|  |  |
| --- | --- |
| * **P01 (\*) Find the last box of a list**   Example:  (my-last '(a b c d))  (D)  **;; The nice, recursive solution:**  (defun my-last (list)  (cond  ((endp list)(error "Empty list"))  ((endp (rest list)) list)  (t (my-last (rest list)))  )  )  ;; **The efficient, iterative solution:**  (defun my-last (list)  (cond  ((endp list) (error "Empty list"))  (t (loop  :for result :on list  :until (endp (rest result))  :finally (return result)))))  **;; The smartass, Common Lisp solution:**  (defun my-last (list)  (last list)) | * **P02 (\*) Find the last but one box of a list.**   Example:  \* (my-but-last '(a b c d))  (C D)  **;; The nice, recursive solution:**  (defun my-but-last (list)  (cond  ((endp list) (error "Empty list"))  ((endp (rest list)) (error "List too short"))  ((endp (rest (rest list))) list)  (t (my-but-last (rest list)))))  **;; The efficient, iterative solution:**  (defun my-but-last (list)  (cond  ((endp list) (error "Empty list"))  ((endp (rest list)) (error "List too short"))  (t (loop  :for result :on list  :until (endp (rest (rest result)))  :finally (return result)))))  **;; The smartass, Common Lisp solution:**  (defun my-but-last (list)  (cond  ((endp list) (error "Empty list"))  ((endp (rest list)) (error "List too short"))  (t (last list 2)))) |
| * **P03 (\*) Find the K'th element of a list.**   Example:  (element-at '(a b c d e) 3)  C  **;; The nice, recursive solution:**  (defun element-at (list index)  (cond  ((not (integerp index)) (error "Non integer index ~A" index))  ((not (plusp index)) (error "Non strictly positive index ~A" index))  ((endp list) (error "List too short."))  ((= 1 index) (first list))  (t element-at (rest list) (1- index)))))  **;; The efficient, iterative solution:**  (defun element-at (list index)  (cond  ((not (integerp index)) (error "Non integer index ~A" index))  ((not (plusp index)) (error "Non strictly positive index ~A" index))  (t  (loop  :for result :on list  :while (plusp (decf index))  :finally (if (endp result)  (error "List too short.")  (return (first result)))))))  **;; The smartass, Common Lisp solution:**  (defun element-at (list index)  (cond  ((not (integerp index)) (error "Non integer index ~A" index))  ((not (plusp index)) (error "Non strictly positive index ~A" index))  (t (elt list (1- index))))) | * **P04 (\*) Find the number of elements of a list.**   **;; The nice, recursive solution:**  (defun number-of-elements (list)  (cond  ((endp list) 0)  (t (1+ (number-of-elements (rest list))))))  **;; A tail-recursive solution (with accumulator):**  (defun number-of-elements (list)  (labels ((count-element (list count)  (cond ((endp list) count)  (t (count-element (rest list) (1+ count))))))  (count-element list 0)))  **;; The efficient, iterative solution:**  (defun number-of-elements (list)  (loop  :for item :in list  :count 1))  **;; The smartass, Common Lisp solution:**  (defun number-of-elements (list)  (length list)) |
| * **P05 (\*) Reverse a list.**   **;; The naive recursive solution, O(nÂ²):**  (defun naive-recursive-reverse (list)  (if (endp list)  '()  (append (naive-recursive-reverse (rest list))  (cons (first list) '()))))  **;; A tail-recursive solution (with accumulator):**  (defun recursive-reverse (list)  (labels ((rev (list acc)  (if (endp list)  acc  (rev (rest list) (cons (first list) acc)))))  (rev list '())))  **;; An iterative solution:**  (defun iterative-reverse-do (list)  (let ((acc '()))  (do ((current list (rest current)))  ((endp current) acc)  (setf acc (cons (first current) acc)))))  (defun iterative-reverse-loop (list)  (loop  :with result = '()  :for item :in list  :do (push item result)  :finally (return result)))  **;; The smartass, Common Lisp solution:**  (defun my-reverse (list)  (reverse list))  ;; (mapcar (lambda (fun) (mapcar fun '(() (a) (a b) (a b c))))  ;; (list (function iterative-reverse-do)  ;; (function iterative-reverse-loop)  ;; (function naive-recursive-reverse)  ;; (function recursive-reverse)  ;; (function my-reverse)))  ;; --> ((NIL (A) (B A) (C B A))  ;; (NIL (A) (B A) (C B A))  ;; (NIL (A) (B A) (C B A))  ;; (NIL (A) (B A) (C B A))  ;; (NIL (A) (B A) (C B A)))  ;; A tail-recursive solution for the reversing of the list in place.  ;; Notice that cl:nreverse may be implemented without reusing the  ;; input list, cl:nreverse may just call cl:reverse.  (defun my-nreverse (list)  (labels ((list-reverse (reverse list)  (if (null list)  reverse  (let ((rest (cdr list)))  (setf (cdr list) reverse)  (list-reverse list rest)))))  (list-reverse nil list)))  ;; (let\* ((list (list 1 2 3 4))  ;; (reversed (my-nreverse list)))  ;; (values list reversed))  ;; --> (1)  ;; (4 3 2 1) | * **P06 (\*) Find out whether a list is a palindrome.**   A palindrome can be read forward or backward; e.g. (x a m a x).  ;; The simple solution, 2n steps in the worst case (a palindrome), n  ;; in the best case (not a palindrome), uses O(n) space:  (defun palindromep (list)  (equal list (reverse list)))  ;; A more complex solution, 3n/2 steps in the worst case, n steps in  ;; the best case, use O(n) space:  (defun reversed-spine (list)  "Returns a list containing each cons cells of the list in reversed order.  Example: (reversed-spine '(1 2 3)) --> '(#1=(3) #2=(2 . #1#) #1=(1 . #2#))  "  (loop  :with result = '()  :for cell :on list  :do (push cell result)  :finally (return result)))  (defun palindromep (list)  (loop  :for left :on list  :for right :in (reversed-spine list)  :until (or (eq left right) (eq (cdr left) right))  :unless (eql (car left) (car right)) :do (return nil)  :finally (return t)))  ;; A tail-recursive solution, O(n) steps, and uses O(n/2) space on the  ;; stack, and O(n/2) space:  (defun palindromep (list)  (labels ((palstep (slow fast reve)  (cond  ((endp fast) (equal slow reve))  ((endp (rest fast)) (equal (rest slow) reve))  (t (palstep (rest slow)  (rest (rest fast))  (cons (first slow) reve))))))  (palstep list list '())))  ;; A smartass solution, O(n) steps, O(n) space (but with a smaller  ;; constant than the other solutions).  (defun palindromep (list)  (loop  :with data = (coerce list 'vector)  :for i :from 0  :for j :from (1- (length data)) :by -1  :while (< i j)  :always (eql (aref data i) (aref data j))))  #-(and)  (mapcar (function palindromep) '((x a m a x)  (x a m m a x)  (x)  ()  (x a m b x)))  ;; --> (T T T T NIL) |
| * **P07 (\*\*) Flatten a nested list structure.**   Transform a list, possibly holding lists as elements into a `flat' list by replacing each list with its elements (recursively).  Example:  \*(my-flatten '(a (b (c d) e)))  (A B C D E)  Hint: Use the predefined functions list and append.  ;; Bad recursive solution, using list and append as hinted, O(nÂ²):  (defun my-flatten (list)  (cond  ((null list) list)  ((atom list) (list list))  ((list (first list)) append (my-flatten (first list))  (my-flatten (rest list))))  (t (append (list (first list))  (my-flatten (rest list))))))  ;; Good, recursive solution, using an accumulator, O(n):  (defun my-flatten (list)  (let ((result '()))  (labels ((collect (item) (push item result))  (collected-result ()(nreverse result))  (walk-list (sublist)  (dolist (item sublist)  (if (listp item)  (walk-list item)  (collect item)))))  (walk-list list)  (collected-result))))  ;; On results overflowing the cache size, we may want to avoid the nreverse:  (defun my-flatten (list)  (let\* ((result (cons 'header nil)) ; this cons avoids a test in collect  (tail result))  (labels ((collect (item) (setf (cdr tail) (cons item nil)  tail (cdr tail)))  (collected-result () (cdr result))  (walk-list (sublist)  (dolist (item sublist)  (if (listp item)  (walk-list item)  (collect item)))))  (walk-list list)  (collected-result))))  ;; An iterative solution:  (defun my-flatten (tree)  "  RETURN: A list containing all the elements of the `tree'."  (loop  :with result = nil  :with stack = nil  :while (or tree stack)  :do (cond  ((null tree)  (setq tree (pop stack)))  ((atom tree)  (push tree result)  (setq tree (pop stack)))  ((listp (car tree))  (push (cdr tree) stack)  (setq tree (car tree)))  (t  (push (car tree) result)  (setq tree (cdr tree))))  :finally (return (nreverse result)))) | * **P08 (\*\*) Eliminate consecutive duplicates of list elements.**   Example:  \* (compress '(a a a a b c c a a d e e e e))  (A B C A D E)  ;; Nice recursive solution:  (defun compress (list)  (labels ((compress-run (item list)  (cond ((null list) (list item))  ((eql item (first list)) (compress-run item (rest list)))  (t (cons item (compress-run (first list) (rest list)))))))  (cond  ((null list) list)  ((null (rest list)) list)  (t (compress-run (first list) (rest list))))))  ;; Iterative solution:  (defun compress (list)  (cond  ((null list) list)  ((null (rest list)) list)  (t (loop  :with result = '()  :with item = (first list)  :for other :in (rest list)  :do (unless (eql item other)  (push item result)  (setf item other))  :finally (push item result) (return (nreverse result))))))  ;; Smartass solution, using Common Lisp reduce:  (defun compress (list)  (reduce (lambda (item result)  (cond  ((endp result) (list item))  ((eql (first result) item) result)  (t (cons item result))))  list  :from-end t  :initial-value '()))  ;; Without :from-end, we need a reverse, and notice the order of the ;; arguments to the function:  (defun compress (list)  (nreverse (reduce (lambda (result item)  (cond  ((endp result) (list item))  ((eql (first result) item) result)  (t (cons item result))))  list  :initial-value '()))) |
