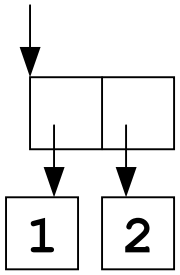
Други деструктивни процедури в езика Scheme

Представяне на списъците в Scheme. Списък на свободната памет

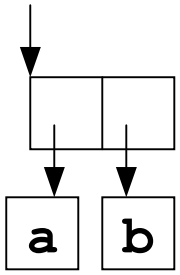
Общ принцип. Всеки обект (всеки S-израз) се представя като указател към някаква клетка. Клетката, съответна на даден примитивен обект (т.е. атом - символен атом, число и т.н.), съдържа представянето на този обект. Клетката, съответна на дадена точкова двойка (такава клетка често се нарича ***cons*** клетка), съдържа двойка указатели към ***car*** и ***cdr*** на тази точкова двойка.

Примери

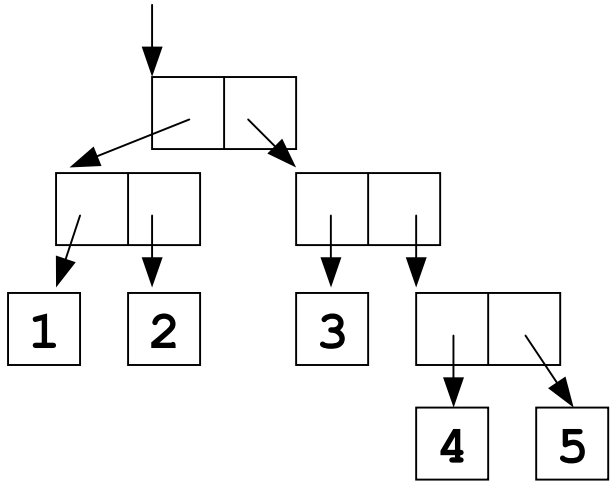
(1.2)



(a.b)

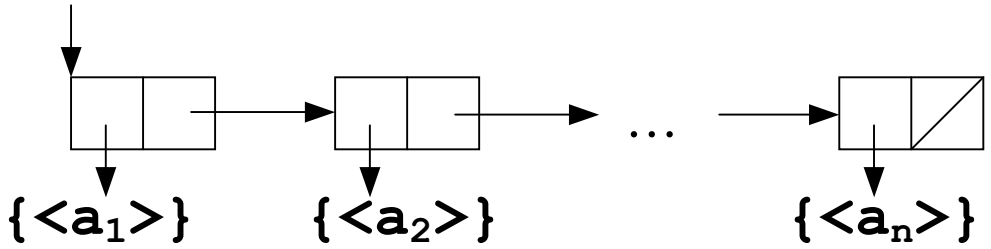


((1.2).(3.(4.5)))



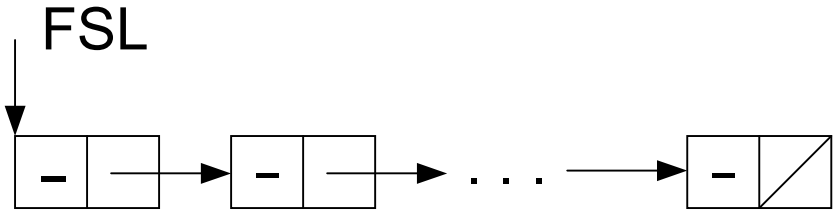
Забележка. Представянията на всички използвани до момента (в даден сеанс) атоми се съхраняват в единствени екземпляри, към които евентуално сочат много указатели. С други думи, интерпретаторът поддържа специална таблица, в която съхранява представянията на всички използвани атоми (или поне представянията на използваните символни атоми). Тази таблица често се нарича таблица на атомите (***таблица на символите***). В нея за всеки символен атом при първото му срещане (използване, цитиране) се отделя специално място (специален ред). След това при всяко използване на даден символен атом като елемент на някакъв S-израз в представянето на S-израза участва указател към реда от таблицата на символите, съответен на този символен атом.

Представянето на списъците е следствие от дефиницията на понятието "списък". По-точно, списъкът **(<a_1> <a_2> ... <a_N>)** се представя както следва:



Забележка. Атомът ***nil*** (***null***) обикновено се представя чрез специален запис (например шестнадесетична нула) в съответното поле за указател.

