

lisp-lazy-seq

February 25, 2016

1 Lazy sequences in (a few lines of) Common Lisp

Copyright (C) 2016 F. Peschanski, CC-BY-SA 3.0

The Clojure programming language comes with a versatile lazy sequence abstraction, which also exists under other names such as generators in e.g. Python, (implicitly) lazy lists in e.g. Haskell, streams in e.g. Ocaml, etc.

There is nothing like in the Common Lisp standard, in particular the CL sequences are IMHO less interesting (providing a too restricted kind of polymorphism).

In this document I address the following question:

- is it easy to implement such an abstraction in Common Lisp ?

And as it is almost always the case in Common Lisp, the answer is :

- yes it is !

And the remaining of this document is the way I would do it.

Disclaimer : the Lisp code below is not of production quality, and it is by no mean a port of the Clojure code, although I guess there aren't a million ways to implement the lazy sequence abstraction...

1.1 What is a lazy sequence ?

Let's start with the basics.

1.1.1 A mathematical detour

According to <https://en.wikipedia.org/wiki/Sequence>,

a (mathematical) **sequence** is an ordered collection of objects in which repetitions are allowed.

Put in other terms, this is an ordered way to encode a multiset, and the simplest encoding is to associate to each element of the multiset a natural number giving its “position” in the ordering relation. The first element is at position 0, the second at position 1, etc. This gives a functional interpretation of a sequence. In this interpretation a sequence can have a finite or infinite domain, in which case we say that the sequence is finite or infinite.

There is also an inductive definition of a finite sequence, being the smallest set composed of either :

- the empty sequence `nil`
- or a non-empty sequence, constructed from a **first** element and the **rest** of the sequence.

You then obtained the definition of a **list**, our ubiquitous data structure (well, its proper, immutable variant) ! The guarantee that the sequence is finite here is that we take the smallest set satisfying the rules.

But what about infinite sequences ?

There is indeed a related mathematical definition of an infinite sequence, being the largest set composed of

- a **head** elements followed by an infinite **tail** sequence

Note that there is no base case and moreover we consider this time the largest set satisfying the unique rule, touching infinity for real ! Such definition is said coinductive and we will not go any deeper in this very interesting theory (if you're interested, cf. <http://www.cs.unibo.it/~sangio/IntroBook.html>).

Note that it is possible to mix the two definitions, by keeping the empty list and identifying **first** with **head**, and **rest** with **tail**.

So our short mathematical detour leads to the following Lisp code :

```
In [1]: (defgeneric head (sequence)
         (:documentation "Taking the head of the 'sequence'."))
```

```
Out[1]: #<STANDARD-GENERIC-FUNCTION CL-JUPYTER-USER::HEAD (0)>
```

```
In [2]: (defgeneric tail (sequence)
         (:documentation "Taking the tail of the 'sequence'"))
```

```
Out[2]: #<STANDARD-GENERIC-FUNCTION CL-JUPYTER-USER::TAIL (0)>
```

For pragmatic reasons we will also need a helper for printing sequences.

```
In [3]: (defgeneric print-cell (c out)
         (:documentation "A printer for cells."))
```

```
Out[3]: #<STANDARD-GENERIC-FUNCTION CL-JUPYTER-USER::PRINT-CELL (0)>
```

1.1.2 Strict vs. lazy sequences

Let's go back to computers and programming... In a computer everything's, in a way, is finite so the finite/infinite dichotomy is at best a view of the mind. We can perhaps more interestingly talk about stict vs. lazy sequences. Both are in fact finite, even though it is sometimes useful to consider lazy sequences as "infinite" (please do mind the double quotes).

A **strict sequence** follows the inductive definition above: it is either empty or non-empty providing a head value and a strict tail sequence. A consequence is that all the elements of strict sequences are computed values.

As long as you abolish **setf**-ing things, we can safely identify strict sequences with lists, hence the following Lisp definitions.

```
In [4]: (defmethod head ((l list))
         (first l))
```

```
Out[4]: #<STANDARD-METHOD CL-JUPYTER-USER::HEAD (LIST) {100729A193}>
```

```
In [5]: (defmethod tail ((l list))
         (rest l))
```

```
Out[5]: #<STANDARD-METHOD CL-JUPYTER-USER::TAIL (LIST) {1007361433}>
```

```
In [6]: (defmethod print-cell ((l list) out)
         (when l
          (format out "~A" (first l))
          (when (rest l)
            (format out " ")
            (print-cell (rest l) out))))
```

```
Out[6]: #<STANDARD-METHOD CL-JUPYTER-USER::PRINT-CELL (LIST T) {1007495973}>
```

```
In [7]: (head '(1 2 3 4 5))
```

```
Out[7]: 1
```

```
In [8]: (tail '(1 2 3 4 5))
```

```
Out[8]: (2 3 4 5)
```

```
In [9]: (print-cell '(1 2 3 4 5) *standard-output*)
```

```
1 2 3 4 5
```

```
Out[9]: NIL
```

Now, a **lazy sequence** is designed after the coinductive case: we know that it has a head and a tail and since we want to be lazy, we do not want to say too early what exactly are the head and tail. In particular, the head is not yet considered a computed value until we need it, and the only thing we know about the tail is that is we need it, then it will also be a lazy sequence.