| * **P09 (\*\*) Pack consecutive duplicates of list elements into sublists**   Example:  \* (pack '(a a a a b c c a a d e e e e))  ((A A A A) (B) (C C) (A A) (D) (E E E E))  ;; Ill choosen name...  (defun pack (list)  (group list))  ;; Nice recursive solution:  (defun group (list)  (labels ((group-run (element group list)  (cond  ((null list) (list (cons element group)))  ((eql element (first list)) (group-run element (cons element group) (rest list)))  (t (cons (cons element group) (group-run (first list) '() (rest list)))))))  (if (null list)  '()  (group-run (first list) '() (rest list)))))  ;; Smartass solution, using Common Lisp reduce:  (defun group (list)  (reduce (lambda (item result)  (cond  ((endp result) (list (list item)))  ((eql (first (first result)) item) (cons (cons item (first result))  rest result)))  (t (cons (list item) result))))  list  :from-end t  :initial-value '())) | * **P10 (\*) Run-length encoding of a list.**   Use the result of problem P09 to implement the so-called run-length encoding data compression method. Consecutive duplicates of elements are encoded as lists (N E) where N is the number of duplicates of the element E.  Example:  \* (encode '(a a a a b c c a a d e e e e))  ((4 A) (1 B) (2 C) (2 A) (1 D)(4 E))  ;; Nice recursive solution, using the same design pattern as in P09:  (defun encode (list)  (labels ((encode-run (element count list)  (cond  ((null list) (list (list count element)))  ((eql element (first list)) (encode-run element (1+ count) (rest list)))  (t (cons (list count element) (encode-run (first list) 1 (rest list)))))))  (if (null list)  '()  (encode-run (first list) 1 (rest list)))))  ;; Nice, functional solution, reusing functions from p09, uses O(n+r) space:  (defun encode (list)  (mapcar (lambda (group) (list (length group) (first group)))  (group list)))  ;; Smartass solution, using Common Lisp reduce:  (defun encode (list)  (reduce (lambda (item result)  (print (list item result))  (cond  ((endp result) (list (list 1 item)))  ((eql (second (first result)) item) (cons (list (1+ (first (first result))) item)  (rest result)))  (t (cons (list 1 item) result))))  list  :from-end t  :initial-value '()))  ;; Smartass solution, using Common Lisp reduce, and a closure; more efficient since we don't call cons twice at each step, but inelegant, since we have to add the last result as an afterthought:  (defun encode (list)  (when list  (let ((count 0)  (last-item nil))  (let ((tail-result (reduce (lambda (item result)  (cond  ((zerop count)  (setf count 1  last-item item)  result)  ((eql item last-item)  (incf count)  result)  (t  (prog1 (cons (list count last-item) result)  (setf count 1 last-item item)))))  list  :from-end t  :initial-value '())))  (cons (list count last-item) tail-result)))))  ;; Iterative solution, uses only O(r) space:  (defun encode (list)  (when list  (loop  :with count = 0  :with last-item = nil  :with result = '()  :for item :in list  :do (cond  ((zerop count) (setf count 1 last-item item))  ((eql item last-item) (incf count))  (t (push (list count last-item) result)  (setf count 1 last-item item)))  :finally (when (plusp count)  (push (list count last-item) result))  (return (nreverse result))))) |
| * **P11 (\*) Modified run-length encoding.**   Modify the result of problem P10 in such a way that if an element has no duplicates it is simply copied into  the result list. Only elements with duplicates are transferred as (N E) lists.    Example:  \* (encode-modified '(a a a a b c c a a d e e e e))  ((4 A) B (2 C) (2 A) D (4 E)) | ;; Simple functional solution:  (defun encode-modified (list)  (mapcar (lambda (item)  (if (= 1 (first item))  (second item)  item))  (encode list)))  ;; See p13 for an iterative solution. |
| * **P12 (\*\*) Decode a run-length encoded list.**   Given a run-length code list generated as specified in problem P11. Construct its uncompressed version. | ;; Nice functional solution:  (defun decode (encoded)  (mapcan (lambda (item)  (if (atom item)  (list item)  (make-list (first item) :initial-element (second item))))  encoded))  ;; Iterative solution:  (defun decode (encoded)  (loop  :with result = '()  :for item :in (reverse encoded)  :do (if (atom item)  (push item result)  (loop  :repeat (first item)  :do (push (second item) result)))  :finally (return result)))  #-(and)  (let ((list '(a a a a b c c a a d e e e e)))  (assert (equal list (decode (encode list))))  (assert (equal list (decode (encode-modified list))))  :success) |
| * **P13 (\*\*) Run-length encoding of a list (direct solution).**   Implement the so-called run-length encoding data compression method directly. I.e. don't explicitly create the  sublists containing the duplicates, as in problem P09, but only count them. As in problem P11, simplify the  result list by replacing the singleton lists (1 X) by X.    Example:  \* (encode-direct '(a a a a b c c a a d e e e e))  ((4 A) B (2 C) (2 A) D (4 E)) | ;; Iterative solution, uses only O(r) space:  (defun encode-modified (list)  (let ((result '())  (count 0)  (last-item nil))  (labels ((collect-result ()  (push (if (= 1 count)  last-item  (list count last-item))  result))  (new-item (item)  (setf count 1  last-item item))  (same-item ()  (incf count))  (return-result ()  (when (plusp count)  (collect-result))  (nreverse result)))  (dolist (item list (return-result))  (cond  ((zerop count) (new-item item))  ((eql item last-item) (same-item))  (t (collect-result)  (new-item item))))))) |
| * **P14 (\*) Duplicate the elements of a list.**   Example:  \* (dupli '(a b c c d))  (A A B B C C C C D D) | ;; Nice functional solution:  (defun dupli (list)  (mapcan (lambda (item) (list item item)) list))  ;; Iterative solution:  (defun dupli (list)  (loop  :for item :in list  :collect item  :collect item))  ;; Badass solution (using solution to p15):  (defun dupli (list)  (repli list 2)) |
| * **P15 (\*\*) Replicate the elements of a list a given number of times.**   Example:  \* (repli '(a b c) 3)  (A A A B B B C C C) | ;; Nice functional solution:  (defun repli (list count)  (mapcan (lambda (item) (make-list count :initial-element item)) list))  ;; Iterative solution:  (defun repli (list count)  (loop :for item :in list :nconc (loop :repeat count :collect item)))  ;; Assembler solution:  (defun repli (list count)  (let\* ((result (cons :head nil)) ; to avoid a test  (tail result)  i item)  (tagbody  (go :begin)  :next  (setq list (cdr list))  :begin  (if (endp list)  (go :done))  (setq item (first list))  (setq i count)  :add-them  (if (zerop i)  (go :next))  (rplacd tail (cons item nil))  (setq tail (cdr tail))  (setq i (1- i))  (go :add-them)  :done)  (cdr result)))  ;; Late student recursive solution:  (defun repli (list count)  (labels ((repli-one (item count)  (if (zerop count)  '()  (cons item (repli-one item (1- count))))))  (if (endp list)  '()  (append (repli-one (first list) count)  (repli (rest list) count))))) |
| * **P16 (\*\*) Drop every N'th element from a list.**   Example:  \* (drop '(a b c d e f g h i k) 3) 🡪 (A B D E G H K)  ;; For some of these, I'm wanting to write code that loops through a list  ;; in parallel with updating some other variable, so I end up with functions like this:  (defun drop (list n)  "returns result of dropping every nth element of list"  (do ((l list (cdr l))  (i 1 (1+ i))  (result nil))  ((null l) (nreverse result))  (unless (= (mod i n) 0)  (push (car l) result))))  ;; I know there's also an approach using LOOP that I think I like better:  (defun drop (list n)  "returns result of dropping every nth element of list"  (loop  for elem in list  for i from 1  unless (= (mod i n) 0)  collect elem))  ;; Another choice that doesn't use MOD:  (defun drop (list n)  (let ((result nil))  (loop while list  do (loop repeat (1- n)  while list  do (push (pop list) result))  (pop list))  (nreverse result))) | ;; Here's a "cute" attempt:  (defun drop (sequence n &aux (i 0))  "Remove every Nth element of SEQUENCE."  (remove-if (lambda (item) (divisible-by-p (incf i) n)) sequence))  (defun divisible-by-p (m n)  "Return true if M is divisible by N, and false otherwise."  (zerop (mod m n)))  ;; Oh, we can get that with the loop version;  (defun drop (list n)  (loop  :for elt :in list  :for i :from 1  :for die-elt-die! = (zerop (mod i n))  :unless die-elt-die!  :collect elt))  ;; Here is my suggestion:  (defun drop (list n &aux (m (1- n)))  (loop  :for a = list :then (cdr b)  :for b = (nthcdr m a)  :if b :nconc (ldiff a b)  :else :nconc a :and :do (loop-finish))) |
| * **P17 (\*) Split a list into two parts; the length of the first part is given.**   Do not use any predefined predicates.    Example:  \* (split '(a b c d e f g h i k) 3)  ( (A B C) (D E F G H I K))  ;; Recursive function:  (defun split (list count)  (if (plusp count)  (destructuring-bind (left right) (split (rest list) (1- count))  (list (cons (first list) left) right))  (list '() list)))  ;; Iterative function:  (defun split (list count)  (loop  :for rest :on list  :repeat count  :collect (first rest) :into left  :finally (return (list left rest))))  ;; Smartass solution, using Common Lisp.  (defun split (list count)  (list (subseq list 0 count)  (subseq list count)))  ;; Smartass solution, using Common Lisp, and sharing the tail:  (defun split (list count)  (list (subseq list 0 count)  (nthcdr count list))) | * **P18 (\*\*) Extract a slice from a list.**   Given two indices, I and K, the slice is the list containing the elements between the I'th and K'th element of  the original list (both limits included). Start counting the elements with 1.    Example:  \* (slice '(a b c d e f g h i k) 3 7)  (C D E F G)  ;; Recursive solution:  (defun slice (list start last)  (cond ((not (<= 1 start last)) (error "Invalid indices."))  ((= 1 start) (first (split list last)))  (t (slice (rest list) (1- start) (1- last)))))  ;; Iterative solution:  (defun slice (list start last)  (loop  :repeat (1- start)  :for rest :on list  :finally (return (loop  :repeat (- last start -1)  :for item :in rest  :collect item))))  ;; Functional solution, using Common Lisp:  (defun slice (list start last)  (let ((rest (nthcdr (1- start) list)))  (ldiff rest (nthcdr (- last start -1) rest))))  ;; Smartass solution, using Common Lisp:  (defun slice (list start last)  (subseq list (1- start) last)) |