За създаване на нови точкови двойки (в частност, на нови списъци) в Lisp се използва т. нар. ***списък на свободната памет*** (Free Storage List, FSL). Това е списък от неизползвани ***cons*** клетки (клетки, предназначени за записване в тях на двойки указатели, т.е. клетки, предназначени за представяне на точкови двойки). Към този списък сочи специален указател и при необходимост (при създаване на нов списък или нова точкова двойка) се използват първите клетки от този списък, като указателят към началото на FSL се премества по-нататък.

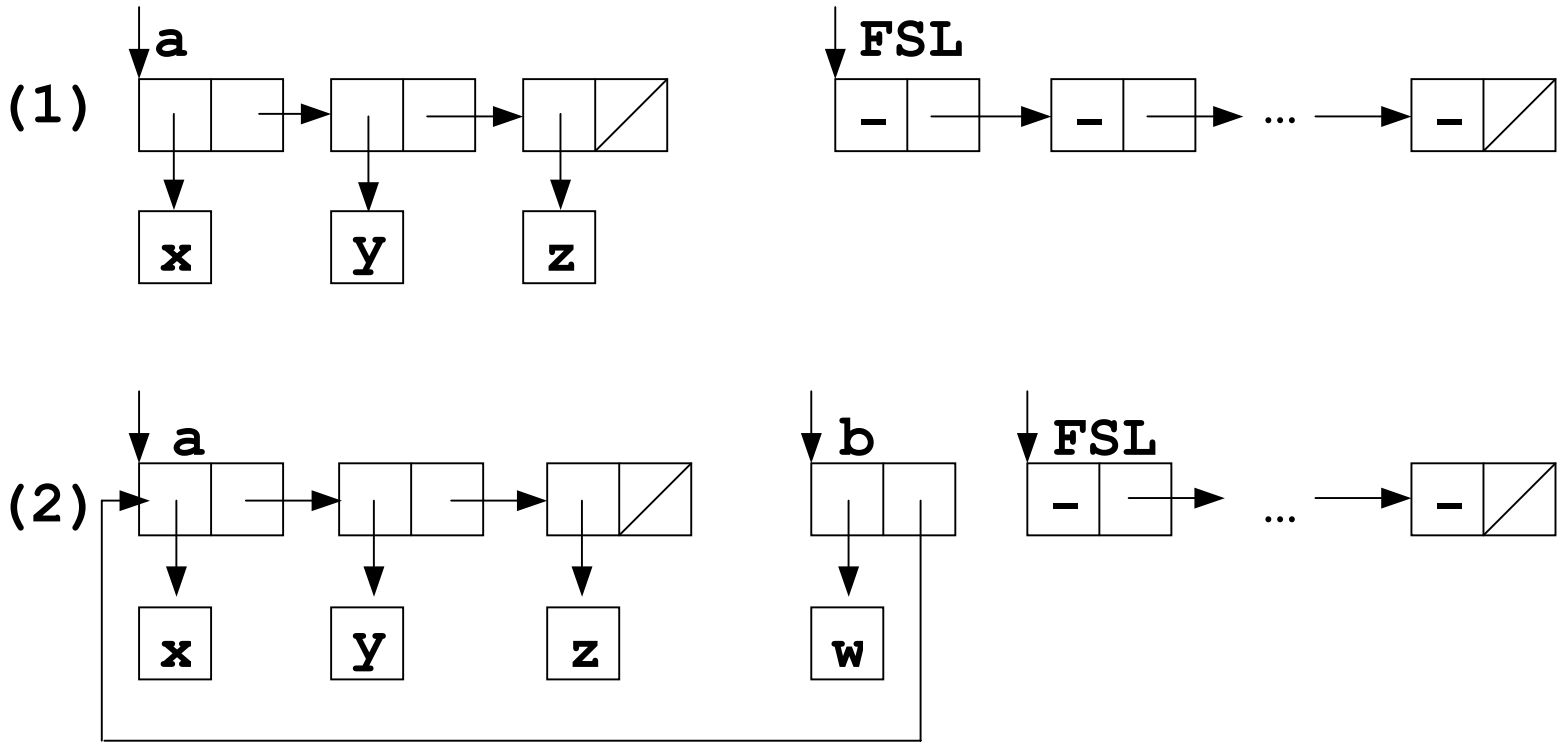


Действие на процедурите cons и append.
Деструктивна процедура append!

