ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

Магдалина Тодорова
magda@fmi.uni-sofia.bg
todorova_magda@hotmail.com
кабинет 517, ФМИ

Тема 10

Деструктивни процедури за работа със списъци. Мутиращи списъци. Еквивалентности и поделяне на обекти. Представяне и реализация на опашка

1. Примитивни мутатори на списъкови структури

Примитивна процедура set-car!

Синтаксис:

(set-car! x y)

където

- x е израз, чиято оценка е точкова двойка;
- y е произволен израз.

Семантика:

Оценяват се изразите x и y. Оценката на set-car!-израза е неопределена. Като страничен ефект, car-ът на [x] се разрушава и се заменя с [y].

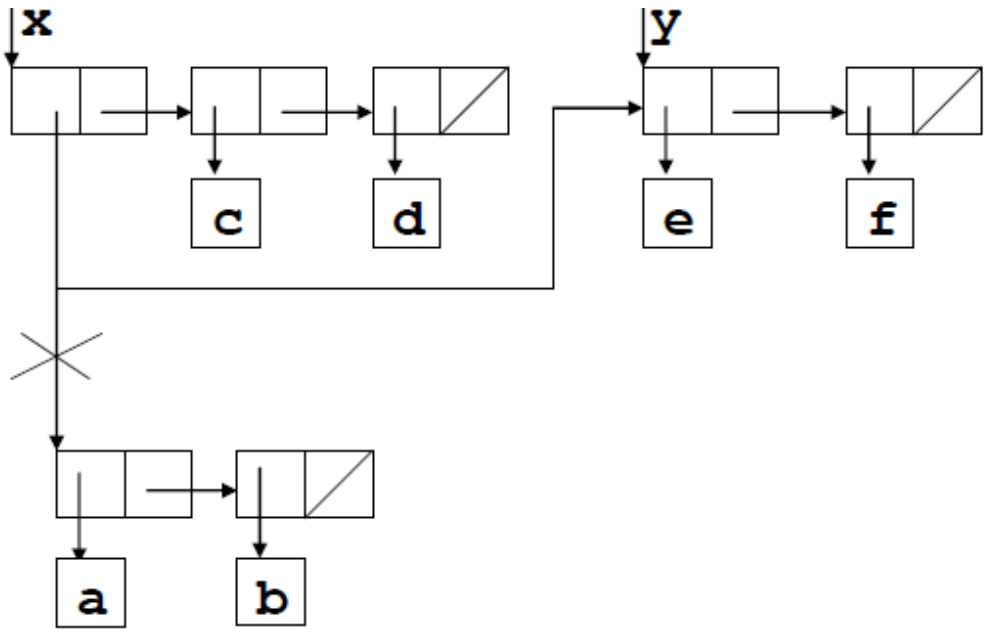
1. Примитивни мутатори на списъкови структури

Забележка: В много реализации оценката на set-car!-израз съвпада с новата стойност на първия аргумент (тази, която се получава в резултат от изпълнението на ***set-car!***).

1. Примитивни мутатори на списъкови структури

Пример:

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



1. Примитивни мутатори на списъкови структури

Примитивна процедура set-cdr!

Синтаксис:

(set-cdr! x y)

където

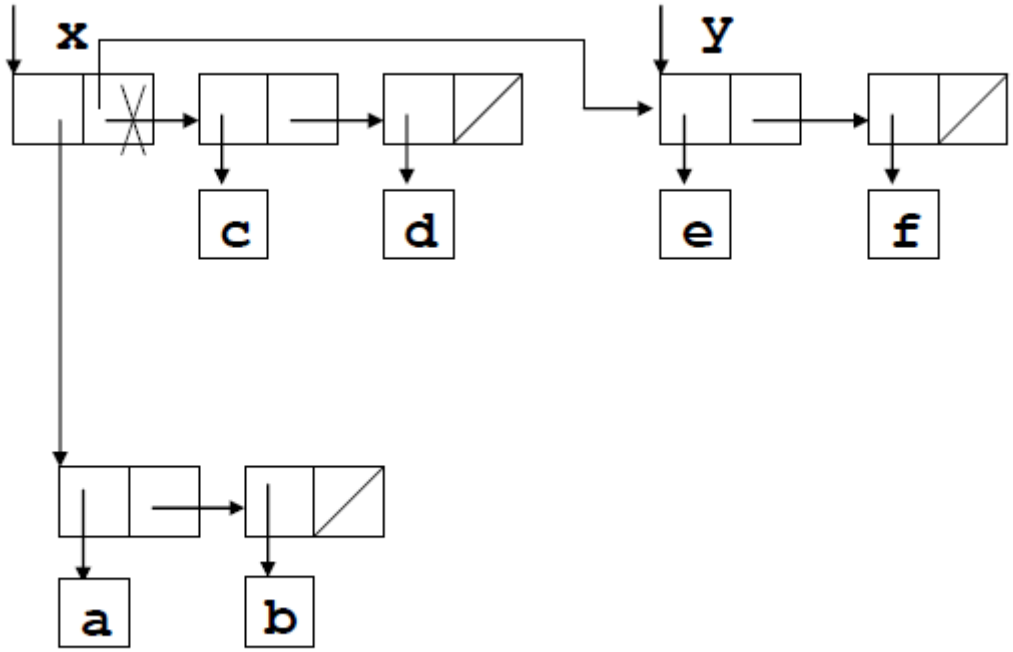
- x е израз, чиято оценка е точкова двойка;
- y е произволен израз.

Семантика:

Оценяват се изразите x и y. Оценката на set-cdr!-израза е неопределена. Като страничен ефект, cdr-ът на [x] се разрушава и се заменя с [y].

1. Примитивни мутатори на списъкови структури

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



2. Приложения на set-car! и set-cdr!