| * **P19 (\*\*) Rotate a list N places to the left.**   Examples:  \* (rotate '(a b c d e f g h) 3)  (D E F G H A B C)    \* (rotate '(a b c d e f g h) -2)  (G H A B C D E F)    Hint: Use the predefined functions length and append, as well as the result of problem P17.  ;; Solution using the function of p17:  (defun rotate (list count)  (if (minusp count)  (rotate list (+ (length list) count))  (destructuring-bind (left right) (split list count)  ;; Notice: depending on the implementation of split, right  ;; might share structure with list, therefore we cannot use  ;; nconc indiscriminately on it!  (append right left))))  ;; Smartass solution, using Common Lisp:  (defun rotate (list count)  (if (minusp count)  (rotate list (+ (length list) count))  (nconc (subseq list count) (subseq list 0 count)))) |  |
| * **P20 (\*) Remove the K'th element from a list.**   Example:  \* (remove-at '(a b c d) 2)  (A C D)  ;; Recursive solution, sharing the tail:  (defun remove-at (list index)  (cond  ((< index 1) (error "Invalid index"))  ((= index 1) (rest list))  (t (cons (first list) (remove-at (rest list) (1- index))))))  ;; Iterative solution, sharing the tail:  (defun remove-at (list index)  (cond  ((< index 1) (error "Invalid index"))  (t (loop  :for rest :on list  :repeat (1- index)  :collect (first rest) :into left  :finally (return (nconc left (rest rest)))))))  ;; Iterative solution, copying the whole result:  (defun remove-at (list index)  (cond  ((< index 1) (error "Invalid index"))  (t (loop  :for item :in list  :for i :from 1  :unless (= i index) :collect item))))  ;; Functional solution, sharing the tail:  (defun remove-at (list index)  (append (subseq list 0 (1- index)) (nthcdr index list)))  ;; Functional solution, copying the whole result:  (defun remove-at (list index)  (append (subseq list 0 (1- index)) (copy-list (nthcdr index list))))  ;; For both these functional solutions, we could replace append by  ;; nconc, since subseq returns a new list, so we could just modify the  ;; last cdr to attach it to the rest.  ;; Smartass solution, using Common Lisp:  (defun remove-at (list index)  (remove-if (constantly t) list :start (1- index) :end index)) | * **P21 (\*) Insert an element at a given position into a list.**   Example:  \* (insert-at 'alfa '(a b c d) 2)  (A ALFA B C D)  ;; Recursive solution, sharing the tail:  (defun insert-at (item list index)  (cond  ((< index 1) (error "Index too small ~A" index))  ((= index 1) (cons item list))  ((endp list) (error "Index too big"))  (t (cons (first list) (insert-at item (rest list) (1- index))))))  ;; Functional solution, using the split function of p17.  (defun insert-at (item list index)  (destructuring-bind (left right) (split list (1- index))  (append left (list item) right)))  ;; Smartass solution, using Common Lisp, sharing the tail:  (defun insert-at (item list index)  (append (subseq list 0 (1- index))  (list item)  (nthcdr (1- index) list)))  ;; Smartass solution, using Common Lisp, copying the whole result:  (defun insert-at (item list index)  (concatenate 'list  (subseq list 0 (1- index))  (list item)  (nthcdr (1- index) list)))  ;; Smartass solution, using Common Lisp, modifying the original list!  (defun insert-at (item list index)  (cond  ((< index 1) (error "Index too small ~A" index))  ((= index 1) (cons item list))  (t (push item (cdr (nthcdr (- index 2) list)))  list)))  ;; (let ((list (list 'a 'b 'c 'd)))  ;; (values list  ;; (insert-at 'alfa list 2)))  ;; --> (A ALFA B C D)  ;; (A ALFA B C D)  ;;  ;; (let ((list (list 'a 'b 'c 'd)))  ;; (values list  ;; (insert-at 'alfa list 1)))  ;; --> (A B C D)  ;; (ALFA A B C D) |
| * **P22 (\*) Create a list containing all integers within a given range.**   If first argument is smaller than second, produce a list in decreasing order.  Example:  \* (range 4 9)  (4 5 6 7 8 9) | ;; Recursive solution:  (defun range (start last)  (if (<= start last)  (cons start (range (1+ start) last))  '()))  ;; Recursive solution, with an accumulator:  (defun range (start last)  (labels ((acc (current result)  (if (<= current last)  (acc (1+ current) (cons current result))  (nreverse result))))  (acc start '())))  ;; Smartass solution, using Common Lisp:  (defun range (start last)  (loop :for i :from start :to last :collect i)) |
| * **P23 (\*\*) Extract a given number of randomly selected elements from a list.**   The selected items shall be returned in a list.  Example:  \* (rnd-select '(a b c d e f g h) 3)  (E D A)    Hint: Use the built-in random number generator and the result of problem P20.  ;; Recursive solution using the function remove-at of P20:  (defun rnd-select (list count)  (if (zerop count)  '()  (let ((i (random (length list)))) ; lenght and elt are O(n) ==> rnd-select is O(nÂ²).  (cons (elt list i) (rnd-select (remove-at list (1+ i)) (1- count)))))) | ;; Iterative solution using the function remove-at of P20:  (defun rnd-select (list count)  (loop  :repeat count  :for len :from (length list) :by -1  :for i = (random len)  :collect (elt list i) ; elt is O(n) ==> rnd-select is O(nÂ²).  :do (setf list (remove-at list (1+ i)))))  ;; This solution extract the items in the same order than in list, by  ;; precomputing a random bitmap. The result is O(n):  (defun rnd-select (list count)  (let ((len (length list)))  (cond  ((zerop count) '()) ; none selected.  ((= count len) list) ; all selected.  ((< 0 count len)  (let ((bits (make-array len :element-type 'bit :initial-element 0)))  (loop  :while (plusp count)  :for i = (random len)  :do (when (zerop (aref bits i))  (setf (aref bits i) 1)  (decf count)))  (loop  :for item :in list  :for indicator :across bits  :when (plusp indicator)  :collect item)))  (t (error "Invalid count, must be between 0 and ~A" len)))))  ;; This other solution remembers the indices selected, and avoids  ;; reusing them. This gives a random order, but the time complexity  ;; is not deterministically bound, only statistically bound (assuming  ;; a correct pseudo-random generator):  (defun rnd-select (list count)  (let ((len (length list)))  (cond  ((zerop count) '()) ; none selected.  ((<= 1 count len)  (loop  :with indices = '()  :with result = '()  :while (plusp count)  :for i = (random len)  :unless (member i indices)  :do (progn  (push i indices)  (push (elt list i) result) ; elt is O(n) ==> rnd-select is O(nÂ²).  (decf count))  :finally (return result)))  (t (error "Invalid count, must be between 0 and ~A" len))))) |
| * **P24 (\*) Lotto: Draw N different random numbers from the set 1..M.**   The selected numbers shall be returned in a list.  Example:  \* (lotto-select 6 49)  (23 1 17 33 21 37) | ;; Note: for this problem any solution of p23 is valid: the order of  ;; lotto draws is irrelevant, therefore we can just chose combinations  ;; in order.  (defun lotto-select (selection set-size)  (rnd-select (range 1 set-size) selection)) |
| * **P25 (\*) Generate a random permutation of the elements of a list.**   Example:  \* (rnd-permu '(a b c d e f))  (B A D C E F)  Hint: Use the solution of problem P23.  ;; Assuming an random order solution for rnd-select:  (defun rnd-permu (list)  (rnd-select list (length list)))  ;; Generating the permutation from scratch:  (defun make-circular (list)  (setf (cdr (last list)) list))  (defun rnd-permu (list)  (when list  (loop  :with len = (length list)  :with choices = (make-circular (copy-list list))  :collect (pop (cdr (nthcdr (random len) choices)))  :while (plusp (decf len))))) | * **Generate the combinations of K distinct objects chosen from the N elements of a list**   Generate the combinations of K distinct objects chosen from the N elements of a list In how many ways can a committee of 3 be chosen from a group of 12 people? We all know that there are  C(12,3) = 220 possibilities (C(N,K) denotes the well-known binomial coefficients). For pure mathematicians, this result may be great. But we want to really generate all the possibilities in a list.  Example:  \* (combination 3 '(a b c d e f))  ((A B C) (A B D) (A B E) ... )  ;; A simple recursive solution:  (defun combinations (count list)  (cond  ((zerop count) '(())) ; one combination of zero element.  ((endp list) '()) ; no combination from noe element.  (t (nconc (mapcar (let ((item (first list))) (lambda (combi) (cons item combi)))  (combinations (1- count) (rest list)))  (combinations count (rest list))))))  ;; (length (combinations 3 '(a b c d e f g h i j k l)))  ;; --> 220 |
| * **P27 (\*\*) Group the elements of a set into disjoint subsets.**   a) In how many ways can a group of 9 people work in 3 disjoint  subgroups of 2, 3 and 4 persons? Write a function that generates  all the possibilities and returns them in a list.    Example: \* (group3 '(aldo beat carla david evi flip gary hugo  ida)) ( ( (ALDO BEAT) (CARLA DAVID EVI) (FLIP GARY HUGO IDA) )  ... )    b) Generalize the above predicate in a way that we can specify a  list of group sizes and the predicate will return a list of  groups.    Example: \* (group '(aldo beat carla david evi flip gary hugo ida)  '(2 2 5)) ( ( (ALDO BEAT) (CARLA DAVID) (EVI FLIP GARY HUGO IDA) )  ... )    Note that we do not want permutations of the group members;  i.e. ((ALDO BEAT) ...) is the same solution as ((BEAT ALDO)  ...). However, we make a difference between ((ALDO BEAT) (CARLA  DAVID) ...) and ((CARLA DAVID) (ALDO BEAT) ...).    You may find more about this combinatorial problem in a good book  on discrete mathematics under the term \"multinomial  coefficients\". | (defun group3 (set)  (group set '(2 3 4)))  (defun group (set sizes)  (cond  ((endp sizes)  (error "Not enough sizes given."))  ((endp (rest sizes))  (if (= (first sizes) (length set))  (list (list set))  (error "Cardinal mismatch |set| = ~A ; required ~A" (length set) (first sizes))))  (t  (mapcan (lambda (combi)  (mapcar (lambda (group) (cons combi group))  (group (set-difference set combi) (rest sizes))))  (combinations (first sizes) set)))))  ;; (map nil 'print (group3 '(aldo beat carla david evi flip gary hugo ida)))  ;; (map nil 'print (group '(aldo beat carla david evi flip gary hugo ida) '(2 2 5))) |

