from2 : int -> seq<int>
```

Both of these functions can be used to generate lazy sequences:

```
let nat10 = from1 10;;
val nat10 : seq<int>

let nat15 = from2 15;;
val nat15 : seq<int>

Seq.nth 5 nat10;;
val it : int = 15

Seq.nth 5 nat15;;
val it : int = 20
```

There is a difference between these results of turning `from` into a function that generates lazy sequences. This difference becomes visible in the presence of the side-effect of printing the computed numbers using the `idWithPrint` function:

```
let rec fromWithPrint1 i =
    cons (idWithPrint i)
         (Seq.delay (fun () -> fromWithPrint1(i+1)));;
val fromWithPrint1 : int -> seq<int>

let rec fromWithPrint2 i =
    Seq.delay (fun () -> cons (idWithPrint i)
                              (fromWithPrint2(i+1)));;
val fromWithPrint2 : int -> seq<int>
```

The first element of a sequence created by using `fromWithPrint1` is eagerly computed when the sequence is created

```
let nat10a = fromWithPrint1 10;;
10
val nat10a : seq<int>
```

since the argument `idWithPrint i` to `cons` is computed, due to the eager evaluation strategy of F#, before `cons` can create a lazy sequence. Therefore, 10 is printed out when the above declaration of `nat10a` is elaborated and not later when an element of the sequence is demanded:

```
Seq.nth 3 nat10a;;
11
12
13
val it : int = 13
```

On the other hand, no element of a sequence created by `fromWithPrint2` is created at declaration time, since the value of that function is delayed, and all side-effects show up when elements are demanded:

```
let nat10b = fromWithPrint2 10;;
val nat10b : seq<int>

Seq.nth 3 nat10b;;
10
11
12
13
val it : int = 13
```

11.4 Example: Sieve of Eratosthenes

The Greek mathematician Eratosthenes (194 – 176 BC) invented a process, called the sieve of Eratosthenes, for generating the sequence of prime numbers. The process starts with the sequence of natural numbers that are greater than 1. The process is described as follows:

1. Select the head element p of the current sequence as the next prime number.
2. Remove multiples of p from the current sequence, yielding a new sequence.
3. Repeat the process from 1. with the new sequence.

Suppose a is the head element of the sequence at the start of some iteration of the process. No number between 2 and $a-1$ divides a since since multiples of those numbers have been removed from the sequence by step 2. in the process.

The first three iterations of the process can be illustrated as follows:

First	:	$\underline{2}$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
Second	:		$\underline{3}$		5		7		9		11		13		15		17	...
Third	:				$\underline{5}$		7				11		13				17	...

The function that removes all multiples of a number from a sequence is called `sift`, and it is declared using the `filter` function for sequences:

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq;;
val sift : int -> seq<int> -> seq<int>
```

The above iterations are exemplified in F# as follows:

```
Seq.initInfinite (fun i -> i+2);;
val it : seq<int> = seq [2; 3; 4; 5; ...]

sift 2 it;;
val it : seq<int> = seq [3; 5; 7; 9; ...]

sift 3 it;;
val it : seq<int> = seq [5; 7; 11; 13; ...]
```

The "Sieve of Erastothenes", corresponding to the process above, is obtained by a recursive application of `sieve` to the sequence `seq` $[2; 3; 4; 5; \ldots]$:

```
let rec sieve sq =
    Seq.delay (fun () ->
                   let p = Seq.nth 0 sq
                   cons p (sieve(sift p (Seq.skip 1 sq))));;
val sieve : seq<int> -> seq<int>

let primes = sieve (Seq.initInfinite (fun n -> n+2));;
val primes : seq<int>
```

The head and tail of a sequence are extracted in the above declaration by the functions `Seq.nth 0` and `Seq.skip 1`, respectively. See also Table 11.2.

The function that finds the n'th prime number is declared as follows:

```
let nthPrime n = Seq.nth n primes;;
val nthPrime : int -> int
```

The 5'th and 100'th prime numbers are fast to compute (remember that numbering starts at 0); but it requires some seconds to compute the 700'th prime number:

```
nthPrime 5;;
val it : int = 13

nthPrime 100;;
val it : int = 547

nthPrime 700;;
val it : int = 5281
```

A re-computation of the 700'th prime number takes the same time as before, and a computations of the 705'th prime number would take approximately the same time. A use of a cached prime number sequence will improve on that:

```
let primesCached = Seq.cache primes;;

let nthPrime1 n = Seq.nth n primesCached;;
val nthPrime1 : int -> int
```

Computing the 700'th prime number takes the same time as before; but a subsequent computation of the 705'th is fast since that computation is based on a prefix containing the first 700 prime numbers.