Процедурата ***cons*** конструира точкова двойка (списък), като за целта използва първата клетка от FSL (и премества по-нататък указателя към началото на FSL).

Пример

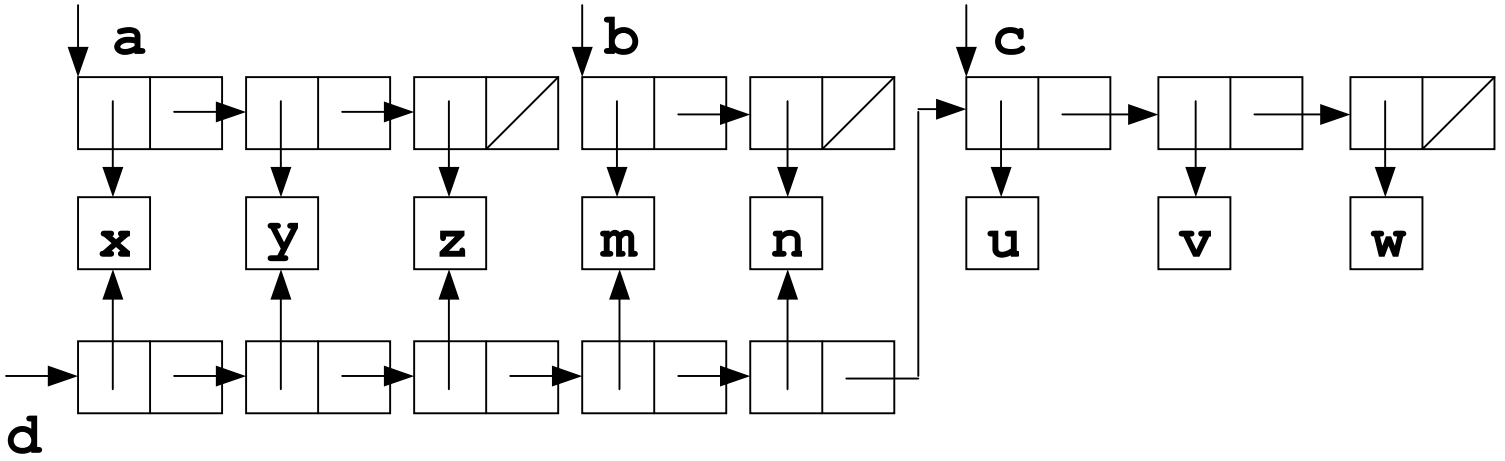
```
(1) (define a '(x y z))
(2) (define b (cons 'w a))
```


(1)
a
x
y
z
FSL
-
-
...
-
(2)
a
x
y
z
b
w
FSL
-
...
-

Вградената процедура ***append*** копира всичките си аргументи (освен последния), като в построените копия заменя крайния ***nil*** с указател към копието на следващия аргумент.

Пример

```
(define a '(x y z))
(define b '(m n))
(define c '(u v w))
(define d (append a b c))
```