* **P28 (\*\*) Sorting a list of lists according to length of sublists**

a) We suppose that a list contains elements that are lists themselves. The objective is to sort the elements of

this list according to their length. E.g. short lists first, longer lists later, or vice versa.

Example:

\* (lsort '((a b c) (d e) (f g h) (d e) (i j k l) (m n) (o)))

((O) (D E) (D E) (M N) (A B C) (F G H) (I J K L))

b) Again, we suppose that a list contains elements that are lists themselves. But this time the objective is to

sort the elements of this list according to their length frequency; i.e., in the default, where sorting is done

ascendingly, lists with rare lengths are placed first, others with a more frequent length come later.

Example:

\* (lfsort '((a b c) (d e) (f g h) (d e) (i j k l) (m n) (o)))

((i j k l) (o) (a b c) (f g h) (d e) (d e) (m n))

Note that in the above example, the first two lists in the result have length 4 and 1, both lengths appear just

once. The third and forth list have length 3 which appears twice (there are two list of this length). And

finally, the last three lists have length 2. This is the most frequent length.

"

;;; a)

;; Simple direct solution, using Common Lisp. This solution may have

;; a bad time complexity because sort may call the key several times.

(defun lsort (llist)

(sort (copy-list llist) (function <) :key (function length)))

;; Simple direct solution, using Common Lisp, taking care of calling

;; length only once. Moreover, we take care of using a vector as

;; temporary structure, so that we use less memory (and sorting

;; vectors may be faster):

(defun lsort (llist)

(map 'list (function cdr)

(sort (map 'vector (lambda (list) (cons (length list) list)) llist)

(function <) :key (function car))))

;; (lsort '((a b c) (d e) (f g h) (d e) (i j k l) (m n) (o)))

;; --> ((O) (D E) (D E) (M N) (A B C) (F G H) (I J K L))

;;; b)

;; Simple direct solution, using Common Lisp:

(defun make-histogram (sequence &key (key (function identity)))

(let ((table (make-hash-table)))

(map nil (lambda (item) (incf (gethash (funcall key item) table 0))) sequence)

table))

(defun lfsort (llist)

(let\* ((data (map 'vector (lambda (list) (cons (length list) list)) llist))

(histo (make-histogram data :key (function car))))

(map 'list (function cdr)

(sort data (function <) :key (lambda (item) (gethash (car item) histo))))))

;; (lfsort '((a b c) (d e) (f g h) (d e) (i j k l) (m n) (o)))

;; --> ((I J K L) (O) (A B C) (F G H) (D E) (D E) (M N))

**Arithmetic**

|  |  |
| --- | --- |
| * **P31 (\*\*) Determine whether a given integer number is prime.**   Example:  \* (is-prime 7)  T  (defun is-prime (n)  (primep n))  (defun primep (n)  (cond  ((minusp n) (primep (- n)))  ((= 1 n) nil)  ((member n '(2 3 5 7)) t) ; primes up to 3Â²=9  ((evenp n) nil)  (t  (loop  :with root = (isqrt n)  :with divisors = (loop :for i :from 3 :to root :by 2 :collect i)  :for d = (pop divisors)  :if (zerop (mod n d))  :do (return nil)  :else :do (setf divisors (delete-if (lambda (x) (zerop (mod x d))) divisors))  :while divisors  :finally (return t))))) | * **P32 (\*\*) Determine the greatest common divisor of two positive integer numbers.**   Use Euclid's algorithm.  Example:  \* (gcd 36 63)  9  ;; Smartass solution using Common Lisp:  (defun my-gcd (p q)  (gcd p q))  ;; Euclide's algorithm:  (defun my-gcd (p q)  (cond  ((= p q) p)  ((< p q) (my-gcd p (- q p)))  (t (my-gcd q (- p q)))))  ;; (loop  ;; :for p :from 1 :below 100  ;; :do (loop :for q :from 1 :below 100  ;; :do (assert (= (my-gcd p q) (gcd p q)) (p q)))  ;; :finally (return :success))  ;; --> :SUCCESS |
| * **P33 (\*) Determine whether two positive integer numbers are coprime.**   Two numbers are coprime if their greatest common divisor equals 1.  Example:  \* (coprime 35 64)  T  (defun coprime (a b)  (= 1 (gcd a b))) | * **P34 (\*\*) Calculate Euler's totient function phi(m).**   Euler's so-called totient function phi(m) is defined as the number  of positive integers r (1 <= r < m) that are coprime to m.    Example: m = 10: r = 1,3,7,9; thus phi(m) = 4. Note the special case: phi(1) = 1.    \* (totient-phi 10)  4    Find out what the value of phi(m) is if m is a prime  number. Euler's totient function plays an important role in one of  the most widely used public key cryptography methods (RSA). In  this exercise you should use the most primitive method to  calculate this function (there are smarter ways that we shall  discuss later).  (defun totient-phi (m)  (loop  :for r :from 1 :below m  :when (coprime r m) :count 1)) |

