

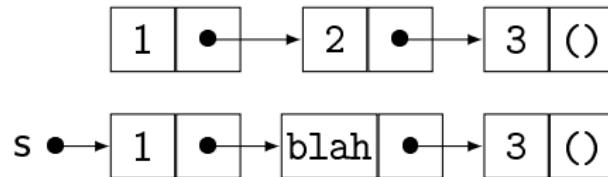
KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA  
UNIVERZITA PALACKÉHO, OLOMOUC

# PARADIGMATA PROGRAMOVÁNÍ 2A MUTACE

Slajdy vytvořili Vilém Vychodil a Jan Konečný

V následujícím nedojde ke změně seznamu, ale k vytvoření nového seznamu

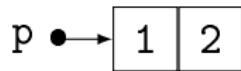
```
(define s '(1 2 3))  
(set! s '(1 blah 3))
```



mutátory páru: procedury `set-car!` a `set-cdr!`

destruktivně změní pár, vrací nedefinovanou hodnotu

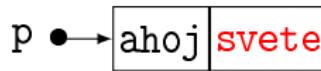
`(define p (cons 1 2))`



`(set-car! p 'ahoj)`

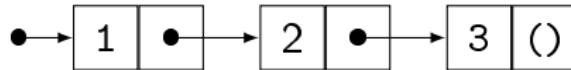


`(set-cdr! p 'svete)`



Příklad:

```
(define s '(1 2 3))
```



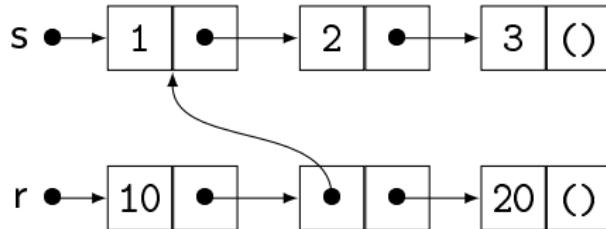
```
(set-car! (cdr s) 'blah)
```

s  $\Rightarrow$  (1 blah 3)

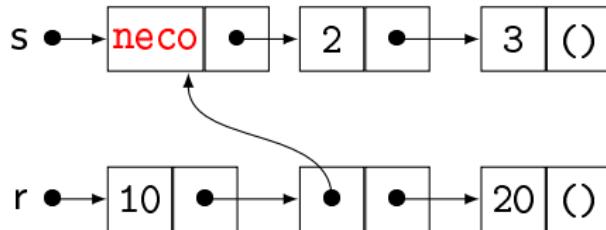


vznikají vzájemně provázané seznamy, nutno dbát zvýšené obezřetnosti!  
zvýšené riziko vzniku chyb: nechtěná mutace seznamů

```
(define s '(1 2 3))  
(define r (list 10 s 20))  
r => (10 (1 2 3) 20)
```



```
(set-car! (cadr r) 'neco)
```



```
r => (10 (neco 2 3) 20)
```

```
s => (neco 2 3)
```

*;; pro  $n = 3$  vytvoř: ((#f #f #f) (#f #f) (#f)), a podobně  
;; asymptotická časová složitost:  $O(n(1 + n)/2)$*

```
(define f-list
  (lambda (n)
    (build-list n
      (lambda (i)
        (build-list (- n i)
          (lambda (x) #f)))))))
```

```
(define s (f-list 4))
```

```
s  $\mapsto$  ((#f #f #f #f) (#f #f #f) (#f #f) (#f))
```

```
(set-car! (car (reverse s)) 'blah)
```

```
s  $\mapsto$  ((#f #f #f #f) (#f #f #f) (#f #f) (blah))
```

```
(set-car! (cadr (reverse s)) 100)
```

```
s  $\mapsto$  ((#f #f #f #f) (#f #f #f) (100 #f) (blah))
```

; ; ta samá procedura, ale efektivnější  
; ; asymptotická časová složitost:  $O(n)$

```
(define f-list
  (lambda (n)
    (if (= n 1)
        '((#f))
        (let ((rest (f-list (- n 1))))
          (cons (cons (caar rest) (car rest)) rest))))))

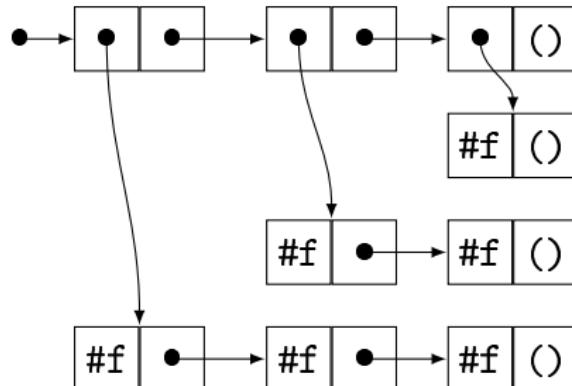
(define s (f-list 4))
s => ((#f #f #f #f) (#f #f #f) (#f #f) (#f))
```