a
b
c
x
y
z
m
n
u
v
w
d

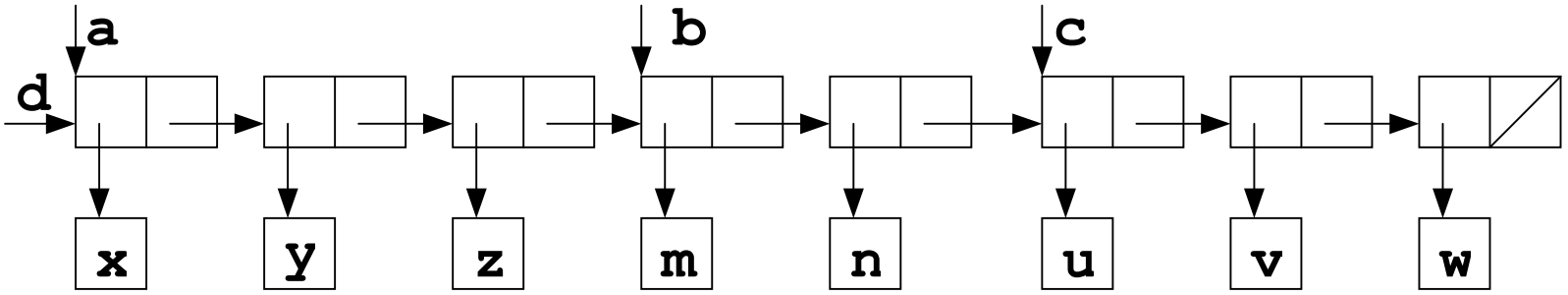
Следователно, ***append*** е сравнително тежка (ресурсоемка) операция, при реализацията на която се правят копия на всички аргументи освен последния. Така, с цел да се запазят (да не се променят) аргументите на ***append***, при реализацията й се губят много време и памет.

Забележка. Ако вместо **(define d (append a b c))** в горния пример бъде оценен само изразът **(append a b c)**, то към конструираната в резултат на оценяването списъчна структура няма да сочи никой от валидните указатели към стойност на някакво име в текущата среда (и съставящите я ***cons*** клетки ще имат статут на неизползвани в текущата среда). Така тази структура ще се превърне в боклук (garbage).

В процеса на работата на интерпретатора на Scheme се натрупва много боклук, който при необходимост се събира с помощта на специална вградена процедура – ***garbage collector*** (***gc***). Процедурата ***gc*** действа на два етапа. На първия етап тя маркира всички използвани ***cons*** клетки, а на втория добавя немаркираните ***cons*** клетки към FSL.

Понякога не е необходимо непременно да се запазват (да не се променят) аргументите на ***append***. В такива случаи вместо ***append*** може да се използва вградената (примитивната) процедура ***append!***. Както се вижда от името ѝ, ***append!*** е деструктивна процедура - тя променя част от аргументите си (всички освен последния), като във всеки от тях заменя крайния ***nil*** с указател към следващия аргумент. Така се спестяват време и памет, но се получават странични ефекти.

Реализация на горния пример с използване на ***append!*** в дефиницията на **d**:



Следователно, след изпълнението на **(define d (append! a b c))** използваните променливи имат следните стойности:

```
a ──> (x y z m n u v w),
b ──> (m n u v w),
c ──> (u v w),
d ──> (x y z m n u v w).
```

Процедурата set-car!

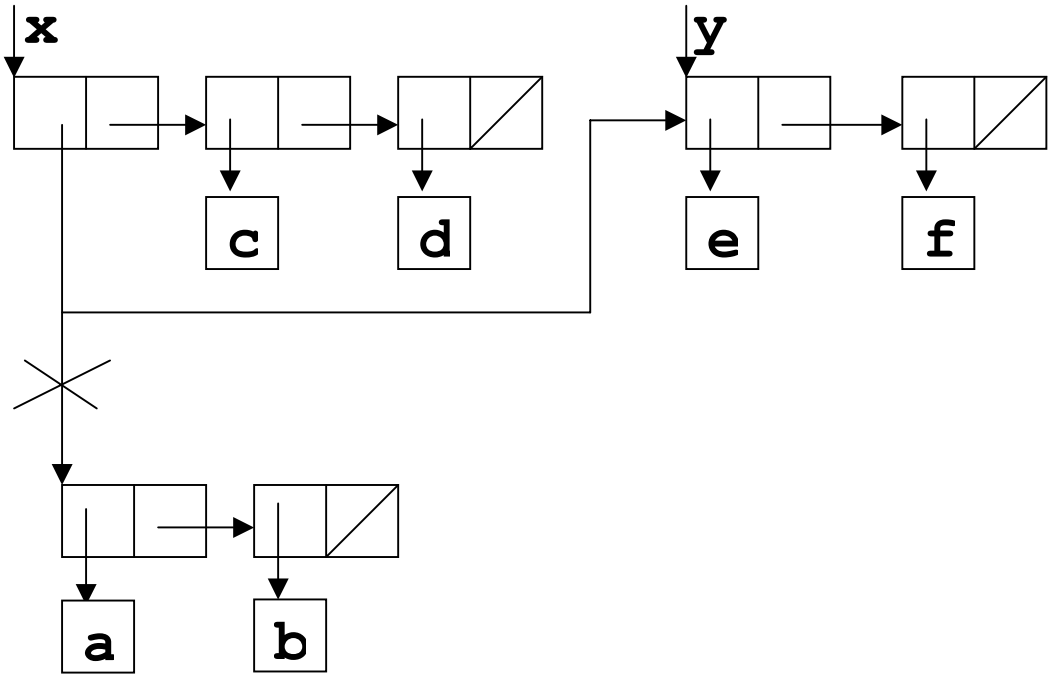
Има два аргумента. Първият аргумент задължително трябва да бъде точкова двойка (в частност, непразен списък). Процедурата ***set-car!*** модифицира първия си аргумент, като в него заменя съществуващия указател към ***car***-а с указател към втория аргумент. По-точно, в резултат на обръщението **(set-car! x y)** се променя **[x]** (**[x]** трябва да е точкова двойка), като съществуващият указател към ***car***-а на **[x]** се заменя с указател към **[y]**.