|  |  |
| --- | --- |
| * **P35 (\*\*) Determine the prime factors of a given positive integer.**   Construct a flat list containing the prime factors in ascending order.  Example:  \* (prime-factors 315)  (3 3 5 7)  ;; Solution using a library function...  (defun compute-primes-to (n)  "  DO: Compute an Eratostene sieve to find all prime numbers up to N.  RETURN: A sorted array of all prime numbers up to n.  "  (cond  ((< n 2) #())  ((= n 2) #(2))  ((= n 3) #(2 3))  (t  (let (bits-max bits bit (prime-count 2) (cur-prime 3) primes)  ;; (SETF N (+ N (IF (ODDP N) 3 2)))  (setf n (- n (if (oddp n) 3 2))) ; base of bits array is 3.  (setf bits-max (/ n 2))  ;; set the bitset to full bits;  (setf bits (make-array (list bits-max) :initial-element 1 :element-type 'bit))  (loop :while (< cur-prime n) :do  (setf bit (+ cur-prime (/ (- cur-prime 3) 2)))  (loop :while (< bit bits-max) :do  (setf (aref bits bit) 0)  (incf bit cur-prime))  (setf bit (1+ (/ (- cur-prime 3) 2)))  ;; search next prime  (setf bit (position 1 bits :start bit))  (if bit  (setf cur-prime (+ bit bit 3)  prime-count (1+ prime-count))  (setf cur-prime n)))  ;; gather the primes.  (setf primes (make-array (list prime-count) :element-type 'integer))  (let ((curnum 0))  (setf (aref primes curnum) 2)  (incf curnum)  (setf (aref primes curnum) 3)  (incf curnum)  (setf cur-prime 3)  (setf bit 0)  (setf bit (position 1 bits :start (1+ bit)))  (loop :while bit :do  (setf cur-prime (+ bit bit 3))  (setf (aref primes curnum) cur-prime)  (incf curnum)  (setf bit (position 1 bits :start (1+ bit)))))  primes))))  (defun factorize (n &optional (primes nil))  "  N: an INTEGER  PRIMES: a VECTOR of prime factors sorted in increasing order. | RETURN: a SEXP of the form: (\* uncomensurate-factor  [ prime | (EXPT prime exponent) ]... [ -1 ] )  "  (let ((primes (or primes (compute-primes-to (1+ (isqrt n)))))  (factors '())  (prime-idx 0) )  (unless (integerp n)  (error "I can only decompose integer values."))  (when (< n 0)  (push -1 factors)  (setf n (- n)))  (loop :while (and (< prime-idx (length primes)) (< 1 n)) :do  (let ((prime (elt primes prime-idx))  (expo 0))  (multiple-value-bind (q r) (truncate n prime)  (loop :while (zerop r) :do  (incf expo)  (setf n q)  (multiple-value-setq (q r) (truncate n prime))))  (when (plusp expo)  (push (if (= 1 expo)  prime  (list 'expt prime expo))  factors)))  (incf prime-idx))  (when (< 1 n)  (push n factors))  (cons '\* factors)))  (defun prime-factors (n)  (mapcan (lambda (factor)  (cond  ((and (listp factor) (eql 'expt (first factor)))  (destructuring-bind (expt prime expo) factor  (declare (ignore expt))  (make-list expo :initial-element prime)))  (t (list factor))))  (nreverse (cdr (factorize n)))))  ;; (prime-factors 315)  ;; --> (3 3 5 7) |

* **P36 (\*\*) Determine the prime factors of a given positive integer (2).**

Construct a list containing the prime factors and their multiplicity.

Example:

\* (prime-factors-mult 315)

((3 2) (5 1) (7 1))

Hint: The problem is similar to problem P13.

;; Solution, using the same library functions as in p35:

(defun prime-factors-mult (n)

(mapcar (lambda (factor)

(cond

((and (listp factor) (eql 'expt (first factor)))

(destructuring-bind (expt prime expo) factor

(declare (ignore expt))

(list prime expo)))

(t

(list factor 1))))

(nreverse (cdr (factorize n)))))

;; (prime-factors-mult 315)

;; --> ((3 2) (5 1) (7 1))

* **P37 (\*\*) Calculate Euler's totient function phi(m) (improved).**

See problem P34 for the definition of Euler's totient function. If

the list of the prime factors of a number m is known in the form

of problem P36 then the function phi(m) can be efficiently

calculated as follows: Let ((p1 m1) (p2 m2) (p3 m3) ...) be the

list of prime factors (and their multiplicities) of a given number

m. Then phi(m) can be calculated with the following formula:

phi(m) = (p1 - 1) \* p1 \*\* (m1 - 1) + (p2 - 1) \* p2 \*\* (m2 - 1) + (p3 - 1) \* p3 \*\* (m3 - 1) + ...

Note that a \*\* b stands for the b'th power of a.

;;; https://secure.wikimedia.org/wikipedia/en/wiki/Euler%27s\_totient\_function#Computing\_Euler.27s\_function

(defun phi (m)

;; (p1 - 1) \* p1 \*\* (m1 - 1)

;; + (p2 - 1) \* p2 \*\* (m2 - 1)

;; + (p3 - 1) \* p3 \*\* (m3 - 1)

;; + ...

(reduce (function \*)

(mapcar (lambda (item)

(destructuring-bind (p-i m-i) item

(\* (1- p-i) (expt p-i (1- m-i)))))

(prime-factors-mult m))))

;; There's something wrong, phi is not equal to totient-phi, so there

;; must be some error in the problem statements. We need to check them.

;; (loop

;; :for n :from 2 :to 100

;; :do (unless (= (totient-phi n) (phi n))

;; (format t "(totient-phi ~A) = ~A /= ~A = (phi ~A)~%"

;; n (totient-phi n) (phi n) n)))

* **P38 (\*) Compare the two methods of calculating Euler's totient function.**

Use the solutions of problems P34 and P37 to compare the algorithms. Take the number of logical inferences as a measure for

efficiency. Try to calculate phi(10090) as an example.

;; TO BE DONE.

* **P39 (\*) A list of prime numbers.**

Given a range of integers by its lower and upper limit, construct a list of all prime numbers in that range.

;; Using library function from problem p35.

;; Note that to compute the primes greater than X, you need to know the primes less or equal to X.

(defun primes-in-range (lower upper)

(remove-if (lambda (x) (< x lower)) (compute-primes-to (1+ upper))))

* **P40 (\*\*) Goldbach's conjecture.**

Goldbach's conjecture says that every positive even number greater than 2 is the sum of two prime numbers. Example: 28 = 5 + 23. It is one of the most famous facts in number theory that has not been proved to be correct in the general case. It has been umerically confirmed up to very large numbers (much larger than we can go with our Prolog system). Write a predicate to find the two prime numbers that sum up to a given even integer.