1.2 Strict sequences in Lisp

It is now time to get some actual code supporting the notions discussed above. First, we consider the case of strict sequences. Of course, implementing strict sequences is neither difficult nor incredibly useful since, as we've seen previously, proper lists serve as quite a good representation of strict sequences. However, I think it is interesting to show the slight modifications involved when going from the strict to the lazy case.

According to the c2 wiki (cf. <http://c2.com/cgi/wiki?ConsCell>):

ConsCells are the building blocks of lists in LispLanguages.

Our version of a cons cell structure for strict sequences is as follows:

```
In [10]: (defstruct cell
          (hd nil)
          (tl nil))
```

```
Out[10]: CELL
```

So a strict cell has two slots : a computed head `hd` and a strict tail `tl`.

And much of what is already done for lisp can be done based on this simple structure.

```
In [11]: (defun ccons (hd tl)
          (make-cell :hd hd :tl tl))
```

```
Out[11]: CCONS
```

```
In [12]: (defmethod head ((c cell))
          (cell-hd c))
```

```
Out[12]: #<STANDARD-METHOD CL-JUPYTER-USER::HEAD (CELL) {1007CB7043}>
```

```
In [13]: (defmethod tail ((c cell))
          (cell-tl c))
```

```
Out[13]: #<STANDARD-METHOD CL-JUPYTER-USER::TAIL (CELL) {1007DC0363}>
```

```
In [14]: (defmethod print-cell ((c cell) out)
          (format out "~A" (cell-hd c))
          (when (cell-tl c)
            (format out " ")
            (print-cell (cell-tl c) out))))
```

```
Out[14]: #<STANDARD-METHOD CL-JUPYTER-USER::PRINT-CELL (CELL T) {1007F0A1B3}>
```

```
In [15]: (defmethod print-object ((c cell) out)
          (format out "#<strict:."
                  (print-cell c out)
                  (format out ">")))
```

```
Out[15]: #<STANDARD-METHOD COMMON-LISP:PRINT-OBJECT (CELL T) {1008039C73}>
```

```
In [16]: (defparameter s1 (ccons 1 (ccons 2 (ccons 3 nil))))
```

```
Out[16]: S1
```

```
In [17]: s1
```

```
Out[17]: #<strict:1 2 3>
```

```
In [18]: (head s1)
```

```
Out[18]: 1
```

```
In [19]: (tail s1)
```

```
Out[19]: #<strict:2 3>
```

We will stop right now to reimplement a list-like datastructure in Lisp: there's something disturbing about the whole exercise!

1.3 Lazy sequences in Lisp

We are now in a position of updating our strict cells to make them lazier.

```
In [20]: (defstruct (lazy-cell (:include cell))
          (genfn nil))
```

```
Out[20]: LAZY-CELL
```

A lazy cell is a specific kind of cell with two possible states:

- either it is computed, in which case the `genfn` slot is `nil` and its `hd` and `tl` slots (included from the `cell` structure) are directly accessible
- or it is not yet computed in which case both the `hd` and `tl` slots are `nil`, and the `genfn` slot contains a generator function.

Initially a lazy cell must be in the not yet computed state, that's the whole point of laziness !

Before asking for a head or tail, it is necessary to switch to the computed state, which happens thanks to the following function.

```
In [21]: (defun compute-lazy-cell (c)
          (let ((val (funcall (lazy-cell-genfn c))))
            (setf (lazy-cell-hd c) (head val))
            (setf (lazy-cell-tl c) (tail val))
            (setf (lazy-cell-genfn c) nil)))
```

```
Out[21]: COMPUTE-LAZY-CELL
```

An important requirement is that the function above may only be called in the not yet computed state. The generator function is then called, which produce a value resulting from the cell computation. The state change is indicated by setting the `genfn` slot to `nil`.

The computation itself results in ... a sequence composed of a head and a tail. In many case we will have a cons pair (hence a kind of sequence) but it can be any kind of sequence distinct (of course distinct from the cell we are working on). For truly lazy sequences, the head should be a computed value, and the tail should be either empty or a (distinct) lazy cell. We will see how to fulfill these requirements below. For now, let's just remember that the extracted informations is stored in the two slots `hd` and `tl`.

