User-extensible sequences

Christophe Rhodes
Goldsmiths, University of London
New Cross Road, London, SE14 6NW
c.rhodes@gold.ac.uk

ABSTRACT

Common Lisp is often touted as the programmable programming language, yet it sometimes places large barriers in the way, with the best of intentions. One of those barriers is a limit to the extensibility by the user of certain core language constructs, such as the ability to define subclasses of built in classes: even where this could be useful with minimal penalties. We introduce the notion of user-extensible sequences, describing a protocol which implementations of such classes should follow. We show examples of their use, and discuss the issues observed in providing support for this protocol in a Common Lisp.

1. INTRODUCTION

Common Lisp is recognized as being an extremely flexible language: one in which linguistic experimentation can take place, one where the method of solving a problem in a domain is first to write an interpreter or a compiler for a language to express concepts in the domain of interest, and then to use that domain-specific language to solve the problem. To some extent, then, it might be surprising to find that the Common Lisp language itself is only conformingly extensible (by the implementor or user) in limited ways. This is at least partially explained by the aims of the standardizers, which included "stricter standardization for portability" (Pitman and Chapman, 1994, section 1.1.2): codification of existing practice was a large part of the X3J13 committee's work, and it is difficult to ensure portability with a highly-extensible language core.

However, the standardization process was not intended to close the door to language development: merely to provide a stable platform which could be agreed on. For example, the CLOS system for object orientation was standardized without very much scope for extensibility; however, it was intentionally (Steele, 1990, chapter 28) upwardly compatible with something close to the Metaobject Protocol described in Kiczales et al. (1991), which is often (though not always) supported by contemporary Lisp implementations.

Meanwhile, other languages and language environments have not stood still; many of Lisp's once-unique features are now to be found in other languages (Norvig, 2002), though some are still not[1]; additionally, sometimes these languages have features not found in any available Lisp implementation. In some cases, such as Aspect-Oriented Programming (Kiczales et al., 1997) these features can be straightforwardly implemented by any interested user – this itself is one of Lisp's unique features not often found elsewhere – but some features need implementation support for them to be used to maximum effect: addition of features by the Lisp programmer can suffice for certain needs, but they are not pervasive in the way that one would want; such extensions, even upwardly-compatible extensions, need to be explicitly used by third-party code.

One such extension is the ability for the user to define new sequence types. While it is possible to recommend ways of accessing and iterating over both Common Lisp sequences and other objects in userspace, there is no way of having those other objects (conceptually sequences though not of type `sequence`) seamlessly interoperate with the standard Common Lisp sequence functionality, or with third-party code which does not follow the recommendation. Additionally, such a userspace implementation is inconvenient, as there are portions of the sequence functionality in Common Lisp which are tedious or tricky to implement[2]; it is more convenient to require implementation of a few simple methods. Possibly most importantly, the need for new names for essentially the same concept – such as `climacs-buffer:size` and `flexichain:nb-elements` (Strandh et al., 2004) for `length`, or `tabcode-syntax:buffer-position-if` (Rhodes et al., 2005) for `position-if`, taking examples from real-world code – is a barrier to clear expression and understanding of Common Lisp code which defines objects which are conceptually sequences, and we aim to remove this barrier with this proposal.

In what follows, there will be reference to new operators without explicit package prefixes: in such cases, the symbols' package should be assumed to be the `sequence` package. A number of new operators have the same name as standardized operators of Common Lisp; except in appendix A, the operator without an explicit prefix should be taken to mean the Common Lisp operator, while the new operators will

---

[1] and it has been argued that their inclusion would convert their host language into a Lisp of some form.

[2] The author, in developing this extension, wrote the implementation first and test cases second: the test cases revealed five implementation errors.

```
(defun foo (x)
  (handler-case
      (etypecase (ignore-errors (make-sequence x 8))
        (null ...) ; make-sequence threw an error
        (list ...)
        (vector ...))
    (type-error (c)
      ;; we get here if make-sequence returned a
      ;; non-list non-vector
      (error "BUG: system threw ~S on ~S" c 'make-sequence))))
```

Figure 1: How to detect violations of the contract of `make-sequence`.

have an explicit `sequence:` prefix.

The rest of this paper is organized as follows: after discussing compatibility issues and related work in sections 1.1 and 1.2, we give an introduction for the prospective user of extensible sequences in section 2, and some examples in section 3, and some details of our implementation are given in section 4. As a snapshot of our work in progress, a more formal specification of the protocols is presented in appendix A, intended in the first instance to stimulate discussion rather than to be the definitive specification.

1.1 Compatibility

> The types [*sic*] vector and the type list are disjoint subtypes of type sequence, but are not necessarily an exhaustive partition of sequence.
>
> Pitman and Chapman (1994, System Class `sequence`)

Although the quote above might suggest that there would be no problem from the point of view of standard conformance for an implementation to offer non-standard types of `sequence`, the outline code in figure 1, as a result of Issue `CONCATENATE-SEQUENCE` (Pitman, 1991), is defined by ANSI CL never to get to the `type-error` branch of the `handler-case`, which was probably not intended by the standardizers. For a fuller discussion of this issue, and a suggestion for a resolution, see Rhodes (2006).

An additional issue is that ANSI CL specifies for the `length` function and the `elt` accessor that their sequence argument should be a *proper sequence*, which is defined in the normative glossary as *a sequence which is not an improper list; that is, a vector or a proper list*; again, this definition was probably not intended to prohibit non-standardized sequences, even though, interpreted strictly, the second half of it implies that no non-standard sequence is a proper sequence.

There is also the issue of cultural compatibility with the body of Common Lisp code available in the wild. Perhaps because of the historical lack of generic sequences offered by Common Lisp implementations, there seems to be little in the way of packages written to exploit the abstract sequence data type; instead, packages choose a concrete sequence type and implement their functionality atop that, manipulating a particular kind of sequence rather than sequences in general[3]. However, it is likely that if user-extensible sequences become available, code will be modified or written afresh to take advantage of it.

It would be possible to implement this proposal (or something like it) entirely portably (in 'userspace'), by defining a new package shadowing many of the standard Common Lisp functions and macros, and implementing the generic functionality on the standard types by trampolining to the standard functions. However, such an implementation strategy has drawbacks:

- Implementation-specific compiler optimizations (such as compiler macros) for the shadowed functions would essentially be lost. As an example, if an implementation has a specialized implementation of `map-into` for arguments known at compile-time to be certain types of vector, that optimization will go unused in a putative userspace implementation in function calls to `sequence-cl:map-into`.
- The userspace implementation of extensible sequences would not interoperate with third-party Common Lisp code: programs already written with generic (not necessarily user-extensible) sequences in mind, using the `common-lisp` package, will not be able to interoperate cleanly with sequences defined using this portable implementation.
- Innocuous-looking uses of explicit package prefixes (for example, `cl:sequence`) would have surprising and potentially difficult-to-debug effects. Other maintenance headaches include how to support both native and portable implementations in library code.

A userspace implentation might be better than nothing, for a transition period, but support from the Lisp implementation is required for seamless operation.

That said, there is an issue regarding portability of libraries using extensible sequences: until it is ubiquitously implemented, use of this facility in bodies of code render those bodies of code unportable in practice. Whether this unportability itself becomes a problem in practice is largely a matter for the user community to decide.

1.2 Related Work

Much inspiration for this proposal was drawn from the Dylan (Shalit, 1996) iterator protocol, which provides for iteration over collections with the open generic function `forward-iteration-protocol`. Where our design is similar to this protocol, it is largely for the same reasons: we do not wish to impose unnecessary run-time overhead (in space or speed) for those uses which need high performance. However, in this proposal, we aim to provide a little more of a layer

---

[3] There are exceptions: for instance, the `split-sequence` library, designed by readers of `news:comp.lang.lisp` in 2001, works without modification on generic sequences of the form described in this paper.

`sequence:length`	`sequence:elt`	`(setf sequence:elt)`
`sequence:adjust-sequence`	`sequence:make-sequence-like`	

Table 1: The protocol functions which must be implemented for a sequence class.

of convenience for the user who does not need to minimize overhead.

Many other languages provide some form of iteration or collection protocol. Python, for instance, allows the implementor of a collection to define a method on `__iter__()` to return an iterator object, which itself must have methods on `__iter__()` and `next()`. Apparently for reasons of efficiency, the Python iterator protocol (Yee and van Rossum, 2001) provides no explicit means for checking for termination of an iteration, instead requiring the iterator to signal an exception when `next()` is called on an iterator representing a terminated iteration.

The Scheme language (Kelsey et al., 1998) definition has little built-in support for user-extensible or even generic sequences. Its community has made one attempt (Miller, 2004) at defining an interface and conventions for collections; however, this SRFI has apparently not seen many Scheme implementations decide to support it natively; as of two years after its finalization, the SRFI status page reports no implementation as supporting it.

The Factor language's sequence interface is conceptually very similar to that described in the following sections for Common Lisp, including the distinction between sequence protocol (defining the fundamental operations that must be implemented for sequence objects) and 'utility words', analogous to functions performing computations over sequences (Pestov, 2006, Sequences), which will work on any object implementing the sequence protocol.

In the Common Lisp world itself, an early (pre-CLOS) attempt to provide generic sequence functionality was presented in Haible (1988) for the GNU CLISP implementation; however, that proposal was never formally exported or documented (Haible, 2006); the author's primary concern with this attempt was in supporting the necessary operations efficiently. Some of these issues of efficiency remain in this proposal, though we believe that they might be resolved in a Lisp implementation supporting sealing and inlining of methods.

2. USER-DEFINED SEQUENCES

In this section, we must draw the distinction between the implementor of a sequence class, and the implementation of Common Lisp which supports this user-extensible sequence facility. Most of the time, we will use 'user' to mean the implementor of a sequence class, and 'implementor' to mean implementor of a Common Lisp implementation, and we hope that it will be clear from context when this does not apply.

The names of the various operators have been chosen to maximize both backward- and forward-compatibility (with Common Lisp as standardized and potential related extensions to Common Lisp such as a collections protocol); we specify operators corresponding to standardized functions such as `find` to be named like `sequence:find`, so that an implementation of Common Lisp 'natively' implementing this proposal can simply import the relevant symbols from the `sequence` package[4]. Thus, we specify `sequence:adjust-sequence` rather than `(setf sequence:length)`, so that an implementation can import `sequence:length` into the standard `common-lisp` package without inadvertantly causing there to be a `setf` function for `length`, in contravention of the constraints on implementations (Pitman and Chapman, 1994, Section 11.1.2.1.1).

We also aim to be not incompatible with similar extensions to Common Lisp, such as a protocol for accessing and iterating over general collections; in this proposal we are dealing with user-defined sequence classes because Common Lisp as standardized has a large library of functions acting on generic sequences, while it has no functions acting on generic collections – and so a 'userspace' implementation of a collections protocol would not pose the interoperability problems discussed in section 1.1.

2.1 Sequence Datatypes

The user can define a direct subclass of `sequence` using `defclass`, specifying `sequence` as one of the superclasses (but not the only one: for code portable between implementations of this proposal, `standard-object` must be in the superclasses list too). The resulting class is `subtypep sequence`, and instances of the class are `typep sequence`.

The fundamental operations defined in Common Lisp relating to sequences are `length`, `elt`, and `(setf elt)`; in order to support these, the user of the extensible sequence facility (the implementor of the sequence class) should define methods on `sequence:length`, `sequence:elt` and `(setf sequence:elt)`.

The set of functions making up the sequence protocol of this proposal (see table 1) consists of the analogues to the three Common Lisp functions above, along with two others: `sequence:adjust-sequence`, which is similar to the `adjust-array` function, but generalized to sequences; and `sequence:make-sequence-like`, which creates a sequence of the the same kind[5] as its first argument, with length given by its second argument.

The rationale for `make-sequence-like` is to support the `subseq` and `copy-seq` operators, and the sequence functions which are defined to return a freshly-consed sequence (such as `substitute`). Additionally, this operator is easier both to specify and to implement efficiently than one like the `make-sequence` function, which requires a full understanding of the Common Lisp type system; `make-sequence-like` can operate as a simple generic function specialized on its sequence argument.

To support the `delete` and `delete-duplicates` functions, we provide `adjust-sequence`, which adjusts various properties, including the length, of a sequence. Users of this pro-

---

[4] According to some interpretations of the standard, the implementation may not simply make `sequence` a nickname for the `common-lisp` package, as the list of package nicknames is standardized as `cl` only.

[5] Neither 'type' nor 'class' is quite right here, as `cons` and `null` are distinct types and classes, while both being subtypes of the sequence type `list`.

Common Lisp Function	Extensible Generic Function
`copy-seq`, `subseq`	`sequence:copy-seq`, `sequence:subseq`
`reduce`	`sequence:reduce`
`reverse`, `nreverse`	`sequence:reverse`, `sequence:nreverse`
`sort`, `stable-sort`	`sequence:sort`, `sequence:stable-sort`
`count`, `find`, `position`	`sequence:count`, `sequence:find`, `sequence:position`
`count-if`, `count-if-not`	`sequence:count-if`, `sequence:count-if-not`
`find-if`, `find-if-not`	`sequence:find-if`, `sequence:find-if-not`
`position-if`, `position-if-not`	`sequence:position-if`, `sequence:position-if-not`
`search`, `mismatch`	`sequence:search`, `sequence:mismatch`
`replace`	`sequence:replace`
`substitute`, `nsubstitute`	`sequence:substitute`, `sequence:nsubstitute`
`substitute-if`, `substitute-if-not`	`sequence:substitute-if`, `sequence:substitute-if-not`
`nsubstitute-if`, `nsubstitute-if-not`	`sequence:nsubstitute-if`, `sequence:nsubstitute-if-not`
`remove`, `delete`	`sequence:remove`, `sequence:delete`
`remove-if`, `remove-if-not`	`sequence:remove-if`, `sequence:remove-if-not`
`delete-if`, `delete-if-not`	`sequence:delete-if`, `sequence:delete-if-not`
`remove-duplicates`, `delete-duplicates`	`sequence:remove-duplicates`, `sequence:delete-duplicates`

Table 2: Common Lisp functions, and the corresponding generic functions specified to be extensible in this protocol.

tocol are encouraged to document the circumstances under which their methods on this operator will preserve the identity of its sequence argument, and must arrange that if a new sequence is allocated, the sequence argument must be unmodified; however, in general, callers (either explicit or implicit) may not assume that the sequence returned from `adjust-sequence` is the same sequence as its argument.

Once methods for these generic functions have been implemented for a sequence class, all the regular Common Lisp sequence functionality will work as expected. However, some sequences may not admit implementations of all these operations, or indeed might offer means to implement certain operations more efficiently, so in the following sections we describe the protocol by which the Lisp implementation provides the sequence functionality.

2.2 Iteration

It is common to iterate over sequences, both in the conception of many of the Common Lisp sequence functions, and in user-defined operations. In this section, we describe the protocol and interface for iterating over sequence contents, both built-in and user-defined. The protocol discussed here has a set of default methods specialized on the `sequence` class, so that any sequence class for which methods on the protocol functions of section 2.1 have been implemented will obey the protocol. However, these default methods cannot take into account the characteristics of the sequence class, and so for efficiency users may wish to override them for their own classes.

The essential concept is of an iterator object to represent the current state of an iteration. It is not necessary for this iterator object to have a distinguished class, as the sequence over which it is iterating is present in all function calls in this protocol (and therefore its class can be used for specialization of methods); indeed, the conceptual object is represented in this protocol by three objects and six functions.

The `make-sequence-iterator` operator constructs one of these iterator objects: after the required sequence argument, it accepts keyword `:start`, `:end` and `:from-end` arguments, and returns nine values. The first three of those values are an iterator state, a limit and the from-end argument; the remaining six are functions which, respectively, return a state one step ahead; test the state against the limit for termination; retrieve the element at the current iteration state from the sequence; set the element at the current iteration state to a new value; return the index corresponding to the current iteration state; and return a distinct iteration state representing a copy of the current one.

The default method on `make-sequence-iterator` is intended for convenience: for most uses, it is unnecessary to construct the nine return values; instead, the default method (specialized to `sequence`) on `make-sequence-iterator` generates the first three of the return values by calling `make-simple-sequence-iterator`, and returns in addition six protocol generic functions: `#'iterator-step` (which advances the iteration state); `#'iterator-endp`, testing an iteration for termination; the `#'iterator-element` reader and the `#'(setf iterator-element)` writer; `#'iterator-index`, returning the sequence index corresponding to the iterator state; and finally `#'iterator-copy`, returning a copy of the iteration state. These functions have methods specialized to `list` and `vector` to provide iterators for the built-in sequence classes.

While implementors of sequence classes may choose to use this CLOS-based iterator protocol (at the potential loss of efficiency through generic function dispatch at each step), users of the iteration protocol (who define functions which perform iterations over sequences) may not assume that the sequence class implementor has done so, and so must call `make-sequence-iterator` or the operators discussed below.

A small dose of syntactic sugar around `make-sequence-iterator` is provided by `with-sequence-iterator`, which binds as if by `multiple-value-bind` the variables in its first argument to the result of applying `make-sequence-iterator` to its second argument, and then executes the body. A slightly simpler macro to use is `with-sequence-iterator-functions`, which binds the six names in its first argument to six local functions (which have dynamic extent and close over the return values from `make-sequence-iterator`) which perform the various iterator manipulations.

For programmer convenience, we also provide a `dosequence` macro, behaving as `dolist` (but for arbitrary sequences), and an extension for `loop` using the loop keywords

`some`	`every`	`notany`
`notevery`	`map`	`map-into`
`concatenate`	`merge`	
`coerce`	`make-sequence`	

Table 3: Common Lisp functions applicable or otherwise relevant to sequences which are not specified as extensible in this protocol; however, the implementation's versions are specified to be applicable to arbitrary sequence types, and to produce the expected result.

`element` and `elements`, in a similar fashion to the `for-as-package` loop path (Pitman and Chapman, 1994, Section 6.1.2.1.7).

It is intended that the iteration protocol described here will be compatible with an iteration protocol for general collections (including hash tables, trees and other similar data structures); at the minimum, any such protocol should be able to specify its behaviour for sequences in terms of the operators described here.

2.3 Sequence Functions

For most of the functions in the sequences chapter of the standard, there is an analogous generic function which can be specialized for user-defined sequence classes. The complete list of generic functions specified to be extensible, and their corresponding Common Lisp function, is given in table 2. Implementations may choose to make the Common Lisp function `eql` to the extensible generic function, or even make the symbols themselves `eql` to each other; however, users may not assume this.

The Common Lisp functions in table 3 are not extensible in this proposal; however, where ANSI CL specifies that they accept sequence arguments, the implementation must accept any user-defined sequence, and where ANSI CL specifies that they accept a sequence type-specifer, they must accept a user-defined sequence name or class object (see also Rhodes (2006) for a discussion of the implications of this with respect to the standard behaviour).

3. EXAMPLES

In this section, we present two examples of uses for the protocols discussed in this paper. Firstly, in section 3.1, we present the implementation of a distinct sequence type along with the method definitions to allow it to interoperate; then, in section 3.2, we demonstrate the definition of a mixin class and methods specialized on it to provide additional functionality to generic sequences implemented as in this paper.

3.1 Queue

As an example of a non-standard sequence, consider implementing a queue data structure, which supports the operations `enqueue` and `dequeue`. Figure 3 shows one way in which such a queue could be implemented, using a list as the storage for the data, and keeping a reference to the cons cell at the back of the queue. Purely for interest, we also implement a variant of the queue which is also funcallable, and arrange so that calling the funcallable queue with no arguments performs dequeueing, while with one argument it enqueues that argument.

```
(length #[a b c d]) ; => 4
(count 1 #[1 2 3]) ; => 1
(remove-if-not #’oddp #{1 2 3}) ; => #{1 3}
(remove-duplicates #[1 2 3 4 5] :end 4
                   :key #’oddp :from-end t)
  ; => #[1 2 5]
#2a#{#[1] #(2)} ; => #2A((1) (2))
```

Figure 2: Examples of using the queues as sequences. For the `#[` and `#{` reader macros and queue print functions, see figure 7 in the appendix.

An implementation of the sequence and iteration protocol for a queue implemented in this fashion is shown in figure 4. Note that we have not paid any particular attention to efficiency: we could improve this implementation by storing the current length in a slot and by using knowledge of the implementation in figure 3 to support `make-sequence-like` more efficiently, without calling `enqueue` many times.

Given the code in figure 4, queues can be used wherever Common Lisp specifies that a sequence is acceptable. For instance, one can ask for the position of an element in the queue using `position`, with all the keywords (`:test`, `:key`, `:start` etc.) that the Common Lisp function accepts; some examples are given in figure 2. Furthermore, this queue implementation will potentially interoperate with bodies of code written for generic sequences, as long as there are no uses of the formal interpretation of the requirements of `make-sequence` and friends.

3.2 Undoable mixin

To illustrate possible uses of the sequence protocol discussed in this document, we present in figure 5 a mixin for implementing some undo functionality for a sequence. The `undoable-mixin` class contains a `record` slot, which records enough information to reconstruct the state of the sequence before an operation; if the user calls the `undo` function on a sequence with this class mixed in, the previous state of the sequence will be reconstructed using `undo-using-record` methods. A more sophisticated version of this might be useful as a component of an editor buffer implementation, for example.

For `(setf elt)`, we record the index and the value at that index before performing the mutation; clearly, this permits reconstructing the previous state of the sequence. For operations such as `fill` and `nreverse`, we could simply make do with this (and an analogous change for the setter function of the iterator protocol; see figure 8 in the appendix for that detail), but this would have the consequence that a single logical operation such as `fill` would require multiple calls to `undo` to undo the state.

Instead, therefore, for `fill`, we record the contents between the bounding index designators, while for `nreverse` we need record nothing, as it is a reversible operation[6]. An alternative strategy for grouping multiple primitive operations, used in the method for recording calls to `nsubstitute` and `delete`, is shown in figure 8 in the appendix.

---

[6] We assume for expository purposes that the destructive sequence functions such as `nreverse` act in-place on the user-extended sequences in which this class will be mixed.

```
(defclass queue (standard-object sequence)
  ((%data :accessor %queue-data) (%pointer :accessor %queue-pointer)))
(defmethod initialize-instance :after ((o queue) &key)
  (let ((head (list nil)))
    (setf (%queue-data o) head (%queue-pointer o) head)))

(defgeneric enqueue (data queue)
  (:argument-precedence-order queue data)
  (:method (data (o queue))
    (setf (cdr (%queue-pointer o)) (list data) (%queue-pointer o) (cdr (%queue-pointer o)))
    o))
(defgeneric dequeue (queue)
  (:method ((o queue))
    (prog1 (cadr (%queue-data o))
      (setf (cdr (%queue-data o)) (cddr (%queue-data o))))))

(defclass funcallable-queue (funcallable-standard-object queue)
  () (:metaclass funcallable-standard-class))
(defmethod initialize-instance :after ((o funcallable-queue) &key)
  (flet ((fun (&optional (new nil new-p)) (if new-p (enqueue new o) (dequeue o))))
    (set-funcallable-instance-function o #'fun)))
```

Figure 3: Basic definitions of a queue data structure, including a funcallable variant. The author acknowledges that the layout of the code in this and subsequent figures is not idiomatic: this is for reasons of vertical space.

```
(defmethod sequence:length ((o queue)) (length (cdr (%queue-data o))))
(defmethod sequence:elt ((o queue) index) (elt (cdr (%queue-data o)) index))
(defmethod (setf sequence:elt) (new-value (o queue) index) (setf (elt (cdr (%queue-data o)) index) new-value))
(defmethod sequence:make-sequence-like ((o queue) length &key (initial-element nil iep) (initial-contents nil icp))
  (let ((result (make-instance (class-of o))))
    (cond
      ((and iep icp)
       (error "supplied both ~S and ~S to ~S" :initial-element :initial-contents 'make-sequence-like))
      (icp (unless (= (length initial-contents) length)
             (error "length mismatch in ~S" 'make-sequence-like))
           (setf (cdr (%queue-data result)) (coerce initial-contents 'list))
           (setf (%queue-pointer result) (last (%queue-data result)))
           result)
      (t (dotimes (i length result) (enqueue initial-element result))))))
(defmethod sequence:adjust-sequence ((o queue) length &key initial-element (initial-contents nil icp))
  (cond
    ((= length 0)
     (setf (cdr (%queue-data o)) nil (%queue-pointer o) (%queue-data o)))
    (t (sequence:adjust-sequence (%queue-data o) (1+ length) :initial-element initial-element)
       (setf (%queue-pointer o) (last (%queue-data o)))
       (when icp (replace (%queue-data o) initial-contents :start1 1)) o)))

(defmethod sequence:make-simple-sequence-iterator ((q queue) &rest args &key from-end start end)
  (declare (ignore from-end start end))
  (apply #'sequence:make-simple-sequence-iterator (cdr (%queue-data q)) args))
(defmethod sequence:iterator-step ((q queue) iterator from-end)
  (sequence:iterator-step (cdr (%queue-data q)) iterator from-end))
(defmethod sequence:iterator-endp ((q queue) iterator limit from-end)
  (sequence:iterator-endp (cdr (%queue-data q)) iterator limit from-end))
(defmethod sequence:iterator-element ((q queue) iterator)
  (sequence:iterator-element (cdr (%queue-data q)) iterator))
(defmethod (setf sequence:iterator-element) (new-value (q queue) iterator)
  (setf (sequence:iterator-element (cdr (%queue-data q)) iterator) new-value))
(defmethod sequence:iterator-index ((q queue) iterator)
  (sequence:iterator-index (cdr (%queue-data q)) iterator))
(defmethod sequence:iterator-copy ((q queue) iterator)
  (sequence:iterator-copy (cdr (%queue-data q)) iterator))
```

Figure 4: Implementation of the sequence protocol for the queues of figure 3.

```
(defclass undoable-mixin ()
  ((recording :initform nil :accessor recording) (record :initform nil :accessor record)))
(defclass setelt-record ()
  ((index :initarg :index :reader index) (value :initarg :value :reader value)))
(defclass fill-record ()
  ((start :initarg :start :reader start) (end :initarg :end :reader end)
   (contents :initarg :contents :reader contents)))
(defclass nreverse-record () ())
(defmacro without-undoing ((object) &body body)
  ‘(let ((old (recording ,object)))
    (unwind-protect (progn (setf (recording ,object) t) ,@body)
      (setf (recording ,object) old))))
(defmacro define-undo-method (name arglist &body body)
  ‘(defmethod ,name :around ,(substitute ’(o undoable-mixin) ’o arglist)
     (if (recording o) (call-next-method) (without-undoing (o) ,@body (call-next-method)))))
(define-undo-method (setf sequence:elt) (new-value o index)
  (push (make-instance ’setelt-record :index index :value (elt o index)) (record o)))
(define-undo-method sequence:fill (o item &key (start 0) end)
  (push (make-instance ’fill-record :start start :end end :contents (subseq (coerce o ’vector) start end))
        (record o)))
(define-undo-method sequence:nreverse (o) (push (make-instance ’nreverse-record) (record o)))
(defun undo (object)
  (undo-using-record object (car (record object))) object)
(defmethod undo-using-record ((o undoable-mixin) (r null)) (error "Nothing to undo"))
(defmethod undo-using-record :after ((o undoable-mixin) r) (pop (record o)))
(defmethod undo-using-record ((o undoable-mixin) (r setelt-record))
  (without-undoing (o) (setf (elt o (index r)) (value r))))
(defmethod undo-using-record ((o undoable-mixin) (r fill-record))
  (without-undoing (o)
    (with-accessors ((start start) (end end) (contents contents)) r (setf (subseq o start end) contents))))
(defmethod undo-using-record ((o undoable-mixin) (r nreverse-record))
  (without-undoing (o) (sequence:nreverse o)))
```

Figure 5: Illustration of a simple implementation of `undo` for sequences: an `undoable-mixin` class which can be mixed in to a sequence class, providing undo functionality (illustrated here for `(setf elt)`, `fill` and `nreverse`).

4. IMPLEMENTATION DETAILS

We have implemented the above proposal in Steel Bank Common Lisp (Newman et al., 2000). The first aspect of SBCL itself which needed to be modified was the type system: the system needed to be informed that `sequence` was no longer simply an alias in that implementation of `(or list vector)`, but was an unsealed (that is, subclassable) class with its own identity in the type system; since the SBCL compiler makes heavy use of type inference, both when compiling user code and when cross-compiling itself, it was also important for the compiler's version of `subtypep` to know that the types `(and sequence vector)` and `(and sequence list)` are equivalent to `vector` and `list` respectively, irrespective of the extensibility of `sequence`.

Additionally, the declared return type of various sequence functions such as `copy-seq` needed to be altered: SBCL has the type `consed-sequence`, which was previously aliased to `(or list (simple-array * (*)))`, expressing that (in SBCL) freshly consed vectors do not have fill-pointers, are not displaced and are not adjustable. In an implementation supporting extensible sequences, this type alias needs to be changed to `(or (simple-array * (*)) (and sequence (not vector)))` to correctly represent the implementation's behaviour[7].

After these modifications to the type system, the implementation of the remainder of this proposal involved two distinct parts: modifying SBCL itself to insert calls to the sequence generic functions, and implementing those generic functions in a userspace library. For reasons of convenience, it was decided to preserve a distinction between Common Lisp functions and their extensible counterparts, trampolining to the latter if a sequence argument was neither a list nor a vector; incremental development of the SBCL internals was eased by the use of a `seq-dispatch` macro, taking a sequence argument and expanding into either two or three cases: one for when the argument is a list, one when it is a vector, and (optionally) one for neither: the third case is only executed when a generic sequence is encountered.

The various compiler transformations and optimizations of sequence functions (such as `find`, `position`, `map`, `coerce`) were largely unaffected, as either the various checks on their applicability were already restrictive enough, or else the optimizations performed were generic to all sequences. The implementation of `find` and `position` (and `-if` and `-if-not` relatives) needed a small amount of alteration to allow the protocol for `sequence:find` and `sequence:position` described in section A.3 to be implemented, while the implementation of `map` was improved in the process of development, being rewritten to use `dynamic-extent` support, resolving a long-standing issue in the system.

Support for defining subclasses of `sequence` from the customized PCL (Bobrow and Kiczales, 1988) which SBCL incorporates to implement CLOS was as simple as modifying the system method for the MetaObject Protocol function `mop:validate-superclass` to allow `sequence` as a direct superclass of classes with metaclasses `standard-class` and `mop:funcallable-standard-class`. In addition, to support

---

[7] SBCL has a sophisticated understanding of the Common Lisp type system, so such complex types do not cause difficulty in type inference.

```
(defun loop-elements-iteration-path (variable data-type prep-phrases)
  (let (of-phrase)
    (loop for (prep . rest) in prep-phrases do
          (ecase prep
            ((:of :in) (if of-phrase
                           (sb-loop::loop-error "Too many prepositions")
                           (setq of-phrase rest)))))
    (destructuring-bind (it lim f-e step endp elt seq)
        (loop repeat 7 collect (gensym))
      (push `(let ((,seq ,(car of-phrase)))) sb-loop::*loop-wrappers*)
      (push `(sequence:with-sequence-iterator (,it ,lim ,f-e ,step ,endp ,elt) (,seq))
            sb-loop::*loop-wrappers*)
    `(((,variable nil ,data-type)) () () nil (funcall ,endp ,seq ,it ,lim ,f-e)
      (,variable (funcall ,elt ,seq ,it) ,it (funcall ,step ,seq ,it ,f-e))))))
(sb-loop::add-loop-path
 '(element elements) 'loop-elements-iteration-path sb-loop::*loop-ansi-universe*
 :preposition-groups '((:of :in)) :inclusive-permitted nil)
```

Figure 6: Extension of `loop` for a loop path to iterate over the elements of generic sequences.

the operations involving sequence type specifiers (such as `map` and `merge`), we used `make-sequence-like` acting on the `mop:class-prototype` of the class named by the type specifier, if that class was a subclass of `sequence`.

SBCL's `loop` facility is based on the MIT implementation, so it was straightforwardly extended using the `add-loop-path` operator as shown in figure 6.

5. CONCLUSIONS

We have described an extension to Common Lisp to allow users to define their own sequence classes in an interoperable manner, and have described the changes necessary to implement this extension in a contemporary implementation. Additionally, we have attempted to justify the need for this extension in terms of expressivity and parsimony, and given simple examples of using it.

We leave for future work the related task of defining a protocol for user-defined collections; Common Lisp sequences are a specialized form of collection, finite and ordered. However, preliminary work suggests that such a collection protocol can be defined to interoperate with the proposals in this document for sequences.

It is intended that a revised version of appendix A should be submitted to some suitable forum for standardization of some form; it is likely that feedback gathered from discussion of this paper will mean that there will be some changes in detail in the revision process, so users should not rely on the description in this paper remaining authoritative.

Acknowledgments

The author thanks Nikodemus Siivola, Marcus Pearce, Eric Marsden, Peter Housel, Cyrus Harmon, Peter Graves, Paul Dietz and Pascal Costanza for fruitful and helpful discussions, and acknowledges support from EPSRC grant number GR/S84750/01.

References

Bobrow, D. G. and Kiczales, G. (1988). The Common Lisp Object System Metaobject Kernel: A Status Report. In *Lisp and Functional Programming*, pages 309–315.

Franz Inc. (2006). Allegro CL version 8.0 documentation. `http://tinyurl.com/3cfu66`.

Gray, D. N. (1989). Issue `STREAM-DEFINITION-BY-USER`. Failed Issue, X3J13, ANSI. `http://tinyurl.com/2d4csd`.

Haible, B. (1988). The Abstract Datatype *Sequence*. Technical report, University of Karlsruhe. `http://tinyurl.com/yy3eys`.

Haible, B. (2006). personal communication.

Kelsey, R., Clinger, W., and Rees, J. (1998). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1).

Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, pages 220–242.

Miller, S. G. (2004). Collections. SRFI 44, `schemers.org`. `http://srfi.schemers.org/srfi-44/srfi-44.html`.

Newman, W. H. et al. (2000). SBCL User Manual. `http://www.sbcl.org/manual/`.

Norvig, P. (2002). Large-Scale Web Services, and the Programming Languages that Build Them. In *International Lisp Conference Proceedings*.

Pestov, S. (2006). Factor documentation. `http://factorcode.org/responder/help/`.

Pitman, K. and Chapman, K., editors (1994). *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI.

Pitman, K. M. (1991). Issue `CONCATENATE-SEQUENCE`. Issue 73, X3J13, ANSI. `http://tinyurl.com/2zt4d6`.

Rhodes, C. (2006). Revisiting `CONCATENATE-SEQUENCE`. Document 3, Common Lisp Document Repository. `http://cdr.eurolisp.org/document/3`.

Rhodes, C., Strandh, R., and Mastenbrook, B. (2005). Syntax Analysis in the Climacs Text Editor. In *International Lisp Conference Proceedings*.

Shalit, A. (1996). *The Dylan Reference Manual.* Addison-Wesley, Redwood City, CA, USA. `http://www.opendylan.org/books/drm/Title`.

Steele, Jr, G. L. (1990). *Common Lisp: The Language.* Digital Press, second edition.

Strandh, R., Villeneuve, M., and Moore, T. (2004). Flexichain: An editable sequence and its gap-buffer implementation. In *European Lisp and Scheme Workshop.* `http://tinyurl.com/yhqwp3`.

Yee, K.-P. and van Rossum, G. (2001). Iterators. PEP 234, Python Software Foundation. `http://www.python.org/dev/peps/pep-0234/`.

APPENDIX

A. SPECIFICATION

In the sections below, all of the generic functions and macros being specified are named by symbols external in the `sequence` package. In cases where the generic function being defined corresponds to a standardized Common Lisp function, it is not specified whether the corresponding Common Lisp function is distinct from the specified generic functions, nor indeed whether the symbol in the `common-lisp` package is distinct from that in the `sequence` package; that is, `#'length` might or might not be `eql` to `#'sequence:length`, and `length` might or might not be `eql` to `sequence:length`. The `sequence` package may have additional, implementation-specific names; `sequence` need not be the `package-name` of the package.

It is not specified whether the methods specified on list and vector are in fact called when the Common Lisp function corresponding to the generic function is called on such data. It is not specified whether the methods specified on sequence are called when the Common Lisp function corresponding to the generic function is called on data of type vector or list.

In implementations which support the Metaobject Protocol as defined in Kiczales et al. (1991), suitable methods on `mop:validate-superclass` should be provided such that no error is signalled for user-defined sequence classes of metaclass `standard-class`; such implementations are additionally encouraged to allow `mop:funcallable-standard-class` as a compatible metaclass for user-defined sequence classes.

A.1 Basic sequence operations

Generic Function `length`
Syntax:
`length` *sequence*

The generic function `length` corresponds to calls to the Common Lisp function `length`.

Primary Method `length` (*l* list)

Primary Method `length` (*v* vector)

The methods provided simply compute the length of the sequence, as if by `length`.

Primary Method `length` (*s* sequence)

This method signals an error of type `type-error`, for compatibility with the requirements of `length`'s argument to be a *proper sequence.*

Generic Function `elt`
Syntax:
`elt` *sequence index*

This generic function corresponds to calls to `elt`.

Primary Method `elt` (*l* list) *index*

Primary Method `elt` (*v* vector) *index*

These methods simply return the element of the provided sequence at the given index, as if by `elt`.

Primary Method `elt` (*s* sequence) *index*

This method signals an error of type `type-error`, for compatibility with the requirements of `elt`'s sequence argument to be a *proper sequence.*

Generic Function `(setf elt)`
Syntax:
`(setf elt)` *new-value sequence index*

This generic function corresponds to calls to `(setf elt)`.

Primary Method `(setf elt)` *new-value* (*l* list) *index*

Primary Method `(setf elt)` *new-value* (*v* vector) *index*

These methods set the element of the provided sequence at the given index to be *new-value*, as if by `(setf elt)`

Primary Method `(setf elt)` *new-value* (*s* sequence) *index*

This method signals an error of type `type-error`, for compatibility with the requirements of `(setf elt)`'s sequence argument to be a *proper sequence.*

Generic Function `make-sequence-like`
Syntax:
`make-sequence-like` *sequence length* `&key`
initial-element initial-contents

This generic function returns a sequence of the same class as its *sequence* argument, with the specified *length.* If neither *initial-element* nor *initial-contents* is supplied, the consequences are undefined if any element of the resulting sequence is read before being written. If *initial-element* is provided, it is used to initialize the resulting sequence; if *initial-contents* is provided, it must be a sequence of length *length*, which is used to initialize the resulting sequence. The consequences are undefined if both *initial-element* and *initial-contents* are supplied.

Primary Method `make-sequence-like` (*l* list) *length* `&key` *initial-element initial-contents*

Primary Method `make-sequence-like` (*v* vector) *length* `&key` *initial-element initial-contents*

No behaviour for these methods is specified beyond that for the generic function.

```
;;; syntax: #[1 2 3 ...] for ordinary queues, #{1 2 3 ...} for funcallable ones
(macrolet ((def (name open close)
               (let ((reader-name (intern (format nil "~A-~A" name 'reader))))
                 `(progn
                   (defun ,reader-name (stream char n)
                     (declare (ignore char n))
                     (let ((contents (read-delimited-list ,close stream t))
                           (result (make-instance ',name)))
                       (dolist (o contents result) (enqueue o result))))
                   (set-dispatch-macro-character #\# ,open ',reader-name)
                   (set-syntax-from-char ,close #\))
                   (defmethod print-object ((o ,name) stream)
                     (when (and *print-readably* (not (eq (class-of o) (find-class ',name))))
                       (call-next-method))
                     (format stream "#~C" ,open)
                     (do ((data (cdr (%queue-data o)) (cdr data)) (i 0 (1+ i)))
                         ((null data) (format stream "~C" ,close) o)
                       (unless (or *print-readably* (not *print-length*))
                         (when (= i *print-length*)
                           (format stream "...~C" ,close)
                           (return o)))
                       (let ((*print-level* (and *print-level* (1- *print-level*))))
                         (write (car data) :stream stream))
                       (unless (null (cdr data)) (format stream " "))))))))
  (def queue #\[ #\]) (def funcallable-queue #\{ #\}))
```

Figure 7: Implementation of reader macros and print functions for queues and funcallable queues. Note the support for correct behaviour in the presence of non-`nil` values of printer control variables.

Primary Method `make-sequence-like` (*s* sequence) *length* `&key` *initial-element initial-contents*

This method signals an error of type `type-error`.

Generic Function `adjust-sequence`
Syntax:
`adjust-sequence` *sequence length* `&key` *initial-element initial-contents*

Primary Method `adjust-sequence` (*l* list) *length* `&key` *initial-element initial-contents*

Primary Method `adjust-sequence` (*v* vector) *length* `&key` *initial-element initial-contents*

This method functions in a similar manner to `adjust-array`, though the implementation may choose to preserve the identity of the argument if it has a fill pointer and the *length* argument is not greater than the size of the vector.

Primary Method `adjust-sequence` (*s* sequence) *length* `&key` *initial-element initial-contents*

This method signals an error of type `type-error`.

A.2 Iteration

Generic Function `make-sequence-iterator`
Syntax:
`make-sequence-iterator` *sequence* `&key` *from-end start end*

This generic function returns nine values: three values corresponding to an iterator state, a limit state and *from-end*, and six functions with signatures and functionality like the `iterator-foo` functions below.

Primary Method `make-sequence-iterator` (*s* sequence) `&key` *from-end* (*start* `0`) *end*

This method returns the three values returned by calling `make-simple-sequence-iterator`, along with the iterator functions `#'iterator-step`, `#'iterator-endp`, `#'iterator-element`, `#'(setf iterator-element)`, `#'iterator-index` and `#'iterator-copy` defined below.

Generic Function `make-simple-sequence-iterator`
Syntax:
`make-simple-sequence-iterator` *sequence* `&key` *from-end start end*

This generic function returns three values: an iterator object, a limit state and *from-end*. These values, along with *sequence*, are to be used for calling to the iterator generic functions, below; the consequences are unspecified if objects not returned by a call to `make-simple-sequence-iterator` are passed as arguments to the iterator functions.

Primary Method `make-simple-sequence-iterator` (*l* list) `&key` *from-end* (*start* `0`) *end*

Primary Method `make-simple-sequence-iterator` (*v* vector) `&key` *from-end* (*start* `0`) *end*

These methods, in combination with methods on the iterator generic functions, below, produce objects of implementation-defined nature to allow iteration to occur.

Primary Method `make-simple-sequence-iterator` (*s* sequence) `&key` *from-end* (*start* `0`) *end*

If *from-end* is *false*, this method returns the three values *start*, *end* (or the length of *s* if *end* is `nil`), and `nil`. If *from-end* is *true*, this method returns the three values `(1- (or` *end* `(length` *s*`)))`, `(1-` *start*`)`, and *from-end*.

```
(defmethod sequence:make-sequence-iterator :around ((o undoable-mixin) &key from-end start end)
  (declare (ignore from-end start end))
  (multiple-value-bind (s l f step endp elt setelt index copy) (call-next-method)
    (values s l f step endp elt
            (lambda (nv s i)
              (if (recording s)
                  (funcall setelt nv s i)
                  (without-undoing (s)
                    (push (make-instance ’setelt-record :index (funcall index s i) :value (funcall elt s i))
                          (record s))
                    (funcall setelt nv s i))))
            index copy)))

(defmacro with-compound-recording ((object) &body body)
  ‘(let ((old (recording ,object)) (or (record ,object)))
    (unwind-protect
         (progn (setf (recording ,object) nil) (setf (record ,object) nil) ,@body)
      (setf (recording ,object) old)
      (setf (record ,object) (cons (make-instance ’compound-record :records (record ,object)) or)))))

(defclass compound-record () ((records :initarg :records :reader records)))
(defmethod undo-using-record ((o undoable-mixin) (r compound-record))
  (dolist (r (records r)) (undo-using-record o r)))
(defmacro define-compound-undo-method (name arglist)
  ‘(define-undo-method ,name ,arglist
     (with-compound-recording (o) (call-next-method))))

(define-compound-undo-method sequence:nsubstitute (new old o &key from-end start end test test-not key count))
(define-compound-undo-method sequence:nsubstitute-if (new old o &key from-end start end key count))
(define-compound-undo-method sequence:nsubstitute-if-not (new old o &key from-end start end key count))

(defclass adjust-sequence-record ()
  ((previous :initarg :previous :reader previous) (discarded :initarg :discarded :reader discarded)))
(define-undo-method sequence:adjust-sequence (o length &rest args)
  (push (make-instance ’adjust-sequence-record :previous (length o)
                       :discarded (when (< length (length o)) (coerce (subseq o length) ’vector)))
        (record o)))
(defmethod undo-using-record ((o undoable-mixin) (r adjust-sequence-record))
  (sequence:adjust-sequence o (previous r))
  (when (discarded r) (setf (subseq o (- (length o) (length (discarded r)))) (discarded r))))

(define-compound-undo-method sequence:delete (item o &key from-end (start 0) end test test-not key count))
(define-compound-undo-method sequence:delete-if (item o &key from-end (start 0) end key count))
(define-compound-undo-method sequence:delete-if-not (item o &key from-end (start 0) end key count))
(define-compound-undo-method sequence:delete-duplicates (o &key from-end (start 0) end test test-not key))
(define-undo-method sequence:replace (o sequence2 &key (start1 0) end1 (start2 0) end2)
  (push (make-instance ’fill-record :start start1 :end end1 :contents (coerce (subseq o start1 end1) ’vector))
        (record o)))

(define-undo-method sequence:sort (o predicate &key key)
  (push (make-instance ’fill-record :contents (coerce o ’vector) :start 0 :end nil) (record o)))
(define-undo-method sequence:stable-sort (o predicate &key key)
  (push (make-instance ’fill-record :contents (coerce o ’vector) :start 0 :end nil) (record o)))
```

Figure 8: Implementation details of the undoable mixin. The setter in the iterator protocol is wrapped by the `:around` method on `make-sequence-iterator`; a specialized undo record is defined for `adjust-sequence`; and undo records for `nsubstitute` and `delete` are represented as sequences of primitive undo records, treated as an atom for the purposes of running undo.

Generic Function `iterator-step`
Syntax:
`iterator-step` *sequence iterator from-end*

This generic function returns a new iterator state representing the advancement of the iteration across *sequence* in the direction indicated by *from-end.*

Primary Method `iterator-step` (*l* list) *iterator from-end*

Primary Method `iterator-step` (*v* vector) *iterator from-end*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `iterator-step` (*s* sequence) *iterator from-end*

If *from-end* is *true*, this returns `(1-` *iterator*`)`; otherwise, it returns `(1+` *iterator*`)`.

Generic Function `iterator-endp`
Syntax:
`iterator-endp` *sequence iterator limit from-end*

This generic function tests the iterator for having reached its end state for iteration across *sequence* indicated in the direction indicated by *from-end.*

Primary Method `iterator-endp` (*l* list) *iterator limit from-end*

Primary Method `iterator-endp` (*v* vector) *iterator limit from-end*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `iterator-endp` (*s* sequence) *iterator limit from-end*

This returns the value of `(=` *iterator limit*`)`.

Generic Function `iterator-element`
Syntax:
`iterator-element` *sequence iterator*

This generic function returns the element of *sequence* at the point of iteration indicated by *iterator.*

Primary Method `iterator-element` (*l* list) *iterator*

Primary Method `iterator-element` (*v* vector) *iterator*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `iterator-element` (*s* sequence) *iterator*

This method returns the value of `(elt` *s iterator*`)`.

Generic Function `(setf iterator-element)`
Syntax:
`(setf iterator-element)` *new-value sequence iterator*

This generic function sets the element of *sequence* at the point of iteration indicated by *iterator* to *new-value.*

Primary Method `(setf iterator-element)` *new-value* (*l* list) *iterator*

Primary Method `(setf iterator-element)` *new-value* (*v* vector) *iterator*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `(setf iterator-element)` *new-value* (*s* sequence) *iterator*

This method returns the value of `(setf (elt` *s iterator*`)` *new-value*`)`.

Generic Function `iterator-index`
Syntax:
`iterator-index` *sequence iterator*

This generic function returns the index of the point indicated by *iterator* in *sequence.*

Primary Method `iterator-index` (*l* list) *iterator*

Primary Method `iterator-index` (*v* vector) *iterator*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `iterator-index` (*s* sequence) *iterator*

This method returns *iterator.*

Generic Function `iterator-copy`
Syntax:
`iterator-copy` *sequence iterator*

This generic function returns an iterator state identifying the same point in an iteration as *iterator*, such that changes to one do not affect the other.

Primary Method `iterator-copy` (*l* list) *iterator*

Primary Method `iterator-copy` (*v* vector) *iterator*

No behaviour for these methods is specified beyond that for the generic function.

Primary Method `iterator-copy` (*s* sequence) *iterator*

This method returns *iterator.*

Macro `with-sequence-iterator` (`&optional` *state limit from-end step endp elt setelt index copy*) (*sequence* `&key` *from-end* (*start* `0`) *end*) `&body` *body*

This macro binds the names in its first argument list as if by `multiple-value-bind` to the values returned by `make-sequence-iterator` applied to its second argument list, and then executes *body.*

Macro `dosequence` (*var sequence-form* `&optional` *result-form*) `&body` *body*

This macro iterates over a sequence, in a similar fashion to `dolist` iterating over a list. The *body* is like a `tagbody`, consisting of a series of tags and statements.

`dosequence` evaluates *sequence-form*, which should produce a sequence. It then executes the body once for each element in the sequence, in the order in which the tags and statements occur, with *var* bound to the element. Then *result-form* is evaluated with *var* bound to `nil`.

An implicit block named `nil` surrounds the entire `dosequence` form. `return` may be used to terminate the loop immediately without performing any further iterations, returning zero or more values.

The scope of the binding of *var* does not include the *sequence-form*, but the *result-form* is included.

It is implementation-dependent whether `dosequence` establishes a new binding of *var* on each iteration or whether it establishes a binding for *var* once at the beginning and then assigns it on any subsequent iterations.

Loop Path `for-as-sequence`

This allows using an iteration variable, as with other loop `for-as-` clauses, as if the following clause were added to the `loop` grammar (Pitman and Chapman, 1994, Macro `loop`):

```
for-as-sequence::=
  var [type-spec] being {each | the}
  {element | elements} {in | of} sequence
```

and as if the `for-as-subclause` clause permitted this `for-as-sequence` clause along with the standardized clauses. As with other iteration control clauses, the variable argument may be a destructuring list. The effect of this clause is to iterate through the contents of a sequence, starting from the zeroth element and terminating at the end of the sequence[8].

A.3 Sequence Function Specifications

Except as specified here, it is implementation-dependent whether methods on these generic functions call other such generic functions or not. For each of these generic functions there is a method, called the “default method” in the descriptions below, where all sequence parameters are specialized on `sequence`, implementing the default behaviour given an implementation of the iteration and basic sequence protocol in the sections above. The user of this protocol is permitted to extend or override these generic functions, but is not permitted to specialize any non-sequence argument.

It is unspecified whether the generic functions specified below are called when the corresponding Common Lisp function has only `list` and `vector` arguments for sequences; however, the implementation is required to provide methods for these generic functions implementing similar behaviour, so that the user may call these generic functions on `list` and `vector` sequences.

Generic Function `count`
Syntax:
`count` *item sequence* `&key` *from-end start end test test-not key*

Generic Function `count-if`
Syntax:
`count-if` *predicate sequence* `&key` *from-end start end key*

Generic Function `count-if-not`
Syntax:
`count-if-not` *predicate sequence* `&key` *from-end start end key*

Generic Function `find`
Syntax:
`find` *item sequence* `&key` *from-end start end test test-not key*

Generic Function `find-if`
Syntax:
`find-if` *predicate sequence* `&key` *from-end start end key*

Generic Function `find-if-not`
Syntax:
`find-if-not` *predicate sequence* `&key` *from-end start end key*

Generic Function `position`
Syntax:
`position` *item sequence* `&key` *from-end start end test test-not key*

Generic Function `position-if`
Syntax:
`position-if` *predicate sequence* `&key` *from-end start end key*

Generic Function `position-if-not`
Syntax:
`position-if-not` *predicate sequence* `&key` *from-end start end key*

Generic Function `subseq`
Syntax:
`subseq` *sequence start* `&optional` *end*

Generic Function `copy-seq`
Syntax:
`copy-seq` *sequence*

The default method on `copy-seq` calls `subseq` with parameters *sequence* and `0`.

Generic Function `fill`
Syntax:
`fill` *sequence item* `&key` *start end*

Generic Function `nsubstitute`
Syntax:
`nsubstitute` *new old sequence* `&key` *from-end start end test test-not key count*

Generic Function `nsubstitute-if`
Syntax:
`nsubstitute-if` *new predicate sequence* `&key` *from-end start end key count*

Generic Function `nsubstitute-if-not`
Syntax:
`nsubstitute-if-not` *new predicate sequence* `&key` *from-end start end key count*

Generic Function `substitute`

---

[8] Note that this behaviour is different from that of the `for-as-across` loop clause on vectors with fill pointers.

Syntax:
`substitute` *new old sequence* `&key` *from-end start end test test-not key count*

Generic Function `substitute-if`
Syntax:
`substitute-if` *new predicate sequence* `&key` *from-end start end key count*

Generic Function `substitute-if-not`
Syntax:
`substitute-if-not` *new predicate sequence* `&key` *from-end start end key count*

The default methods on `substitute`, `substitute-if` and `substitute-if-not` call `copy-seq` on *sequence*, and then call `nsubstitute`, `nsubstitute-if` and `nsubstitute-if-not` respectively on their arguments with the provided sequence replaced by the copy.

Generic Function `replace`
Syntax:
`replace` *sequence1 sequence2* `&key` *start1 end1 start2 end2*

Generic Function `nreverse`
Syntax:
`nreverse` *sequence*

Generic Function `reverse`
Syntax:
`reverse` *sequence*

The default method on `reverse` calls `nreverse` on the result of `copy-seq` on *sequence*.

Generic Function `reduce`
Syntax:
`reduce` *function sequence* `&key` *from-end start end initial-value*

Generic Function `mismatch`
Syntax:
`mismatch` *sequence1 sequence2* `&key` *from-end start1 end1 start2 end2 test test-not key*

Generic Function `search`
Syntax:
`search` *sequence1 sequence2* `&key` *from-end start1 end1 start2 end2 test test-not key*

Generic Function `delete`
Syntax:
`delete` *item sequence* `&key` *from-end start end test test-not key count*

Generic Function `delete-if`
Syntax:
`delete-if` *predicate sequence* `&key` *from-end start end key count*

Generic Function `delete-if-not`
Syntax:
`delete-if-not` *predicate sequence* `&key` *from-end start end key count*

Generic Function `remove`
Syntax:
`remove` *item sequence* `&key` *from-end start end test test-not key count*

Generic Function `remove-if`
Syntax:
`remove-if` *predicate sequence* `&key` *from-end start end key count*

Generic Function `remove-if-not`
Syntax:
`remove-if-not` *predicate sequence* `&key` *from-end start end key count*

The default methods on `remove`, `remove-if` and `remove-if-not` call `copy-seq` on *sequence*, and then call `delete`, `delete-if` and `delete-if-not` respectively on their arguments with the provided sequence replaced by the copy.

Generic Function `delete-duplicates`
Syntax:
`delete-duplicates` *sequence* `&key` *from-end start end test test-not key*

Generic Function `remove-duplicates`
Syntax:
`remove-duplicates` *sequence* `&key` *from-end start end test test-not key*

The default method on `remove-duplicates` calls `copy-seq` on *sequence*, and then calls `delete-duplicates` on its arguments with the provided sequence replaced by the copy.

Generic Function `sort`
Syntax:
`sort` *sequence predicate* `&key` *key*

Generic Function `stable-sort`
Syntax:
`stable-sort` *sequence predicate* `&key` *key*

The default method on `sort` behaves as if it constructs a vector with the same elements as *sequence*, calls `sort` on that vector, then replaces the elements of *sequence* with the elements of the sorted vector.

A.4 Other affected functions

Six functions in the `SEQUENCES` chapter of the ANSI CL standard do not have the structure for user-extensibility in the same way as the generic functions discussed in section A.3: they are `concatenate`, `map`, `merge`, `make-sequence`, `coerce` and `map-into`.

Taking `map-into` first, we simply specify that an implementation of user-extensible sequences must implement the `map-into` function such that target sequences and source sequences of arbitrary subtype of `sequence` are supported given a complete implementation of the sequence and iteration protocol of sections A.1 and A.2. An implementation is not prohibited from providing a means of customizing the behaviour of `map-into`, but neither is it required to.

A means of extending the behaviour of `concatenate`, `map`, `merge`, `make-sequence` and `coerce` is likewise left unspecified, though again an implementation of this document must provide for the functionality of these functions to be available for objects of arbitrary sequence type and type specifiers naming a concrete subtype of `sequence`; for details of how this interacts with the ANSI CL specified behaviour of these functions, see Rhodes (2006). It is likely that any documented fashion of extending these five functions will use the same mechanism for all of them.

As for functions which are not in the `SEQUENCES` chapter of the ANSI CL specification, the intent is that where a particular behaviour is specified for a sequence argument, that behaviour should be implemented for the user-extensible sequences. For instance, the short-circuiting quantifiers `some`, `every`, `notevery` and `notany` should accept sequence arguments of arbitrary class; `make-array`'s `:initial-contents` argument (and the `#a` array reader) should accept a sequence of sequences, as specified, including user-extended sequences.

Particular attention must be paid to the `read-sequence` and `write-sequence` functions. Although Gray (1989) does not suggest that these functions be extensible in the manner of the other stream functions, it is likely that this is because `read-sequence` and `write-sequence` were added to the language after the extensible streams proposal was made, and indeed Common Lisp implementations have provided `stream-read-sequence` and `stream-write-sequence` generic functions for user customizeability. Lisps providing both an implementation of this proposal and extensible streams based on Gray (1989) should document the effect of calling `read-sequence` and `write-sequence` with arguments being instances of non-standardized classes. The simple-streams (Franz Inc., 2006, Chapter 83) analogue does not even support all of the standard Common Lisp sequence types: only strings and octet vectors are acceptable for filling with `read-sequence` of a simple-stream; therefore, there is no significant interaction between simple-streams and the user-extensible sequences documented herein.