Example:

\* (goldbach 28)

(5 23)

(defun goldbach (n)

(check-type n (integer 4))

(assert (evenp n))

(let ((primes (compute-primes-to n))

(bits (make-array (1+ n) :element-type 'bit :initial-element 0)))

(loop

:for p :across primes :do (setf (aref bits p) 1))

(loop

:for p :across primes

:when (plusp (aref bits (- n p)))

:do (return (list p (- n p)))

:finally (return '()))))

;; (loop :for i :from 4 :to 100 :by 2 :collect (goldbach i))

;; ((2 2) (3 3) (3 5) (3 7) (5 7) (3 11) (3 13) (5 13) (3 17) (3 19)

;; (5 19) (3 23) (5 23) (7 23) (3 29) (3 31) (5 31) (7 31) (3 37) (5

;; 37) (3 41) (3 43) (5 43) (3 47) (5 47) (7 47) (3 53) (5 53) (7 53)

;; (3 59) (3 61) (5 61) (7 61) (3 67) (5 67) (3 71) (3 73) (5 73) (7

;; 73) (3 79) (5 79) (3 83) (5 83) (7 83) (3 89) (5 89) (7 89) (19 79)

;; (3 97))

|  |  |
| --- | --- |
| * **P41 (\*\*) A list of Goldbach compositions.**   Given a range of integers by its lower and upper limit, print a list of all even numbers and their Goldbach composition.  Example:  \* (goldbach-list 9 20)  10 = 3 + 7  12 = 5 + 7  14 = 3 + 11  16 = 3 + 13  18 = 5 + 13  20 = 3 + 17    In most cases, if an even number is written as the sum of two prime numbers, one of them is very small. Very rarely, the primes  are both bigger than say 50. Try to find out how many such cases there are in the range 2..3000.    Example (for a print limit of 50):  \* (goldbach-list 1 2000 50)  992 = 73 + 919  1382 = 61 + 1321  1856 = 67 + 1789  1928 = 61 + 1867 | (defun goldbach-list (lower upper &optional limit)  (if limit  (loop  :for n :from (max 4 (\* 2 (ceiling lower 2))) :to (\* 2 (truncate upper 2)) :by 2  :do (destructuring-bind (p q) (goldbach n)  (when (and (<= limit p) (<= limit q))  (format t "~A = ~A + ~A~%" n p q))))  (loop  :for n :from (max 4 (\* 2 (ceiling lower 2))) :to (\* 2 (truncate upper 2)) :by 2  :do (destructuring-bind (p q) (goldbach n)  (format t "~A = ~A + ~A~%" n p q)))))  ;; (goldbach-list 1 2000 50)  ;; 992 = 73 + 919  ;; 1382 = 61 + 1321  ;; 1856 = 67 + 1789  ;; 1928 = 61 + 1867  ;; --> NIL |

**Logic and Codes**

|  |  |
| --- | --- |
| * **P46 (\*\*) Truth tables for logical expressions.**   Define predicates and/2, or/2, nand/2, nor/2, xor/2, impl/2 and  equ/2 (for logical equivalence) which succeed or fail according to  the result of their respective operations; e.g. and(A,B) will  succeed, if and only if both A and B succeed. Note that A and B  can be Prolog goals (not only the constants true and fail).    A logical expression in two variables can then be written in  prefix notation, as in the following example:  and(or(A,B),nand(A,B)).    Now, write a predicate table/3 which prints the truth table of a  given logical expression in two variables.    Example:  \* table(A,B,and(A,or(A,B))).  true true true  true fail true  fail true fail  fail fail fail  ;; The first question is somewhat meaningless in Lisp, since we have  ;; values, not goals, with applicative evaluation, where the  ;; subexpressions are evaluated before calling a function.  (defun .and. (a b) (cond (a b) (t nil)))  (defun .or. (a b) (cond (a) (b) (t nil)))  (defun .nand. (a b) (not (.and. a b)))  (defun .nor. (a b) (not (.or. a b)))  (defun .xor. (a b) (cond (a (not b)) (b t) (t nil)))  (defun .impl. (a b) (.or. (not a) b))  (defun .equ. (a b) (cond (a b) (t (not b))))  ;; On the other hand, AND and OR are defined as macro in lisp, to  ;; implement short-cut evaluation. Notice that in the case of xor and  ;; equ, it doesn't matter, we need to evaluate both in any case.  (defmacro =and= (a b) `(cond (,a ,b) (t nil)))  (defmacro =or= (a b) `(cond (,a) (,b) (t nil)))  (defmacro =nand= (a b) `(=or= (not ,a) (not ,b)))  (defmacro =nor= (a b) `(=and= (not ,a) (not ,b)))  (defmacro =xor= (a b) `(cond (,a (not ,b)) (,b t) (t nil)))  (defmacro =impl= (a b) `(=or= (not ,a) ,b))  (defmacro =equ= (a b) `(cond (,a ,b) (t (not ,b))))  ;; For the second question, we can write a function taking two  ;; variable names (two symbols) and a symbolic expression (sexp), and  ;; print a truth table. Here is a solution using EVAL, which is not  ;; safe, since expr could be bound to any lisp form, even something  ;; "dangerous" such as a delete-file form.  (defun table (a b expr)  (loop :for p :in '(t nil) :do  (loop :for q :in '(t nil) :do  (let ((v (eval `(let ((,a ,p) (,b ,q)) ,expr))))  (format t "~:[fail~;true~] ~:[fail~;true~] ~:[fail~;true~]~%" p q v))))) | ;; Here is a solution where we implement our own evaluation function:  (defmacro nand (a b) `(or (not ,a) (not ,b)))  (defmacro nor (a b) `(and (not ,a) (not ,b)))  (defun xor (a b) (cond (a (not b)) (b t) (t nil)))  (defmacro impl (a b) `(or (not ,a) ,b))  (defun equ (a b) (cond (a b) (t (not b))))  (defun evaluate-boolean (expression bindings)  "Evaluates the boolean expression. Returns t or NIL.  expression := variable | constant | '(' operator expression expression ')' | '(' 'not' expression ')' .  constant := 'true' | 'fail' .  variable := symbol .  operator := 'and' | 'or' | 'nand' | 'nor' | 'xor' | 'impl' | 'equ' .  bindings is a list of pairs (variable . constant).  (cond  ((eq expression 'true) t)  ((eq expression 'fail) nil)  ((symbolp expression) (let ((pair (assoc expression bindings)))  (if pair  (progn  (assert (member (cdr pair) '(true fail)))  (eql 'true (cdr pair)))  (error "No variable named ~A in the bindings." expression))))  ((atom expression) (error "Invalid atom ~A in the expression." expression))  (t (case (length expression)  ((2) (destructuring-bind (op subexpression) expression  (case op  ((not) (not (evaluate-boolean subexpression bindings)))  (otherwise (error "Invalid operator ~A" op)))))  ((3) (destructuring-bind (op left right) expression  (case op  ((and) (and (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((or) (or (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((nand) (nand (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((nor) (nor (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((xor) (xor (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((impl) (impl (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  ((equ) (equ (evaluate-boolean left bindings) (evaluate-boolean right bindings)))  (otherwise (error "Invalid operator ~A" op)))))  (otherwise (error "Invalid expression ~A" expression))))))  (defun table (a b expr)  (loop :for p :in '(true fail) :do  (loop :for q :in '(true fail) :do  (let ((v (evaluate-boolean expr (list (cons a p) (cons b q)))))  (format t "~:[fail~;true~] ~:[fail~;true~] ~:[fail~;true~]~%"  (eql 'true p) (eql 'true q) v)))))  ;; (table 'a 'b '(and a (or a b)))  ;; true true true  ;; true fail true  ;; fail true fail  ;; fail fail fail  ;;NIL |

* **P47 (\*) Truth tables for logical expressions (2).**

Continue problem P46 by defining and/2, or/2, etc as being operators. This allows to write the logical expression in the more natural way, as in the example: A and (A or not B). Define operator precedence as usual; i.e. as in Java.

Example:

\* table(A,B, A and (A or not B)).

true true true

true fail true

fail true fail

fail fail fail

"

;; Again, this question doesn't make much sense in lisp.

;;

;; To have fun, we could interpret it as requesting parsing a list in infix notation and translating it to prefix notation.

;;

;; (infix-to-prefix '(a and (a or not b))) --> (and a (or a (not b)))

;;

;; but notice that we need a list anyways, and that parenthesized subexpressions are actually sublists, there's no parentheses to be parsed.

;;

;; On the other hand, we could write a full lexer and parser for prolog syntax, but that would be out of scope for this exercise.

;;

;;

;; Java operator precedences are:

;;

;; Priority Operators Operation Associativity

;; 1 [ ] array index left

;; () method call

;; . member access

;; 2 ++ pre- or postfix increment right

;; -- pre- or postfix decrement

;; + - unary plus, minus

;; ~ bitwise NOT

;; ! boolean (logical) NOT

;; (type) type cast

;; new object creation

;; 3 \* / % multiplication, division, remainder left

;; 4 + - addition, substraction left

;; + string concatenation

;; 5 << signed bit shift left left

;; >> signed bit shift right

;; >>> unsigned bit shift right

;; 6 < <= less than, less than or equal to left

;; > >= greater than, greater than or equal to

;; instanceof reference test

;; 7 == equal to left

;; != not equal to

;; 8 & bitwise AND left

;; & boolean (logical) AND

;; 9 ^ bitwise XOR left

;; ^ boolean (logical) XOR

;; 10 | bitwise OR left

;; | boolean (logical) OR

;; 11 && boolean (logical) AND left

;; 12 || boolean (logical) OR left

;; 13 ? : conditional right

;; 14 = assignment right

;; \*= /= += -= %=

;; <<= >>= >>>=

;; &= ^= |= combinated assignment

;; (operation and assignment)

;;

;; There are no NAND, NOR, EQU, or IMPL, but there are several AND and

;; OR, with different precedences! What a fucking problem statement!

;; So we will write a parser that is parameterized by the precedences,and we'll see later what is needed.

;; For this, we use a simple recursive-descend parser generator:

(load "rdp.lisp")

(use-package :com.informatimago.rdp)

;; Prefix or suffix operators must have arity = 1, and therefore don't have any specific associativity dirrection other than their being prefix or suffix.