An alternative to the above declaration of `primesCached` is to let the sieve function return a cached sequence:

```
let rec sieve sq =
    Seq.cache
       (Seq.delay (fun () ->
                   let p = Seq.nth 0 sq
                   cons p (sieve(sift p (Seq.skip 1 sq)))));;
```

11.5 Limits of sequences: Newton-Raphson approximations

The Newton-Raphson method for computing the square root of a number is based on the fact that $\sqrt{a}$, for $a \geq 0$, is the limit of a sequence:

$$\texttt{seq}\,[x_0; x_1; x_2; \ldots]$$

where $x_0 > 0$ and

$$x_{i+1} = (a/x_i + x_i)/2 \tag{11.1}$$

for any $i \geq 0$.

It is easy to generate such infinite sequences using F# and to use the elements to approximate the square root of a number a within a given tolerance.

The computation of the next element in the sequence on the basis of a and the current element x follows from (11.1):

```
let next a x = (a/x + x)/2.0;;
val next : float -> float -> float
```

This function should be iterated over a sequence and to this end two auxiliary functions are defined:

```
let rec iter f x = function
                   | 0 -> x
                   | n -> iter f (f x) (n-1);;
val iter : ('a -> 'a) -> 'a -> int -> 'a

let iterate f x = Seq.initInfinite (fun i -> iter f x i);;
val iterate : ('a -> 'a) -> 'a -> seq<'a>
```

where `iter` makes an n-fold application of a function:

$$\texttt{iter}\ f\ x\ n = f(f(f(\cdots f(x)\cdots))) = f^n(x)$$

and `iterate` gives a sequence on the basis of the iteration of a function:

$$\texttt{iterate}\ f\ x = \texttt{seq}\,[x; f(x); f(f(x)); f(f(f(x)))\ldots]$$

The "Newton-Raphson" sequence for $\sqrt{2}$ that starts at 1.0 is:

```
iterate (next 2.0) 1.0;;
val it : seq<float> = seq [1.0;1.5;1.416666667;1.414215686;...]
```

The following function traverses trough a sequence *sq* using an enumerator (see Section 8.12) and returns the first element where the distance to the next is within a given tolerance *eps*:

```
let rec inTolerance (eps:float) sq =
    let f = enumerator sq
    let nextVal() = Option.get(f())
    let rec loop a = let b = nextVal()
                     if abs(a-b) > eps then loop b else a
    loop(nextVal());;
```

The sequence argument `sq` is assumed to be infinite and the enumerator `f` will always return some value, which is exploited by the function `nextVal`. The square roots of a can be computed according to Newton-Raphson's method within tolerance 10^{-6}.

```
let sRoot a = inTolerance 1E-6 (iterate (next a) 1.0);;
val sRoot : float -> float

sRoot 2.0;;
val it : float = 1.414213562
```

This example illustrates the expressiveness of infinite sequences, but the above programs can definitely be improved, in particular when considering efficiency issues. See, for example, Exercise 11.5 and Exercise 11.6. But notice that none of these sequence-based solutions is as efficient as a solution that just remembers the last computed approximation and terminates when the next approximation is within the given tolerance.

11.6 Sequence expressions

A *sequence expression* is a special kind of computational expression (cf. Chapter 12) that allows a step-by-step generation of sequences.

The corresponding general form of expressions to generate sequences is

seq { *seqexp* }

where *seqexp* is a sequence expression.

Suppose that we aim at generating a sequence of type `seq<'a>`. Then the following sequence expressions are fundamental for the generation:

- The sequence expression `yield` x, with x : `'a`, adds the element x to the sequence.
- The sequence expression `yield!` sq, with sq : `seq<'a>`, adds the sequence sq to the sequence.

The following examples show three ways of generating the integer sequence `seq` $[1, 2; 3]$ using `yield` and `yield!` constructs:

```
let s123a = seq {yield 1
                 yield 2
                 yield 3 };;

let s123b = seq {yield 1
                 yield! seq [2; 3] };;

let s123c = seq {yield! seq [1; 2]
                 yield 3           };;
```

These examples all show a *combination* of sequence expressions of the form:

$$seqexp_1$$
$$seqexp_2$$

The meaning of this combination is the sequence obtained by appending the sequence denoted by $seqexp_1$ with the sequence denoted by $seqexp_2$.

Expressions denoting sequences may in general be obtained using functions from the `Seq` library; but sequence expressions have the advantage that the generated sequences by construction are lazy, but not cached, and elements are therefore generated by demand only. Hence an explicit use of `Seq.delay` can be avoided when using sequence expressions.

This implicit delaying of a sequence is exploited in the following declaration of the function `from` from Section 11.2:

```
let rec from i = seq {yield i
                      yield! from (i+1)};;
val from : int -> seq<int>

let s10 = from 10;;
val s10 : seq<int>

Seq.nth 5 s10;;
val it : int = 15
```

Construct	Legend
`yield` *exp*	generate element
`yield!` *exp*	generate sequence
$seqexp_1$ $seqexp_2$	combination of two sequences by appending them
`let` *pat* = *exp* *seqexp*	local declaration
`for` *pat* `in` *exp* `do` *seqexp*	iteration
`if` *exp* `then` *seqexp*	filter
`if` *exp* `then` *seqexp* `else` *seqexp*	conditional

Table 11.2 *Constructs for sequence expressions*

A summary of constructs used in sequence expressions is given in Table 11.2.

