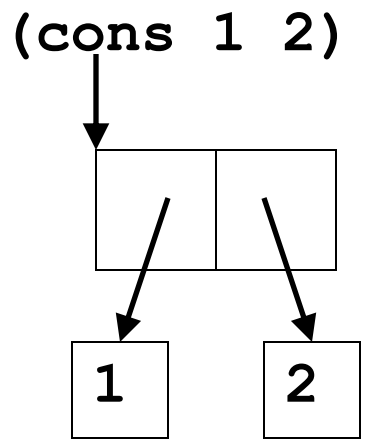
Списъци в езика Scheme

Представяне на данните в езика Scheme

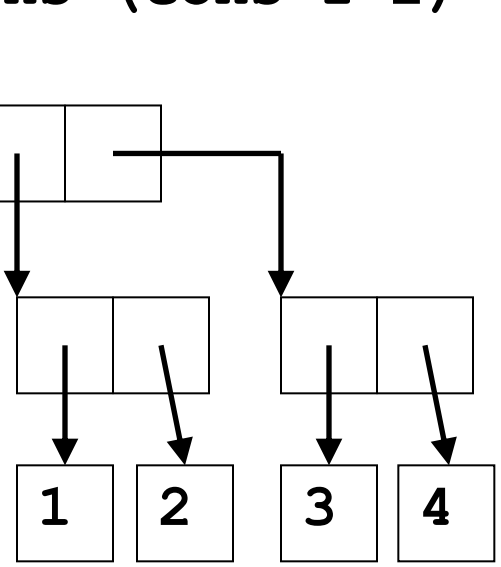
Ще разгледаме едно междинно логическо ниво, което се намира между нивото на текстовия запис на изразите и нивото на конкретното физическо представяне, зависещо от решенията на автора на конкретната реализация. Поддържането на това ниво се смята за задължително за авторите на реализации. За описанието му се използват диаграми, съдържащи клетки (кутии, boxes) и указатели.

Всеки обект се представя като указател към някаква клетка. Клетката, съответна на даден примитивен обект (каквито са например числата), съдържа представянето на този обект. Клетката, съответна на дадена точкова двойка, съдържа двойка указатели към представянията на ***car*** и ***cdr*** на тази точкова двойка.

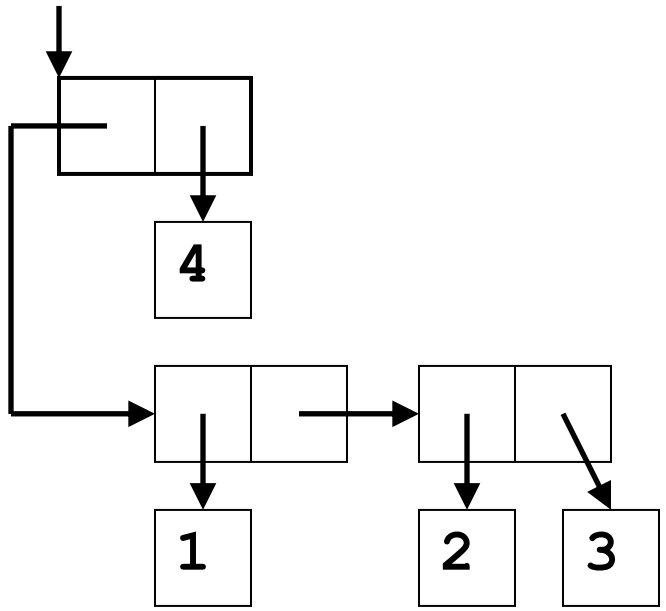
Примери



```
(cons (cons 1 2) (cons 3 4))
```



```
(cons (cons 1 (cons 2 3)) 4)
```



Важни следствия и бележки по отношение на данните в езика Scheme

1. Елементите на точковите двойки могат да бъдат както числа (т.е. примитивни обекти), така и други точкови двойки. Следователно, точковите двойки дават възможност с тяхна помощ да бъдат представяни йерархични данни - данни, съставени от части, които също са съставени от различни части и т.н.

2. Разгледаните досега типове данни (числа и точкови двойки) са частни случаи на т. нар. ***символни изрази*** (***S-изрази***). Това са обекти, които могат да бъдат произволни символи (а не само числа) или да се състоят от произволни символи.

S-изрази в Scheme. Символни атоми

Дефиниция на понятието S-израз

S-изразите в езика Lisp (в частност, в Scheme) се дефинират както следва:

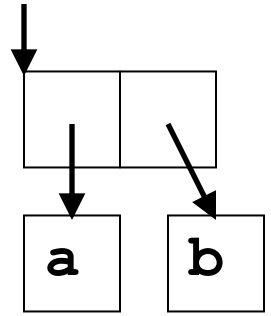
1) Атомите (числата, символните низове и т. нар. символни атоми или символи) са S-изрази.
2) Ако **s1** и **s2** са S-изрази, то **(s1.s2)** също е S-израз.
3) Няма други S-изрази освен тези, описани в т. 1) и 2).

Важни допълнения и бележки:

- атомите (числата, символните низове и символните атоми) са примитивните типове данни в Lisp (Scheme);
- символните атоми (символите) от синтактична гледна точка са идентификатори;
- S-изразите са най-общият тип данни в Scheme. Всички останали типове (и най-вече списъците) са техни частни случаи.

Действие на специалната форма quote

Пример. Как се конструира точковата двойка, представена чрез следната диаграма:



Очевидно записът **(cons a b)** не е подходящ, тъй като при него се предполага оценяване на **a** и **b** и формиране на точкова двойка от получените оценки. В случая е необходима процедура (по-точно, специална форма), която не оценява аргумента си и го връща като оценка такъв, какъвто е, без да го променя.

Такава е примитивната процедура (специалната форма) ***quote***.

Общ вид на обръщението към ***quote***:

(quote <S-израз>) или **'<S-израз>**

Семантика:

(quote <S-израз>) ⟶ <S-израз>

Примери

```
'a ⟶ a
'25 ⟶ 25
'(a.b) ⟶ (a.b)
''a ⟶ 'a
```

Примитивни предикати за проверка на типа на даден S-израз

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

Списък на по-важните предикати за проверка на типа на даден S-израз
(този списък ще бъде допълнен по-късно при въвеждането на понятието "списък")

`(pair? obj)` ⟶

- ***#t***, ако **[obj]** е точкова двойка;
- ***#f***, в противния случай.

`(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

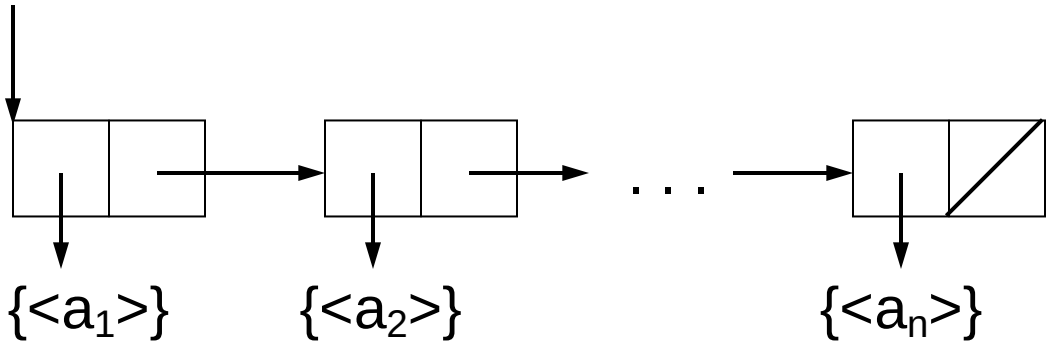
```
(pair? '(a.b)) ⟶ #t
(pair? 'a) ⟶ #f
(atom? 'a) ⟶ #t
(atom? 58) ⟶ #t
(atom? '(a.b)) ⟶ #f
(string? "a string") ⟶ #t
(string? 'no-string) ⟶ #f
```

Забележка

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

Дефиниция на списъците като точкови двойки. Конструиране на списъци и цитиране на елементи на даден списък

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



Забележка. С **{<S-израз>}** тук означаваме представянето на **<S-израз>**.

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

(<a_1>.(<a_2>.(... .(<a_n>.()) ...)))

или също:

(cons '<a_1> (cons '<a_2>
(cons ... (cons '<a_n> '()) ...)))

Тук **()** е означение на т. нар. ***празен списък*** (в DrScheme, но не и в DrRacket, той е еквивалентен на вградения (примитивния) атом ***null***, който служи за означаване на края на всеки списък – в графичните диаграми обикновено ***null*** се означава като диагонална линия в съответната клетка). Горният списък се записва още по следния начин:

(<a_1> <a_2> ... <a_n>)

Тук **<a_i>** се наричат ***елементи на списъка***. Елементите на един списък могат да бъдат произволни S-изрази (в частност, символни атоми или други списъци).

Следователно, понятието „списък“ в езика Scheme се дефинира по следния начин:

(1) празният списък **()** е списък;

(2) ако **lst** е списък, то точковата двойка **(a.lst)**, където **a** е произволен S-израз, също е списък.

Тази дефиниция в частност означава, че всеки непразен списък е точкова двойка. Празният списък няма елементи и **не е** точкова двойка.

За проверка на това, дали един S-израз е еквивалентен на празния списък, се използва примитивната процедура (вградения предикат) ***null?***:

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

Конструирането на списъци се извършва най-често с помощта на примитивните процедури ***list*** и ***cons***.

Процедурата ***list*** има произволен брой аргументи. Тя конструира и връща като резултат списък от оценките на аргументите си:

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

Горният запис е еквивалентен на

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

което означава, че ***cons*** може да се използва за конструиране на списък, а също и за добавяне на елемент отпред (преди първия елемент) в списък:

```
(cons 'a '(b c d)) ⟶ (a b c d)
```

Извличане на елементи на списък

От записа на списъците във вид на точкови двойки следва, че

(car '(<a_1> <a_2> ... <a_n>)) ⟶ <a_1> и
(cdr '(<a_1> <a_2> ... <a_n>)) ⟶ (<a_2> ... <a_n>)

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

Забележка. По стандарт оценките на **(car ′())** и **(cdr ′())** са неопределени, тъй като празният списък е атом, а не точкова двойка.

В DrScheme/DrRacket обръщения от този вид не са коректни.

Следователно поредните елементи на един списък могат да бъдат извличани (цитирани) както следва:

- първият елемент на списъка: **(car <списък>)**;
- вторият елемент:

(car (cdr <списък>)) ≡ (cadr <списък>);

- третият елемент:

(car (cdr (cdr <списък>))) ≡ (caddr <списък>);

- четвъртият елемент:

(car (cdr (cdr (cdr <списък>)))) ≡ (cadddr <списък>) и т.н.

Извличането на n-тия пореден елемент (n>0) на даден списък l може да стане с помощта на следната процедура:

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

Основни примитивни процедури за работа със списъци в езика Scheme

Намиране на 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. Ако това не е така, тя може да бъде дефинирана например по следните начини:

- в рекурсивен стил

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

- в итеративен стил

```
(define (length l)
   (define (length-iter arg count)
           (if (null? arg)
               count
               (length-iter (cdr arg)
                            (+ 1 count))))
   (length-iter l 0))
```

Обединяване на елементите на произволен брой списъци - примитивна процедура append

(append l_1 l_2 ... l_n) ⟶ списък, който съдържа елементите на **[l_1]**, следвани от елементите на **[l_2]**, ... , елементите на **[l_n]**

Примери

```
(append '(a b) '(c d)) ⟶ (a b c d)
(append '((a b) (c d)) '(x y) '(k l)) ⟶
   ((a b) (c d) x y k l)
(append '(a b) '()) ⟶ (a b)
```

Забележка. Процедурата ***append*** е вградена, при това има произволен брой аргументи. Вариантът й с два аргумента може да бъде дефиниран в рекурсивен стил както следва:

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

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)) ⟶ (e (c d) (a b))
```

Предикати за проверка на равенство и проверка за принадлежност към списък

Проверка за равенство

Обща бележка. Точният механизъм на действие на тези предикати е свързан с изясняване на някои допълнителни подробности от представянето на обектите в Scheme. Затова сега действието им ще бъде въведено не съвсем точно и ще бъде уточнено по-нататък.

`(eq? s1 s2)` ⟶

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

Забележки

1) Обикновено предикатът ***eq?*** се използва за проверка на това, дали **[s1]** и **[s2]** са еднакви символни атоми (има и други случаи, в които ***eq?*** дава резултат ***#t***, но те са много малко).

2) За сравняване на числа е добре да се използва аритметичният предикат **=** или предикатът ***eqv?*** (той обединява в определен смисъл действието на ***eq?*** и **=**).

`(equal? s1 s2)` ⟶

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

Проверка за принадлежност към списък

`(memq item l)` ⟶

- **#f**, ако **[item]** не съвпада (в смисъл на ***eq?***) с никой от елементите на списъка **[l]**;
- тази част от списъка **[l]**, която започва с първото срещане на елемент, равен (в смисъл на ***eq?***) на **[item]**.

Примери

```
(memq 'a '(c a b a d)) ⟶ (a b a d)
(memq 'a '(b c d)) ⟶ #f
(memq '(a b) '((a b) c d)) ⟶ #f
```

Примерна дефиниция на функционален аналог на процедурата ***memq***:

```
(define (memq item l)
   (cond ((null? l) #f)
         ((eq? (car l) item) l)
         (else (memq item (cdr l)))))
```

Примитивната процедура ***(member item l)*** има същото действие като ***memq***, но при нея сравнението се извършва с помощта на предиката ***equal?*** (а не с ***eq?***, както е при ***memq***).

Примери

```
(member 'a '(b a c a d)) ⟶ (a c a d)
(member '(a b) '((a b) (c d))) ⟶ ((a b) (c d))
(member '(c d) '((a b) (c d))) ⟶ ((c d))
(member 'a '((a b) (c d))) ⟶ #f
```

Примерна дефиниция на функционален аналог на процедурата ***member***:

```
(define (member item l)
   (cond ((null? l) #f)
         ((equal? (car l) item) l)
         (else (member item (cdr l)))))
```

Работа със списъци с вложения

Досега разглеждахме и дефинирахме процедури, свързани с изследване (обхождане) на елементите на даден списък, които се намират на най-високо ниво на вложение. При тях изследвахме едно и също гранично условие - проверка за изчерпване на броя на елементите на изходния списък (проверка за достигане на празен списък). В общия (непразния) случай се извършваше определена обработка с използване на първия елемент на списъка и задачата се редуцираше до по-проста чрез формиране на рекурсивно обръщение към същата процедура с аргумент - ***cdr*** от изходния списък.

Ако поставената задача е такава, че изисква изследване на елементите на дадения списък и в дълбочина, тогава към горното гранично условие (проверката за изчерпване на броя на елементите на списъка) трябва да се добави още едно, свързано с изследване (проверка) за изчерпване на текущия елемент в дълбочина (достигане на атомарна структура на текущия елемент на списъка) или понякога – с описание на случая на атомарен аргумент (т.е. с разширяване на типа на аргумента на процедурата: от списък към атом или списък).

Пример 1. Намиране на броя на атомите, които се намират на произволно ниво на вложение в даден списък (намиране на броя на атомите в даден списък).

Първи начин

```
(define (count-atoms l)      ;;; брои всички атоми
   (cond ((null? l) 0)
         ((atom? (car l))
          (+ 1 (count-atoms (cdr l))))
         (else (+ (count-atoms (car l))
                  (count-atoms (cdr l))))  ))
```

Втори начин

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

Пример 2. Дефиниране на процедура, която обръща реда на елементите на даден списък на всички нива на вложение (обобщение на примитивната процедура ***reverse***).

Пример за действието на процедурата:

```
                          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)
```

А. Примерна дефиниция на процедура, аналогична по действие на примитивната процедура ***reverse***

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

Б. Дефиниция на процедурата **deep-reverse**

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

} могат да се обединят

След обединяване на посочените два реда от горната дефиниция се получава окончателният вариант на дефиницията на процедурата **deep-reverse**:

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

Процедури от по-висок ред за работа със списъци

Акумулиране (комбиниране) на елементите на даден списък

Идея. Нека разгледаме следните процедури, предназначени съответно за събиране и умножаване на елементите на даден списък (предполага се, че тези елементи са числа).

Събиране на елементите на даден списък

```
(define (sum-list l)
   (if (null? l)
       0
       (+ (car l) (sum-list (cdr l)))))
```

Умножаване на елементите на даден списък

```
(define (product-list l)
   (if (null? l)
       1
       (* (car l) (product-list (cdr l)))))
```

Горните две дефиниции могат да бъдат обобщени до следната акумулираща процедура от по-висок ред, натрупваща или комбинираща по някакъв начин елементите на даден списък, която има като аргументи съответната комбинираща процедура (**combiner**), подходяща начална стойност на резултата (**init**) и дадения списък (**l**):

```
(define (accum combiner init l)
   (if (null? l)
       init
       (combiner (car l)
                 (accum combiner init (cdr l)))
))
```

Тази процедура може да се използва за дефиниране на редица конкретни полезни процедури, например:

`(accum + 0 l)` ⟶ сумата от елементите на **[l]** (действа като **sum-list**);

`(accum * 1 l)` ⟶ произведението на елементите на **[l]** (действа като **product-list**);

`(accum cons '() l)` ⟶ **[l]** (копира елементите на **[l]** и връща резултат, който е ***equal*** с **l**).

Трансформиране (изобразяване) на даден списък чрез прилагане на една и съща процедура към всеки от неговите елементи

Примерна дефиниция на процедура, която прилага дадена процедура към всеки от елементите на даден списък и връща списък от получените оценки:

```
(define (map proc l)
   (if (null? l)
       '()
       (cons (proc (car l))
             (map proc (cdr l))) ))
```

Забележка. При процедурата **map** няма натрупване (акумулиране) на резултатите от прилагането на процедурата **[proc]** върху елементите на **[l]** в смисъла, в който това става при дефинираната по-горе процедура **accum**. Процедурата **map** връща винаги списък от получените резултати.

В действителност съществува вградена (примитивна) процедура ***map*** с подобно на описаното по-горе действие.

Обръщението към примитивната процедура ***map*** изглежда по следния начин:

(map <процедура> <списък>)

Действието на тази процедура е следното. Оценяват се **<процедура>** и **<списък>**, процедурата **[<процедура>]** се прилага едновременно (псевдопаралелно) към всеки от елементите на списъка **[<списък>]** (като при това не се оценяват още веднъж елементите на **[<списък>]**) и като оценка се връща списъкът от получените резултати.

Примери

```
(map car '((a 1) (b 2) (c 3) (d 4))) ⟶
   (a b c d)
(map (lambda (y) (+ y y)) '(1 2 3 4 5)) ⟶
   (2 4 6 8 10)
```

Процедурата ***map*** стои в основата на една (трета поред след рекурсията и итерацията) от основните стратегии за управление на изчислителния процес при програмиране на езика Lisp - т.нар. ***изобразяване*** (mapping). Основните характеристики на рекурсивните и итеративните процеси вече бяха разгледани. **Същността на процесите на изобразяване (mapping) се свежда до едновременно (псевдопаралелно) извършване на един и същ тип обработка върху елементите на даден списък и формиране на списък от получените резултати.**

Примерна задача: филтриране на елементите на даден списък. Да се дефинира процедура **list-filter**, която по дадени едноаргументна процедура - предикат **filter** и списък **l** връща като резултат списък от онези елементи на **l**, които преминават успешно през филтъра (за които процедурата **filter** връща стойност "истина").

Пример, илюстриращ действието на процедурата list-filter:

```
(list-filter odd? '(1 2 3 4 5)) ⟶ (1 3 5)
```

Решение

Първи начин (с рекурсия)

```
(define (list-filter filter l)
   (cond ((null? l) '())
         ((filter (car l))
          (cons (car l)
                (list-filter filter (cdr l))))
         (else (list-filter filter (cdr l))) ))
```

Втори начин (с итерация)

```
(define (list-filter filter l)
   (define (iter arg acc)
      (cond ((null? arg) acc)
            ((filter (car arg))
             (iter (cdr arg)
                (append acc (list (car arg)))))
            (else (iter (cdr arg) acc))))
   (iter l '()))
```

ИЛИ СЪЩО

```
(define (list-filter filter l)
   (define (iter arg acc)
      (cond ((null? arg) (reverse acc))
            ((filter (car arg))
             (iter (cdr arg)
                   (cons (car arg) acc)))
            (else (iter (cdr arg) acc))))
   (iter l '()))
```

Трети начин (с изобразяване)

```
(define (list-filter filter l)
   (accum append
          '()
          (map (lambda (x)
                  (if (filter x) (list x) '()))
               l)))
```

Прилагане на процедура към списък от аргументи – примитивна процедура apply

Идея за процедурата ***apply*** може да се получи например от последната дефиниция на процедурата **list-filter**. В тази дефиниция беше използвано обръщение към дефинираната преди това процедура **accum**, за да се приложи процедурата ***append*** върху аргументи - елементите на даден списък. С други думи, **accum** беше използвана, за да може да се приложи ***append*** върху списък от подходящи аргументи. В общия случай за подобни цели може да се използва примитивната (вградената) процедура ***apply***.

Общ вид на обръщението към ***apply***:

`(apply <процедура> <списък-от-арг>)`

Действие. Оценяват се **<процедура>** и **<списък-от-арг>**. Нека **[<списък-от-арг>]** е **(arg_1 arg_2 ... arg_n)**. Процедурата ***apply*** предизвиква прилагане на процедурата **[<процедура>]** върху аргументи **arg_1**, **arg_2**, ... , **arg_n**, като при това тези аргументи не се оценяват още един път, и връща получения резултат.

Примери

```
(apply + '(2 5)) ⟶ 7
(apply max '(2 7 8 9 5)) ⟶ 9
(apply append '((1) (2) () (3))) ⟶ (1 2 3)
```

Ако се върнем отново на първоначалната идея, то последната дефиниция на процедурата **list-filter**, която следва методологията на изобразяването, може да се запише още и по следния начин:

```
(define (list-filter filter l)
   (apply append
          (map (lambda (x)
                  (if (filter x) (list x) '()))
               l)))
```

Намирането на броя на елементите на даден списък, които преминават през даден филтър, може да стане с помощта на следната процедура:

```
(define (count-filter filter l)
   (apply +
          (map (lambda (x)
                 (if (filter x) 1 0))
               l)))
```

Забележка. Оценката на първия аргумент на ***apply*** трябва да бъде процедура, която следва общото правило за оценка на комбинации, т.е. процедура, която не е специална форма. Поради тази особеност на ***apply*** следната дефиниция на процедура за намиране на първия от елементите на даден списък, който преминава успешно през даден филтър, е **некоректна**:

```
(define (find-filter filter l)
   (apply or
          (map (lambda (x)
                 (if (filter x) x #f))
               l)))
```

Двукратно оценяване на даден израз – примитивна процедура eval

Идея за тази процедура може да се получи от горните примери, илюстриращи действието на процедурата ***apply***. По същество действието на ***apply*** се свежда до формиране на подходящо обръщение към зададената като аргумент процедура и активиране, т.е. оценяване на това обръщение. Например, процесът на оценяване на **(apply + ′(2 5))** може да се сведе до следните стъпки:

- формиране на обръщението **(+ 2 5)**;
- оценяване на това обръщение.

За формирането на обръщението могат да се използват различни средства на езика Scheme, например конструкцията **(cons ′+ ′(2 5))**; за оценяването на това обръщение може да се използва например примитивната процедура ***eval***.

Общ вид на обръщението към ***eval*** (в по-простия му вариант):

(eval <израз>)

Действие. Оценява се **<израз>** в текущата среда и оценката на получената оценка (отново в текущата среда) се връща като резултат.

Следователно, **(eval <израз>) ⟶ [[<израз>]]**.

Пример 1

```
> (define a 'b)
a
> (define b 'c)
b
> (eval a)
c
```

Пример 2

```
(eval (cons '+ '(1 2 3))) ⟶ 6
```

или също

```
> (define l '(+ 1 2 3))
l
> (define m l)
m
> (eval m)
6
```