;; x not not not not/1 suffix (((x not) not) not)

;; not not not x not/1 prefix (not (not (not x)))

;; Infix operators must have an arity > 1 (usually 2), and must have either left or right associativity.

;; a op b op c op/2 left (a op b) op b

;; a op b op c op/2 right a op (b op c)

;; We define an operator level as a list containing the precedence level (smaller number, higher precendence), the arity, the position-and-associativity (:prefix, :suffix, :infix-left or :infix-right), and the list of operators:

(defstruct (level (:type list))

(precedence 0 :type integer)

(arity 0 :type integer)

(position :prefix :type (member :prefix :suffix :infix-left :infix-right))

(operators '() :type list))

(defun leftify (operators expr)

"

Transforms a right-associative operation tree EXPR of OPERATORS, into a leflt-associative one. (op a (op b c)) --> (op (op a b) c)

"

(labels ((flatten (expr flattened)

(cond

((atom expr) (cons expr flattened))

((member (first expr) operators)

(flatten (third expr) (list\* (first expr) (second expr) flattened)))

(t (cons expr flattened))))

(unflatten-left (expr flattened)

(if (endp flattened)

expr

(unflatten-left (list (first flattened) expr (second flattened))

(rest (rest flattened))))))

(let ((flattened (nreverse (flatten expr '()))))

(unflatten-left (first flattened) (rest flattened)))))

(defun production-var (n)

"Makes a production variable $n"

(intern (format nil "$~A" n)))

(defun token-to-lisp (token)

;; rdp tokens are (terminal "text" position)

;; Since we take care of naming our operator terminals as lisp

;; operators, we can just extract them from the tokens.

;; For variables, we cl:read the text.

(if (atom token)

token

(case (first token)

((identifier) (read-from-string (second token)))

((true) 't)

((fail) 'nil)

(otherwise (first token)))))

(defun generate-operator-level-rules (non-terminal inferior-non-terminal level)

"Generates a grammar rule to parse the given operator level, and

build the corresponding expression tree."

(let ((op-non-terminal (intern (format nil "~A-OP" non-terminal))))

(ecase (level-position level)

((:prefix)

`((--> ,non-terminal

(alt ,op-non-terminal ,inferior-non-terminal)

:action $1)

(--> ,op-non-terminal

(alt ,@(level-operators level))

,@(make-list (level-arity level) :initial-element inferior-non-terminal)

;; (opt (seq (alt ,@(level-operators level))

;; ,@(make-list (level-arity level) :initial-element inferior-non-terminal))

;; ,inferior-non-terminal)

:action (list (token-to-lisp $1)

,@(loop

:repeat (level-arity level)

:for i :from 2 :collect (production-var i))))))

((:suffix)

`((--> ,non-terminal

(alt ,op-non-terminal ,inferior-non-terminal)

:action $1)

(--> ,op-non-terminal

,@(make-list (level-arity level) :initial-element inferior-non-terminal)

(alt ,@(level-operators level))

:action (list (token-to-lisp ,(production-var (1+ (level-arity level))))

,@(loop

:repeat (level-arity level)

:for i :from 1 :collect (production-var i))))))

;; We're using a recursive-descend parser, so we can have only

;; right-recursive rules. Therefore we will just collect the list

;; of operations at the grammar level, and implement the

;; associativity in the action.

;;

;; (--> factor

;; term op factor) ; :infix-right term op (term op term)

;;

;; (--> factor

;; factor op term) ; :infix-left (term op term) op term

((:infix-left :infix-right)

(assert (= 2 (level-arity level))

(level) "Infix operators with an arity different from 2 are not implemented.")

`((--> ,non-terminal

,op-non-terminal

:action ,(if (eql :infix-left (level-position level))

`(if (and (listp $1) (= 3 (length $1)))

(leftify ',(level-operators level) $1)

$1)

`$1))

(--> ,op-non-terminal

,inferior-non-terminal (opt (alt ,@(level-operators level)) ,op-non-terminal)

:action (if $2

(destructuring-bind (op right) $2

(list (token-to-lisp op) $1 right))

$1)))))))

(defun generate-operator-grammar (name operator-levels)

"Generate a RDP grammar for the operators given in OPERATOR-LEVELS.

This will create a function named PARSE-{NAME}."

(let\* ((levels (sort (copy-list operator-levels) (function >)

:key (function level-precedence)))

(non-terminals (nconc (loop

:for level :in levels

:collect (intern (format nil "~{~A~^/~}-FACTOR" (level-operators level))))

(list 'term)))

(terminals (nconc (mapcan (lambda (level)

(mapcar (lambda (operator) (list operator (string-downcase operator)))

(level-operators level)))

levels)

'((true "true")

(fail "fail")

(identifier "[A-Za-z][-A-Za-z0-9]\*"))))

(rules `((--> ,(first (last non-terminals))

(alt constant variable parenthesized-expression)

:action $1)

;; We need to wrap terms in an identity operator to

;; avoid lefitification of the first non-terminal:

;; (a impl b) impl (c impl d) must stay that way.

(--> parenthesized-expression

"(" ,(first non-terminals) ")"

:action (list 'identity $2))

(--> constant (alt true fail)

:action (token-to-lisp $1))

(--> variable identifier

:action (token-to-lisp $1))

,@(reduce (function append)

(mapcar (function generate-operator-level-rules)

non-terminals

(rest non-terminals)

levels)

:from-end t))))

#+debug

(print `(com.informatimago.rdp:generate-grammar

,name

:terminals ',terminals

:start ',(first non-terminals)

:rules ',rules))

(com.informatimago.rdp:generate-grammar

name

:terminals terminals

:start (first non-terminals)

:rules rules)))

(defparameter \*operators\* '(( 2 1 :prefix (not))

( 8 2 :infix-left (and nand))

( 9 2 :infix-left (xor equ))

(10 2 :infix-left (or nor))

(12 2 :infix-left (impl))))

(generate-operator-grammar 'logical-expression \*operators\*)

(defun test/operator-grammar ()

(loop

:for (source expected)

:in '(("a" a)

("true" t)

("fail" nil)

("a and b" (and a b))

("(a impl b) impl (c impl d)" (impl (identity (impl a b)) (identity (impl c d))))

("a and b and c and d" (AND (AND (AND a b) c) D))

("a and b and c or d and e and f or g and i and j"

(or (or (and (and a b) c) (and (and d e) f)) (and (and g i) j)))

("(a xor b) equ (not a xor not b)"

(equ (identity (xor a b)) (identity (xor (not a) (not b))))))

:do (let ((result (handler-case (PARSE-LOGICAL-EXPRESSION source)

(error (err) (princ err) (terpri) :error))))

(assert (equal result expected)

(source)

"Parsing the logical expression ~S~% gave ~S ~%instead of expected ~S"

source result expected)))

:success)

(defun remove-identity (expr)

(cond

((atom expr) expr)

((eql 'identity (first expr)) (remove-identity (second expr)))

(t (cons (first expr) (mapcar (function remove-identity) (rest expr))))))

;; (table 'a 'b (remove-identity (parse-logical-expression "a and (a or not b)")))

;; true true true

;; true fail true

;; fail true fail

;; fail fail fail

;; --> NIL

* **P48 (\*\*) Truth tables for logical expressions (3).**

Generalize problem P47 in such a way that the logical expression may contain any number of logical variables. Define table/2 in a way that table(List,Expr) prints the truth table for the expression Expr, which contains the logical variables enumerated in List.

Example:

\* table([A,B,C], A and (B or C) equ A and B or A and C).

true true true true

true true fail true

true fail true true

true fail fail true

fail true true true

fail true fail true

fail fail true true

fail fail fail true"

(defun table (variables expr)

(loop

:with width = (length variables)

:with size = (expt 2 width)

:for i :from (1- size) :downto 0 ; to display from true to fail.

:do (let\* ((bindings (loop

:for bit :from (1- width) :by -1

:for var :in variables

:collect (cons var

(if (logbitp bit i)

'true

'fail))))

(v (evaluate-boolean expr bindings)))

(format t "~{~:[fail~;true~] ~} ~:[fail~;true~]~%"

(mapcar (lambda (binding) (EQL 'TRUE (CDR BINDING))) BINDINGS)

V))))

;; (table '(a b c) (remove-identity (parse-logical-expression "(a and (b or c) equ a and b or a and c)")))

;; true true true true

;; true true fail true

;; true fail true true

;; true fail fail true

;; fail true true true

;; fail true fail true

;; fail fail true true

;; fail fail fail true

;; --> NIL

* **P49 (\*\*) Gray code.**

An n-bit Gray code is a sequence of n-bit strings constructed according to certain rules. For example,

n = 1: C(1) = ['0','1'].

n = 2: C(2) = ['00','01','11','10'].

n = 3: C(3) = ['000','001','011','010',Â´110Â´,Â´111Â´,Â´101Â´,Â´100Â´].