Оценката на обръщението към ***set-car!*** по стандарт е неопределена. В използваната реализация съвпада с новата стойност на първия аргумент (тази, която се получава след изпълнението на ***set-car!***).

Пример

```
(define x '((a b) c d))
(define y '(e f))
(set-car! x y)
```

Графично представяне:



Следователно, след изпълнението на обръщението към процедурата ***set-car!*** променливата **x** има стойност **((e f) c d)**.

Процедурата set-cdr!

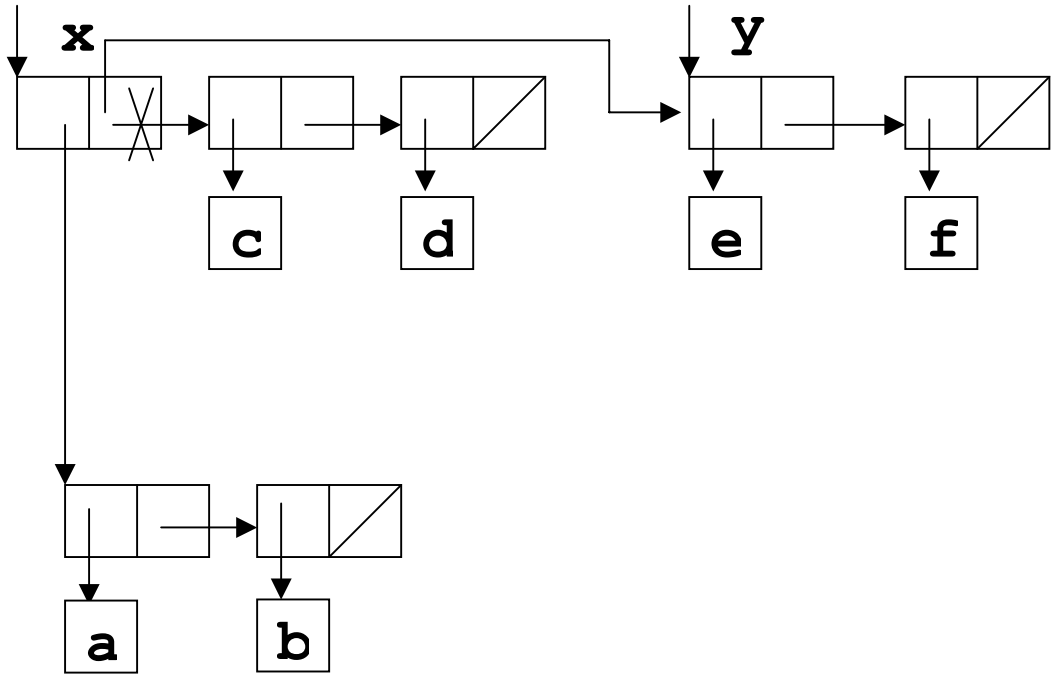
Обръщението към нея има същия вид и същите аргументи, както и обръщението към ***set-car!***, но в резултат на действието ѝ указателят към ***cdr***-а на първия аргумент се заменя с указател към втория аргумент. По-точно, в резултат на обръщението **(set-cdr! x y)** се променя **[x]** (и тук **[x]** задължително трябва да е точкова двойка, в частност – непразен списък), като съществуващият указател към ***cdr***-а на **[x]** се заменя с указател към **[y]**.