Example: Sieve of Eratosthenes

The Sieve of Eratosthenes can also be formulated using sequence expressions.

The function `sift` a sq makes a filtering of the elements of the sequence sq by removing multiples of a. This is expressed using an iteration over sq and a filter:

```
let sift a sq = seq {for n in sq do
                         if  n % a <> 0 then
                             yield n          };;
val sift : int -> seq<int> -> seq<int>
```

The use of sequence expressions gives, in this case, a more "verbose" alternative to the succinct formulation using `Seq.filter` in Section 11.4.

The use of sequence expressions in the declaration of `sieve` sq is attractive since the explicit delay of the recursive call can be avoided and the combination sequence expressions is a brief alternative to using `Seq.append`:

```
let rec sieve sq =
    seq {let p = Seq.nth 0 sq
         yield p
         yield! sieve(sift p (Seq.skip 1 sq)) };;
val sieve : seq<int> -> seq<int>
```

The remaining parts of the example are unchanged, for example:

```
let primes = sieve(Seq.initInfinite (fun n -> n+2));;
val primes : seq<int>

let nthPrime n = Seq.nth n primes;;
val nthPrime : int -> int
```

Example: Search in a directory

We shall now use sequences in connection with the search for certain files in a directory. Although there is a finite number of files to be considered, it is natural to exploit the laziness of sequences since a search typically succeeds before the whole directory structure is exhausted.

The function `allFiles` declared below gives a sequence of all files in a directory, including those in subdirectories. The sequence of files and directories are extracted using the functions `Directory.GetFiles` and `Directory.GetDirectories` from the library `System.IO`:

```
open System.IO;;

let rec allFiles dir = seq {
  yield! Directory.GetFiles dir
  yield! Seq.collect allFiles (Directory.GetDirectories dir)};;
val allFiles : string -> seq<string>
```

The function

```
Seq.collect: ('a -> seq<'c>) -> seq<'a> -> seq<'c>
```

combines the map and concatenate functionality. The value of `Seq.collect` fsq is obtained (lazily) by applying f to each element e_i of the sequence sq. The result is obtained by concatenation of all the sequences $f(e_i)$, for $i = 0, 1, 2, \ldots$.

Hence the expression

```
Seq.collect allFiles (Directory.GetDirectories dir)
```

recursively extracts all files in sub-directories of `dir` and concatenates the results.

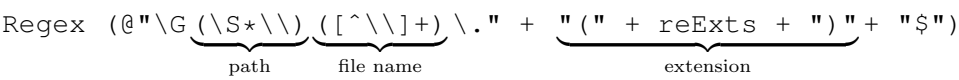
The sequence of all files in the directory `C:\mrh\Forskning\Cambridge\`, for example, can be extracted as follows:

```
// set current directory
Directory.SetCurrentDirectory @"C:\mrh\Forskning\Cambridge\";;

let files = allFiles ".";;
val files : seq<string>

Seq.nth 100 files;;
val it : string = ".\BOOK\Satisfiability.fs"
```

A composite file name like `.\BOOK\Satisfiability.fs` can be split into three parts: a path `.\BOOK\`, a file name `Satisfiability` and an extension `fs`. These three parts can be extracted from a composite file name using regular expressions (see Page 222):

- The path matches a sequence of characters ending with a backslash "\", that is, it matches the regular expression "`\S*\\`".
- The file name matches a non-empty sequence of characters not containing backslash "\", that is, it matches the regular expression "`[^\\]+`".

Suppose that `reExts` is a regular expression matching certain file extensions. Then the following regular expression (where the string is obtained by joining several strings) matches composite file names having one of these file extensions:



Note that the extension is the suffix of the composite file name that starts just after the last period (`.`). The regular expression has three capturing groups (enclosed in normal brackets `(` and `)`) so that the path, the name and the extension can be extracted from a matched composite file name. The function `captureSingle` from the `TextProcessing` library (cf. Table 10.4) is used to extract captured strings in the following declaration of `searchFiles` *files exts* that gives those files in the sequence *files* that have an extension in the list *exts*:

```
open System.Text.RegularExpressions;;

let rec searchFiles files exts =
    let reExts =
        List.foldBack (fun ext re -> ext+"|"+re) exts ""
    let re = Regex (@"\G(\S*\\)([^\\]+)\.(" + reExts + ")$")
    seq {for fn in files do
            let m = re.Match fn
            if m.Success
            then let path = TextProcessing.captureSingle m 1
                 let name = TextProcessing.captureSingle m 2
                 let ext  = TextProcessing.captureSingle m 3
                 yield (path, name, ext)      };;
val searchFiles : seq<string> -> string list
                  -> seq<string * string * string>
```

