```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; STREAMS with DELAYED EVALUATION
;;;
;;;  Ideas from Pavel Curtis, Richard Waters, Guy Steele and
;;;    the Common LISP community.  Most of the code by
;;;    George Lugar and William Stubblefield.
;;;
;;;  DELAY and FORCE permit fine grain program control of when
;;;  a function or subprogram is executed.  Instead of flow of
;;;  control being managed by glue logic, loops, and function
;;;  nesting, evaluation is either delayed or called when needed
;;;  by the dynamic context of the program.
;;;
;;;  Conceptually, a STREAM is a dynamically available data
;;;  structure, implemented by a GENERATOR function:
;;;
;;;  {first-data-item <generator-function-for-next-data-item>}
;;;
;;;  When the first item is used by a process, the generator is
;;;  called to construct the next data-item.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmacro delay (exp) `(function (lambda () ,exp)))
(defun force (function-closure) (funcall function-closure))

;;;add a new first element to a stream
(defmacro cons-stream (new stream)
  `(cons ,new (delay ,stream)))

;;;get the first element
(defun first-stream (stream)
  (car stream))

;;;the rest of the stream is a generator function
(defun rest-stream (stream)
  (force (rest stream)))

;;;test for empty stream
(defun empty-streamp (stream)
  (null stream))

;;;make an empty stream
(defun make-empty-stream ()
  nil)

;;;append two streams
(defun combine-streams (s1 s2)
  (append s1 s2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;GENERIC-FILTER for LISTS
```

```
(defun filter (lst test)
  (cond ((null lst) nil)
        ((funcall test (first lst))
         (cons (first lst) (filter (rest lst) test)))
        (T (filter (rest lst) test))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;GENERIC-FILTER for STREAMS

(defun filter-stream (stream test)
  (cond ((empty-streamp stream) (make-empty-stream))
        ((funcall test (first-stream stream))
         (cons-stream (first-stream stream)
                      (filter-stream (rest-stream stream) test)))
        (T (filter-stream (rest-stream stream) test))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;FIBONACCI STREAM FILTER
;;;   consider generating the first N odd Fibonacci numbers
;;;   using the minimum of computational effort
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;the standard mathematical definition of Fibonacci numbers
;;;  fib[1] = 0
;;;  fib[2] = 1
;;;  fib[n] = fib[n-1] + fib[n-2]
(defun fib (n)
  (cond ((= n 1) 0)
        ((= n 2) 1)
        (T (+ (fib (- n 1)) (fib (- n 2))))))

(defun collect-fibs (n)
  (let ((result nil))
    (dotimes (i n (reverse result))  ;iterator i starts at 0, so we must
      (push (fib (+ i 1)) result)))) ;add 1 to start fib counter at 1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;an efficient doloop implementation which collects prior results
(defun memo-fibs (n)
  (let ((memo '(1 0)))
    (dotimes (i (- n 2))
      (push (+ (first memo) (second memo)) memo))
    (reverse memo)))

;;;a memoized doloop with an arbitrary filter
(defun filter-fibs (n filter-test-fn termination-test-fn)
  (do ((acc '(1 0))
       (test-acc nil))
      ((funcall termination-test-fn test-acc n)
       (reverse test-acc))
    (let ((new (+ (first acc) (second acc))))
```

```
      (push new acc)
      (when (funcall filter-test-fn new)
        (push new test-acc)))))

#|
;;; code fitted with lots of debug options,
;;;   to illustrate inline debug tools
(defun filter-fibs (n filter-test-fn termination-test-fn)
  (do ((acc '(1 0))
       (test-acc nil))
      ((funcall termination-test-fn test-acc n)
;(print (list 'VALUES-PRIOR-TO-DO-EXIT acc test-acc n))
;(break "At exit: count ~A fibs ~A filtered-fibs ~A" acc test-acc n)
       (reverse test-acc))
    (let ((new (+ (first acc) (second acc))))
;(break "new fib value: ~A"  new)
      (push new acc)
;(print (list 'new-stack acc))
      (when (funcall filter-test-fn new)
        (push new test-acc)))))
|#

(defun have-sufficient (lst n)
  (= (length lst) n))

(defun three-fives (n)
  (at-least-digits n 3 5))

;;; given a number N, find at least THIS-MANY of a given digit DIGIT
(defun at-least-digits (n this-many digit)
  (let ((digit-string (format nil "~D" n)) ;convert symbol to string
        (test-char (digit-char digit)))    ;convert symbol to character
    (count-digits-test digit-string this-many test-char)))

(defun count-digits-test (string num char)
  (>= (count char string) num))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;an implementation based on streams
;;;  cleaner, more maintainable, and as efficient as FILTER-FIBS
;;;  better than FILTER-FIBS when the termination test is complex
(defun fibonacci-stream ()
  (fib-stream-aux 0 1))

(defun fib-stream-aux (fib1 fib2)
  (cons-stream (+ fib1 fib2)
               (fib-stream-aux fib2 (+ fib1 fib2))))

(defun accumulate (n stream)
  (cond ((zerop n) nil)
        (T (cons (first-stream stream)
                 (accumulate (- n 1) (rest-stream stream))))))
```

```
(defun filter-stream-fibs (n test-fn)
  (accumulate n (filter-stream (fibonacci-stream) test-fn)))


;;;most abstract
(defun accumulate-filtered-stream (number-items generator-fn filter-fn)
  (accumulate number-items
    (filter-stream (funcall generator-fn)  filter-fn)))


#|
;;;filter examples
(filter '(1 3 -9 5 -2 -7 6) #'plusp)
;;==>  (1 3 5 6)
(filter '(1 2 3 4 5 6 7 8 9) #'evenp)
;;==>  (2 4 6 8)
(filter '(1 a b 3 c 4 7 d) #'numberp)
;;==>  (1 3 4 7)

;;;try these
(filter (collect-fibs 20) #'oddp)    ;yields only 13 odd fibs, so guess
(filter (collect-fibs 30) #'oddp)    ;to get 20 odd fibs
(time (filter (collect-fibs 30) #'oddp))
;;;don't try (filter (collect-fibs <some-guess>) #'three-fives)

;;;try these
(filter-fibs 20 #'oddp #'have-sufficient)
(filter-fibs 8 #'three-fives #'have-sufficient)
(time (dotimes (i 10) (filter-fibs 8 #'three-fives #'have-sufficient)))

;;;try these
(filter-stream-fibs 20 #'oddp)
(filter-stream-fibs 8 #'three-fives)
(time (dotimes (i 10) (filter-stream-fibs 8 #'three-fives)))

;;;try this
(accumulate-filtered-stream 8 #'fibonacci-stream #'three-fives)
|#
```