## ROZDÍL OPROTI PŘEDCHOZÍMU:

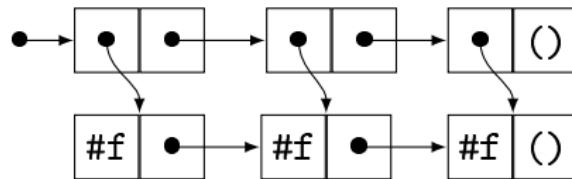
```
(set-car! (car (reverse s)) 'blah)
s => ((#f #f #f blah) (#f #f blah) (#f blah) (blah))

(set-car! (cadr (reverse s)) 100)
s => ((#f #f 100 blah) (#f 100 blah) (100 blah) (blah))
```

(f-list 3) ... první verze používající *build-list*



(f-list 3) ... druhá verze (rekurzivní)



program = data = mutovatelný seznam  
je možné destruktivně modifikovat samotný program (!!)

```
(define proc
  (lambda (x)
    (display (list "Input parameter: " x))
    (newline)
    (set-car! x (+ (car x) 1))
    x))
```

```
(define test (lambda () (proc '(0))))
```

```
(test) => (1)
```

vytištěno: (Input parameter: (0))

```
(test) => (2)
```

vytištěno: (Input parameter: (1))

⋮

*; ; konstrukce mutovatelného páru*

```
(define cons  
  (lambda (x y)
```

*; ; modifikátory vazby symbolů x a y*

```
(define set-x! (lambda (value) (set! x value)))  
(define set-y! (lambda (value) (set! y value))))
```

*; ; dispatch*

```
(lambda (signal)  
  (cond ((equal? signal 'car) x)  
        ((equal? signal 'cdr) y)  
        ((equal? signal 'set-car!) set-x!)  
        ((equal? signal 'set-cdr!) set-y!)  
        (else 'unknown-signal))))
```

*;; selektory car a cdr*

```
(define car (lambda (pair) (pair 'car)))
(define cdr (lambda (pair) (pair 'cdr)))
```

*;; mutace prvního prvku*

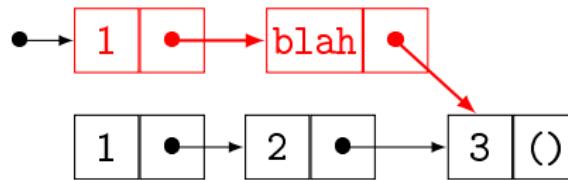
```
(define set-car!
  (lambda (pair value)
    ((pair 'set-car!) value)))
```

*;; mutace druhého prvku*

```
(define set-cdr!
  (lambda (pair value)
    ((pair 'set-cdr!) value)))
```

list-set - klasická nedestruktivní (funkcionální) verze

```
(define list-set
  (lambda (l index value)
    (let aux ((l l)
              (i 0))
      (if (= i index)
          (cons value (cdr l))
          (cons (car l) (aux (cdr l) (+ i 1)))))))
```



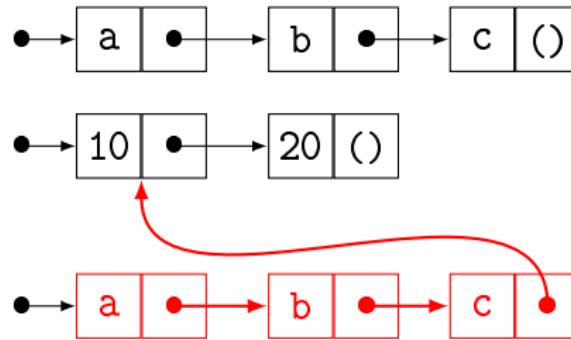
list-set! destruktivní modifikace prvku

```
(define list-set!
  (lambda (l index value)
    (let iter ((l l)
              (i 0))
      (if (= i index)
          (set-car! l value)
          (iter (cdr l) (+ i 1))))))
  l))
```



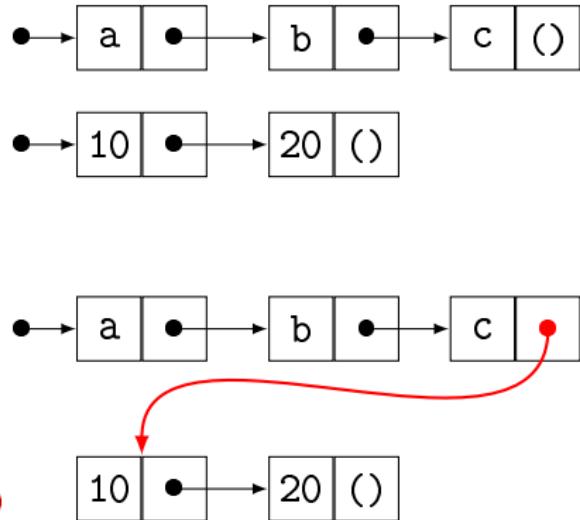
## klasický nedestruktivní (funkcionální) append2

```
(define append2
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append2 (cdr l1) l2)))))
```