Find out the construction rules and write a predicate with the following specification:

% gray(N,C) :- C is the N-bit Gray code

Can you apply the method of \"result caching\" in order to make the predicate more efficient, when it is to be used repeatedly?

;; In Lisp, we will instead implement a function (gray n) producing the list of gray codes C(n). Here is a solution giving the codes as strings of #\0 or #\1:

(defun gray (n)

(if (= 1 n)

(list "0" "1")

(let ((gray-1 (gray (1- n))))

(nconc (mapcar (lambda (code) (concatenate 'string "0" code))

gray-1)

(mapcar (lambda (code) (concatenate 'string "1" code))

(nreverse gray-1))))))

;; Since we call (gray (1- n)) only once per call to (gray n), there's no gain in memoizing gray, unless it is called repeatitively, in which case, memoizing could help as caching any other function.

;; (gray 3) --> ("000" "001" "011" "010" "110" "111" "101" "100")

;; Here is a version giving the codes as integers:

(defun gray (n)

(if (= 1 n)

(list 0 1)

(let ((gray-1 (gray (1- n))))

(nconc gray-1

(mapcar (lambda (code) (dpb 1 (byte 1 (1- n)) code))

(reverse gray-1))))))

;; (gray 3) --> (0 1 3 2 6 7 5 4)

* **P50 (\*\*\*) Huffman code.**

First of all, consult a good book on discrete mathematics or algorithms for a detailed description of Huffman codes!

We suppose a set of symbols with their frequencies, given as a list of fr(S,F) terms. Example:

[fr(a,45),fr(b,13),fr(c,12),fr(d,16),fr(e,9),fr(f,5)]. Our

objective is to construct a list hc(S,C) terms, where C is the

Huffman code word for the symbol S. In our example, the result

could be Hs = [hc(a,'0'), hc(b,'101'), hc(c,'100'), hc(d,'111'),

hc(e,'1101'), hc(f,'1100')] [hc(a,'01'),...etc.]. The task shall

be performed by the predicate huffman/2 defined as follows:

% huffman(Fs,Hs) :- Hs is the Huffman code table for the frequency table Fs

"

#-(and) "

The algorithm described in:

\"A Method for the Construction of Minimum-Redundancy Codes\" David A. Huffman, Procedings of the I.R.E.

<http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf> is:

compute huffman code:

input: message ensemble (set of (message, probability)).

base D.

output: code ensemble (set of (message, code)).

algorithm:

1- sort the message ensemble by decreasing probability.

2- N is the cardinal of the message ensemble (number of different messages).

3- compute the integer n\_0 such as 2<=n\_0<=D and (N-n\_0)/(D-1) is integer.

4- select the n\_0 least probable messages, and assign them each a digit code.

5- substitute the selected messages by a composite message summing their probability, and re-order it.

6- while there remains more than one message, do steps thru 8.

7- select D least probable messages, and assign them each a digit code.

8- substitute the selected messages by a composite message summing their probability, and re-order it.

9- the code of each message is given by the concatenation of the code digits of the aggregate they've been put in.

"

(defun make-code (base type digits)

"

Returns a STRING or VECTOR representing the code made of the

concatenation of the digits in the DIGITS list.

BASE is the base of the code. When TYPE is STRING, BASE is limited to 36.

TYPE

is either STRING or VECTOR, indicates the type of the codes.

If BASE is two, then a bit-vector are used, otherwise vectors

of unsigned-bytes.

DIGITS is a list of digits (integers between 0 and (1- BASE)).

"

(check-type base integer)

(ecase type

((string)

(assert (<= 2 base 36) (base) "When TYPE is STRING, BASE must be between 2 and 36")

(make-array (length digits)

:element-type 'base-char

:initial-contents (mapcar (function digit-char) digits)))

((vector)

(make-array (length digits)

:element-type (if (= 2 base)

'bit

`(unsigned-byte ,base))

:initial-contents digits))))

(defstruct composite

"

While building the Huffman codes, the symbols are gathered in a composite symbol tree. Each node maintains the code digit, the frequency of all the messages its aggregates, and the list of its subnodes.

"

symbols frequency code)

(defun collect-codes (composite digits)

"

Returns an a-list of all the leaf nodes in the composite tree, ie. The symbols, with their code (reversed list of digits).

"

(mapcan (lambda (symbol)

(if (composite-p (first (composite-symbols symbol)))

(collect-codes symbol (cons (composite-code symbol) digits))

(list (cons (first (composite-symbols symbol))

(cons (composite-code symbol) digits)))))

(composite-symbols composite)))

(defun huffman (symbols &key (base 2) (type 'string))

"

Builds the Huffman code ensemble for the given message ensemble.

Returns an a-list (symbol . code).

SYMBOLS

is an a-list (symbol . frequency)

BASE

is the base of the code. When TYPE is STRING, BASE is limited to 36.

TYPE

is either STRING or VECTOR, indicates the type of the codes.

If BASE is two, then a bit-vector are used, otherwise vectors

of unsigned-bytes."

(let\* ((symbols (sort (mapcar (lambda (symbol)

(make-composite :symbols (list (car symbol))

:frequency (cdr symbol)))

symbols)

(function <) :key (function composite-frequency)))

(n (length symbols))

(n0 (if (= 2 base)

2

;; (n-n0)/(d-1) is integer <=> (r-n0)/(d-1) is integer

;; with r being the remainder of n / (d-1).

(let ((r (mod n (1- base))))

(case r

(0 (1- base))

(1 base)

(otherwise r))))))

(assert (zerop (mod (- n n0) (1- base))))

(if (<= n base)

(loop ; Less than base symbols, we just assign all of them a single-digit code.

:for symbol :in selection

:for code :from 0

:collect (cons (first (composite-symbols symbol))

(make-code base type (list code))))

(loop ; more than BASE symbols. Let's apply Huffman's algorithm.

:for selection-size = n0 :then base

:for selection = (subseq symbols 0 selection-size)

:for composite = (make-composite :symbols selection

:frequency (reduce (function +) selection

:key (function composite-frequency)))

:do (progn

(loop ; codify the selection.

:for symbol :in selection

:for code :from (1- (length selection)) :downto 0

; to keep the same numbering as in Huffman's paper.

; We could as well just use :for code :from 0

:do (setf (composite-code symbol) code))

(setf symbols (merge 'list (list composite) (nthcdr selection-size symbols)

(function <) :key (function composite-frequency))))

:while (rest symbols)

:finally (return (mapcar (lambda (symbol)

(cons (car symbol) (make-code base type (reverse (cdr symbol)))))

(collect-codes (first symbols) '())))))))

;; (huffman '((a . 45) (b . 13) (c . 12) (d . 16) (e . 9) (f . 5)))

;; --> ((A . "1") (C . "011") (B . "010") (F . "0011") (E . "0010") (D . "000"));;

;; (huffman '((a . 45) (b . 13) (c . 12) (d . 16) (e . 9) (f . 5)) :type 'vector)

;; --> ((A . #\*1) (C . #\*011) (B . #\*010) (F . #\*0011) (E . #\*0010) (D . #\*000));;

;; (huffman '((a . 45) (b . 13) (c . 12) (d . 16) (e . 9) (f . 5)) :type 'string :base 3)

;; --> ((D . "2") (C . "12") (B . "11") (F . "101") (E . "100") (A . "0"))

(defparameter \*example-1\*

'((1 . 0.20)

(2 . 0.18)

(3 . 0.10)

(4 . 0.10)

(5 . 0.10)

(6 . 0.06)

(7 . 0.06)

(8 . 0.04)

(9 . 0.04)

(10 . 0.04)

(11 . 0.04)

(12 . 0.03)

(13 . 0.01)))

;; (map nil (function print)

;; (sort (huffman \*example-1\* :base 2 :type 'string)

;; (function <) :key (function car)))

;; ( 1 . "10")

;; ( 2 . "000")

;; ( 3 . "111")

;; ( 4 . "110")

;; ( 5 . "011")

;; ( 6 . "00100") ; 6 and 7 have the same probability, so this result is valid,

;; ( 7 . "0101") ; it depends on the way sort orders them.

;; ( 8 . "00110")

;; ( 9 . "01001")

;; (10 . "01000")

;; (11 . "00101")

;; (12 . "001110")

;; (13 . "001111")

(defparameter \*example-2\*

'((1 . 0.22)

(2 . 0.20)

(3 . 0.18)

(4 . 0.15)

(5 . 0.10)

(6 . 0.08)

(7 . 0.05)

(8 . 0.02)))

;; (map nil (function print)

;; (sort (huffman \*example-2\* :base 4 :type 'string)

;; (function <) :key (function car)))

;; (1 . "1")

;; (2 . "2")

;; (3 . "3")

;; (4 . "00")

;; (5 . "01")

;; (6 . "02")

;; (7 . "030")

;; (8 . "031")