Потоци. Работа с крайни и безкрайни потоци в езика Scheme

Дефиниция на потоците и основни процедури за работа с потоци

Дефиниция на понятието "поток"

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 на потока към неговия край). Както ще стане ясно по-нататък, потоците са структури от данни, чиято употреба е полезна най-вече в случаите, когато те имат голям брой (и дори безкрайно много) елементи.

Дефиниции на процедури за работа с потоци

Обща идея. По аналогия със списъците ще дефинираме някои процедури (включително такива от по-висок ред) за работа 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))))
```

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).

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)))))
```

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))))
```

Реализация на потоците в езика Scheme

Една възможност за реализация на потоците е свързана с използването на списъци (с представянето им във вид на списъци). Ако се избере този подход, тогава

head съвпада с ***car***, ***tail*** съвпада с ***cdr***,
cons-stream съвпада с ***cons***,
the-empty-stream съвпада с ***nil (null)***,
empty-stream? съвпада с ***null?*** .

Оказва се обаче, че списъците не са подходяща (ефективна) реализация на потоците.

Пример. Да се дефинира процедура, която връща като резултат второто просто число в интервала [10000, 1000000].

Примерно решение, което използва дефинираните по-горе процедури:

```
(define (second-prime)
   (head (tail (filter-stream
                  prime?
                  (enum-stream 10000 1000000)))))
```

Анализ на предложеното решение. В резултат на оценяването на обръщението към **enum-stream** (при реализация с използване на списъци) ще се формира списък от почти 1000000 елемента, като при това от този списък ще се използват само първите му десетина елемента (останалите почти 1000000 елемента изобщо няма да се използват). Разбира се, предложеният подход за решаване на тази задача далеч не е най-добрият, но той е подбран не само за да се покаже, че списъците не са подходящо средство за реализация на потоците. Идеята тук е, че много често програмистът не знае предварително каква част от дадена структура е действително необходима за решаването на конкретната задача. В такива случаи оценяването (формирането) на цялата структура може да се окаже излишно и дори крайно неефективно (когато тази структура има голям брой елементи).

Основната идея на действителната реализация на потоците, която ги прави подходящи структури за представяне на редици, съдържащи много голям брой елементи, дори безкрайно много елементи, е следната. 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 (примитивни). 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)))
```

Тази реализация на потоците вече е достатъчно ефективна в смисъла, в който проблемът беше обсъждан по-горе.

Например, ако се върнем на разгледания пример за намиране на второто просто число в интервала [10000, 1000000], процесът на оценка на израза

```
(head (tail (filter-stream
               prime?
               (enum-stream 10000 1000000))))
```

изглежда по следния начин.

Оценяването на **`(enum-stream 10000 1000000)`** води до оценяване на съответното обръщение към ***cons-stream***, чиято оценка е

```
(cons 10000 (delay (enum-stream 10001 1000000))).
```

Следователно, получава се поток, представен от точковата двойка с първи елемент 10000 и втори елемент - "пакетиран" израз, оценяването на който може да бъде извършено при необходимост. Този поток се филтрира от **filter-stream** (с помощта на **prime?**), като за целта се проверява дали неговият първи елемент (числото 10000) е просто число. Това в случая не е така, поради което се поражда (предизвиква) оценяване на обръщението към **filter-stream** с аргумент – опашката (***tail***) на същия поток. Обръщението към ***tail*** форсира оценяването на "пакетирания" ("отложения") израз, върнат от **enum-stream**, в резултат на което се получава **(cons 10001 (delay (enum-stream 10002 1000000)))**. Тук **filter-stream** отново установява, че ***car*** (***head***) от този израз (числото 10001 = 73.137) не е просто число и отново се обръща към ***tail***, като го форсира да продължи оценяването и т.н.

Този процес продължава до получаване на първото просто число в разглеждания интервал (числото 10007), при което **filter-stream** съгласно дефиницията си връща

```
(cons-stream (head stream)
                    (filter-stream proc
                                   (tail stream))).
```

При това в конкретния случай горният израз има вида

```
(cons 10007
      (delay
         (filter-stream prime?
            (cons 10008
               (delay
                  (enum-stream 10009 1000000))))))
```

Този резултат се предава като аргумент на ***tail*** - а в основния израз. Сега този ***tail*** форсира отложеното оценяване на **filter-stream**, който от своя страна форсира **enum-stream**, докато бъде намерено следващото просто число.