destruktivní spojení dvou seznamů

```
(define append2!
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (let iter ((l l1))
          (if (null? (cdr l))
              (begin
                (set-cdr! l l2)
                l1)
              (iter (cdr l)))))))
```



při spojení seznamů dochází k mutaci prvního argumentu:

```
(define x '(a b c))  
(define y '(10 20))  
(append2! x y)  => (a b c 10 20)  
x                  => (a b c 10 20)  
y                  => (10 20)
```

neplatí v případě prázdného seznamu (není to pář)

```
(define x '())  
(define y '(10 20))  
(append2! x y)  => (10 20)  
x                  => ()  
y                  => (10 20)
```

můžeme rozšířit na libovolné argumenty:

```
(define append!
  (lambda lists
    (foldr append2! '() lists)))
```

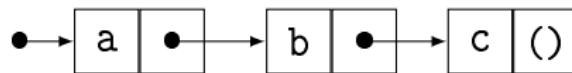
dochází k destrukci všech kromě posledního  
všechny neprázdné seznamy se postupně provážou

```
(define a '(a b c))
(define b '(#t #f))
(define c '(2 4 6 8))
(define d '(foo bar baz))
```

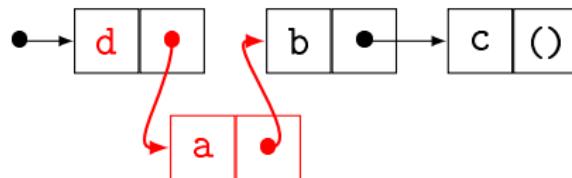
(append! a b c d)	⇒	(a b c #t #f 2 4 6 8 foo bar baz)
a	⇒	(a b c #t #f 2 4 6 8 foo bar baz)
b	⇒	(#t #f 2 4 6 8 foo bar baz)
c	⇒	(2 4 6 8 foo bar baz)
d	⇒	(foo bar baz)

*destruktivní přidávání prvku do seznamu*

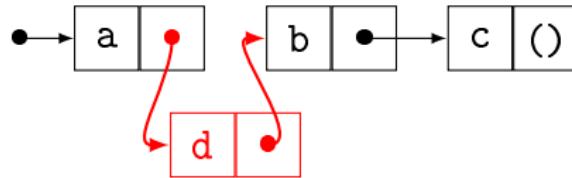
(define s '(a b c))



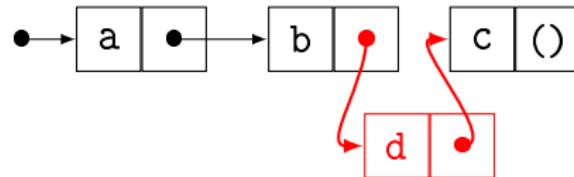
(list-insert! s 0 'd)



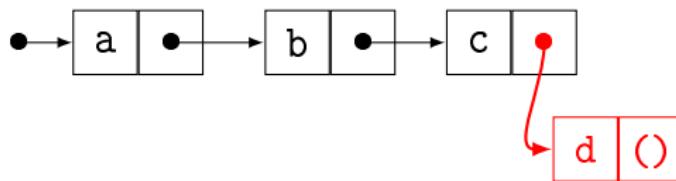
(list-insert! s 1 'd)



(list-insert! s 2 'd)



(list-insert! s 3 'd)



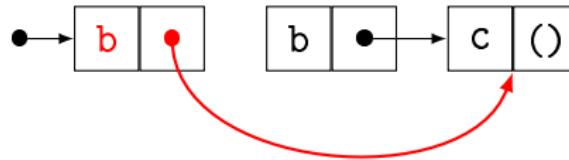
destruktivní přidávání prvku do NEPRÁZDNÉHO seznamu  
pro prázdný seznam nelze (nejsou mutovatelné)

```
(define list-insert!
  (lambda (l index value)
    (if (= index 0) ; vkládání na začátek
        (begin
          (set-cdr! l (cons (car l) (cdr l)))
          (set-car! l value))
        (let iter ((l l)
                  (index index)) ; vkládání doprostřed
          (if (= index 1)
              (set-cdr! l (cons value (cdr l)))
              (iter (cdr l) (- index 1)))))))
```

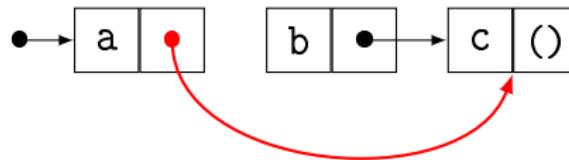
*destruktivní odebírání prvku ze seznamu*

(define s '(a b c))

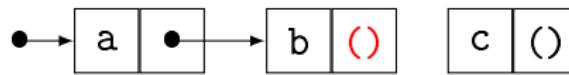
(list-delete! s 0)



(list-delete! s 1)



