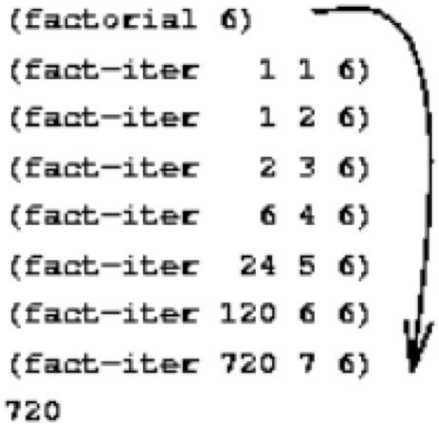
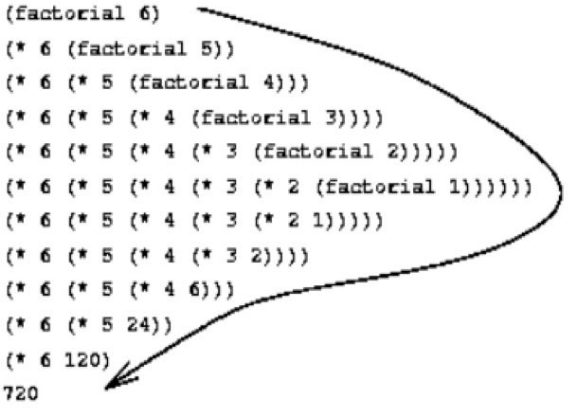
Tail Recursive vs Not Tail Recursive



Only uses the registers



Uses the stack

Iteracija	Rekurzija

```
(define (foo-iter x0 x1 x2 x3 n)
  (define (foo-iter-iter x0 x1 x2 x3 n )
          (let ((x4 (- (+ (* 2 x3) (* 3 x1) x0) x2)) )
          (cond
          ([eq? n 1]  x4)
          (else (foo-iter-iter x1 x2 x3 x4 (- n 1))))))
  (foo-iter-iter x0 x1 x2 x3 (- n 3)))
```

```
; ekspanzija -> <- redukcija
(define (foo-rek x0 x1 x2 x3 n)
  (cond
          ([eq? n 0] x0)
          ([eq? n 1] x1)
          ([eq? n 2] x2)
          ([eq? n 3] x3)
          ([eq? n 4] (- (+ (* 2 x3) (* 3 x1) x0) x2))
          (else (- (+ (* 2 (foo-rek x0 x1 x2 x3 (- n 1)))
                    (* 3 (foo-rek x0 x1 x2 x3 (- n 3)))
                    (foo-rek x0 x1 x2 x3 (- n 4)))
          (foo-rek x0 x1 x2 x3 (- n 2))))))
```

```
(define (permute l)
  (if (null? l)
     '(())
     (append-map (lambda (p)
                 (map (lambda (n)
                        (insert p n (car l)))
                      (seq 0 (length p))))
               (permute (cdr l)))))
```

PERMUTE

- append-map = flatmap

Naslov2

(define (map proc lst)
  (cond
  ((null? lst) '())
  (else (cons (proc (car lst)) (map proc (cdr lst))))))
	MAP
	apply proc to every elem
	return lst

(define (filter pred? lst)
  (cond
  ([null? lst] '())
  ([pred? (car lst)] (cons (car lst) (filter pred? (cdr lst))))
  (else (filter pred? (cdr lst)))))
	FILTER
	return list of elem if
	predicate? #t for elem

(define (length lst)
  (define (len-iter lst cntr)
  (cond
  ([null? lst] cntr)
  (else (len-iter (cdr lst) (+ 1 cntr)))))
  (len-iter lst 0))
	LENGTH
	ret len of lst

(define (reverse lst)
  (if [null? lst] '()
  (append (reverse (cdr lst)) (list (car lst)))))
	REVERSE
	reverse lst

(define (count predicate? lst)
  (cond
  ([null? lst] 0)
  ([predicate? (car lst)] (+ 1 (count predicate? (cdr lst))))
  (else (+ (count predicate? (cdr lst)) 0))))
	COUNT
	# of elem in lst

