ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

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

Тема 6

СПИСЪЦИ

1. S-израз

S-изразите са основна конструкция на езика Лисп. Чрез тях се създават съставни структури от данни, в т.ч. списъци.

Дефиниция:

а) всеки атом (число, символ, низ, #t, #f) е S-израз;

б) ако **X** и **Y** са S-изрази, то **(X . Y)** е също S-израз;

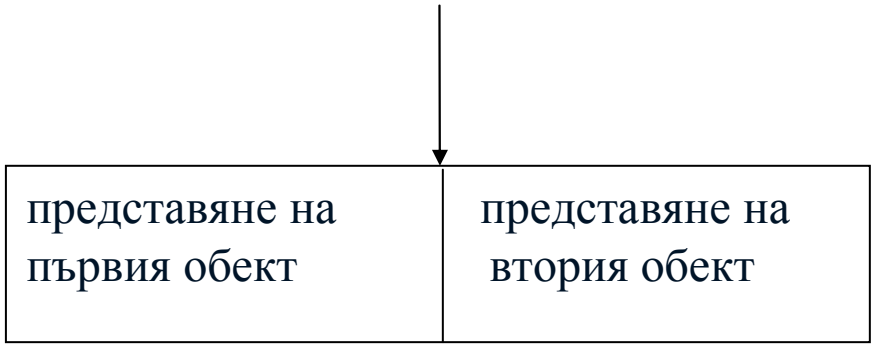
в) S-изразите се определят само по правилата а) и б).

Двойката **(X . Y)** се нарича още точкова двойка. Така всеки S-израз е или атом, или точкова двойка.

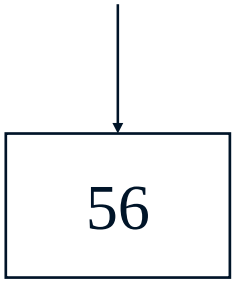
Пример: Нека A, B, C, и D са атоми. Тогава **((A . B) . (C . D))** е S-израз.

1. S-израз

Удобен начин за графично представяне на точкова двойка е чрез указател към двойна кутия.

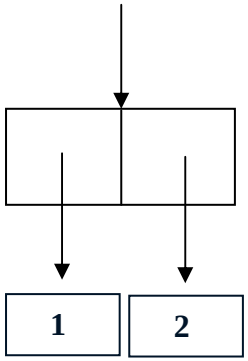


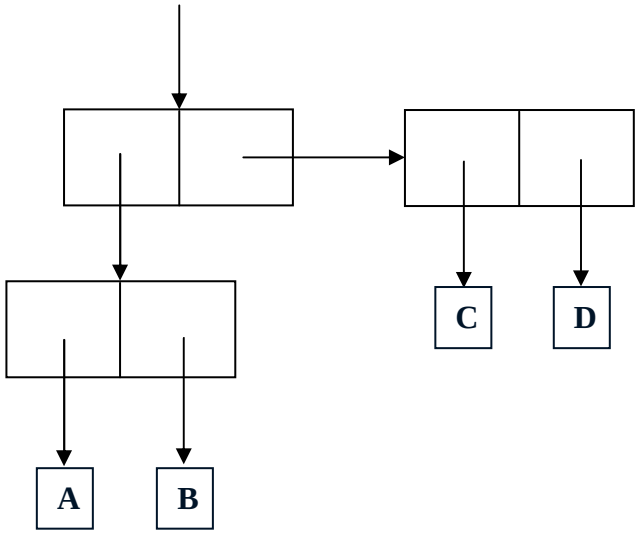
Удобен начин за графично представяне на атом е чрез указател към единична кутия.



1. S-израз

Примери:





(1 . 2)

((A . B) . (C . D))

Примитивни предикати за проверка типа на даден S-израз

В езика Scheme са предвидени някои вградени (примитивни) предикати, които проверяват какъв е типът на оценката на аргумента им.

`(pair? obj)` ⟶
- **#*t***, ако **[obj]** е точкова двойка;
- **#*f***, в противния случай.

1. S-израз

`(atom? obj)` ⟶
- ***#t***, ако **[obj]** е атом;
- ***#f***, в противния случай.

`(number? obj)` ⟶
- ***#t***, ако **[obj]** е число;
- ***#f,*** в противния случай.

`(string? obj)` ⟶
- ***#t***, ако **[obj]** е символен низ;
- ***#f***, в противния случай.

`(symbol? obj)` ⟶
- ***#t***, ако **[obj]** е символен атом;
- ***#f***, в противния случай.

Примери:

(pair? '(x . y)) ⟶ #t

(pair? 'abc) ⟶ #f

(atom? 'abc) ⟶ #t

(atom? 58) ⟶ #t

(atom? '(a . b)) ⟶ #f

(string? "string") ⟶ #t

(string? 'string) ⟶ #f

Забележка:

В DrScheme/DrRacket функцията *atom?* не е вградена. Вместо (atom? obj) може да се използва (not (pair? obj)).

2. Примитивни процедури за работа с двойки

Процедура cons

Предназначение: Конструира точкова двойка.

Синтаксис:

(cons a1 a2)

където

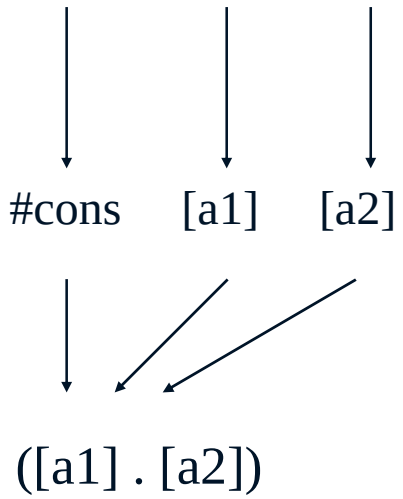
- cons е специален символ (съкращение на construct);
- a1 и a2 са S-изрази, т.е. атоми или точкови двойки.

Семантика:

Резултатът от прилагането на cons към a1 и a2 е точкова двойка от оценките на a1 и a2.

2. Примитивни процедури за работа с двойки

(cons a1 a2) в средата E



(cons 1 5) → (1 . 5)
(cons "a" "b") → ("a" . "b")
(cons ′p ′q) → (p . q)
(cons 1 (cons 2 (cons 3 nil))) → (1 . (2 . (3 . nil)))

2. Примитивни процедури за работа с двойки

Процедура car

Предназначение: Car намира първия обект на двойка, т.е. car е селектор.

Синтаксис:

(car x)

където

- car е специален символ и е съкращение на Contents Address Register.
- x е израз, чиято оценка е точкова двойка.

Семантика:

Намира първия обект на двойката, която е оценка на *x*, т.е. ако оценката на *x* е двойката (y . z), то (car x) намира **y**.

2. Примитивни процедури за работа с двойки

Процедура cdr

Предназначение: Cdr намира втория обект на двойка, т.е. cdr е селектор.

Синтаксис:

(cdr x)

където

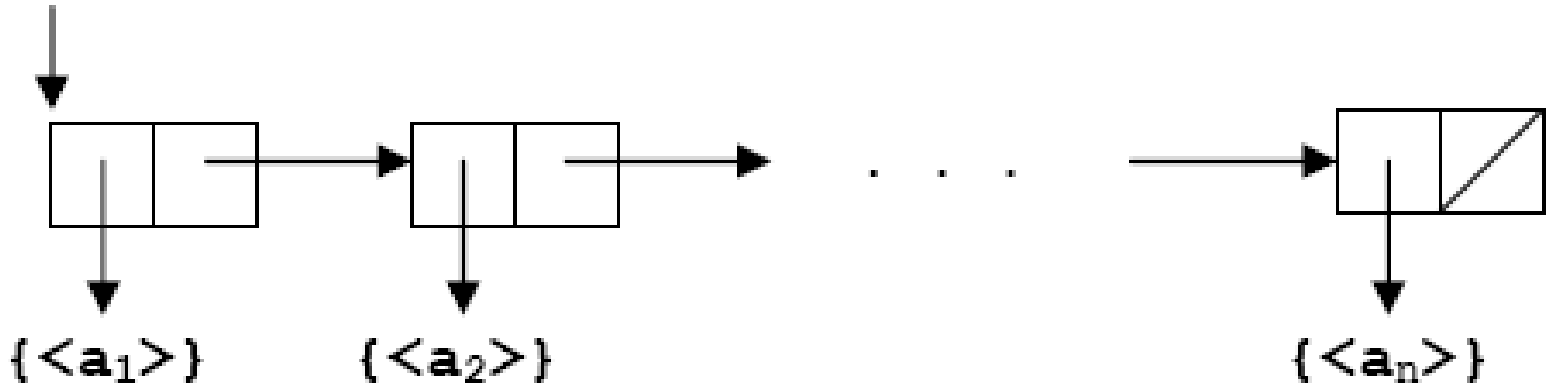
- cdr е специален символ и е съкращение на Contents Decrement Register.
- x е израз, чиято оценка е точкова двойка.

Семантика:

Намира втория обект на двойката, която е оценка на *x*, т.е. ако оценката на *x* е двойката (y . z), то (cdr x) намира z.

3. Списъците като точкови двойки

Списъците са точкови двойки, които представят крайни редици от елементи, изобразени графично по следния начин:



3. Списъците като точкови двойки

Записът във вид на точкова двойка на S-израза (списъка), представен с помощта на горната диаграма, е следният:

(cons <b1> (cons <b2> (cons ... (cons <bn> '()) ...)))

или също:

([<b1>] . ([<b2>] . (... . ([<bn>] . ()) ...)))

където $\{<a_i>\}$ е графичното представяне на [<bi>], i = 1, 2, …, n.

3. Списъците като точкови двойки

Изразът

([<b1>] . ([<b2>] . (... . ([<bn>] . ()) ...)))

съкратено се записва по следния начин:

([<b1>] [<b2>] ... [<bn>])

() е означение на ***празния списък*** (в DrRacket е еквивалентен на ***null***)

3. Списъците като точкови двойки

Списък

(1) празният списък **()** е списък;

(2) ако **a** е произволен S-израз, а **L** е списък, точковата двойка **(a . L)**, също е списък.

Следствия:

1) Всеки непразен списък е точкова двойка.
2) Празният списък няма елементи и **не е** точкова двойка. Празният списък е атом.

```
> (not (pair? '()))
#t
```

3. Списъците като точкови двойки

За проверка дали даден S-израз е еквивалентен на празния списък, се използва примитивната процедура ***null?***:

`(null? obj)` —> { *#t*, ако `[obj]` е ***nil*** (или *()*, или *#f*); *#f*, в противния случай.

3. Списъците като точкови двойки

Конструиране на списъци – примитивни процедури list и cons

Процедура list

Синтаксис:

(list <a1> <a2> ... <an>) ⟶ ([<a1>] [<a2>] ... [<an>])

където <a1> <a2> ... <an> са произволни S-изрази.

Семантика:
Горният запис е еквивалентен на

(cons <a1> (cons <a2> (cons ... (cons <an> nil) ...)))

3. Списъците като точкови двойки

Забележка:

От записа

(cons <a1> (cons <a2> (cons ... (cons <an> nil) ...)))

следва, че ***cons*** може да се използва както за конструиране на списък, така и за включване на елемент пред първия елемент на списък.

Пример:

(cons 'x '(a b c d e f)) ⟶ (x a b c d e f)

3. Списъците като точкови двойки

Примери:

(cons 1 (cons 2 (cons 3 '()))) → (1 2 3)

(list 1 2 3) → (1 2 3)

(define L '(1 2 3)) ; дефинира списъка L = (1 2 3)

3. Списъците като точкови двойки

Представяне на списък в паметта:

Нека

>(define L '(1 2 3))

В резултат се получва:

глоб. среда

Векторна памет

	0	1	2	3	4	5	6	7	…
car-	…	…	n3	n1	…	n2	…	…	…
cdr-	…	…	e0	p5	…	p2	…	…	…

3. Списъците като точкови двойки

Извличане на елементи на списък

От записа

(cons <a1> (cons <a2> (cons ... (cons <an> '()) ...)))

следва, че примитивната процедура ***car*** може да се използва за извличане (намиране) на първия елемент на даден списък, а процедурата ***cdr*** – за получаване на списъка без неговия първи елемент (за получаване на опашката на дадения списък).

Пример:

(car '(<a1> <a2> ... <an>)) ⟶ <a1>
(cdr '(<a1> <a2> ... <an>)) ⟶ (<a2> ... <an>)

3. Списъците като точкови двойки

Забележка.

По стандарт **(car '())** и **(cdr '())** са неопределени. В реализациите DrScheme/DrRacket **(car '())** и **(cdr '()) съобщават за грешка.**

Поредните елементи на списъка L (ако съществуват) се намират както следва:

първият елемент на списък: **(car L)**

вторият елемент: **(car (cdr L))** ≡ **(cadr L)**;

третият елемент: **(car (cdr (cdr L)))** ≡ **(caddr L)**;

четвъртият елемент: **(car (cdr (cdr (cdr L))))** ≡ **(cadddr L)**

и т.н.

4. Основни операции над списъци

а) ***Извличане на n-тия пореден елемент (n>0) на даден списък L***

```
(define (nth n L)
    (if (= n 1) (car L)
        (nth (- n 1) (cdr L))))
```

Пример:

```
> (define L '(1 2 3 4 5))
> (nth 1 L)
1
> (nth 5 L)
5
> (nth 6 L)
car: contract violation
expected: pair?
given: ()
```

б) ***Намиране броя на елементите (дължината) на даден списък - примитивна процедура length***

(length L) ——> число, равно на броя на елементите на **[L]**
(**[L]** трябва да е списък)

Забележка. Процедурата ***length*** е вградена. Може да бъде дефинирана по следния начин:

- в рекурсивен стил

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

4. Основни операции над списъци

- ***в итеративен стил***

```
(define (length L)
  (define (length-iter li count)
        (if (null? li) count
            (length-iter (cdr li) (+ 1 count))))
  (length-iter L 0))
```

Пример:

```
(length '(1 2 3)) ->
(length-iter '(1 2 3) 0) ->
(length-iter '(2 3) 1) ->
(length-iter '(3) 2) ->
(length-iter '() 3) ->
3
```

в) *Конкатенация на списъци*

Примитивна процедура append

(append L1 L2 ... Ln) ⟶ списък, който съдържа елементите на **[L1]**, следвани от елементите на **[L2]**, ... , елементите на **[Ln]**

Примери:

(append '(a b c) '(d e f g)) ⟶ (a b c d e f g)

(append '((a b) (c d) e f) '(g h) '(p q)) ⟶ ((a b) (c d) e f g h p q)

(append '(a b c d) '()) ⟶ (a b c d)

Забележка.

Процедурата ***append*** е вградена. Има произволен, но краен брой аргументи. Вариантът й с два аргумента може да се дефинира в рекурсивен стил по следния начин:

```
(define (append2 L1 L2)
    (if (null? L1)  L2
        (cons (car L1) (append2 (cdr L1) L2))))
```

г) ***Обръщане на реда на елементите на даден списък***

Примитивна процедура reverse

(reverse L) ⟶ списък, съставен от елементите на **[L]**, но взети в обратен ред (**[L]** трябва да бъде списък).

Примери:

(reverse '(a b c d)) ⟶ (d c b a)

(reverse '()) ⟶ ()

(reverse '((a b c) (d e f g) h p q)) ⟶ (q p h (d e f g) (a b c))

Забележка: Обръщането е на най-външно ниво, т.е. елементите на подсписъците на списъка L (ако има такива) не се обръщат.

4. Основни операции над списъци

Дефиниране на reverse

Първи начин: чрез append (рекурсивен вариант)

```
(define (reverse L)
    (if (null? L) L
        (append (reverse (cdr L)) (list (car L)))))
```

Втори начин*: *без append (итеративен вариант)

```
define (reverse L)
  (define (rev x y)
      (if (null? x) y
          (rev (cdr x) (cons (car x) y))))
  (rev L '()))
```

При втория начин параметърът y на вложената процедура rev последователно натрупва елементите на списъка x в обратен ред. Затова този начин се нарича обръщане на елементите на списък с *натрупващ параметър*, а първият – *без натрупващ параметър*.

5. Работа със списъци, съдържащи списъци

а) ***Пресмятане броя на атомите, които се намират на произволно ниво на влагане в даден списък***

Първи начин (като брои празния списък):

```
(define (atom? a)
    (not (pair? a)))

(define (count-atoms L)
    (cond ((null? L) 0)
          ((atom? (car L)) (+ 1 (count-atoms (cdr L))))
          (else (+ (count-atoms (car L)) (count-atoms (cdr L)))) ))
```

5. Работа със списъци, съдържащи списъци

```
(define (count-atoms L)
    (cond ((null? L) 0)
          ((atom? (car L)) (+ 1 (count-atoms (cdr L))))
          (else (+ (count-atoms (car L)) (count-atoms (cdr L))))))
```

Примери:

```
> (count-atoms '(1 2 (2 (3 4)) (8 (9 (10)))))
8
> (count-atoms '(1 2 (2 (3 4)) () () (8 (9 (10)))))
10
```

***Втори начин (като НЕ брои празния списък, който е атом)**:*

```
(define (count-atoms L)                ; брои атомите,
   (cond ((null? L) 0)                 ; различни от ()
         ((atom? L) 1)
         (else (+ (count-atoms (car L))
                  (count-atoms (cdr L))))))
```

Примери:

```
> (count-atoms '(1 2 (2 (3 4)) (8 (9 (10)))))
8
> (count-atoms '(1 2 (2 (3 4)) () () (8 (9 (10)))))
8
```

5. Работа със списъци, съдържащи списъци

б) ***Обръщане реда на елементите на даден списък на всички нива на влагане***

Пример:

reverse (1 (2 3) (4 (5 6))) ———> ((4 (5 6)) (2 3) 1)

deep-reverse (1 (2 3) (4 (5 6))) —> (((6 5) 4) (3 2) 1)

Дефиниране:

```
(define (deep-reverse L)
    (cond ((null? L) '())
          ((atom? L) L)
          (else (append (deep-reverse (cdr L))
                        (list (deep-reverse (car L)))))))
```

5. Работа със списъци, съдържащи списъци

```
(define (deep-reverse L)
    (if  (atom? L) L
        (append (deep-reverse (cdr L))
                (list (deep-reverse (car L))))))
```

6. Работа със списъци като множества от елементи (числа)

а) *Принадлежност на елемент на множество от числа*

```
(define (member? x A)
    (cond ((null? A) #f)
          ((= x (car A)) #t)
          (else (member? x (cdr A)))))
```

6. Работа със списъци като множества от елементи (числа)

б) *Сечение на множества*

```
(define (intersection A B)
    (cond ((null? A) '())
          ((null? B) '())
          ((member? (car A) B) (cons (car A)
                                     (intersection (cdr A) B)))
          (else (intersection (cdr A) B))))
```

6. Работа със списъци като множества от елементи (числа)

в) *Разлика на множества A \ B*

```
(define (diff A B)
    (cond ((null? B) A)
          ((null? A) '())
          ((member? (car A) B)  (diff (cdr A) B))
          (else (cons (car A) (diff (cdr A) B)))))
```

6. Работа със списъци като множества от елементи (числа)

г) *Обединение на множества*

```
(define (union A B)
   (append (diff A B) B))

(define (union1 A B)
    (cond ((null? A) B)
          ((null? B) A)
          ((member? (car A) B) (union (cdr A) B))
          (else (cons (car A) (union (cdr A) B)))))
```

7. Други процедури за работа със списъци от числа

Изтриване на първо срещане на елемент от списък от числа

```
(define (del x L)
    (cond ((null? L) '())
          ((= x (car L)) (cdr L))
          (else (cons (car L) (del x (cdr L))))))
```

Изтриване на всички срещания на елемент от списък

```
(define (del_all x L)
    (cond ((null? L) '())
          ((= x (car L)) (del_all x (cdr L)))
          (else (cons (car L) (del_all x (cdr L))))))
```

7. Други процедури за работа със списъци от числа

Намиране на последния елемент на непразен списък

```
(define (last L)
    (if (null? (cdr L)) (car L)
        (last (cdr L))))
```

Изтриване на последния елемент на непразен списък

```
(define (del_last L)
    (if (null? (cdr L)) '()
        (cons (car L) (del_last (cdr L)))))
```

Изтриване на последния елемент на непразен списък

```
(define (withoutlast L)
   (reverse (cdr (reverse L))))
```

7. Други процедури за работа със списъци от числа

намиране на минимален елемент на непразен списък

```
(define (min_el L)
    (if (null? (cdr L)) (car L)
        (min (car L) (min_el (cdr L)))))
```

7. Други процедури за работа със списъци от числа

Сортиране на списък от числа по метода на пряката селекция

```
(define (sort1 L)
    (if (null? L) '()
        (let ((m (min_el L)))
            (cons m (sort1 (del m L))))))
```

7. Други процедури за работа със списъци от числа

Забележка: Има вградена процедура за сортиране **sort**.

Примери за прилагане на sort:

```
> (sort '(3 2 1 5 5 3 2 1) <)
(1 1 2 2 3 3 5 5)

> (sort '(3 1 2 4 6 5 5 1 2) >)
(6 5 5 4 3 2 2 1 1)

> (sort '("Ivan" "Katia" "Anna" "Bilyana") string<?)
("Anna" "Bilyana" "Ivan" "Katia")
```