(list-delete! s 2)



destruktivní mazání prvku z aspoň DVOUPRVKOVÉHO seznamu  
jednoprvkové seznamy nelze zmutovat na prázdné

```
(define list-delete!
  (lambda (l index)
    (if (= index 0)
        (begin
          (set-car! s (cadr s)) ; mazání z první pozice
          (set-cdr! s (cddr s)))
        (let iter ((l l) ; mazání ze zbytku seznamu
                  (index index))
            (if (= index 1)
                (set-cdr! l (cddr l))
                (iter (cdr l) (- index 1)))))))
```

destruktivní mazání prvku z aspoň DVOUPRVKOVÉHO seznamu  
jednoprvkové seznamy nelze zmutovat na prázdné

```
(define list-delete!
  (lambda (l index)
    (if (= index 0)
        (begin
          (set-car! s (cadr s)) ; mazání z první pozice
          (set-cdr! s (cddr s)))
        (let iter ((l l) ; mazání ze zbytku seznamu
                  (index index))
            (if (= index 1)
                (set-cdr! l (cddr l))
                (iter (cdr l) (- index 1)))))))
```

## Efektivní implementace FRONTY: vkládání a mazání v $O(1)$

*; ; vytvoř prázdnou frontu*

(define make-queue

(lambda ()

(cons '() '())))

*; ; testuj, zdali je daná fronta prázdná*

(define empty-queue?

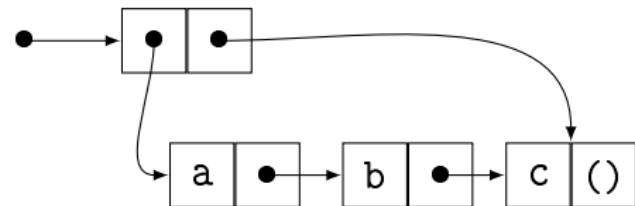
(lambda (queue)

(and (null? (car queue))

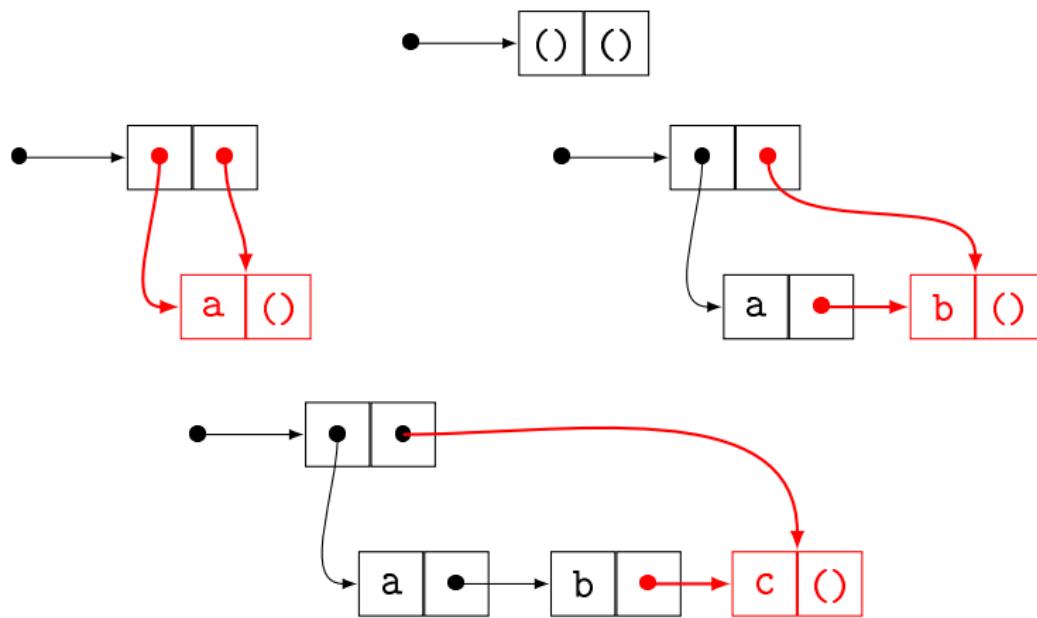
(null? (cdr queue)))))

*; ; vrát' prvek na vrcholu fronty*

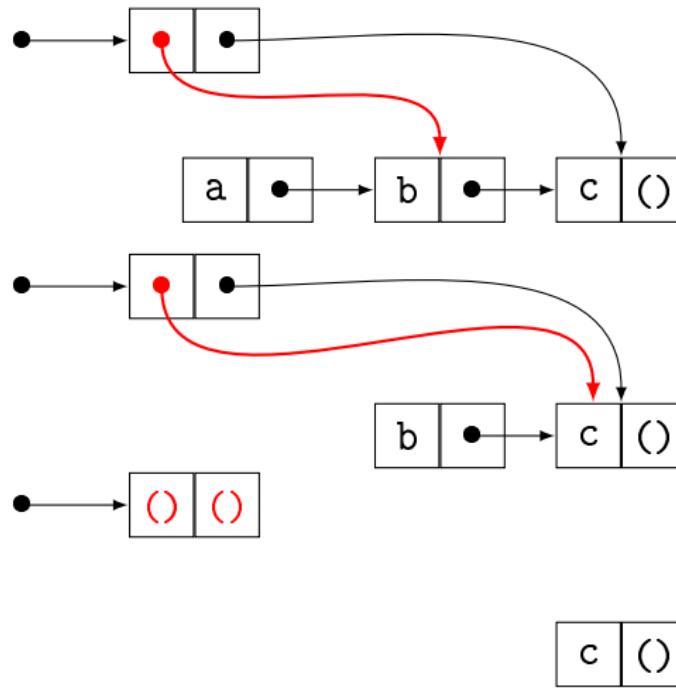
(define queue-get caar)