Оценката на обръщението към ***set-cdr!*** по стандарт е неопределена. В използваната реализация съвпада с новата стойност на първия аргумент (тази, която се получава след изпълнението на ***set-cdr!***).

Пример

```
(define x '((a b) c d))
(define y '(e f))
(set-cdr! x y)
```

Графично представяне:



Следователно, след изпълнението на обръщението към процедурата ***set-cdr!*** променливата **x** има стойност **((a b) e f)**.

Характерни приложения на процедурата set-cdr!

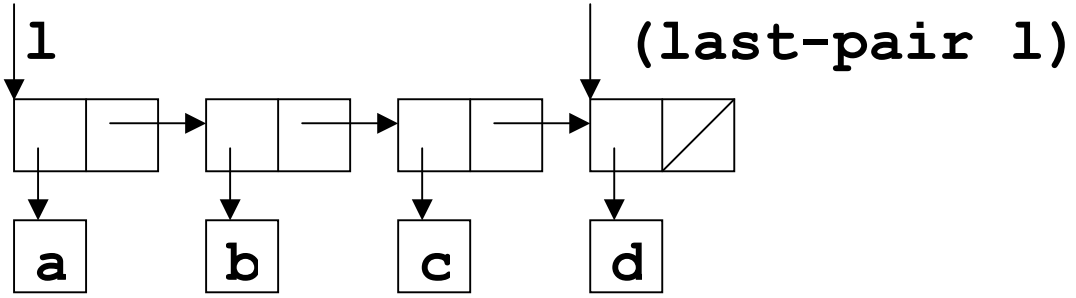
Преди да разгледаме някои характерни приложения на процедурата ***set-cdr!***, ще въведем още една (недеструктивна) примитивна процедура - процедурата ***last-pair***.

Общ вид и оценка на обръщението към ***last-pair***:

(last-pair l) ⟶ последната точкова двойка на списъка **[l]**, т.е. списък, съдържащ последния елемент на **[l]** (**[l]** задължително трябва да бъде списък). Оценката на **(last-pair '())** е ***()***.

Графично представяне

Нека **l ⟶ (a b c d)**. Тогава **(last-pair l) ⟶ (d)**.

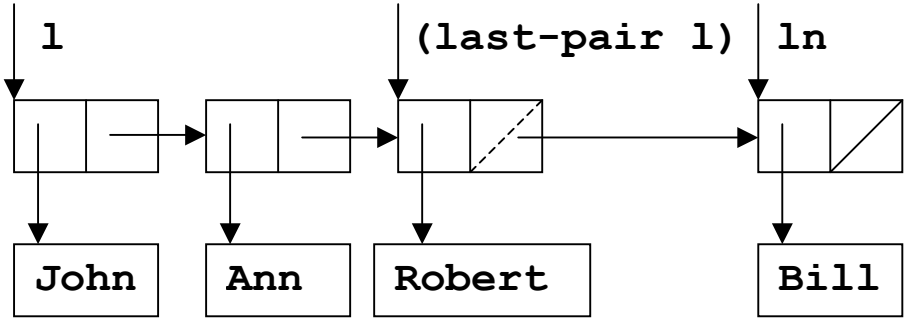


Деструктивно (физическо) добавяне на елемент в края на даден списък

Пример

```
> (define l '(John Ann Robert))
l
> (define ln '(Bill))
ln
> (set-cdr! (last-pair l) ln)
(Robert Bill)
> l
(John Ann Robert Bill)
```

Графично представяне:



Извод. Ако искаме да добавим в края на списъка **[l]** нов елемент, който е стойност на **el**, това може да стане посредством конструкцията:

```
(set-cdr! (last-pair l) (list el)).
```

Дефиниране на деструктивна процедура
за добавяне на елемент към непразен асоциативен списък

```
(define (insert! key value a-list)
   (let ((found (assq key a-list)))
      (if (null? found)
          (set-cdr! a-list
                    (cons (cons key value)
                          (cdr a-list)))
          (set-cdr! found value))
      a-list))
```