The search for F# files (with extensions `fs` or `fsi`) is initiated as follows:

```
let funFiles =
    Seq.cache (searchFiles (allFiles ".") ["fs";"fsi"]);;
val funFiles : seq<string * string * string>
```

In this case a cached sequence is chosen so that a search can exploit the already computed part of the sequence from previous searches:

```
Seq.nth 6 funFiles;;
val it : string * string * string = (".\BOOK\","Curve","fsi")

Seq.nth 11 funFiles;;
val it : string * string * string
       = (".\BOOK\", "Satisfiability", "fs")
```

11.7 Specializations of sequences

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>` and any .NET framework type that implements this interface can be used as a sequence. Since lists and arrays are specializations of sequences they can be used as sequence arguments for the functions in the `Seq` library.

Specialization of `Seq`*-library functions*

Using the function `Seq.append`

```
Seq.append: seq<'a> -> seq<'a> -> seq<'a>
```

a sequence can, for example, be formed by appending two lists:

```
let sq1 = Seq.append [1; 2; 3] [4; 5; 6];;
val sq1 : seq<int>
```

by appending two arrays (see Section 8.10)

```
let sq2 = Seq.append [|1; 2; 3|] [|4; 5; 6|];;
val sq2 : seq<int>
```

and by appending a list and an array:

```
let sq3 = Seq.append [1; 2; 3] [|4; 5; 6|];;
val sq3 : seq<int>
```

Similarly, the range *rng* in a `for`-construct: `for` *i* `in` *rng* `do` *seqexp* can be a sequence. This is exploited in the following example where the range is the first eleven natural numbers:

```
let squares = seq {for i in 0..10 do yield i*i };;
val squares : seq<int>
```

Range expressions

In a construction of a sequence constant, like

```
seq [1; 2; 3; 4];;
val it : seq<int> = [1; 2; 3; 4]
```

the sequence builder `seq` is actually a built-in function on sequences:

```
seq;;
val it : (seq<'a> -> seq<'a>) = <fun:clo@16-2>
```

and when applying `seq` to the list in the above example, we are just exploiting that lists are specializations of sequences.

Hence the range expressions $[b\ ..\ e]$ and $[b\ ..\ s\ ..\ e]$ for lists described in Section 4.2 can, in a natural manner, be used to generate finite sequences:

```
let evenUpTo n = seq [0..2.. n];;
val evenUpTo : int -> seq<int>
```

Sequences constructed using this function will not be lazy:

```
let e20 = evenUpTo 20;;
val e20 : seq<int> = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20]
```

The answer from the system shows that the sequences constructed in this way will not be lazy, as the answer shows all eleven elements. The reason is that the construct `[0..2.. n]` eagerly will construct a list.

11.8 Type providers and databases

The *Language-Integrated Query* or *LINQ* component of the .NET framework gives query support for different kinds of sources like SQL databases and XML documents. We shall now exploit that LINQ queries for SQL databases return values of type `IEnumerable<T>`, and hence the result of these queries can be viewed as F# sequences.

An obstacle in doing so is that the type system of an SQL database is different from that of F#. To overcome this obstacle a *type provider* for SQL is used. An F# type provider for SQL makes it possible to work directly with the tables of the database in a type safe manner. There are type providers for other external sources than SQL databases, and it is possible to make user-defined type providers; but we will just illustrate the use of a type provider for SQL in connection with a database for a simple *product-register* application.

A database for a Product Register

Suppose for the moment that the product register is a database `ProductRegister` containing the two tables shown in Figure 11.1.

Part:

PartId	PartName	IsBasic
0	"Part0"	1
1	"Part1"	1
2	"Part2"	0
3	"Part3"	0

PartsList:

PartsListId	PartId	Quantity
2	0	5
2	1	4
3	1	3
3	2	4

- The SQL-types of the attributes are:

  PartId, PartsListId, Quantity: `int`, PartName: `varchar(50)` and IsBasic: `bit`

- The table Part has PartId as key.
- The table PartsList has (PartsListId, PartId) as composite key.

Figure 11.1 A Product Register Database with two tables: Part and PartsList

The table `Part` stores information about four parts named: "`Part0`", ... ,"`Part3`", where "`Part0`" and "`Part1`" are basic parts (with empty parts lists) since their `IsBasic` attribute is 1 (the SQL representation of true), while "`Part2`" and "`Part3`" are composite parts since their `IsBasic` attribute is 0 (representing false). The `PartId` attribute of the `Part` table is a unique identifier, that is, a *key*, for the description of a part.

The table `PartsList` contains the parts lists for all the composite parts. The attribute pair (`PartsListId`, `PartId`) is a composite key. A row (pid, id, q) in this table, therefore, describes that exactly q pieces of the part identified by id is required in the parts list of the composite part identified by pid. For example, the parts list for "`Part3`" comprises 3 pieces of "`Part1`" and 4 pieces of "`Part2`".

Starting on Page 274 it is shown how this database can be created and updated. But before that we address the issue of making queries to this database from F#.

F# Type Providers for SQL

A *type provider* is a component that automatically generates types and functions for external data sources like SQL databases and Web services. Below we shall use a built-in SQL type provider for F#. In order to access and use this type provider, the following assemblies should be referenced: `FSharp.Data.TypeProviders.dll`, `System.Data.dll` and `System.Data.Linq.dll`, and namespaces should be opened (see Figure 11.2 for the complete code) prior to the execution of the following declarations:

```
type schema =
    SqlDataConnection<"Data Source=IMM-NBMRH\SQLEXPRESS;
                       InitialCatalog=ProductRegister;
                       Integrated Security=True">;;
type schema

let db = schema.GetDataContext();;
val db:
    schema.ServiceTypes.SimpleDataContextTypes.ProductRegister
```

The SQL type provider: `SqlDataConnection` has a *connection string* as argument. This connection string has three parts: A definition of the *data source*, in this case an SQL

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif

open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema =
    SqlDataConnection<"Data Source=IMM-NBMRH\SQLEXPRESS;
                       Initial Catalog=ProductRegister;
                       Integrated Security=True">;;

let db = schema.GetDataContext();;
```

Figure 11.2 Creating a type provider for the `ProductRegister` database

server, the *initial catalog*, in this case the database name, and the *integrated security*, which in this case is true, meaning that the .NET credentials of the current user will be used for authentication.

The type `schema` contains all the generated types that represent the database and `db` is an object containing the database tables. The two database tables can be accessed as follows:

```
let partTable = db.Part;;
val partTable : Data.Linq.Table<schema.ServiceTypes.Part>

let partsListTable = db.PartsList;;
val partsListTable:
    Data.Linq.Table<schema.ServiceTypes.PartsList>
```

The answers from the F# system do not reveal the F# values of these two tables.

They are in fact lazy sequences. For example:

```
partTable;;
val it : Data.Linq.Table<schema.ServiceTypes.Part> =
  seq [Part {IsBasic = true; PartId = 0; PartName = "Part0";};
       Part {IsBasic = true; PartId = 1; PartName = "Part1";};
       Part {IsBasic = false; PartId = 2;PartName = "Part2";};
       Part {IsBasic = false; PartId = 3;PartName = "Part3";}]
```

where all the elements are shown in this case just because the interactive environment always prints a short prefix of a lazy sequence. The elements of this sequence are objects belonging to a class `Part` that has the attributes of the table as public fields:

```
let r = Seq.nth 2 partTable;;
val r : schema.ServiceTypes.Part

r.PartId;;
val it : int = 2

r.PartName;;
val it : string = "Part2"

r.IsBasic;;
val it : bool = false
```

Note that the SQL types `bit` and `varchar(50)` are translated to the F# types `bool` and `string`, respectively, by the type provider.

The list of F# elements of the `PartsList` table is obtained as follows:

```
Seq.toList partsListTable;;
val it : schema.ServiceTypes.PartsList list =
  [PartsList {PartId = 0; PartsListId = 2; Quantity = 5;};
   PartsList {PartId = 1; PartsListId = 2; Quantity = 4;};
   PartsList {PartId = 1; PartsListId = 3; Quantity = 3;};
   PartsList {PartId = 2; PartsListId = 3; Quantity = 4;}]
```

Database queries can be expressed using the functions from the sequence library since the tables in the database can be accessed like sequences when using the above type provider. The names of all composite parts can, for example, be extracted as follows:

```
Seq.fold
  (fun ns (r:schema.ServiceTypes.Part)
           -> if r.IsBasic then ns else r.PartName::ns)
  []
  partTable;;
val it : string list = ["Part3"; "Part2"]
```

Query expressions

We shall now introduce *query expressions* as means for extracting information from the database. A query expression is a computation expression (just like sequence expressions) and it occurs in expressions of the form:

```
query { queryexp }
```

The construct `select` v adds the element v to the answer to the query just like `yield` v adds an element to a sequence:

```
query {select (1, "a") };;
val it : IQueryable<int * string> = seq [(1, "a")]
```

The value of this query expression has type `IQueryable<int * string>`. The type `IQueryable<T>` is a specialization of `IEnumerable<T>` and, therefore, values of type `IQueryable<T>` can be treated as sequences.

There is a rich collection of query-expression constructs that translates to SQL queries. We will now introduce a small part these constructs by illustrating how the following operations of relational algebra can be expressed: projection, selection and join.

Projection

A *projection* operation extracts certain columns of a table and such a projection can be expressed using an *iteration*.

For example, a query for the projection of the `Part` table with respect to `PartName` and `IsBasic` is declared as follows:

```
let q1 = query {for part in db.Part do
                select (part.PartName, part.IsBasic) };;

q1;;
val it : IQueryable<string * bool> =
  seq [("Part0", true); ("Part1", true);
       ("Part2", false); ("Part3", false)]
```

Selection

A *selection* operation extracts certain rows of a table and such a selection can be expressed using an iteration together with a *where*-clause and a selection.

For example, the query selecting the composite parts from the `Part` table is declared by:

```
let q2 =
  query {for part in db.Part do
         where (not part.IsBasic)
         select (part.PartId, part.PartName, part.IsBasic)};;

q2;;
val it : IQueryable<int * string * bool> =
  seq [(2, "Part2", false); (3, "Part3", false)]
```

Join

A *join* operation combines the rows of two tables A and B. There are many different kinds of such combinations that are supported by SQL and query expressions. We shall here just consider what is called an *equi*-join, where a row $a \in A$ is combined with a row $b \in B$ only if $a.L_A = b.L_B$, where L_A is a given attribute of A and L_B is a given attribute of B.

By an equi-join of `PartsList` and `Part` tables with `PartsListId` of `PartsList` equal to `PartId` of `Part` we can extract tuples from `PartsList` where identifiers for parts list are replaced by their names:

```
let q3 = query {for pl in db.PartsList do
                join part in db.Part on
                   (pl.PartsListId = part.PartId)
                select(part.PartName, pl.PartId, pl.Quantity)
               };;
q3;;
val it : IQueryable<string * int * int> =
  seq [("Part2", 0, 5); ("Part2", 1, 4);
       ("Part3", 1, 3); ("Part3", 2, 4)]
```

Hence "`Part2`" is a composite part consisting of 5 pieces of the part with `PartId` equal to 0 and 4 pieces of the part with `PartId` equal to 1. By the use of nested joins we can make a query where these identifiers also are replaced by their names. The result of `q4` cannot be used in a join since the elements have a tuple type and not a record type. We therefore introduce a record type:

```
type partListElement =
    {PartName:string; PartId:int; Quantity:int}
```

In the following nested join, the local query `qa` is the variant of `q3` that gives elements of type `partListElement`:

```
let q4 =
  query {let qa = query {for pl in db.PartsList do
                         join part in db.Part on
                             (pl.PartsListId = part.PartId)
                         select {PartName = part.PartName;
                                 PartId = pl.PartId;
                                 Quantity = pl.Quantity} }
         for pl in qa do
         join part in db.Part on
            (pl.PartId = part.PartId)
         select(pl.PartName, part.PartName, pl.Quantity) };;
q4;;
val it : IQueryable<string * string * int> =
  seq
    [("Part2", "Part0", 5); ("Part2", "Part1", 4);
     ("Part3", "Part1", 3); ("Part3", "Part2", 4)]
```

Aggregate operations

In SQL there are so-called aggregate operations that depend on a whole table or all the values in a column of a table, such as counting the number of elements in a table or finding the average of the elements in a column. There are also query-expression constructs for these functions, for example, `count` that counts the number of elements selected so far, `exactlyOne` that returns the single element selected, and raises an exception if no element or more than one element have been selected, and `contains` v that checks whether v is among the so far selected elements.

The following function counts the number of rows in `Part`. Since we shall use consecutive numbers $0, 1, \ldots, n-1$ as identifiers for existing parts, the number of rows n is the next identifier that can be used as a key. This function is therefore named `nextID`:

```
let nextId() = query {for part in db.Part do
                      count };;
val nextId : unit -> int
```

The function `getDesc` extracts the description of a given identifier

```
let getDesc id =
    query {for part in db.Part do
           where (part.PartId=id)
           select (part.PartName,part.IsBasic)
           exactlyOne                          };;
val getDesc : int -> string * bool
```

where the description consists of the name and truth values of the `Name` and `IsBasic` attributes. For example:

```
nextId();;
val it : int = 4

getDesc 3;;
val it : string * bool = ("Part3", false)
```

```
getDesc 4;;
System.InvalidOperationException: Sequence contains no elements
```

The predicate `containsPartId` checks whether a given identifier is in the `Part` table:

```
let containsPartId id = query {for part in db.Part do
                               select part.PartId
                               contains id };;
val containsPartId : int -> bool

containsPartId 3;;
val it : bool = true
containsPartId 4;;
val it : bool = false
```

Example: Parts Break Down

We shall now consider the problem of computing a parts list containing all the basic parts needed to produce a given part. By a parts list we shall now understand a list of pairs: $[(id_1, k_1), \ldots, (id_n, k_n)]$ where id_i is the identifier of a part and k_i is the quantity needed of that part.

The following function extracts the parts list for a given part:

```
let getPartsList id =
    query {for pl in db.PartsList do
           where (pl.PartsListId = id)
           select (pl.PartId,pl.Quantity) };;
val getPartsList : int -> IQueryable<int * int>

getPartsList 3;;
val it : IQueryable<int * int> = seq [(1, 3); (2, 4)]
```

We shall need functions for adding a pair (id, k) to given parts list, for merging two parts lists and for multiplying all quantities in a parts list by a constant. These functions are "usual" auxiliary list functions:

```
let rec add pl (id,q) =
    match pl with
    | []                          -> [(id,q)]
    | (id1,q1)::pl1 when id=id1 -> (id,q+q1)::pl1
    | idq::pl1                    -> idq :: add pl1 (id,q);;
val add : ('a * int) list -> 'a * int -> ('a * int) list
          when 'a : equality

let mergePartsList pl1 pl2 = List.fold add pl1 pl2;;
val mergePartsList :
  ('a * int) list -> ('a * int) list -> ('a * int) list
  when 'a : equality

let mult k pl = List.map (fun (id,q) -> (id,k*q)) pl;;
val mult : int -> ('a * int) list -> ('a * int) list
```

The following function `partBreakDown` that computes a parts list containing all basic parts needed for producing a given part is declared in mutual recursion with the function `partsListBreakDown` that computes a parts list containing all basic parts needed for producing a given parts list. These functions access the database to extract the description and the parts list of a given part using `getDesc` and `getPartsList`.

```
let rec partBreakDown id =
    match getDesc id with
    | (_,true)  -> [(id,1)]
    | _         ->
        partsListBreakDown(Seq.toList(getPartsList id))
and partsListBreakDown = function
    | (id,q)::pl -> let pl1 = mult q (partBreakDown id)
                    let pl2 = partsListBreakDown pl
                    mergePartsList pl1 pl2
    | []         -> [];;
val partBreakDown : int -> (int * int) list
val partsListBreakDown : (int * int) list -> (int * int) list

partBreakDown 3;;
val it : (int * int) list = [(1, 19); (0, 20)]

partBreakDown 1;;
val it : (int * int) list = [(1, 1)]
```

Creating a database

Executing the F# program in Figure 11.3 will setup the `ProductRegister` database with empty `Part` and `PartList` tables.

Updating a database

The type `scheme` contains service types and constructors for elements of the tables in the database. For example

```
new schema.ServiceTypes.Part(PartId=id, PartName=s, IsBasic=b)
```

generates a new part object that can belong to the `Part` table.

Table objects like `db.Part` and `db.PartsList` have members `InsertOnSubmit` and `InsertAllOnSubmit` that you can give a single row and a collection of rows, respectively, to be inserted in the tables. These insertions are effectuated only when the function `SubmitChanges` from the LINQ `DataContext` type has been applied.

Consider for example the following function that inserts a basic part into the `Part` table given its part name:

```
open System.Configuration
open System.Data
open System.Data.SqlClient

let connString = @"Data Source=IMM-NBMRH\SQLEXPRESS;
                   Initial Catalog=ProductRegister;
                   Integrated Security=True";;
let conn = new SqlConnection(connString)

conn.Open();;

let execNonQuery conn s =
   let comm = new SqlCommand(s, conn, CommandTimeout = 10)
   comm.ExecuteNonQuery() |> ignore;;

execNonQuery conn "CREATE TABLE Part (
   PartId int NOT NULL,
   PartName varchar(50) NOT NULL,
   IsBasic bit NOT NULL,
   PRIMARY KEY (PartId))";;

execNonQuery conn "CREATE TABLE PartsList (
   PartsListId int NOT NULL,
   PartId int NOT NULL,
   Quantity int NOT NULL,
   PRIMARY KEY (PartsListId, PartId))";;
```

Figure 11.3 F# program creating the tables of `ProductRegister`

```
let addBasic s =
    let id = nextId()
    let part = new schema.ServiceTypes.Part(PartId = id,
                                            PartName = s,
                                            IsBasic = true)
    db.Part.InsertOnSubmit(part)
    db.DataContext.SubmitChanges()
    Some id;;
val addBasic : string -> int option
```

The function generates a key for the part and this key is returned by the function.

The insertion of a composite part into the database is based on its name s and its parts list: $[(id_1, k_1), \ldots (id_n, k_n)]$. Such an insertion is only meaningful when the identifiers id_i, for $1 \leq i \leq n$, are already defined in the `Part` table and when all the quantities k_i, for $1 \leq i \leq n$, are positive integers. This well-formedness constraint is checked by the following function:

```
let isWellFormed pl =
    List.forall (fun (id,k) -> containsPartId id && k>0) pl;;
val isWellFormed : (int * int) list -> bool
```

If this well-formedness constraint is satisfied, then the following function inserts a new composite part into the `Part` table and its parts list into the `PartsList` table:

```
let addComposite s pl =
  if isWellFormed pl
  then
    let id = nextId()
    let part =
      new schema.ServiceTypes.Part(PartId=id,
                                   PartName=s,
                                   IsBasic = false)
    let partslist =
      List.map
        (fun (pid,k) ->
           new schema.ServiceTypes.PartsList(PartsListId=id,
                                             PartId=pid,
                                             Quantity=k))
         pl
    db.Part.InsertOnSubmit(part)
    db.PartsList.InsertAllOnSubmit(partslist)
    db.DataContext.SubmitChanges()
    Some id
  else None;;
val addComposite : string -> (int * int) list -> int option
```

The tables in Figure 11.1 are generated from an initial `ProductRegister` database with two empty tables by evaluation of the following declarations:

```
let id0 = Option.get (addBasic "Part0");;
val id0 : int = 0

let id1 = Option.get (addBasic "Part1");;
val id1 : int = 1

let id2 =
    Option.get (addComposite "Part2" [(id0,5);(id1,4)]);;
val id2 : int = 2

let id3 =
    Option.get (addComposite "Part3" [(id1,3);(id2,4)]);;
val id3 : int = 3
```

Summary

This chapter has introduced the notion of sequence, which is an ordered, possibly infinite, collection of elements where the computation of elements is on demand only. Sequences are convenient to use in applications where you are dealing with elements that are processed one after they other. Functions from the sequence part of the collection library of F# have been introduced together with cached sequences that prevents a recomputation of already computed sequence elements. Furthermore, sequences can be defined in F# using sequence expressions defining a step-by-step process for generating the elements.

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>` and any .NET framework type that implements this interface can be used as a sequence. This has been studied in connection with the *Language-Integrated Query* or *LINQ* component of the .NET framework. LINQ gives query support for different kinds of data sources like SQL databases and XML repositories. We have used LINQ in connection with a database for a simple product-register application, where an F# type provider made it possible to work with values from the external data sources (an SQL databases in this case) in a type safe manner. The concept of query expressions was introduced since it gives powerful support for expressing queries on SQL databases in F#.

Exercises

11.1 Make a declaration for the sequence of odd numbers.

11.2 Make a declaration for the sequence of numbers $1, 1, 2, 6, \ldots, n!, \ldots$.

11.3 Make a declaration for the sequence of `seq` $[1; 1; 2; 6; \ldots; n!; \ldots]$, where the $i+1$'st element is generated from the $i'th$ element by multiplication with $i+1$.

11.4 Declare a function that, for given i and n, selects the sublist $[a_i; a_{i+1}; \ldots; a_{i+n-1}]$ of a sequence `seq` $[a_0; a_1; \ldots]$.

11.5 The declaration of the function `iterate` f on Page 260 has the drawback that $f^n\, x$ is computed when the $n'th$ element is demanded. Give an alternative declaration of this function using the property that the $n+1$'st element of the sequence can be computed from the n'th element by an application of f.

11.6 Have a look at the `unfold` function from the `Seq` library. Make a declaration of the `sRoot` function from Section 11.5 using `Seq.unfold`. That declaration should be based on the idea that the sequence generation is stopped when the desired tolerance is reached. Measure the possible performance gains.

11.7 The exponential functions can be approximated using the Taylor's series:

$$e^x = \frac{1}{0!} + \frac{x^1}{1!} + \cdots + \frac{x^k}{k!} + \cdots \tag{11.2}$$

1. Declare a function that for a given x can generate the sequence of summands in (11.2). Hint: Notice that the next summand can be generated from the previous one.
2. Declare a function that accumulates the elements of a sequence of floats. I.e. given a sequence `seq` $[x_0; x_1; x_2; \ldots]$ it generates the sequence `seq` $[x_0; x_0 + x_1; x_0 + x_1 + x_2; \ldots]$.
3. Declare a function to generate the sequence of approximations for the function e^x on the basis of (11.2).
4. Declare a function to approximate e^x within a given tolerance.

11.8 The Madhava-Leibniz series (also called Gregory-Leibniz series) for π is:

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Use this series to approximate π. (Note that there are other series for π, which converge much faster than the above one.)

11.9 Declare a sequence denoting the following enumeration of the integers:

$$0, -1, 1, -2, 2, -3, 3, \ldots$$

11.10 Use the functions in the `Seq` library to declare a function `cartesian` *sqx sqy* that gives a sequence containing all pairs (x, y) where x is a member of sqx and y is a member of sqy. Make an alternative declaration using sequence expressions.

11.11 Solve Exercise 11.3 using sequence expressions.

11.12 Solve Exercise 11.7 using sequence expressions.

11.13 Solve Exercise 11.8 using sequence expressions.

11.14 Solve Exercise 11.9 using sequence expressions.

11.15 Give a database-based solution to the cash-register example introduced in Section 4.6.

11.16 Give a database-based solution to Exercise 4.23.