# Princip vkládání prvku na konec fronty



# Princip smazání prvku ze začátku fronty



*; ; vlož prvek na konec fronty*

```
(define queue-insert!
  (lambda (queue elem)
    (if (empty-queue? queue)
        (begin
          (set-car! queue (cons elem (car queue)))
          (set-cdr! queue (cdr queue)))
        (begin
          (set-cdr! (cdr queue) (list elem))
          (set-cdr! queue (cddr queue)))))))
```

*; ; smaž prvek ze začátku fronty*

```
(define queue-delete!
  (lambda (queue)
    (if (not (empty-queue? queue))
        (begin
          (set-car! queue (cdar queue))
          (if (null? (car queue))
              (set-cdr! queue '()))))))
```

```
(define q (make-queue))
```

```
q  $\Rightarrow$  ()
```

```
(queue-insert! q 10)
```

```
(queue-insert! q 20)
```

```
(queue-insert! q 30)
```

```
q  $\Rightarrow$  ((10 20 30) 30)
```

```
(queue-get q)  $\Rightarrow$  10
```

```
(queue-delete! q)
```

```
(queue-insert! q 40)
```

```
q  $\Rightarrow$  ((20 30 40) 40)
```

```
(queue-delete! q)
```

```
(queue-delete! q)
```

```
q  $\Rightarrow$  ((20 30 40) 40)
```

## Cyklické seznamy

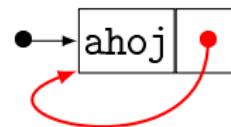
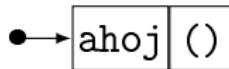
```
(define a '(ahoj))
```

```
(set-cdr! a a)
```

a  $\Rightarrow$  (ahoj ahoj ahoj ...)

$\Rightarrow$  DrScheme vypíše:

```
#0=(ahoj . #0#)
```



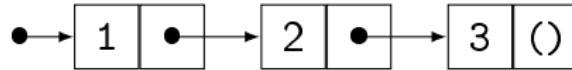
tradičně napsaný `length` selhává

```
(define length
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l))))))
```

(length a)  $\Rightarrow$   $\infty$

(length a)  $\Rightarrow$  length: exp. arg. of type <prop. list>;  
given #0=(ahoj . #0#)

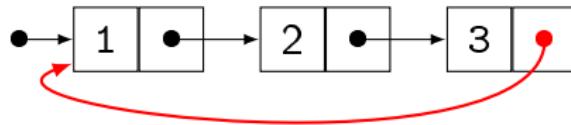
```
(define a '(1 2 3))
```



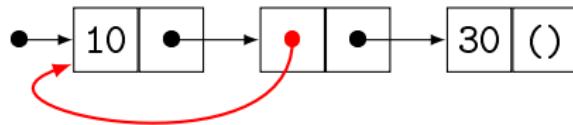
```
(set-cdr! (cddr a) a)
```

a  $\Rightarrow$  (1 2 3 1 2 3 1 2 3 ...)

$\Rightarrow$  #0=(1 2 3 . #0#)



```
(define a '(10 20 30))  
  
(set-car! (cdr a) a)  
a      ⇒ (10 (10 (10 (10 ... 30) 30) 30) 30)  
        ⇒ #0=(10 #0# 30)  
(length a) ⇒ 3
```



```
(define a '#f))
```

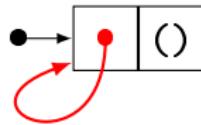


```
(set-car! a a)
```

a  $\Rightarrow (((\dots)))$

$\Rightarrow \#0=(\#0\#)$

(length a)  $\Rightarrow 1$



zacyklení lineárního seznamu (vrací nedefinovanou hodnotu)  
do posledního páru místo () vloží ukazatel na první pár

```
(define cycle!
  (lambda (l)
    (let iter ((aux l))
      (if (null? (cdr aux))
          (set-cdr! aux l)
          (iter (cdr aux)))))))
```

```
(define s '(a b c d e))
(cycle! s)
s => #0=(a b c d e . #0#)
```