(define (accumulate proc init lst)
  (cond
  ([null? lst] init)
  (else (proc (car lst) (accumulate proc init (cdr lst))))))
	ACCUMULATE
	Accumulate,FoldR,Reduce,
	Compress,Inject
	-> apply proc to lst, ret 1
	val
	-> proc npr. :
	  (lambda (x y) (+ x y)

(define (member elem lst)
  (cond
  ([null? lst] #f)
  ([eq? elem (car lst)] #t)
  (else (member elem (cdr lst)))))
	MEMBER
	-> check if elem is in lst

(define (unique lst)
  (define (uniquer lst lst-un)
  (cond
  ([null? lst] lst-un)
  ([not (member (car lst) lst-un)]
  (uniquer (cdr lst) (append lst-un (list (car lst)))))
  (else (uniquer (cdr lst) lst-un))))
  (uniquer lst '()))
	UNIQUE
	-> ret lst of unique elem in
	lst

Naslov2

(define (flatten lst)
 (cond
  ([null? lst] '())
  ([pair? (car lst)]
  (append (flatten (car lst))
  (flatten (cdr lst))))
  (else (cons (car lst) (flatten (cdr lst))))))
	FLATTEN
	-> in > list of lists
	-> out > list
	flattens list
	of lists
	of lists
	of ...

(define (flatmap proc seq) (accumulate append nil (map proc seq)))	FLATMAP ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

(define (sorted? lst)
 (define (sort-up? lst)
  (cond
  ([null? lst] #t)
  ([null?(cdr lst)] #t)
  ([>= (car lst) (car (cdr lst))] (sort-up? (cdr lst)))
  (else #f)))
 	
 (define (sort-dwn? lst)
  (cond
  ([null? lst] #t)
  ([null?(cdr lst)] #t)
  ([<= (car lst) (car (cdr lst))] (sort-dwn? (cdr lst)))
  (else #f)))
 	
 (or (sort-up? lst) (sort-dwn? lst)))
	creating a single list out of a list of sublists *after* applying a procedure to each sublist
	
	(flatmap (lambda (n)
	(if (odd? n) (list (* n n)) '()))
	'(1 2 3 4 5 6))
	->(1 9 25)
	
	(flatmap vector->list
	'(#(1) #(2 3) #(4)))
	-> (1 2 3 4)
	
	;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
	SORTED?
	-> gleda dali je lst sortiran

(define (zip lst1 lst2)
 (define (zip-iter lst1 lst2 lst-rez)
  (cond
  ([null? lst1] lst-rez)
  ([null? lst2] lst-rez)
  (else (zip-iter (cdr lst1) (cdr lst2)
   (cons (cons (car lst1) (car lst2))
    lst-rez)))))
 (zip-iter lst1 lst2 '()))
;;;
(define (zip-rek l1 l2)
 (if (or (null? l1) (null? l2))
  '()
  (cons (cons (car l1) (car l2)) (zip (cdr l1) (cdr l2)))))
	ZIP
	
	zip operator that takes two lists and gives a list of pairs of those elements.
	
	the len of out should be the len of shorter list
	
	(zip '(1 2 3 4) '(a b c d))
	 -> '((1 . a) (2 . b)
	  (3 . c) (4 . d))

(flatmap
 (lambda (x)
  (map
   (lambda (y) (list y x))
   '(1 2 3)))
 '(a b c))
	CARTESIAN PRODUCT
	
	->((1 a) (2 a) (3 a)
	 (1 b) (2 b) (3 b) (1 c)
	 (2 c) (3 c))

Naslov2

naslov 3

(let ((var val) (var val)) <scope>)		
(cons Relem Lelem)	--->	‘ (Relem . Lelem) ,(cons 1 2) -> (1 . 2)
(append Rlst Llst)	--->	‘(Rlst Llst) , (append '(1) '(2)) -> (1 2)