➢ Деструктивно конкатениране на списъци

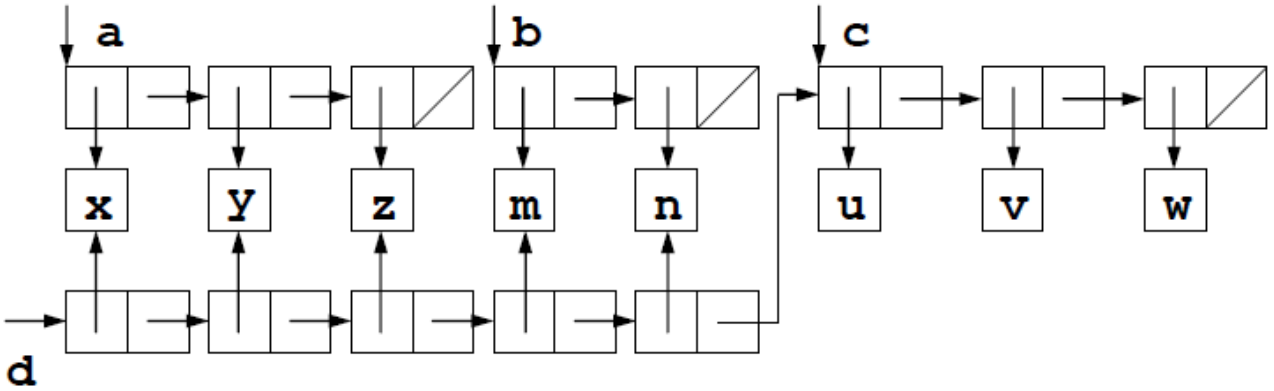
Вградената в DrRacket процедура ***append*** е ***недеструктивна.***

Тя копира всичките си аргументи (освен последния) на друго място в паметта, като в построените копия заменя записа за край на списък с указател към копието на следващия аргумент.

2. Приложения на set-car! и set-cdr!

Пример:

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



2. Приложения на set-car! и set-cdr!

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!***.

Процедура ***append!*** е деструктивна – променя всичките си аргументи освен последния, като във всеки от тях заменя записа за край на списък с указател към следващия аргумент.

Така се спестяват време и памет, но се получават странични ефекти.

Ще реализираме append!.

2. Приложения на set-car! и set-cdr!

Задача.

Нека x и y са дадени списъци, като **x не е празен**. Да се дефинира процедура append2!, която конкатенира x с y, като разрушава cdr-а на последната двойна кутия, представяща списъка x.

2. Приложения на set-car! и set-cdr!

```
(define (last x)                    ; x не е празен списък
    (if (null? (cdr x)) x
        (last (cdr x))))

(define (append2! x y)
    (set-cdr! (last x) y)
    x)

(define (append! . L)
   (accumulate append2! '() L))
```

2. Приложения на set-car! и set-cdr!

➢Изтриване на елемент от списък

Задача.

Даден е непразен списък L. Да се дефинира деструктивна процедура, която изключва n-тия елемент на L.
Предполага се, че n-тият елемент на списъка съществува.

Опит за решение:

```
(define (delete! L n)
    (if (= n 1)
        (begin (set! L (cdr L)) L)
        (begin (set-cdr! (n_sub_list (- n 1) L)
                         (cdr (n_sub_list n L)))
               L)))
```

2. Приложения на set-car! и set-cdr!

Процедурата (n_sub_list n L) намира указател към n-тия елемент на непразния списък L. Предполага се, че n-тият елемент на L съществува.

```
(define (n_sub_list n L)
    (if (= n 1) L
        (n_sub_list (- n 1) (cdr L))))
```

2. Приложения на set-car! и set-cdr!

Експерименти с delete!

```
>(define d  '(1 2 3 4))
>d
(1 2 3 4)

>(delete! d 2)
(1 3 4)
>d
(1 3 4)
```

2. Приложения на set-car! и set-cdr!