## OTÁZKA: jak detektovat cyklus v seznamu?

naivní (nefunkční) řešení

```
(define cycle?  
  (lambda (l)  
    (if (null? l)  
        #f  
        (cycle? (cdr l)))))
```

potřebujeme porovnávat fyzické umístění párů v paměti

predikát `equal?` nám NIJAK NEPOMŮŽE, protože:

```
(define a (cons 1 2))  
(define b (cons 1 2))  
(equal? a b)  => #t ... problém  
(set-car! a 10)  
a  => (10 . 2)  
b  => (1 . 2)
```

## Predikát `eq?` je #t

- na číslech, právě když mají shodnou reprezentaci
- na symbolech, právě když se jmenují stejně
- na párech, právě když mají stejné uložení v paměti

`(eq? 1.2 1.2)`  $\Rightarrow$  #f

`(eq? 2 2)`  $\Rightarrow$  #t

`(eq? 'ahoj 'ahoj)`  $\Rightarrow$  #t

`(eq? (cons 1 2) (cons 1 2))`  $\Rightarrow$  #f

## Predikát `eqv?` je #t

- na číslech, právě když jsou numerická stejná
- na symbolech, právě když se jmenují stejně
- na párech, právě když mají stejné uložení v paměti

`(eqv? 1.2 1.2)`  $\Rightarrow$  #t

`(eqv? 2 2)`  $\Rightarrow$  #t

`(eqv? 'ahoj 'ahoj)`  $\Rightarrow$  #t

`(eqv? (cons 1 2) (cons 1 2))`  $\Rightarrow$  #f

```
(define both
  (lambda (type? x y)
    (and (type? x) (type? y))))
```

```
(define eqv?
  (lambda (x y)
    (if (and (both number? x y)
              (or (both exact? x y)
                  (both inexact? x y)))
        (= x y)
        (eq? x y))))
```

```
(define equal?
  (lambda (x y)
    (or (eqv? x y)
        (and (both pair? x y)
              (equal? (car x) (car y))
              (equal? (cdr x) (cdr y))))))
```

test zacyklenosti lineárního seznamu

```
(define cyclic?
  (lambda (l)
    (let test ((rest (if (null? l)
                           '()
                           (cdr l))))
        (cond ((null? rest) #f)
              ((eq? rest l) #t)
              (else (test (cdr rest)))))))
```

příklad použití

(cyclic? '())	⇒	#f
(cyclic? '(a b c))	⇒	#f
(define s '(a b c))		
(cycle! s)		
(cyclic? s)	⇒	#t

odcyklení lineárního seznamu

rozetnutí cyklu vložením () místo ukazatele na počátek

(define uncycle!

```
(lambda (l)
  (let iter ((aux l))
    (if (eq? (cdr aux) l)
        (set-cdr! aux '())
        (iter (cdr aux))))))
```

zacyklením a odcyklením nemusíme získat výchozí seznam

(define s '(a b c))

(cycle! s)

(set! s (cdr s))

s  $\Rightarrow$  #0=(b c a . #0#)

(uncycle! s)

s  $\Rightarrow$  (b c a)

Analogicky jako existují `eq?`, `eqv?` a `equal?`  
mají své varianty i `member` a `assoc`

`member` ... používá k porovnání prvků `equal?`

`memu` ... používá k porovnání prvků `eqv?`

`memq` ... používá k porovnání prvků `eq?`

`(member ' (a) ' (1 2 (a) 3 4))`  $\Rightarrow$  `((a) 3 4)`

`(memq ' (a) ' (1 2 (a) 3 4))`  $\Rightarrow$  `#f`

`assoc` ... používá k porovnání klíčů `equal?`

`assu` ... používá k porovnání klíčů `eqv?`

`assq` ... používá k porovnání klíčů `eq?`

`(assoc ' (2) ' ((1 . a) ((2) . b) (3 . c)))`  $\Rightarrow$  `((2) . b)`

`(assq ' (2) ' ((1 . a) ((2) . b) (3 . c)))`  $\Rightarrow$  `#f`

## Test cyklu do hloubky

- `cyclic?` testuje pouze jeden typ zacyklení
- selhává na mnoha cyklických strukturách

Příklad:

```
(define s '(a b c d e f))

(set-car! (cdddr s) (cdr s))
s  =>  (a . #0=(b c #0# e f))

(length s)    =>  6
(cyclic? s)  =>  #f
```

## Test cyklu do hloubky

- během sestupu seznamem si udržujeme seznam již navštívených párů
- procedura využívá `memq`

```
(define depth-cyclic?
  (lambda (l)
    (let ((found '()))
      (let test ((l l))
        (if (pair? l)
            (if (memq l found)
                #t
                (begin
                  (set! found (cons l found))
                  (or (test (car l))
                      (test (cdr l))))))
            #f))))
```

