19.2. ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ: Списъци. Функции от по-висок ред за работа със списъци. Потоци и отложено оценяване.

Списъци. Основни операции със списъци. Процедури от по-висок ред за работа със списъци. Потоци. Основни операции с потоци. Функции от по-висок ред за работа с потоци. Отложено оценяване. Работа с безкрайни потоци.

Списъци.

Основна конструкция на лиспоподобните функционални езици е **S-изразът**:

1) Атомите (символите, числата, низовете, а също булевите константи #t и #f) са S-изрази;

2) Ако X и У са S-изрази, то (X. У) е S-израз;

3) Няма други S-изрази освен тези, описани в т. 1) и 2).

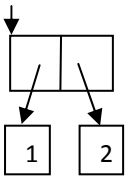
S-изразите са най-общият тип данни в Scheme; всички останали типове (и най-вече списъците) са техни частни случаи. S-изразът (X. У) се нарича **точкова двойка** или само **двойка**. Следователно всеки S-израз е или атом, или точкова двойка. Елементите на двойките могат да бъдат както числа (т.е. примитивни обекти), така и други двойки. ⇒ двойките дават възможност с тяхна помощ да бъдат представяни йерархични данни.

Чрез процедурата ***cons*** (construct) се конструира двойка, т.е. ***cons*** е конструктор на двойки. Синтаксисът му е **(cons a1 a2)**, където a1 и a2 са изрази, чиито оценки са S-изрази. Резултатът от прилагането на ***cons*** към a1 и a2 е точкова двойка от оценките на a1 и a2.

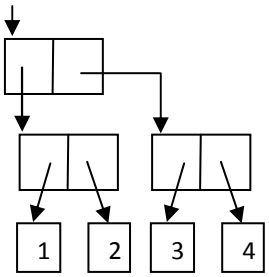
Графично една точкова двойка се представя чрез указател към двойна кутия (кутия, която съдържа двойка указатели).

Примери:

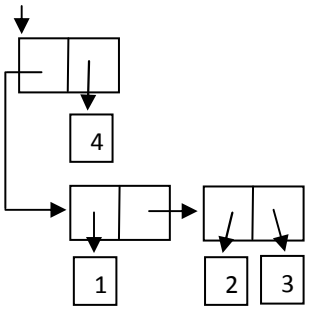
(cons 1 2)



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

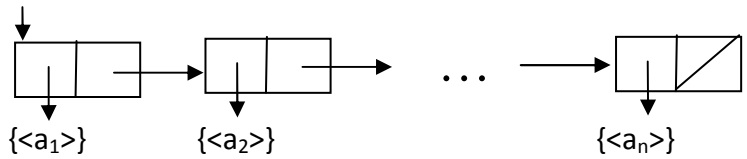


(cons (cons 1 (cons 2 3)) 4)



Процедурата ***car*** (Contents Address Register) намира първия обект на двойка, а ***cdr*** (Contents Decrement Register) намира втория обект на двойка, т.е. ***car*** и ***cdr*** са селектори. Синтаксисът им е **(car x)** и **(cdr x)**, където x е израз, чиято оценка е точкова двойка.

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



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

Примитивната процедура (специалната форма) ***quote*** не оценява аргумента си и го връща като оценка такъв, какъвто е, без да го променя. Общият вид на обръщението към ***quote*** е **(quote <S-израз>)** или **'<S-израз>**. Например **'a** ⟶ a, **'(a.b)** ⟶ (a.b), **''a** ⟶ 'a.

Записът във вид на точкова двойка на S-израза (списъка), представен с помощта на горната диаграма, е следният: **(<a_1>.(<a_2>.(... .(<a_n>.nil) ...)))** или също: **(cons '<a_1> (cons '<a_2> (cons ... (cons '<a_n> nil) ...)))**. Тук ***nil*** (***null*** в DrScheme) е означение на един вграден (примитивен) специален атом, който служи за означаване на края на всеки списък (графично обикновено ***nil*** се означава като диагонална линия в съответната клетка). Атомът ***nil*** се смята за еквивалентен на празния списък. По-точно, според стандарта на Scheme атомът ***nil***, празният списък ***()*** и системната константа ***#f*** са еквивалентни.

Горният списък се записва още по следния начин: **(<a_1> <a_2> ... <a_n>)**.

За проверка дали един S-израз е еквивалентен на ***nil***, се използва предиката ***null?***:

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

Списъци се конструират с примитивната процедура с произволен брой аргументи ***list***. Тя конструира и връща като резултат списък от оценките на аргументите си: **(list <a_1> <a_2> ... <a_n>)** ⟶ ([<a_1>] [<a_2>] ... [<a_n >])**.** Този запис е еквивалентен на **(cons < a_1> (cons < a_2> (cons ... (cons < a_n > nil) ...)))**, което означава, че ***cons*** може да се използва за конструиране на списък, а също и за добавяне на елемент отпред в списък: **(cons 'a '(b c d))** ⟶ (a b c d).

Основни операции със списъци.

1) Намиране на броя елементи (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)
```

2) Извличане на n-ия пореден елемент на списък:

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

(nth (- n 1) (cdr l))))

3) 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 ѝ 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))))
```

4) 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))

5) Предикати за проверка за равенство

(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 същ 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.

6) Проверка за принадлежност към списък

(memq item l**)** ⟶
- ***()***, 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]**.

Забележка: Оценката на втория аргумент на ***memq*** задължително трябва да бъде списък.

Примери:

(memq 'a '(c a b a d)) ⟶ (a b a d)
(memq 'a '(b c d)) ⟶ ()
(memq '(a b) '((a b) c d)) ⟶ ()

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

```
(define (memq item l)
        (cond ((null? l) '())
              ((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))) ⟶ ()

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

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

Забележка: написаното за ***memq*** и ***member*** е вярно за стандарта на езика и за някои конкретни реализации, например за PC Scheme, но не е съвсем точно за използваната напоследък във ФМИ среда на DrScheme.

Процедури от по-висок ред за работа със списъци.

1) Акумулиране (комбиниране) на елементите на даден списък

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 nil l) ——> **[l]** (копира елементите на **[l]** и връща резултат, който е ***equal*** с **l**).

2) 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 **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 **accum**. **map** връща винаги списък от получените резултати.

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

3) Филтриране на елементите на списък – 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").

```
(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))) ))
```

Пример: **(list-filter odd? '(1 2 3 4 5))** ⟶ (1 3 5)

4) Прилагане на процедура към списък от аргументи – примитивна 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)

Забележка: Оценката на първия аргумент на ***apply*** трябва да е процедура, която следва общото правило за оценка на комбинации, т.е. процедура, която не е специална форма.

Потоци.

Потокът е редица от елементи, която може да се дефинира с помощта на основните примитивни процедури за работа с потоци както следва: ако **x** има стойност, равна на оценката на **(cons-stream a b)**, то **(head x)** ⟶ **[a]** и **(tail x)** ⟶ **[b]**.

Поток без елементи се означава с ***the-empty-stream***, а примитивен предикат за проверка за празен поток (поток без елементи) е ***empty-stream?***.

Следователно, потоците са съставен тип данни, чийто конструктор е примитивната процедура ***cons-stream*** и чийто селектори са примитивните процедури ***head*** (която извлича първия елемент на даден поток) и ***tail*** (която извлича опашката на даден поток, т.е. връща поток, получен от дадения поток чрез премахване на първия му елемент).

Една от важните характерни черти на потоците е, че при тях се предполага строго последователен достъп до елементите (в посока от началото към края). Употребата на потоци е полезна най-вече, когато те имат голям брой (и дори безкрайно много) елементи.

Основни операции с потоци.

Обединяване на два потока (аналог на ***append*** с два аргумента при списъци):

```
(define (append-stream stream1 stream2)
        (if (empty-stream? stream1)
            stream2
            (cons-stream (head stream1) (append-stream (tail stream1) stream2))))
```

Функции от по-висок ред за работа с потоци.

1) Трансформиране на поток (аналог на ***map*** при списъци: прилага дадена процедура към всеки от елементите на даден поток и формира поток от съответните резултати):

```
(define (map-stream procedure stream)
        (if (empty-stream? stream)
            the-empty-stream
            (cons-stream (procedure (head stream))
                         (map-stream procedure (tail stream)))))
```

Примери за употреба на дефинираните процедури:

```
(define str1
          (cons-stream 1
            (cons-stream 2
              (cons-stream 3
                  the-empty-stream))))
(map-stream (lambda (x) (* 2 x)) str1)
```

В резултат се получава поток с елементи 2, 4, 6 (но не разполагаме с примитивна процедура, позволяваща извеждане отведнъж на всички елементи на потока).

2) Филтриране (по зададена процедура – филтър и поток връща нов поток, съставен от тези елементи на изходния, които преминават през филтъра):

```
(define (filter-stream proc stream)
        (cond ((empty-stream? stream) the-empty-stream)
              ((proc (head stream))
                (cons-stream (head stream)
                             (filter-stream proc (tail stream))))
                            (else (filter-stream proc (tail stream)))))
```

3) Формиране на поток от числата, които се намират между две дадени числа **low** и **high**:

```
(define (enum-stream low high)
        (if (> low high)
            the-empty-stream
            (cons-stream low (enum-stream (+ low 1) high))))
```

Отложено оценяване.

Реализацията на потоците се основава на т. нар. **забавени изчисления** (**отложено оценяване**, **delayed evaluation**), които са средство за "пакетиране" на съответните изрази (аргументи и резултати) и позволяват тези изрази да бъдат оценявани едва тогава, когато оценяването им е наистина необходимо.

В този смисъл основната идея при реализацията на ***cons-stream*** е тази процедура да конструира потока само частично и да предава така конструирания от нея специален обект на програмата (процедурата), която го използва. Ако тази програма се опита да получи достъп до неконструираната част, то се конструира допълнително само тази част от потока, която е реално необходима за продължаване на процеса на оценяване. Оценяването (конструирането) на останалата част от потока отново се отлага до евентуално възникване на необходимост от достъп до нови елементи и т.н.

За целта се използва специалната форма ***delay***. В резултат на оценяването на обръщението **(delay <израз>)** се получава "пакетиран" ("отложен") вариант на **<израз>**, който позволява реалното оценяване на **<израз>** да се извърши едва тогава, когато това е абсолютно

необходимо. По-точно, "пакетираният" израз, който е оценка на обръщението към ***delay***, може да бъде използван за възобновяване на оценяването на **<израз>** с помощта на процедурата ***force***. Процедурите ***delay*** и ***force*** са вградени. Тяхното действие може да се разглежда още и по следния начин. Формата ***delay*** "пакетира" даден израз така, че той да бъде оценен при повикване. Следователно, може да се смята, че ***delay*** е такава специална форма, която не оценява аргумента си и за която **(delay <израз>) ⟶** (lambda () <израз>)**.**

Обратно, ***force*** просто извиква процедурата без аргументи, получена от ***delay***. Следователно, ***force*** може да се реализира като процедура:

```
(define (force delayed-object)
        (delayed-object))
```

При тези дефиниции е в сила

(force (delay expr)) ⟶ (force (lambda () expr)) ⟶ ((lambda () expr)) ⟶ [expr].

Тогава ***cons-stream*** може да се разглежда като специална форма (която не оценява втория си аргумент), дефинирана така, че **(cons-stream a b) ⟶ (cons a (delay b)).**

Селекторите ***head*** и ***tail*** могат да бъдат дефинирани като процедури както следва:

```
(define (head stream) (car stream))
(define (tail stream) (force (cdr stream)))
```

Работа с безкрайни потоци.

Използват се два основни подхода при конструирането на **безкрайни потоци** – т. нар. неявно/индиректно и явно/директно конструиране. При първия подход се използват специални процедури – генератори, а при втория – рекурсия върху съответния генериран поток (по отношение на вече генерираните части на потока).

Неявно (индиректно) конструиране на безкрайни потоци

Пример 1: Генериране на безкраен поток от положителните цели числа.

```
(define (integers-from n)
        (cons-stream n (integers-from (+ 1 n))))
        (define integers (integers-from 1))
```

Пример 2: Генериране на безкраен поток от числата на Фибоначи.

```
(define (fibgen a b)
        (cons-stream a (fibgen b (+ a b))))
        (define fibs (fibgen 0 1))
```

Обяснение на решението от Пример 1: Предложената дефиниция е смислена, тъй като **integers** има за стойност точкова двойка с глава – числото 1 и опашка - "пакетиран" израз, който може да генерира поток от целите числа, започващи от 2. Така генерираният поток е безкраен, но във всеки момент може да се работи само с негова крайна част. В този смисъл нашата програма никога няма да "знае", че целият безкраен поток не е наличен (тъй като никога няма да се наложи да бъде разглеждан целият поток).

Намиране на n-тия елемент на безкраен поток:

```
(define (nth-of-stream n stream)
        (if (= n 1)
           (head stream)
           (nth-of-stream (- n 1) (tail stream))))
```

Събиране на елементите на два потока:

```
(define (add-streams s1 s2)
        (cond ((empty-stream? s1) s2)
              ((empty-stream? s2) s1)
              (else (cons-stream
                     (+ (head s1) (head s2))
                     (add-streams (tail s1)
                     (tail s2))))))
```

Забележка: Ако потоците **s1** и **s2** са безкрайни, проверките на двете гранични условия не са необходими (не са необходими проверките в първите две клаузи на ***cond***-а).

Умножаване на всички елементи на даден поток с константа:

```
(define (scale-stream const stream)
        (map-stream (lambda (x) (* x const))
        stream))
```

Явно (директно) конструиране на безкрайни потоци

По-горе генерирахме потоци с помощта на специални процедури - генератори. Възможен е и друг, директен вариант на дефиниране на потоци, при който се използват предимствата, които дава отложеното оценяване (използва се рекурсия върху самия генериран поток, а не се използват процедури - генератори).

Пример 1: Дефиниране на безкраен поток от единици: **(define ones (cons-stream 1 ones))**

Така **ones** се дефинира като точкова двойка, чийто ***car*** е 1 и чийто ***cdr*** е "пакетиран" израз, при форсирането на който ще се получи **[ones]**. Оценяването на ***cdr*** от тази точкова двойка отново дава 1 и "пакетиран" израз от описания вид и т.н. С други думи, вече генерираната част от потока се използва за дефиниране на следващите му елементи.

Пример 2: Генериране на безкраен поток от целите положителни числа.

```
(define ints
        (cons-stream 1
        (add-streams ones ints)))
```

Пример 3: Генериране на безкраен поток от числата на Фибоначи.

```
(define fibs
        (cons-stream 0
           (cons-stream 1
              (add-streams (tail fibs) fibs))))
```