Експерименти с delete!

```
>(delete! d 1)
(3 4)

>d
(1 3 4)
```

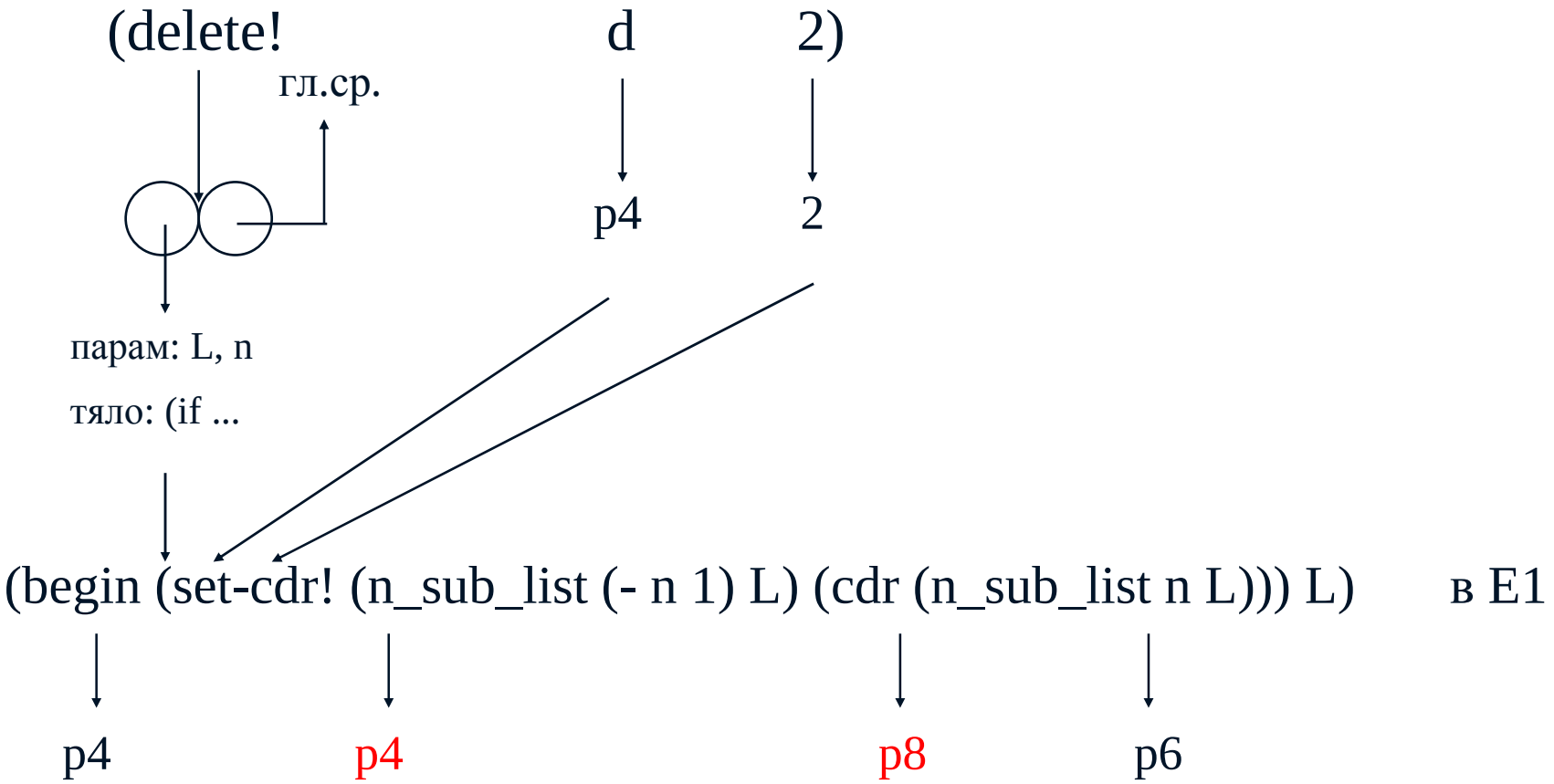
В този случай, обръщението към delete! връща правилен резултат, но променливата d не се е променила.

2. Приложения на set-car! и set-cdr!

парам: L, n
тяло: (if (= n 1)
(begin (set! L (cdr L)) L)
(begin (set-cdr! (n_sub_list (- n 1) L)
(cdr (n_sub_list n L)))
L))

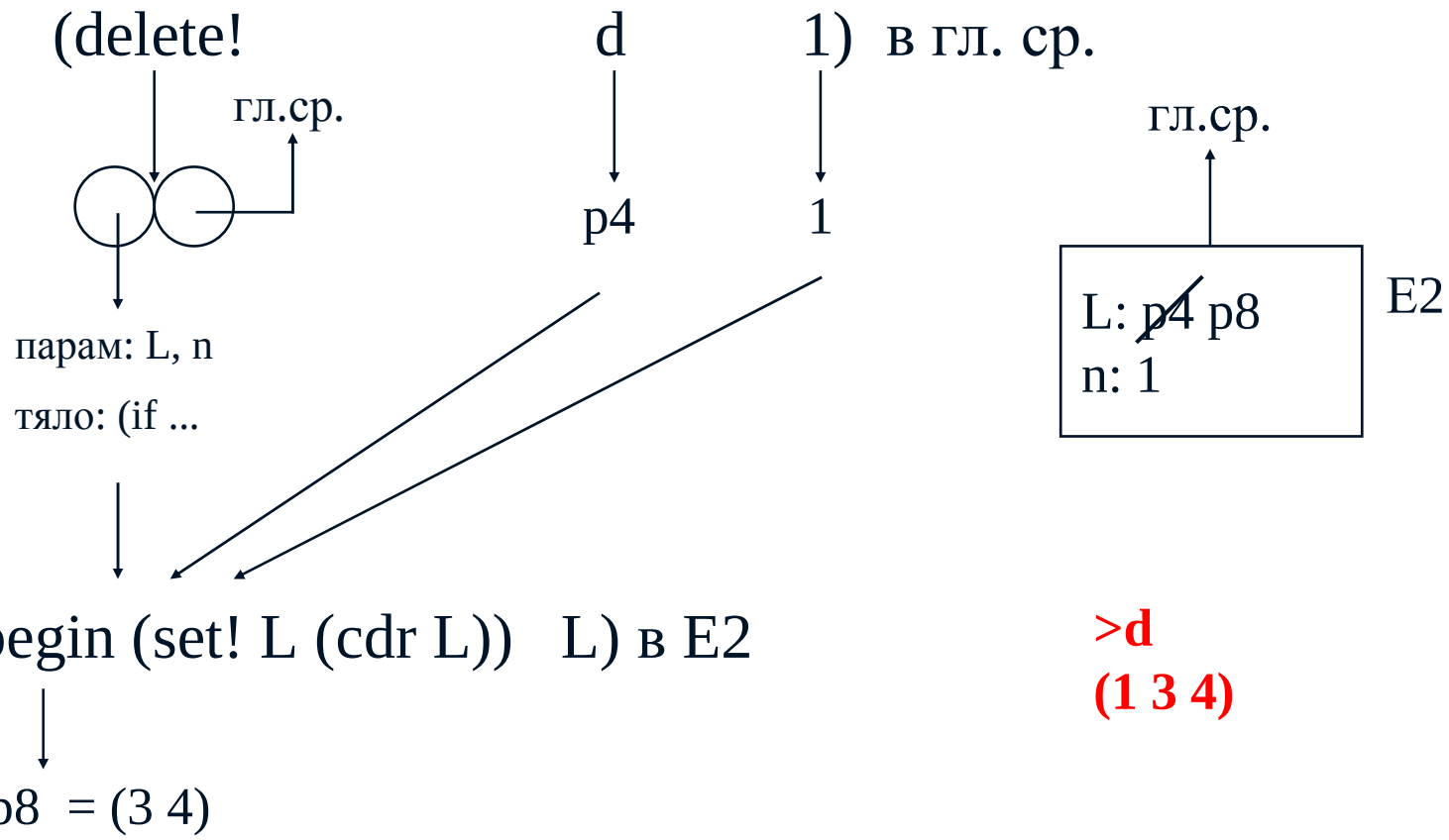
2. Приложения на set-car! и set-cdr!