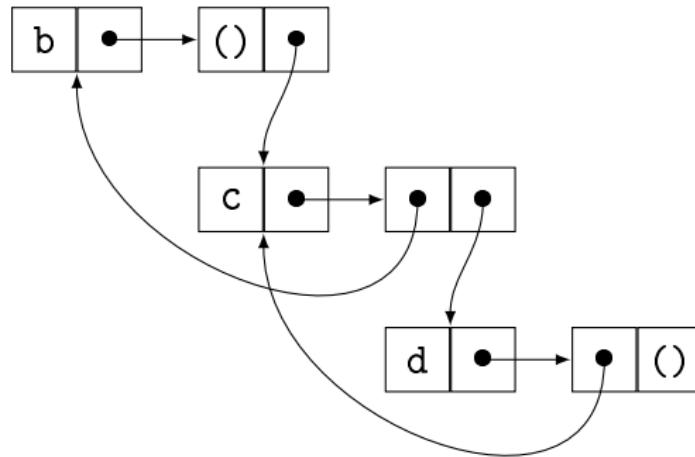
## Obousměrné seznamy: speciální cyklické struktury

- jednotlivé buňky budou ve tvaru (elem . (ptr1 . ptr2)), kde
  - elem je libovolný element uložený v buňce
  - ptr1 je ukazatel na předchozí buňku
  - ptr2 je ukazatel na následující buňku (má stejnou roli jako cdr)

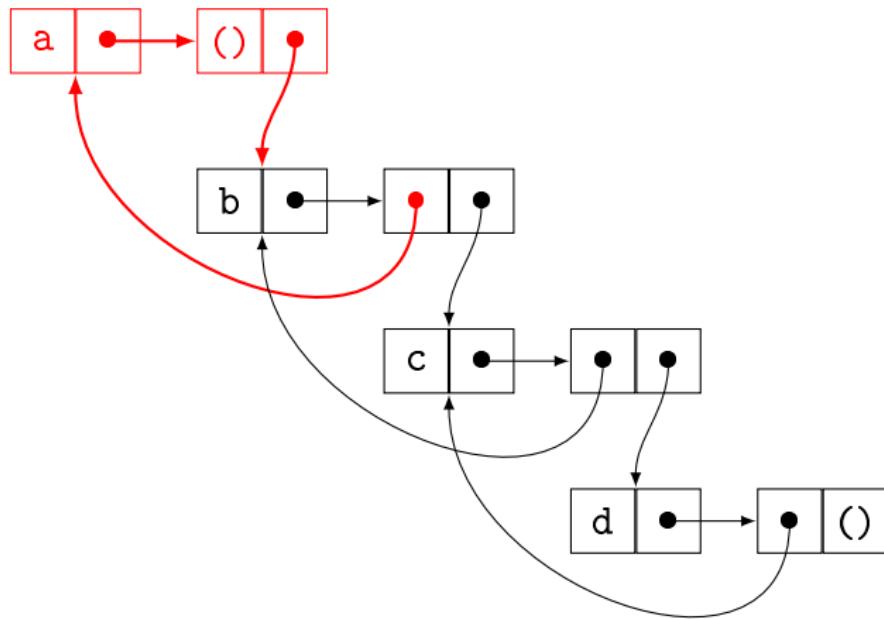
; ; konstruktor obousměrného seznamu

```
(define cons-dlist
  (lambda (elem dlist)
    (let ((new-cell (cons elem
                           (cons '()
                                 dlist))))
      (if (not (null? dlist))
          (set-car! (cdr dlist) new-cell)
          new-cell)))
```

Princip konstrukce obousměrného seznamu (`consd`-list značeno `consd`)  
`(define s (consd 'b (consd 'c (consd 'd '()))))`



```
(define r (consd 'a s))
```



```
;; selektory car, cdr a cir
(define dlist-car (lambda (dlist) (car dlist)))
(define dlist-cdr (lambda (dlist) (cddr dlist)))
(define dlist-cir (lambda (dlist) (cadr dlist)))
```

Příklad: (zkracujeme jména na consd, dcar, dcdr a dcir)

```
(define s (consd 'a (consd 'b (consd 'c (consd 'd '())))))
  => #0=(a () . #1=(b #0# . #2=(c #1# d #2#)))
(dcar s) => a
(dcir s) => ()
(dcdr s) => #0=(b (a () . #0#) . #1=(c #0# d #1#))
(dcar (dcdr s)) => b
(dcir (dcdr s)) => #0=(a () . #1=(b #0# . #2=(c #1# d #2#)))
(dcdr (dcdr s)) => #1=(c #0=(b (a () . #0#) . #1#) d #1#)
```

## Předávání argumentů procedurám

```
(define add2
  (lambda (x)
    (set! x (+ x 2))
    x))
```

```
(define val 10)
(add2 val)  ⇒  12
val          ⇒  10
```

- Chceme umožnit předávat argumenty „odkazem“
- Vytvoříme: BOX = mutovatelný kontainer na hodnotu

# Metody předávání argumentů procedurám

Předávání parametrů = metoda navázání *parametrů* na *formální argumenty*  
souvisí s *příkazem přiřazení*:  $A := B$