Конструиране на зациклени списъци

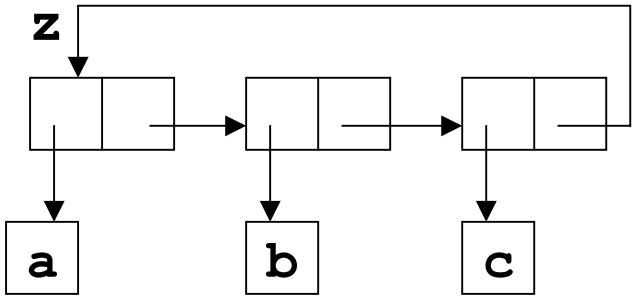
Пример. Нека разгледаме следната процедура:

```
(define (make-cycle l)
   (set-cdr! (last-pair l) l)
   l)
```

Нека е дефинирана още и променливата **z**:

```
(define z (make-cycle '(a b c)))
```

Графична илюстрация:



Примерни процедури за работа със зациклени списъци

Пример 1. Дефиниция на функция, която проверява дали в даден списък се съдържа цикъл

```
(define (cycled? l)
   (let ((elements (list l)))
      (define (traverse x)
         (cond ((atom? x) #f)
               ((memq (car x) elements) #t)
               ((memq (cdr x) elements) #t)
               (else (set! elements
                           (cons (car x)
                                 (cons (cdr x) elements)))
                     (or (traverse (car x))
                         (traverse (cdr x))))))
      (traverse l)))
```

Пример 2. Дефиниция на функция, която намира броя на ***cons*** клетките, участващи в представянето на даден списък

```
(define (count-pairs l)
   (let ((pairs (list l)))
      (define (count x)
         (if (atom? x)
             0
             (+ (cond ((memq (car x) pairs) 0)
                      (else (set! pairs
                                  (cons (car x) pairs))
                            (count (car x))))
                (cond ((memq (cdr x) pairs) 0)
                      (else (set! pairs
                                  (cons (cdr x) pairs))
                            (count (cdr x))))
                1)))
      (count l)))
```

Допълнителни странични ефекти при използване на поделена памет

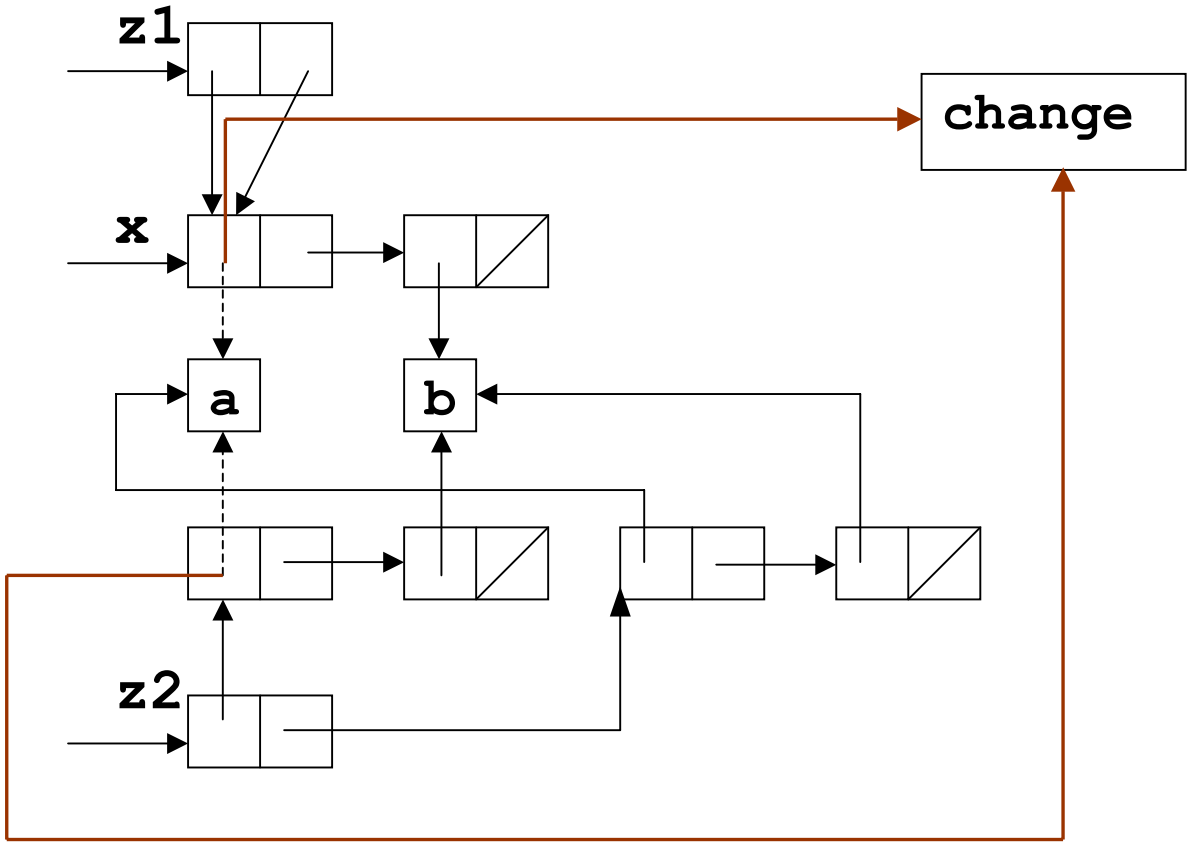
Идея. В много случаи се използват такива дефиниции на променливи, които водят до вътрешни представяния на съответните стойности, които включват едни и същи (т.е. общи) участъци от паметта. В такива случаи, ако се извърши деструктивна (физическа, "хирургическа") промяна в едната стойност, то автоматично като страничен ефект се променя и другата стойност.

