How to Hurt Your Friends with (loop)

Daniel Lyons
fusion@storytotell.org

Why?

- Fast
- Powerful
- Widely-used

Like format you can probably do
more with it than you realize.

Even though you hate it, you may have
to read it in someone else's code.

On my machine, 578 of 1068 library files
contain at least one loop form (54.12%).

It's fast as hell.

loop as List Comprehension

- Python:
  ```
  [ f(x) for x in L if p(x) ]
  ```
- Haskell:
  ```
  [ f(x) | x <- L, p(x) ]
  ```
- Erlang:
  ```
  [ f(X) || X <- L, p(X) ].
  ```
- Loop:
  ```
  (loop for x in L when (p x) collect (f x))
  ```

Things You Can Do With loop

- Variable initialization and stepping
- Value accumulation
- Conditional and unconditional execution
- Pre- and post-loop operations
- Termination whenever and however you like

This is taken from the list of kinds of loop clauses from CLtL Ch. 26
- variable initialization and stepping
- value accumulation
- termination conditions
- unconditional execution
- conditional execution
- miscellaneous operations

Initialization

- for/as
- with
- repeat

for and as mean the same thing
with is a single ‘let’ statement
repeat runs the loop a specified number of times

for Is Really Complicated

- for var from expr to expr
- for var in expr
- for var on expr

Of course from can also be downfrom or upfrom, and to can also be downto, upto, below or above
for var in expr:
  Loops over every item in the list
  Optionally by some stepping function
for var on expr loops over the cdrs of a list, giving you each successive sublist
  Loops over each sublist of the list,
  e.g. ‘(1 2 3) ‘(2 3) `(3)
  Also optionally with a stepping function

for Is Really Complicated

- for var = expr1 [then expr2]
- for var across vector

for var = expr1:
  Initially set var to expr1, then call expr2 each subsequent iteration of the loop. Or, just keep calling expr1 if no expr2 provided.

for var across vector: just set var to each element of the vector

for Is Really Complicated

- for var being {each|the} {hash-key|hash-keys|hash-value|hash-values} {in|of} hashtable [using ({hash-value|hash-key} other-var)
- Wow, that's easy to understand

This is for iterating the keys or values of a hash table
The using construct lets you access the other value in the pair (the key if iterating by value, or value if iterating by key)

for Is Really Complicated

- for var being {each|the} {symbol|present-symbol|external-symbol|symbols|present-symbols|external-symbols} {in|of} package

This is for iterating through the symbols of a package

Accumulation

- collect
- append/nconc
- sum
- count
- minimize/maximize

collect accumulates a list of results
append/nconc appends each value to a list it builds, nconc destructively
sum computes the sum of all the items
count computes the count of all the items
minimize and maximize give you the minimal and maximal values

End Test Control

- while/until
- always/never/thereis
- loop-finish

while and until terminate the loop when their test returns false or true
always and never terminate the loop with nil when their form evaluates to nil or not nil, otherwise returning t.
thereis is like never but returns the value
loop-finish ends the loop early

Execution

- Conditional: when/if/unless expr clause {and clause}* else clause [end]
- Unconditional: do {expr}*

"Misc"

- initially
- finally
- named
- Destructuring assignment

initially just executes some code at the beginning
finally execute some code at the end, or with return, change the result of the loop
named lets you name your loop. I have no idea why

"Misc"

- of-type to declare types for your values
- and can be used to define parallel actions or assignment
- For readability, you can use symbols instead of atoms

```
(loop :for i :in '(1 2 3)...)
```

Some examples

- rectangles.lisp in spacial-trees:

```
(defgeneric minimum-bound (one two))
(defmethod minimum-bound ((r1 rectangle) (r2 rectangle))
  (make-rectangle
   :lows #+slow (mapcar #'boundmin (lows r1) (lows r2))
   (loop for l1 in (lows r1) for l2 in (lows r2)
         collect (boundmin l1 l2))
   :highs #+slow (mapcar #'boundmax (highs r1) (highs r2))
   (loop for h1 in (highs r1) for h2 in (highs r2)
         collect (boundmax h1 h2))))
```

- Loop tends to be pretty snappy

Some examples

- lw-buffering.lisp from acl-compat:

```
(defun read-elements (socket-stream sequence start end reader-fn)
  (let* ((len (length sequence))
         (chars (- (min (or end len) len) start)))
    (loop for i upfrom start
          repeat chars
          for char = (funcall reader-fn socket-stream)
          if (eq char :eof) do (return-from read-elements i)
          do (setf (elt sequence i) char))
    (+ start chars)))
```

Some examples

- md5.lisp:

```
(defmacro with-md5-round ((op block) &rest clauses)
  (loop for (a b c d k s i) in clauses
        collect
        `(setq ,a (mod32+ ,b (rol32 (mod32+ (mod32+ ,a (,op ,b ,c ,d))
                                            (mod32+ (aref ,block ,k)
                                                    ,(aref *t* (1- i))))
                                    ,s)))
        into result
        finally
        (return `(progn ,@result))))
```

Some examples

- from cffi-cmucl.lisp:

```
(defun foreign-funcall-type-and-args (args)
  "Return an ALIEN function type for ARGS."
  (let ((return-type nil))
    (loop for (type arg) on args by #'cddr
          if arg collect (convert-foreign-type type) into types
          and collect arg into fargs
          else do (setf return-type (convert-foreign-type type))
          finally (return (values types fargs return-type)))))
```

Some examples

- url-rewrite.lisp

```
(defun starts-with-scheme-p (string)
  "Checks whether the string STRING represents a URL which starts with
a scheme, i.e. something like 'https://' or 'mailto:'."
  (loop with scheme-char-seen-p = nil
        for c across string
        when (or (char-not-greaterp #\a c #\z)
                 (digit-char-p c)
                 (member c '(#\+ #\- #\.) :test #'char=))
do (setq scheme-char-seen-p t)
else return (and scheme-char-seen-p
                 (char= c #\:))))
```

- There's a 200 line monster a few lines later

Some examples

```
;; performs the standard wc function on a file
(defun wc-file (filename)
  (with-open-file (file filename)
    (loop
       ;; prep for word counting
       :with last-was-whitespace :of-type boolean = t
       :and this-is-whitespace :of-type boolean

       ;; move across the characters of this array
       :for char :of-type character = (read-char file nil)
       :while char

       ;; count the bytes
       :count char :into bytes

       ;; count the words
       :do (setf this-is-whitespace (> (char-code char) 32))
       :when (and (not this-is-whitespace) last-was-whitespace)
         :count char :into words

       ;; we need to know the previous character's status for this to work
       :do (setf last-was-whitespace this-is-whitespace)

       ;; count the lines
       :when (char= char #\Newline)
         :count char :into lines

       ;; print and return it
       :finally (format t "~8@A ~7@A ~7@A ~A~%" lines words bytes filename)
       :finally (return (list lines words bytes)))))
```

I wrote this last week because I felt like duplicating the functionality of the UNIX utility ‘wc’

Competition

- iterate
  more extensible, arguably more readable
- SERIES
  lazier, more Haskell-esque

Pitfalls

- Not very extensible
- Jim pointed out you can't do this with `loop`:

```
(iter (for x from 1 to 10)
      (finding x maximizing (sin x)))
```

clsql has a loop extension, for var being each tuple in "SQL", but it's fairly convoluted how he achieved it, and relies on non-standard APIs.
I see no way to create new loop accumulators.