$A \dots L$ -hodnota ... paměťové místo, na které ukládáme

$B \dots R$ -hodnota ... obsah, který ukládáme

- **Volání hodnotou (Call by Value)**

- volané proceduře jsou předány  $R$ -hodnoty argumentů
- hodnoty jsou uchovávány (vázány) v lokálním prostředí
- volaná procedura nemůže přiřazovat hodnoty přes argumenty
- jazyky: Scheme, LISP, C

- **Volání odkazem (Call by Reference)**

- volané proceduře jsou předány  $L$ -hodnoty argumentů
- volaná procedura má k dispozici odkazy na úložiště hodnot
- přiřazení do proměnné v těle procedury mění hodnotu argumentu v prostředí ze kterého byla procedura volána
- jazyky: C++ (reference &), PL1

# Metody předávání argumentů procedurám (pokr.)

- **Volání hodnotou-výsledkem (Call by Value-Result)**
  - někdy se principu říká „Copy-restore Linkage“
  - volané procedury jsou předány *L*-hodnoty
  - hodnoty jsou uchovávány (vázány) v lokálním prostředí
  - po dokončení výpočtu se provede kopie lokálně uložených hodnot na paměťová místa předaných argumentů
  - zdánlivě totéž jako „volání odkazem“
  - rozdíl je například při paralelním vyhodnocování
  - jazyky: FORTRAN, MPD
- **Volání jménem (Call by Name)**
  - volané procedury jsou předána jména argumentů
  - pokaždé, když je během vyhodnocování těla procedury naraženo na argument zastupovaný jménem, je toto jméno vyhodnoceno
  - jazyky: Algol 60, makra v jazyku C (přísně vzato nejsou procedury)

## BOX (mutovatelný kontainer na hodnotu)

následující procedura vytvoří box: nový objekt reagující na dva signály  
signál SET ... zapiš hodnotu do boxu, signál GET ... vrat' hodnotu z boxu

```
(define make-box
  (lambda (value)
    (lambda (signal . new-value)
      (cond ((equal? signal 'get) value)
            ((equal? signal 'set)
             (set! value (car new-value)))
            (else "neznamy signal")))))
```

Příklad:

```
(define val (make-box 10))
(val 'get)      => 10
(val 'set 100)
(val 'get)      => 100
```

## Příklad:

vypočti faktoriál a zapiš výsledek do argumentu  
procedura vždy vrací symbol `hotovo`

```
(define proc
  (lambda (box n)
    (letrec ((f (lambda (n)
                  (if (= n 1)
                      1
                      (* n (f (- n 1)))))))
      (box 'set (f n))
      'hotovo)))
```

použití:

```
(proc val 20)  ⇒  hotovo
(val 'get)     ⇒  2432902008176640000
```

## Další mutovatelná element: vektory (analogie pole)

vytváření vektorů pomocí hodnot

(vector)	⇒	#0()
(vector 10 20 30)	⇒	#2(10 20 30)
(vector 10 10 10)	⇒	#3(10)
(vector 'ahoj 'svete)	⇒	#2(ahoj svete)
(vector 1 #f 'blah)	⇒	#3(1 #f blah)

vrat' délku vektoru

(vector-length (vector))	⇒	0
(vector-length (vector 'a 'b 'c))	⇒	3

vytváření vektoru o dané délce (hodnoty jsou nespecifikované)

(make-vector 10)	⇒	#10(0)
------------------	---	--------

naplnění vektoru jednou hodnotou

```
(define v (make-vector 10))  
(vector-fill! v 'blah)
```

v  $\Rightarrow$  #10(blah)

vytváření vektoru o dané délce s počátečním naplněním

```
(make-vector 10 'blah)  $\Rightarrow$  #10(blah)
```

získání hodnoty podle indexu / mutace hodnoty

```
(define v (vector 'a 'b 'c 'd 'e 'f))
```

```
(vector-ref v 2)  $\Rightarrow$  c
```

```
(vector-set! v 2 'blah)
```

```
(vector-ref v 2)  $\Rightarrow$  blah
```

v  $\Rightarrow$  #6(a b blah d e f)

můžeme definovat další procedury, třeba:

```
;; build-vector (analogicky jako build-list)
(define build-vector
  (lambda (len f)
    (let ((new-vector (make-vector len)))
      (let iter ((i 0))
        (if (>= i len)
            new-vector
            (begin
              (vector-set! new-vector i (f i))
              (iter (+ i 1))))))))
```

## Časová složitost práce se seznamy a vektory:

- „stejné operace“ mají jinou složitost

build-list	$O(n)$	build-vector	$O(n)$
car	$O(1)$	vector-car	$O(1)$
cdr	$O(1)$	vector-cdr	$O(n)$
cons	$O(1)$	cons-vector	$O(n)$
length	$O(n)$	vector-length	$O(1)$
list-ref	$O(n)$	vector-ref	$O(1)$
list-set!	$O(n)$	vector-set!	$O(1)$
map	$O(n)$	vector-map	$O(n)$