	0	1	2	3	4	5	6	7	8	9	…
car-					n1	n4	n2		n3		…
cdr-					p6	e0	p8		p5		…



2. Приложения на set-car! и set-cdr!

	0	1	2	3	4	5	6	7	8	9	…
car-					n1	n4	n2		n3		…
cdr-					p8	e0	p8		p5		…



2. Приложения на set-car! и set-cdr!

За да се изтрие първият елемент на списъка, в конкретния случай може да се прекопира втората двойка, представяща списъка, в първата, т.е. да се замени

(set! L (cdr L))

с

(set-car! L (cadr L))

(set-cdr! L (cddr L))

Последното е възможно, ако списъкът има повече от един елемент.

А ако списъкът има точно един елемент?

2. Приложения на set-car! и set-cdr!

Препоръка:

При деструктивни операции над списък пред първия елемент на списъка да се постави фиктивен елемент (сентинел).

Тази техника ще приложим при работа с: опашка и асоциативен списък.

3. Еквивалентности

В лиспоподобните езици за програмиране за сравнение на обекти за еквивалентност са въведени понятията:

- *идентичност*,
- *операционна еквивалентност* и
- *равенство*.

3. Еквивалентности

а) ***Идентичност на обекти***

Дефиниция:

• два символа са идентични, ако се състоят от едни и същи знаци, записани в един и същ ред, т.е. ако се печатат по един и същ начин;

• две числа с една и съща стойност могат да са, но могат и да не са идентични. Това зависи от реализацията на езика. В повечето реализации на езика са идентични само целите числа от определен диапазон;

• ***в останалите случаи два обекта са идентични, ако се представят вътрешно чрез един и същ указател***.

3. Еквивалентности

Примитивен предикат eq?

Синтаксис:

(eq? obj1 obj2)

където

- eq? е специален символ;
- obj1 и obj2 са произволни обекти.

Семантика:

Оценяват се obj1 и obj2. Обръщението към eq? връща #t, ако [obj1] и [obj2] са идентични и #f – в противен случай.

3. Еквивалентности

За по-лесно определяне на идентичността на два обекта могат да се използват следните ***правила***:

• два обекта от различни типове не са идентични;

• два обекта от един и същ тип, но с различни оценки не са идентични;

• #t е идентично на себе си, #f е идентично на себе си и () е идентичен на себе си. (В стандарта на езика Scheme не е указано дали #f и () са идентични или не);

• два символа, състоящи се от едни и същи знаци, записани в един и същ ред, т.е. печатат се по един и същ начин, са идентични;

3. Еквивалентности

• два низа, състоящи се от едни и същи знаци, записани в един и същ ред, може да са, но може и да не са идентични (зависи от реализацията);

•два обекта, създадени по различно време чрез един и същ cons, list, let, let* или lambda-израз не са идентични;

• две процедури, създадени чрез един и същ lambda-израз по едно и също време са идентични.

3. Еквивалентности

Примери:

```
(eq? 'a 3)                  #f
(eq? "hi" '(hi))            #f
(eq? 312 312)               #t
(eq? 999999 999999)         #f
(eq? #t #t)                 #t
(eq? #t #f)                 #f
(eq? 'ab 'ab)               #t
(eq? "a" "a")               #f      ; в DrRacket е #t
(eq? "abcd" "abcd")         #f      ; в DrRacket е #t
(eq? (= 3 3) (null? '()))   #t
```

3. Еквивалентности

```
(define x (list 1 2 3))
(eq? x x)                       #t
(define f (lambda (x) x))
(eq? f f)                       #t
(eq? car car)                   #t
(eq? (cons 1 2) (cons 1 2))     #f
(let ((f (lambda (x) x))
      (g (lambda (x) x)))
  (eq? f g))                    #f
```

3. Еквивалентности

б) ***Операционна еквивалентност***

Операционната еквивалентност е обобщение на идентичността. Следните ***правила*** определят операционната еквивалентност на обекти:

- два идентични обекта са операционно еквивалентни;
- два обекта от различен тип не са операционно еквивалентни;
- два числови обекта са операционно еквивалентни, ако са равни в смисъла на =;

3. Еквивалентности

• два низа, състоящи се от едни и същи знаци, записани в един и същ ред, са операционно еквивалентни;

• два мутиращи обекта са операционно еквивалентни, ако са идентични;

• две процедури са операционно еквивалентни, ако имат едни и същи аргументи, връщат едни и същи оценки и извършват едни и същи действия (имат едни и същи странични ефекти).

За установяване на операционната еквивалентност на обекти се използва предикатът eqv?

3. Еквивалентности

Примитивен предикат eqv?

Синтаксис:

(eqv? obj1 obj2)

където

- eqv? е специален символ;
- obj1 и obj2 са произволни изрази.

Семантика:

Оценяват се obj1 и obj2. Оценката на eqv?-израза е #t, ако [obj1] и [obj2] са операционно еквивалентни обекти и #f – в противен случай.

3. Еквивалентности

Забележка: За повечето реализации на езика, разликата между идентичността и операционната еквивалентност е единствено в сравняването на числа и низове.

Примери:

```
(eqv? 'a 3)                    #f
(eqv? "hi" '(hi))              #f
(eqv? 999999 999999)           #t
(eqv? #t #t)                   #t
(eqv? 'ab 'ab)                 #t
(eqv? "abcd" "abcd")           #t
(eqv? "abcd" "abce")           #f
```

3. Еквивалентности

Примери:

```
(eqv? (= 3 3) (null? '()))          #t
(eqv? car car)                      #t
(eqv? (cons 1 2) (cons 1 2))        #f
(let ((f (lambda (x) x))
      (g (lambda (x) x)))
  (eqv? f g))                       #f
```

3. Еквивалентности

в) ***Равенство***

Дефиниция: Два обекта са равни, ако имат една и съща структура и съдържание, т.е. ако се печатат по един и същ начин.

Чрез предиката equal? се установява дали два обекта са равни.

На практика, предикатът equal? най-често се използва за сравняване на списъци. Добрата дефиниция на equal? трябва да допуска две числа да са equal? в смисъл на =.

3. Еквивалентности

Примитивен предикат equal?

Синтаксис:

(equal? obj1 obj2)

където

- equal? е специален символ;
- obj1 и obj2 са произволни изрази.

Семантика:

Оценяват се obj1 и obj2. Оценката на equal?-израза е #t, ако [obj1] е равно на [obj2] и е #f – в противен случай.

3. Еквивалентности

Примери:

```
(equal? "hi" '(hi))                          #f
(equal? '(a b) '(a b))                       #t
(equal? 'a 'b)                               #f
(equal? 3156 3156)                           #t
(equal? 3.4 (+ 3.0 0.4))                     #t
(equal? (cons 'a 'b) (cons 'a 'b))           #t
(equal? "hi" "hi")                           #t
(equal? car car)                             #t
(let ((f (lambda (x) x)))
   (equal? f f))                             #t
```

4. Поделяне на обекти

Поделяне. Обект се поделя, ако се използва едновременно от два или повече обекта.

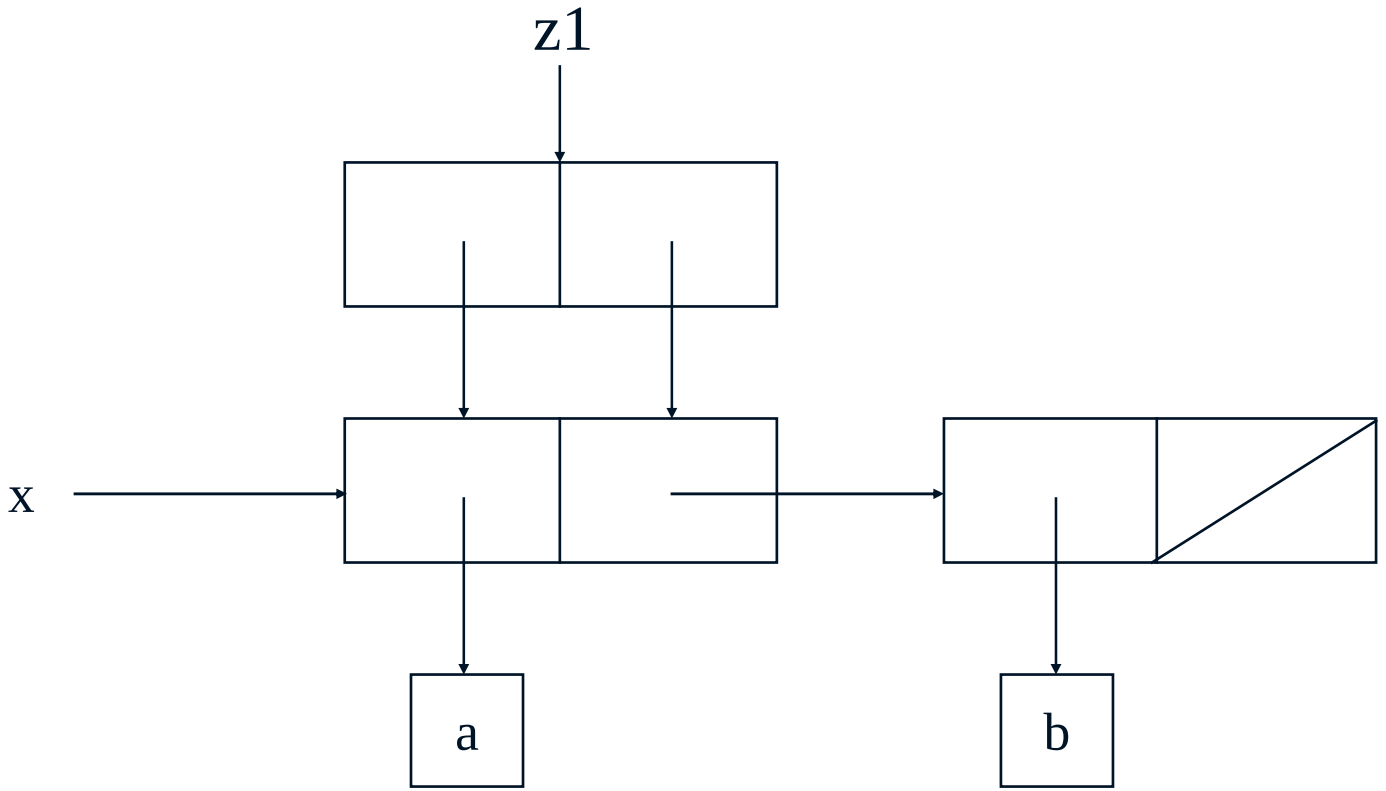
Поделянето може да е предизвикано от:

- програмиста;
- реализацията.

4. Поделяне на обекти

Пример (поделяне, предизвикано от програмиста):

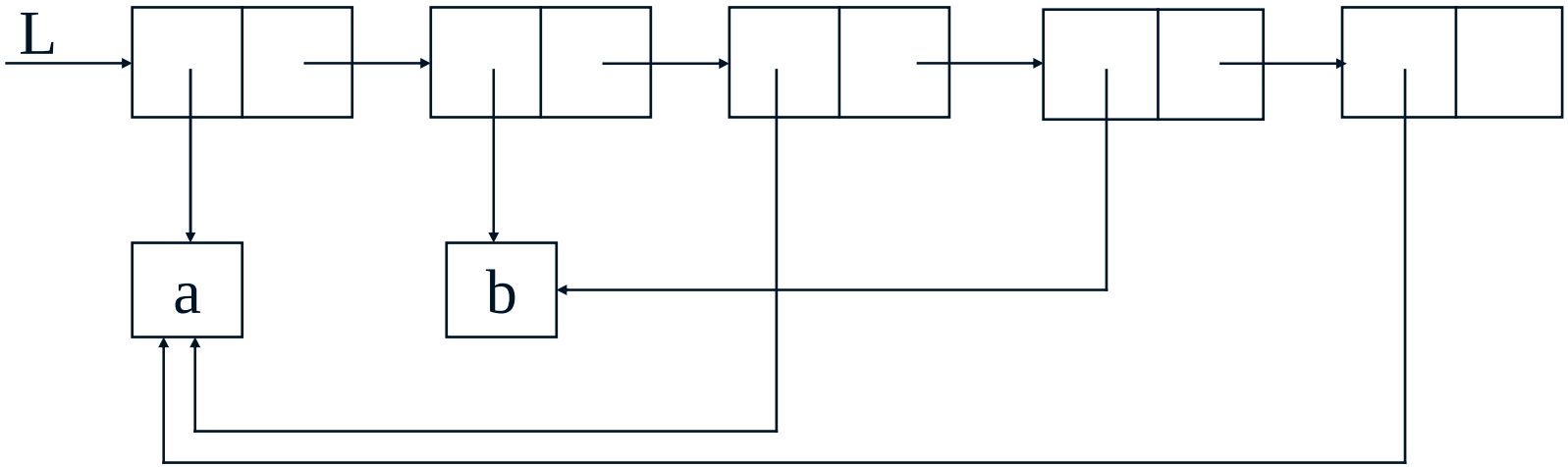
```
>(define x (list 'a 'b))
x
>(define z1 (cons x x))
z1
```



4. Поделяне на обекти

Пример (поделяне, предизвикано от реализацията):

```
>(define L '(a b a b a))
L
>L
(a b a b a)
```



4. Поделяне на обекти

Поделянето на обекти може да се установи чрез предиката eq?.

Пример:

(eq? (car z1) (cdr z1)) → #t

(eq? (car L) (caddr L)) → #t

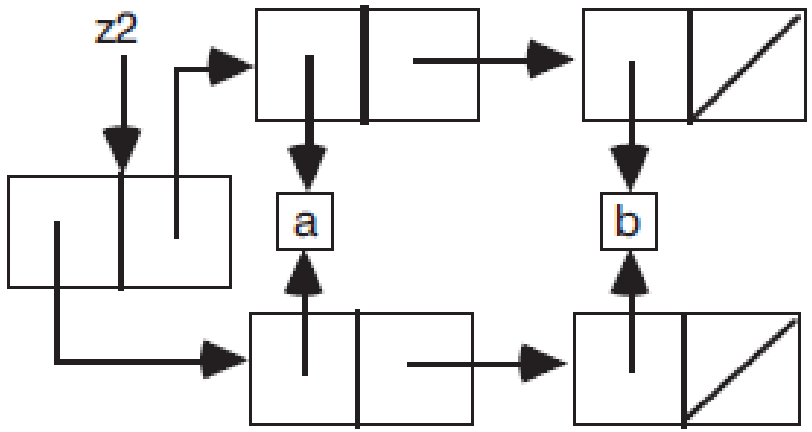
(eq? (caddr L) (cadr (cdddr L))) → #t

4. Поделяне на обекти

Нека списъкът z2 е дефиниран по следния начин:

(define z2 (cons (list 'a 'b) (list 'a 'b)))

Графичното представяне на z2 има вида:



(eq? (caar z2) (cadr z2)) #t
(eq? (cadar z2) (caddr z2)) #t

4. Поделяне на обекти

Ако се оценят z1 и z2, се получава:

>z1

((a b) a b)

>z2

((a b) a b)

Следователно, z1 и z2, разглеждани на ниво списък са равни.

(equal? z1 z2) е #t

В първия случай списъкът (a b) се поделя от car-а и cdr-а на z1, а във втория случай не е така – създадени са две копия на (a b).

4. Поделяне на обекти

Поделянето е напълно неоткриваемо, ако се работи само с примитивните процедури *car*, *cdr* и *cons*. Ако се използват мутатори, поделянето става съществено и може да предизвика проблеми.

Пример: Списъците *z1* и *z2* са равни, но извършването на една и съща деструктивна операция над тях води до получаване на различни резултати.

```
>(set-car! (car z1) 'x)
((x b) x b)
>(set-car! (car z2) 'x)
((x b) a b)
```

5. Представяне и реализация на опашка

Опашка – това е крайна редица от елементи. Операцията включване на елемент се осъществява само в единия край на редицата (края на опашката), а операцията изключване - само в другия й край (началото на опашката).

Примери:

Операция:	*Получена опашка*:
(define q (make-queue))	()
(insert-queue! q ′a)	(a)
(insert-queue! q ′b)	(a b)
(delete-queue! q)	(b)
(insert-queue! q ′c)	(b c)
(insert-queue! q ′d)	(b c d)
(delete-queue! q)	(c d)

5. Представяне и реализация на опашка

Опашката може да се разглежда като структура, дефинирана чрез следното множество от операции:

а) *конструктор*

(make-queue) – създава празна опашка

б) *селектор*

(front queue) – намира обекта, който е в началото на непразна опашка или сигнализира за грешка, ако опашката е празна.

в) *мутатори*

(insert-queue! queue item) – включва елемента *item* в опашката *queue* и връща модифицираната опашка.

(delete-queue! queue) – изключва елемент от опашката *queue* и връща модифицираната опашка.

5. Представяне и реализация на опашка

г) предикат

(empty-queue? queue) - връща #t, ако опашката *queue* е празна и #f – в противен случай.

Тъй като опашката е редица от елементи, тя може да се представи чрез списък. Началото на опашката ще бъде car-ът на списъка.

Включването на елемент в опашка е еквивалентно на конкатенирането на списъка, представящ опашката със списък, съдържащ един елемент – този, който ще се включва. Изключването на елемент от опашка означава да се вземе cdr-ът на списъка, представящ опашката.

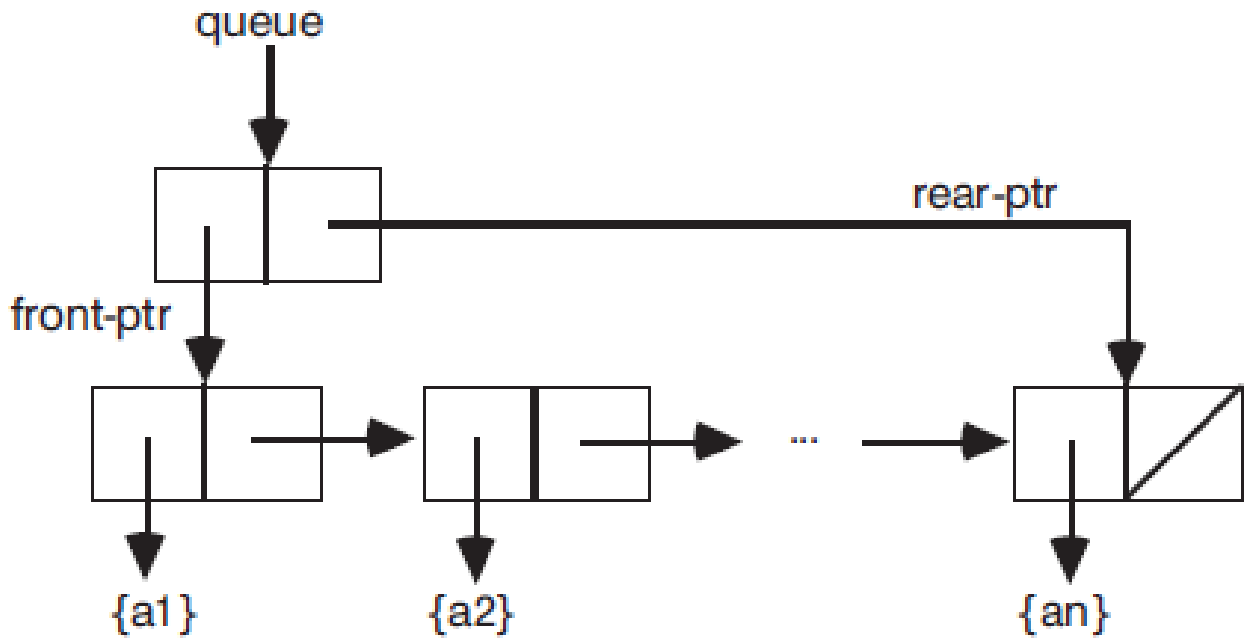
5. Представяне и реализация на опашка

Това представяне е неефективно, защото за да се включи нов елемент, трябва да се сканира списъкът, представящ опашката, докато се достигне краят му. Тъй като единственият метод за обхождане на списък е последователното прилагане на операцията *cdr*, сканирането на списък от *n* елемента ще изисква време от порядък O(n).

Освен това при изключване на елемент от опашка с един елемент, а също при включване на елемент в празна опашка, ще възникват проблеми.

5. Представяне и реализация на опашка

Модификация на представянето (добавяме *сентинел* пред първия елемент на опашката):



5. Представяне и реализация на опашка

Основни операции над опашка:

а) *Проверка дали опашка е празна*

```
(define (empty-queue? queue)
    (null? (car queue)))
```

б) *Конструктор на опашка*

```
(define (make-queue)
    (cons '() '()))
```

5. Представяне и реализация на опашка

Основни операции над опашка:

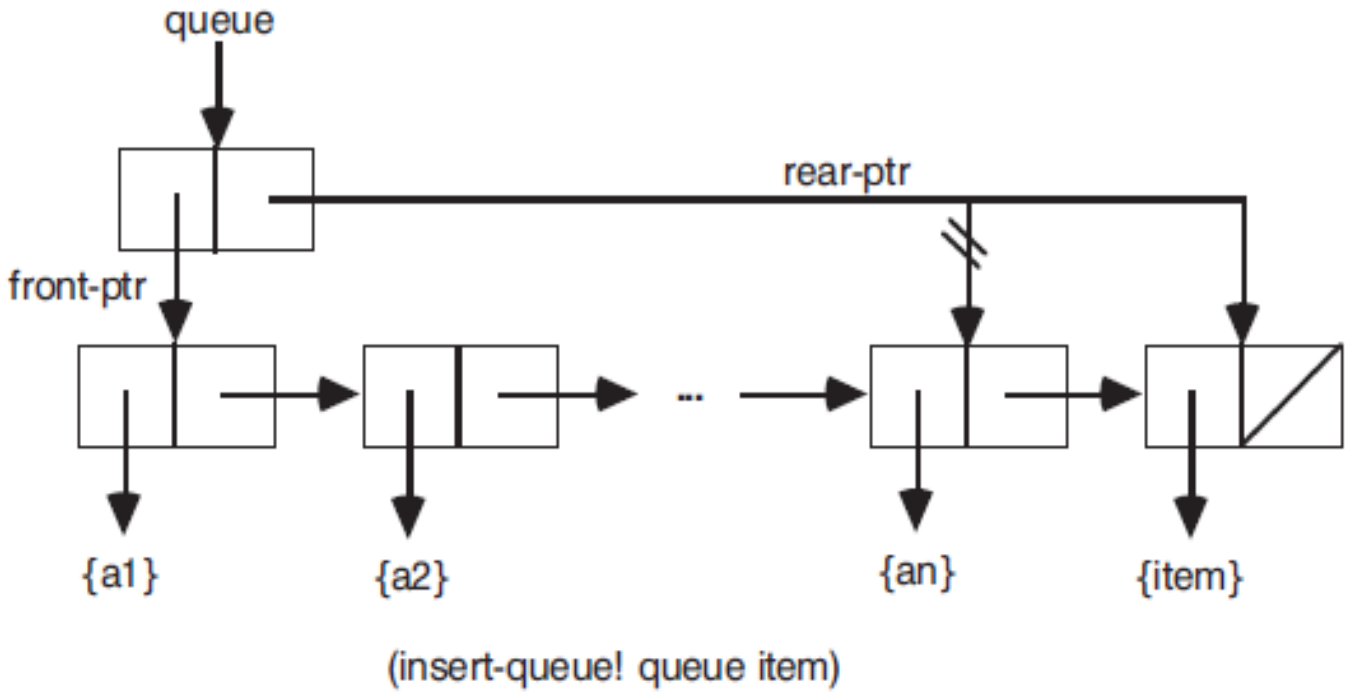
в) *Намиране на елемента в началото на опашката*

```
(define (front queue)
    (if (empty-queue? queue)
        (error "front is called for an empty queue" queue)
        (car (car queue))))
```