Пример

```
> (define x (list 'a 'b))
x
> (define z1 (cons x x))   ;;; z1 се дефинира чрез x
z1
> (define z2 (cons (list 'a 'b) (list 'a 'b)))
z2
> z1
((a b) a b)
> z2
((a b) a b)
> (define (set-to-change! l) (set-car! (car l) 'change) l)
set-to-change!
> (set-to-change! z1)
((change b) change b)
```

```
> z1
((change b) change b)
> z2
((a b) a b)
> (set-to-change! z2)
((change b) a b)
> z2
((change b) a b)
> x
(change b)
```

Графична илюстрация:



Обяснение на действието на процедурите за проверка на равенство eq? и equal? върху символни атоми и списъци

Точната формулировка на оценката на обръщението към ***eq?*** и ***equal?*** е следната:

`(eq? s1 s2)` ⟶

- ***#t***, ако **[s1]** и **[s2]** са идентични (т.е. ако **[s1]** и **[s2]** са означения на един и същ участък от паметта);
- ***()***, в противния случай.

`(equal? s1 s2)` ⟶

- ***#t***, ако **[s1]** и **[s2]** са еквивалентни S-изрази (т.е. или са еднакви символни атоми, или са еднакви низове, или са равни числа от един и същ тип, или са точкови двойки (в частност, списъци) с еквивалентни ***car*** и ***cdr***);
- ***()***, в противния случай.

От отбелязаното по-горе следва, че ако **[s1]** и **[s2]** са еднакви символни атоми (а се оказва, че същото се отнася и за символните низове), то те са равни както в смисъла на предиката ***equal?***, така и в смисъла на предиката ***eq?***.

Както вече беше отбелязано, за сравняване на числа по принцип е добре да се използва числовият предикат **=** или предикатът ***eqv?*** (този предикат обединява действието на **=** и ***eq?***).

Ако **[s1]** и **[s2]** са еднакви точкови двойки (в частност, непразни списъци), т.е. точкови двойки с еднакви ***car*** и ***cdr***, то те са равни в смисъла на ***equal?***, но в повечето случаи не са равни в смисъла на ***eq?***.

Пример, обясняващ действието на ***equal?*** и ***eq?*** върху списъци:

```
> (define l1 (list 'a 'b 'c))
l1
> (define l2 (list 'a 'b 'c))
l2
> (define l3 l2)
l3
> (equal? l1 l2)
#t
> (equal? l2 l3)
#t
> (eq? l1 l2)
()
> (eq? l2 l3)
#t
```

Графична илюстрация: