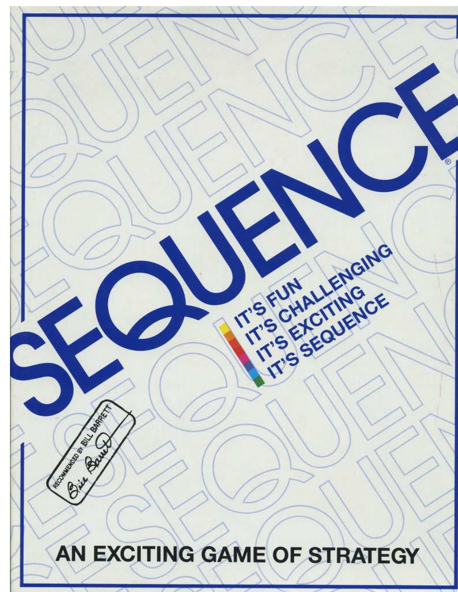
Sequence Iterators



Tobias-Christian Rittweiler

trittweiler@common-lisp.net

November 10, 2009

Scope of the library

- Sequence Iterators: library for writing functions operating on sequences;
- it does NOT define an iteration protocol,
- it's supposed to be layered on TOP of such a protocol,
  - for example, Christophe Rhodes' `SB-SEQUENCES`
    Rhodes,C.: User-extensible Sequences in Common Lisp;
    in International Lisp Conference Proceedings, 2007.
- it aims to provide:
  - tools to write such functions conveniently
  - yet not blatantly inefficiently

Current state of affairs

Ever wrote a function operating on sequences?

You have to deal with

- two definitions: one for lists, the other for vectors (unless you want to bury the list case in n^2)
- `:key`, and `:test`/`:test-not`
- `:start` and `:end`
  - bound checking
  - trimming of input sequence
- `:from-end`
  - in very worst case, you have to add two new definitions
- result type should match input type
  - think of specialized vectors

API Overview

Key component

- `with-sequence-iterator`

On top

- `dosequence` and `dosequences*`

Utilities

- `check-sequence-bounds`
- `make-sequence-like`
- `canonicalize-key`
- `canonicalize-test`

Utilities

`(canonicalize-key key-designator)`

> "Canonicalizes *key-designator* to a function object.
> If it's `nil`, the `identity` function is returned."

`(canonicalize-test test &optional test-not)`

> "Canonicalizes *test* and *test-no*t to a function object.
> If both are given, an error is signaled.
> If neither is given, the `eql` function is returned."

`(check-sequence-bounds sequence start end &optional length)`

> "Signals an error ... if *start*, and *end* are not valid bounding
> indices for sequence *sequence*."

- arguments are places where normalized values are stored
- including appropriate type annotations

Utilities

```
(make-sequence-like sequence length &key initial-element
                                         initial-contents)
```

> "Returns a new sequence of length *length* and of the same type as *sequence*."

An example:

```
(defun subseq (sequence start &optional end)
  (check-sequence-bounds sequence start end)
  (let ((result (make-sequence-like sequence (- end start))))
    (replace result sequence :start2 start :end2 end)))
```

Let's recall..

Ever wrote a function operating on sequences?

You have to deal with

- two definitions: one for lists, the other for vectors
- `:key`, and `:test`/`:test-not` ✔
- `:start` and `:end`
    - bound checking ✔
    - trimming of input sequence
- `:from-end`
- result type should match input type ✔

With-Sequence-Iterator

```
(with-sequence-iterator (iterator-name sequence &key start
                                                      end
                                                      from-end
                                                      place)
  ...)
```

- iterator returns
  1. boolean indicating when the iterator exhausted
  2. the current element of *sequence*
  3. the index of that element
- iterator traverses *sequence*
  - starting from `:start`
  - until `:end`,
  - in reverse order if `:from-end` is given
- bounding indices are not checked for validity (use `check-sequence-bounds`)
- `:place` can be used to destructively modify *sequence* during traversal

Dosequence

```
(dosequence (var sequence &optional result &key start
                                                  end
                                                  from-end
                                                  place)
  ...)
```

- convenience macro
  - with-sequence-iterator
  - + loop until iterator exhausts
- *var* can also be of form `(elt idx)`
- like all `do`-style macros, body is executed in an implicit `block` and `tagbody`
- (everyone aware of semantics of `&optional` + `&key`?)

Dosequence (an example)

```
(defun find (item sequence &key (start 0) end from-end
                                 key test test-not)
  (check-sequence-bounds sequence start end)
  (let ((key  (canonicalize-key key))
        (test (canonicalize-test test test-not)))
    (dosequence (elt sequence nil :start start :end end
                                   :from-end from-end)
      (when (funcall test item (funcall key elt))
        (return elt)))))
```

Dosequence (another example)

```
(defun map-into-subseq (sequence function &key (start 0) end from-end key)
  (check-sequence-bounds sequence start end)
  (let ((key (canonicalize-key key)))
    (dosequence (elt sequence sequence :start start :end end
                                       :from-end from-end
                                       :place ptr)
      (setf (ptr) (funcall function (funcall key elt))))))
```

Dosequences*

```
(dosequences* ((var sequence &optional result &key start
                                                    end
                                                    from-end
                                                    place)
               &rest more-clauses)
  ...)
```

- convenience macro to iterate through multiple sequences simultaneously
  - multiple `with-sequence-iterators`
  - + loop until one of the iterators exhausts
- returns the *result* of the clause belonging to the iterator exhausted first
- `dosequences*` because it runs iterators sequentially (à la `do*`)

Dosequences* (an example)

```
(defun begins-with-subseq (prefix sequence &key key test test-not
                                                (start1 0) end1
                                                (start2 0) end2)
  (check-sequence-bounds prefix   start1 end1)
  (check-sequence-bounds sequence start2 end2)
  (let ((key  (canonicalize-key key))
        (test (canonicalize-test test test-not)))
    (dosequences* ((p prefix   t   :start start1 :end end1)
                   (s sequence nil :start start2 :end end2))
      (unless (funcall test (funcall key p) (funcall key s))
        (return nil)))))
```

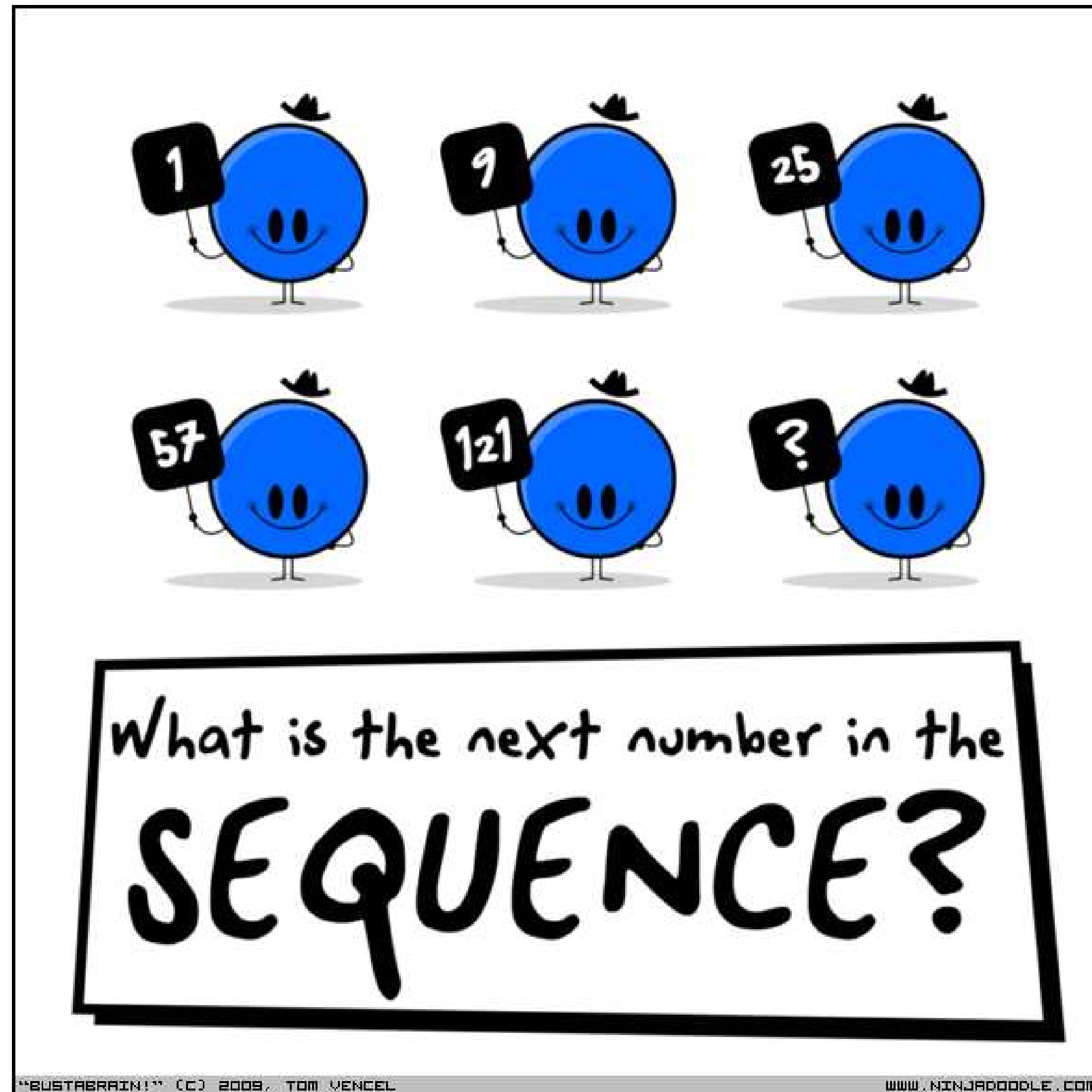
Let's recall again..

Ever wrote a function operating on sequences?

You have to deal with

- two definitions: one for lists, the other for vectors ✔
- `:key`, and `:test`/`:test-not` (✔)
- `:start` and `:end`
  - bound checking (✔)
  - trimming of input sequence ✔
- `:from-end` ✔
- result type should match input type (✔)

Short break...



Performance

Compared reimplementation of standard functions (`find`, `count`, `subseq`, ...) with builtin equivalents.

On SBCL:

- sequence iterators are pretty fast
- depending on grade of builtin optimization: 1.3x - 2x slower

On CCL:

- sequence iterators are an order of magnitude slower
- I haven't investigated why (perhaps because the current code assumes good type derivation)

Open issues

- declarations (e.g. `ignore`)
  - currently not allowed in body of `dosequence`/`dosequences*`
  - have to split body, and insert declarations at right places
  - luckily, I wrote a library for that in 2008:

    `http://common-lisp.net/project/parse-declarations/`

Open issues (cont.)

- `copy-sequence-iterator`
  - How do we traverse lists in reverse order?
  - we create a reversed list of pointers (for `:place`) and traverse that
  - Implementing `search`: nested `with-sequence-iterators`
    `(on same sequence, different :start)`
  - that's costy!
  - `copy-sequence-iterator` could create a copy with some shared state.

Open issues (cont.)

- output sequence allocation
  - Often you do not in advance how big the result sequence must be
    - E.g. when implementing `remove-if`
  - various possible solutions
    - iterate twice (first to calculate exact output size, then to fill output sequence)
    - allocate input-sized output sequence, fill, shrink
    - incremently allocate chunks, fill, concatenate (similiar to “list arrays”)
    - anything else? (anonymous reader, I count on thee!)
  - it is not yet clear to me what to prefer (perhaps depend on `(> space speed)`?)
  - should probably check prior work..

Long-term issues

- on top of Sequence Iterators, library which provides additional sequence functions
  - `split-sequence`,
  - `take-while`, `drop-while`,
  - `partition`, `group`,
  - `replace-match`, `replace-all-matches`

- underneath Sequence Iterators
  - add generic iteration protocol
  - then, implement standard sequence functions with Sequence Iterators
  - voilà: portable extensible sequences for Common Lisp

The End

Comming soon:

`http://common-lisp.net/project/sequence-iterators/`