Now we can define the head and tail methods in a straightforward way.

```
In [22]: (defmethod head ((c lazy-cell))
          (when (lazy-cell-genfn c)
              (compute-lazy-cell c))
          (lazy-cell-hd c))
```

```
Out[22]: #<STANDARD-METHOD CL-JUPYTER-USER::HEAD (LAZY-CELL) {1008961273}>
```

```
In [23]: (defmethod tail ((c lazy-cell))
          (when (lazy-cell-genfn c)
              (compute-lazy-cell c))
          (lazy-cell-tl c))
```

```
Out[23]: #<STANDARD-METHOD CL-JUPYTER-USER::TAIL (LAZY-CELL) {1008A3E813}>
```

In both cases the first step is to check whether we are in the computed state (i.e. `genfn` is `nil`) or not. If not, the `compute-lazy-cell` function is called, and then the value of the `hd` (resp. `tl`) slot is returned.

And that is about all we need for manipulating lazy sequences !

```
In [24]: (defmethod print-cell ((c lazy-cell) out)
          (if (lazy-cell-genfn c)
              (format out "?")
              (progn
                  (format out "~A" (lazy-cell-hd c))
                  (when (lazy-cell-tl c)
                      (format out " ")
                      (print-cell (lazy-cell-tl c) out))))))
```

```
Out[24]: #<STANDARD-METHOD CL-JUPYTER-USER::PRINT-CELL (LAZY-CELL T) {1008BA5213}>
```

```
In [25]: (defmethod print-object ((c lazy-cell) out)
          (format out "#<lazy:")
          (print-cell c out)
          (format out ">"))
```

```
Out[25]: #<STANDARD-METHOD COMMON-LISP:PRINT-OBJECT (LAZY-CELL T) {10054974E3}>
```

1.4 Constructing lazy sequences

In order to construct lazy sequences, we can define a lazy variant of `cons`.

```
In [26]: (defmacro lazy-cons (hd tl)
          '(make-lazy-cell :hd nil :tl nil :genfn (lambda () (cons ,hd ,tl))))
```

```
Out[26]: LAZY-CONS
```

Note that we have to define it as a macro, otherwise we would compute both the `hd` and `tl` parameters ! The `(lambda () ...)` used as a generator function is used to delay the computation of the head and tail. Hence, of course, strict sequences are required to implement the lazy ones (we could have used `cells` also but let's use lists everywhere they work).

1.4.1 Constructing a finie sequence

Any finite sequence should be constructible in a lazy way, so let's try.

```
In [27]: (defparameter ls1 (lazy-cons (+ 1 0) (lazy-cons (+ 1 1) (lazy-cons (+ 1 2) nil))))
```

```
Out[27]: LS1
```

```
In [28]: LS1
```

```
Out[28]: #<lazy:?>
```

The output above confirms that nothing has been computed yet. The question mark marks (pun intended!) the not yet computed part of the sequence (everything, for now). Of course, taking either the head or the tail of the lazy sequence will trigger some computation.

```
In [29]: (head ls1)
```

```
Out[29]: 1
```

The computed value of the head is 1 as expected, and we can see the effect on the sequence itself.

```
In [30]: ls1
```

```
Out[30]: #<lazy:1 ?>
```

The computed part of the sequence now comprises the first element. Of course, we can dig a little deeper.

```
In [31]: (head (tail ls1))
```

```
Out[31]: 2
```

```
In [32]: ls1
```

```
Out[32]: #<lazy:1 2 ?>
```

To make the sequence fully computed, we need to dig a little deeper.

```
In [33]: (tail (tail (tail ls1)))
```

```
Out[33]: NIL
```

```
In [34]: ls1
```

```
Out[34]: #<lazy:1 2 3>
```

Note that the list is still considered lazy but it is in computed state now so there's no computation to delay anymore.

One important take away of the previous example is that laziness can only be obtained through side effects, however very controlled (and overallly "safe") ones.

1.4.2 Constructing an “infinite” sequence

Once computed a sequence must of course be finite, but the lazy sequence abstraction allows to define intentionally infinite sequences. A good example is the sequence of natural numbers. Let’s try a definition using the `lazy-cons` macro.

```
In [35]: (defun nats (&optional (n 0))
          (lazy-cons n (nats (1+ n))))
```

```
Out[35]: NATS
```

```
In [36]: (defparameter ls2 (nats))
```

```
Out[36]: LS2
```

```
In [37]: LS2
```

```
Out[37]: #<lazy:?>
```

```
In [38]: (head ls2)
```

```
Out[38]: 0
```

```
In [39]: (head (tail ls2))
```

```
Out[39]: 1
```

```
In [40]: (head (tail (tail ls2)))
```

```
Out[40]: 2
```

```
In [41]: ls2
```

```
Out[41]: #<lazy:0 1 2 ?>
```

In this state, only the finite prefix 0 1 2 of the lazy sequence has been computed. To dig deeper, we can start to implement some utility functions.

The first one consists in taking a finite prefix of the sequence, materializing it (of course!) as a list.

```
In [42]: (defun take (n s)
          "Returns the list of the 'n' first elements of the sequence 's'."
          (loop repeat n
                for cell = s then (tail cell)
                when cell
                collect (head cell)))
```

```
Out[42]: TAKE
```

```
In [43]: (take 10 ls2)
```

```
Out[43]: (0 1 2 3 4 5 6 7 8 9)
```

```
In [44]: ls2
```

```
Out[44]: #<lazy:0 1 2 3 4 5 6 7 8 9 ?>
```

Note that since we keep a reference to the first cell (with head 0), the more we consume informations in the LS2 sequence the more we increase the amount of memory required to store the sequence cells. There is a Clojure proverb saying:

don't hold to your head!

The example above is a good illustration of what happens in this case.
One way to skip unwanted cells is to use the `drop` function defined below:

```
In [45]: (defun drop (n s)
          "Drops the first 'n' elements of the sequence 's'."
          (loop repeat n
                 for cell = s then (tail cell)
                 when (not cell) do (return cell)
                 finally (return cell)))
```

```
Out[45]: DROP
```

Let's try to drop a few natural numbers.

```
In [46]: (drop 10 (nats 0))
```

```
Out[46]: #<lazy:?>
```

The result is a lazy sequence without any computed cell... Of course, it's more useful if we take some information from the sequence:

```
In [47]: (head (drop 10 (nats 0)))
```

```
Out[47]: 9
```

```
In [48]: (take 10 (drop 10000 (nats 1)))
```

```
Out[48]: (10000 10001 10002 10003 10004 10005 10006 10007 10008 10009)
```

Of course, be careful not holding to your head as in the `take` case.

```
In [49]: (drop 100 ls2)
```

```
Out[49]: #<lazy:?>
```

```
In [50]: ls2
```

```
Out[50]: #<lazy:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 251
```


1.5 The three stooges: map, filter and reduce

Lazy sequences, as you can see, are not difficult to introduce in Common Lisp. And thanks to the `lazy-seq` macro it is rather easy to write generator functions such as `iterate` above. We will also see that sequences are an adequate tool to perform traversals of structured data.

But once the sequences have been created, what we want to do is to manipulate them. The three most important manipulations are: mappings, filterings and reductions. The corresponding functions are only a few lines away.

Because there is already a `map` in Common Lisp (although not that useful), let call our version `maps` for mapping sequence.

```
In [54]: (defun maps (f s)
          (when s
            (lazy-seq (cons (funcall f (head s)) (maps f (tail s))))))
```

Out[54]: MAPS

```
In [55]: (take 10 (maps (lambda (x) (* x 10)) (nats 1)))
```

Out[55]: (10 20 30 40 50 60 70 80 90 100)

Filtering is not much more difficult.

```
In [56]: (defun filters (pred s)
          (when s
            (if (funcall pred (head s))
                (lazy-seq (cons (head s) (filters pred (tail s))))
                (filters pred (tail s)))))
```

Out[56]: FILTERS

```
In [57]: (take 10 (filters #'evenp (nats 1)))
```

Out[57]: (2 4 6 8 10 12 14 16 18 20)

```
In [58]: (take 10 (filters #'oddp (nats 1)))
```

Out[58]: (1 3 5 7 9 11 13 15 17 19)

The reduction function is very powerful, although unlike the map and filter cases, it requires the whole sequence to be computed. Efficiency is prominent here, and we have to be careful not feeding the reduction with an infinite sequence.

```
In [59]: (defun reduces (f init s)
          (loop for cell = s then (tail cell)
                and acc = init then (funcall f acc (head cell))
                when (not cell) do (return acc)))
```

Out[59]: REDUCES

```
In [60]: (reduces #'+ 0 (take 10 (nats 1)))
```

Out[60]: 55

1.6 The final word ...

In a few lines of Lisp we were able to implement (a prototypal version of) the sequence abstraction (through the `head` and `tail` generic functions) and their lazy representation. The `lisp-lazy-seq` library contains much more facilities for your enjoyment but you now know how all this works.

That's all folks!

```
In [ ]:
```