5. Представяне и реализация на опашка

Основни операции над опашка:

г) *Включване на елемент в опашка*



(insert-queue! queue item)

5. Представяне и реализация на опашка

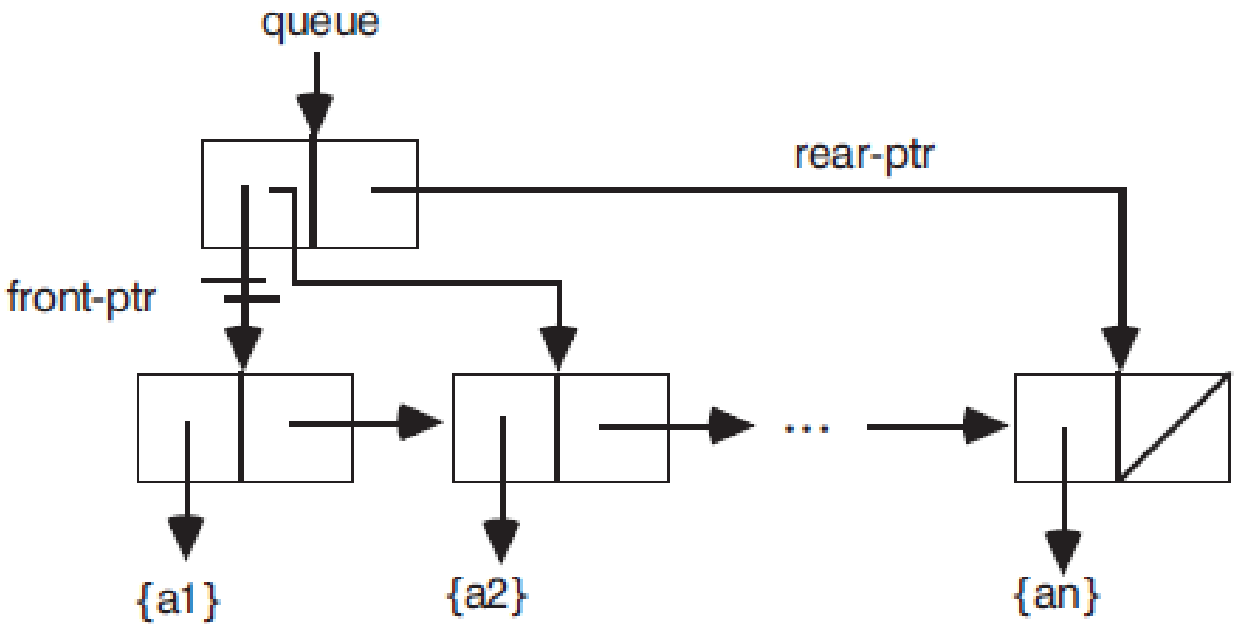
Основни операции над опашка:

```
(define (insert-queue! queue item)
    (let ((new-pair (cons item '())))
        (if (empty-queue? queue)
            (begin (set-car! queue new-pair)
                   (set-cdr! queue new-pair)
                   queue)
            (begin (set-cdr! (cdr queue) new-pair)
                   (set-cdr! queue new-pair)
                   queue))))
```

5. Представяне и реализация на опашка

Основни операции над опашка:

д) *Изключване на елемент от опашка*



5. Представяне и реализация на опашка

Основни операции над опашка:

```
(define (delete-queue! queue)
    (if (empty-queue? queue)
        (error "deleting an element from an empty queue" queue)
        (begin (set-car! queue (cdr (car queue)))
               queue)))
```

5. Представяне и реализация на опашка

Експерименти:

```
>(define q (make-queue))
>q
(())
>(insert-queue! q 1)
((1) 1)
>(insert-queue! q 2)
((1 2) 2)
>(insert-queue! q 3)
((1 2 3) 3)
>(delete-queue! q)
((2 3) 3)
```

5. Представяне и реализация на опашка

```
(define (print-queue! queue)
    (if (empty-queue? queue) (princ " ")
        (begin
            (princ (front queue))
            (princ " ")
            (print-queue! (delete-queue! queue)))))
```

display

извежда елементите на опашка, но я разрушава.

```
(define (print-queue queue)
    (if (empty-queue? queue) (princ " ")
        (car queue)))
```

извежда елементите на опашка без да я разрушава.