Generic Sequences Manual

David E. Sorokin <david.sorokin@gmail.com>,
Yoshkar-Ola, Russia

December 11, 2012

Contents

1 Introduction **2**

2 Generic Sequences **3**
2.1 Basic Definitions 3
2.2 Predefined Sequences 5
2.3 List of Functions 5

3 Iterating Sequences **14**

4 Sequence Comprehension **15**

5 Lazy Streams **17**

Chapter 1

Introduction

Generic Sequences is a library for Common Lisp. It introduces the generic *sequences*, which are a hybrid between the ordinary lists, lazy streams and iterable sequences that can be found in other programming languages such as C#, F#, Java and Scala.

Each sequence returns an *enumerator* or `NIL`. The `NIL` value means that the sequence is empty. The enumerator is a cons cell, which `CAR` part has the ordinary meaning for Common Lisp, i.e. it returns the current element, while the `CDR`-part is different. That part is already a function that returns either the next enumerator or `NIL` in case of reaching the end of the sequence. In other words, there is a delay before receiving the next element of the sequence, which actually makes many sequences lazy.

Unlike the lazy streams, the enumerator always returns a new continuation of the sequence through the `CDR` part. It recalculates the next element anew, while the stream would memoize it. When iterating, this makes the enumerator more efficient computationally as the memoization would require some kind of locking in the multi-threaded environment.

At the same time, unlike the C# enumerators, our enumerator is like a list cell, which allows defining the sequence in a more easy and declarative way as if we defined an ordinary list. Macros introduced below simplify this process. There exists even the sequence comprehension similar to the F# sequence expression syntax and the `yield` construction from C#.

Unfortunately, the approach has a drawback. When iterating the sequence, it allocates a lot of small short-term objects on the heap. But the tests show that the modern List-machines have efficient garbage collectors and sometimes the consing is relatively fast, at least in comparison with calling the generic functions. Along with simplicity of defining the sequences, it was the second reason why I decided to apply the described representation to the enumerators. They are not just as slow as they might seem!

Chapter 2

Generic Sequences

Except for some special cases, the symbols considered below are contained in the GEN-SEQ package from the ASDF system GENERIC-SEQUENCES. To perform the tests, I entered the following commands, assuming that I had already installed the corresponded ASDF system:

```
(require 'asdf)
(asdf:load-system :generic-sequences)

(defpackage test-seq (:use :cl :gen-seq))
(in-package :test-seq)
```

2.1 Basic Definitions

To create a new sequence manually, we have to define the methods for the following two generic functions. This is all you need to make your data structure a full-fledged sequence.

[Generic function]
seqp *seq => generalized-boolean*

Test whether the argument is a sequence.

[Generic function]
seq-enum *seq => (or enum null)*

Return either an enumerator of the sequence or NIL if the sequence is empty.

[Type]
seq

Defines the type of generic sequences. It uses the SEQP predicate to test.

The second method returns either an enumerator or NIL for the empty sequence. The enumerator is actually the cons cell. The CAR part has an ordinary meaning and it returns the current element of the sequence. The CDR

part of the cell is a function that returns either the next enumerator with the next element of the sequence or NIL in case of reaching the end of the sequence.

You should not rely on the internal implementation of the enumerator but use the corresponded macros, for example, which hide the fact that the CDR part is actually a function.

[Macro]
enum-cons *item enum => enum*

> Construct a new enumerator with the specified element and the next enumerator. It is like the standard CONS function. Only the next enumerator is delayed through creating the lambda.

[Macro]
enum-car *enum => item*

> Return the current element of the sequence. This is like the CAR function.

[Macro]
enum-cdr *enum => (or enum null)*

> Calculate the next enumerator of the sequence or return NIL. This is similar to the CDR function but the macro calculates a new enumerator each time it is called. It can make the sequence lazy. Unlike the lazy streams and ordinary lists, the CDR part is not memoized.

[Macro]
enum-append *&rest enums => enum*

> Append the enumerators and create a new enumerator that returns the elements from the specified enumerators sequentially. All enumerators but the first are delayed through creating the lambdas.

These macros allow us to define an enumerator as it was a list. For example, it easy to create an enumerator traversing the nodes of the binary tree. There is only one subtle thing. All resulting enumerators will be delayed but the first enumerator. To delay the first enumerator, we have to pass in it to another structure in a lambda.

[Macro]
make-seq *enum => seq*

> Given the specified enumerator, create a new sequence. This enumerator is wrapped in a lambda. So, it is recalculated anew each time the sequence is iterated.

[Macro]
delay-seq *seq => seq*

> Delay the sequence. This input sequence will be wrapped in the lambda expression, which allows using the sequence in recursive definitions. Please see also the SEQ->STREAM function.

For example, the following function creates a lazy sequence that returns the first *N* elements of the input sequence.

```
(defun seq-take (n seq)
  "Take the first N elements of the sequence."
  (make-seq
    (labels ((traverse (enum acc)
               (cond
                ((null enum) nil)
                ((zerop acc) nil)
                (t (enum-cons
                    (enum-car enum)
                    (traverse (enum-cdr enum)
                              (1- acc)))))))
      (traverse (seq-enum seq) n))))
```

2.2 Predefined Sequences

Lists and vectors implement the protocol of the generic sequences. Moreover, any sequence can be converted to a list or vector.

[Function]
seq->list *seq => list*

Convert the sequence to a list.

[Function]
seq->vector *seq => vector*

Convert the sequence to a vector.

```
? (seqp '(1 2 3))
T
? (seqp #(1 2 3))
T
? (seq->list #(1 2 3))
(1 2 3)
? (seq->vector '(1 2 3))
#(1 2 3)
```

2.3 List of Functions

There is a plenty of functions that operate on sequences or create new sequences, usually lazy sequences.

[Function]
seq-null *seq => generalized-boolean*

Test whether the sequence is empty.

```
? (seq-null '(1 2 3))
NIL
? (seq-null nil)
T
? (seq-null #())
T
```

[Function]
seq-car *seq => item*

Return the first element of the sequence.

```
? (seq-car #(1 2 3))
1
```

[Function]
seq-cdr *seq => seq*

Return the tail of the sequence, i.e. excluding the first element.

```
? (seq->list (seq-cdr #(1 2 3)))
(2 3)
```

[Function]
seq-cons *item seq => seq*

Construct a new sequence that begins with the specified item and ends with the sequence.

```
? (seq->list (seq-cons 1 #(2 3)))
(1 2 3)
```

[Function]
seq-append *&rest seqs => seq*

Append the specified sequences and return a new sequence.

```
? (seq->list (seq-append '(0 1) #(2 3)))
(0 1 2 3)
```

[Function]
seq-equal *seq-1 seq-2 &key test key => generalized-boolean*

Test two sequences for equality.

```
? (seq-equal '(1 2 3) #(1 2 3))
T
```

[Function]
seq-compare *seq-1 seq-2 test &key key => fixnum*

Compare two sequences where the test function for items must return <0 (less), >0 (greater) or 0 (when equal elements).

```
? (seq-compare '(1 4 3) #(1 2 3)
               (lambda (x y)
                 (cond
                   ((< x y) -1)
                   ((> x y) 1)
                   ((= x y) 0))))
1
```

[Function]
seq-length *seq => fixnum*

Return the length of the specified sequence.

```
? (seq-length #(1 2 3))
3
```

[Function]
seq-elt *seq index => item*

Access the element of the sequence specified by index.

```
? (seq-elt #(1 2 3) 1)
2
```

[Function]
seq-remove *item seq &key test key => seq*

Like the REMOVE function but applied to the generic sequence. It returns a lazy sequence.

```
? (seq->list (seq-remove 2 #(1 2 3)))
(1 3)
```

[Function]
seq-remove-if *test seq &key key => seq*

Like the REMOVE-IF function but applied to the generic sequence. It returns a lazy sequence.

```
? (seq->list (seq-remove-if #'zerop #(1 0 2 0 3)))
(1 2 3)
```

[Function]
seq-remove-if-not *test seq &key key => seq*

Like the REMOVE-IF-NOT function but applied to the generic sequence. It returns a lazy sequence.

```
? (seq->list (seq-remove-if-not #'zerop #(1 0 2 0 3)))
(0 0)
```

[Function]
seq-map *function &rest seqs => seq*

Like the MAPCAR function but applied to generic sequences. It returns a lazy sequence.

```
? (seq->list (seq-map (lambda (x) (list x :a)) #(1 2 3)))
((1 :A) (2 :A) (3 :A))
```

[Function]
seq-mappend *function &rest seqs => seq*

Map the function over the elements of the sequences and append together all the results. It returns a lazy sequence. This is like SEQ-MAP but only the input function must return a sequence.

```
? (seq->list (seq-mappend (lambda (x) (list x :a)) #(1 2 3)))
(1 :A 2 :A 3 :A)
```

[Function]
seq-reduce *function seq &key key initial-value => value*

Like REDUCE but applied to the generic sequences.

```
? (seq-reduce #'+ #(1 2 3 4 5))
15
```

[Function]
seq-zip *&rest seqs => seq*

Return a lazy sequence that returns lists of items from the provided sequences.

```
? (seq->list (seq-zip #(:a :b :c) '(1 2 3)))
((:a 1) (:b 2) (:c 3))
```

[Function]
seq-foreach *function &rest seqs => no-values*

Apply the specified function to the sequences for receiving a side-effect.

```
? (seq-foreach #'write #(1 2 3))
123
```

[Function]
seq-take *n seq => seq*

Take the first N elements of the sequence and return a lazy sequence.

```
? (seq->list (seq-take 3 #(1 2 3 4 5 6 7 8 9 0)))
(1 2 3)
```

[Function]
seq-take-while *predicate seq &key key => seq*

`SEQ-TAKE-WHILE` takes a predicate function taking a single argument and a sequence. It returns a lazy sequence of all items in the original sequence, up until the first item for which the predicate function returns `NIL`.

```
? (seq->list (seq-take-while #'oddp #(1 3 2 4 5)))
(1 3)
```

[Function]
seq-take-while-not *predicate seq &key key => seq*

> SEQ-TAKE-WHILE-NOT takes a predicate function taking a single argument and a sequence. It returns a lazy sequence of all items in the original sequence, up until the first item for which the predicate function returns T.

```
? (seq->list (seq-take-while-not #'evenp #(1 3 2 4 5)))
(1 3)
```

[Function]
seq-take-nth *n seq => seq*

> TAKE-NTH takes two arguments, a number and a sequence. It returns a sequence of items from the supplied sequence, taking the first item and every Nth item, where N is the supplied number.

```
? (seq->list (seq-take-nth 2 #(1 2 3 4)))
(1 3)
```

[Function]
seq-drop *n seq => seq*

> Drop the first N elements of the sequence and return the rest.

```
? (seq->list (seq-drop 2 #(1 2 3 4)))
(3 4)
```

[Function]
seq-drop-while *predicate seq &key key => seq*

> SEQ-DROP-WHILE takes a predicate function taking a single argument and a sequence. It returns a sequence of all items in the original sequence, starting from the first item for which the predicate function returns NIL.

```
? (seq->list (seq-drop-while #'oddp #(1 3 2 4 5)))
(2 4 5)
```

[Function]
seq-drop-while-not *predicate seq &key key => seq*

> SEQ-DROP-WHILE-NOT takes a predicate function taking a single argument and a sequence. It returns a sequence of all items in the original sequence, starting from the first item for which the predicate function returns T.

```
? (seq->list (seq-drop-while-not #'evenp #(1 3 2 4 5)))
(2 4 5)
```

[Function]
seq-split *n seq => (seq . seq)*

Split the sequence at the `N`-th element and return the both parts as a list.

```
? (destructuring-bind (x y)
      (seq-split 2 #(1 2 3 4))
    (values (seq->list x)
            (seq->list y)))
(1 2)
(3 4)
```

[Function]
seq-split-if *predicate seq &key key => (seq . seq)*

`SEQ-SPLIT-IF` takes a predicate function taking a single argument and a sequence. It splits the sequence at the first item for which the predicate function returns `T` and and then `SEQ-SPLIT-IF` returns the both parts as a list.

```
? (destructuring-bind (x y)
      (seq-split-if (lambda (x) (>= x 3))
                    #(1 2 3 4))
    (values (seq->list x)
            (seq->list y)))
(1 2)
(3 4)
```

[Function]
seq-split-if-not *predicate seq &key key => (seq . seq)*

`SEQ-SPLIT-IF-NOT` takes a predicate function taking a single argument and a sequence. It splits the sequence at the first item for which the predicate function returns `NIL` and and then `SEQ-SPLIT-IF-NOT` returns the both parts as a list.

```
? (destructuring-bind (x y)
      (seq-split-if-not (lambda (x) (< x 3))
                        #(1 2 3 4))
    (values (seq->list x)
            (seq->list y)))
(1 2)
(3 4)
```

[Function]
seq-member *item seq &key test key => seq*

Search a sequence for an item and return the tail of the sequence beginning with this element; otherwise `NIL` is returned.

```
? (seq->list (seq-member 3 #(1 2 3 4)))
(3 4)
```

[Function]
seq-member-if *predicate seq &key key => seq*

Search a sequence for a top-level item for which the predicate returns `T` and return the tail of the sequence beginning with this element; otherwise `NIL` is returned.

```
? (seq->list (seq-member-if (lambda (x) (= x 3)) #(1 2 3 4)))
(3 4)
```

[Function]
seq-member-if-not *predicate seq &key key => seq*

Search a sequence for a top-level item for which the predicate returns `NIL` and return the tail of the sequence beginning with this element; otherwise `NIL` is returned.

```
? (seq->list (seq-member-if-not (lambda (x) (< x 3)) #(1 2 3 4)))
(3 4)
```

[Function]
seq-find *item seq &key test key => item*

Search a sequence for an item and return this element; otherwise `NIL` is returned.

```
? (seq-find 5 #(1 2 3 4) :key #'1+)
4
```

[Function]
seq-find-if *predicate seq &key key => item*

Search a sequence for an item for which the predicate returns `T` and return this element; otherwise `NIL` is returned.

```
? (seq-find-if #'evenp #(1 2 3 4))
2
```

[Function]
seq-find-if-not *predicate seq &key key => item*

Search a sequence for an item for which the predicate returns `NIL` and return this element; otherwise `NIL` is returned.

```
? (seq-find-if-not #'oddp #(1 2 3 4))
2
```

[Function]
seq-position *item seq &key test key => index*

Search a sequence for an element and return the index within the sequence; otherwise, `NIL` is returned.

```
? (seq-position 3 #(1 2 3 4))
2
```

[Function]
seq-position-if *predicate seq &key key => index*

Search a sequence for an element for which the predicate returns `T` and return the index within the sequence; otherwise, `NIL` is returned.

```
? (seq-position-if (lambda (x) (= x 3)) #(1 2 3 4))
2
```

[Function]
seq-position-if-not *predicate seq &key key => index*

Search a sequence for an element for which the predicate returns `NIL` and return the index within the sequence; otherwise, `NIL` is returned.

```
? (seq-position-if-not (lambda (x) (< x 3)) #(1 2 3 4))
2
```

[Function]
seq-every *predicate &rest seqs => generalized-boolean*

Like the EVERY function but applied to the generic sequences.

```
? (seq-every #'characterp "abc")
T
```

[Function]
seq-some *predicate &rest seqs => generalized-boolean*

Like the SOME function but applied to the generic sequences.

```
? (seq-some #'= '(1 2 3 4 5) #(5 4 3 2 1))
T
```

[Function]
seq-notany *predicate &rest seqs => generalized-boolean*

Like the NOTANY function but applied to the generic sequences.

```
? (seq-notany #'> '(1 2 3 4) #(5 6 7 8) '(9 10 11 12))
T
```

[Function]
seq-notevery *predicate &rest seqs => generalized-boolean*

Like the NOTEVERY function but applied to the generic sequences.

```
? (seq-notevery #'< '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))
NIL
```

[Function]
seq-repeatedly *function => seq*

Return an infinite lazy sequence obtained by calling the function repeatedly.

```
? (seq->list
   (seq-take 3 (seq-repeatedly (lambda () "hello"))))
("hello" "hello" "hello")
```

[Function]
seq-iterate *function initial-value => seq*

It returns an infinite lazy sequence obtained by starting with the supplied value, and then by calling the supplied function passing the previous item in the sequence as its argument.

```
? (seq->list (seq-take 5 (seq-iterate #'1+ 5)))
(5 6 7 8 9)
```

[Function]
seq-repeat *value => seq*

It returns an infinite lazy sequence consisting of the argument value repeated endlessly.

```
? (seq->list (seq-take 5 (seq-repeat :a)))
(:A :A :A :A :A)
```

[Function]
seq-range *&key start end step => seq*

`SEQ-RANGE` returns a lazy sequence of numbers from the start (inclusive, 0 by default) to the end (exclusive, nil by default) incremented by the step (1 by default).

```
? (seq->list (seq-take 5 (seq-range)))
(0 1 2 3 4)
```

[Function]
seq-cycle *seq => seq*

It returns a lazy infinite sequence obtained by successively repeating the values in the supplied sequence.

```
? (seq->list (seq-take 5 (seq-cycle '(:a :b :c))))
(:A :B :C :A :B)
```

[Function]
seq-interpose *value seq => seq*

SEQ-INTERPOSE takes two arguments, a value and a sequence. It returns a lazy sequence obtained by inserting the supplied value between the values in the sequence.

```
? (seq->list (seq-interpose :a '(1 2 3 4)))
(1 :A 2 :A 3 :A 4)
```

Chapter 3

Iterating Sequences

The library includes an additional ASDF system GENERIC-SEQUENCES-ITERATE that defines package GEN-SEQ-ITER. It allows iterating the sequences within the ITER macro.

To test, we define the TEST-SEQ-ITER package:

```
(require 'asdf)
(asdf:load-system :generic-sequences-iterate)

(defpackage test-seq-iter (:use :cl :iter :gen-seq :gen-seq-iter))
(in-package :test-seq-iter)
```

The GEN-SEQ-ITER package defines a new clause for the ITER macro. It looks like the clause that is used for iterating the lists and vectors but only it uses symbol IN-SEQ.

```
? (iter (for n from 1 to 5)
        (for x in-seq (seq-range))
        (collect x))
(0 1 2 3 4)
```

Chapter 4

Sequence Comprehension

The library also contains ASDF system `GENERIC-SEQUENCES-CONT` that defines package `GEN-SEQ-CONT`. This is a *sequence comprehension* similar to the sequence expression syntax from F# and the `yield` expression from C#.

To test, we define the `TEST-SEQ-CONT` package:

```
(require 'asdf)

(asdf:load-system :generic-sequences-iterate)
(asdf:load-system :generic-sequences-cont)

(defpackage test-seq-cont
  (:use :cl :iter :cl-cont :gen-seq :gen-seq-iter :gen-seq-cont))

(in-package :test-seq-cont)
```

The macros of the sequence comprehension are as follows.

[Macro]
with-enum/cc *&body body => enum*

Return an enumerator with support of special macros `YIELD/CC`, `YIELD-ENUM/CC` and `YIELD-SEQ/CC`.

[Macro]
with-seq/cc *&body body => seq*

Return a lazy sequence with support of special macros `YIELD/CC`, `YIELD-ENUM/CC` and `YIELD-SEQ/CC`.

[Macro]
yield/cc *item => enum*

Yield a new element and suspend the control flow until the next element will be requested.

[Macro]
yield-enum/cc *enum => enum*

Yield the elements specified by the enumeration.

[Macro]
yield-seq/cc *seq* => *enum*

Yield the elements specified by the sequence.

The sequence comprehension is based on the CL-CONT package, about which the names of the macros indicate. It means that the comprehension has some limitations, for example, it cannot process the UNWIND-PROTECT form but it can process the loops.

Let us take the following example.

```
(defun reader (x)
  (format t "Read: x = ~S~%" x))

(defun writer ()
  (with-seq/cc
    (iter (for n from 1 to 5)
          (for x in-seq (seq-range))
          (progn
            (format t "Write: x = ~S~%" x)
            (yield/cc x)))))
```

To illustrate the laziness, the functions contain the debugging code. When running, we receive the next output:

```
? (seq-foreach #'reader (writer))
Write: x = 0
Read: x = 0
Write: x = 1
Read: x = 1
Write: x = 2
Read: x = 2
Write: x = 3
Read: x = 3
Write: x = 4
Read: x = 4
; No value
```

Finally, the macros of the sequence comprehension have a very simple definition.

```
(in-package :generic-seq-cont)

(defmacro with-enum/cc (&body body)
  `(with-call/cc ,@body nil))

(defmacro with-seq/cc (&body body)
  `(make-seq (with-enum/cc ,@body)))

(defmacro yield/cc (item)
  `(call/cc (lambda (k) (enum-cons ,item (funcall k)))))

(defmacro yield-enum/cc (enum)
  `(call/cc (lambda (k) (enum-append ,enum (funcall k)))))

(defmacro yield-seq/cc (seq)
  `(yield-enum/cc (seq-enum ,seq)))
```

Chapter 5

Lazy Streams

Sometimes it is useful to have a sequence that always returns the same elements but does it lazily by demand. Such a sequence is called a *lazy stream*. The ASDF system `GENERIC-SEQUENCES-STREAM` defines these streams in package `GEN-SEQ-STREAM`.

To test, we can define a new package.

```
(require 'asdf)
(asdf:load-system :generic-sequences-stream)

(defpackage test-seq-stream (:use :cl :gen-seq :gen-seq-stream))
(in-package :test-seq-stream)
```

The `GEN-SEQ-STREAM` package defines only one function.

[Function]
seq->stream *seq &key thread-safe => seq*

> Convert the sequence to a lazy stream that must always return the same elements. The `THREAD-SAFE` parameter is `NIL` by default.

To show a difference, we define a sequence that returns random numbers each time it is iterated. The stream always returns the same elements.

```
(setf *random-seq* (seq-take 10 (seq-repeatedly (lambda () (random 10)))))
(setf *stream-seq* (seq->stream *random-seq*))
```

When running, I received the following results:

```
? (seq->list *random-seq*)
(8 4 8 3 6 2 1 6 8 3)
? (seq->list *random-seq*)
(2 6 0 4 1 9 5 2 5 3)
? (seq->list *stream-seq*)
(7 7 9 8 0 1 9 1 5 9)
? (seq->list *stream-seq*)
(7 7 9 8 0 1 9 1 5 9)
```