Тогава като аргумент на ***head*** - а в основния израз се предава

```
(cons 10009
      (delay (filter-stream
                 prime?
                 (cons 10010
                       (delay
                          (enum-stream 10011
                                       1000000))))))
```

и крайният резултат е 10009.

Обща бележка. По този начин действително се получи максимална ефективност, тъй като не се оцени нищо, което не е необходимо за формирането на крайния резултат. При това, моделът на оценяване чрез заместване е валиден и в този случай.

Работа с безкрайни потоци

Основни идеи

Демонстрираната по-горе техника на реализация на потоците действително позволява те да бъдат използвани като ефективни структури за представяне на редици, които могат да имат голям брой елементи и дори безброй много елементи.

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).

Дефиниции на някои полезни функции
за работа с (безкрайни) потоци

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))
```

Сливане на два сортирани потока в резултантен сортиран поток
(дублиращите се елементи се включват в резултата
в един екземпляр)

```
(define (merge-streams s1 s2)
   (cond ((empty-stream? s1) s2)
         ((empty-stream? s2) s1)
         (else (let ((h1 (head s1)) (h2 (head s2)))
                  (cond ((< h1 h2)
                         (cons-stream h1
                            (merge-streams (tail s1) s2)))
                        ((> h1 h2)
                         (cons-stream h2
                            (merge-streams s1 (tail s2))))
                        (else (cons-stream h1
                                  (merge-streams (tail s1)
                                     (tail s2)))))))))
```

Явно (директно) конструиране на безкрайни потоци

Обща идея. 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 (чрез рекурсия по отношение на дефинирания поток от единици).

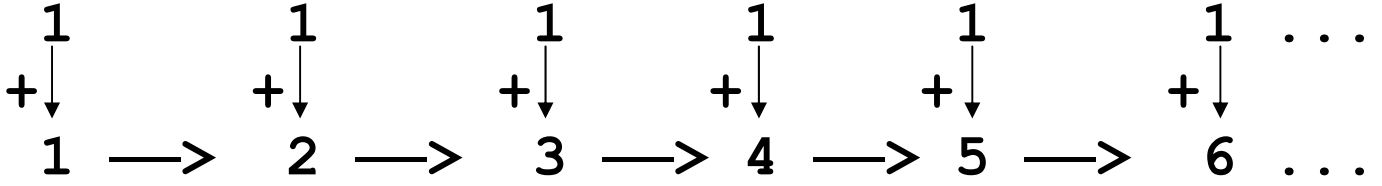
```
(define ones (cons-stream 1 ones))
```

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.

Идея. Всеки елемент на потока (от втория включително нататък) е равен на предишния, увеличен с 1.

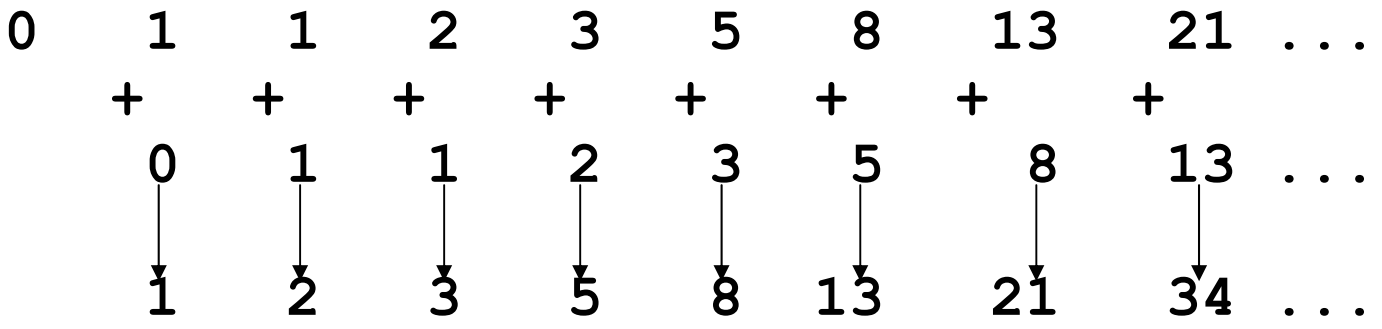
```
(define ints
   (cons-stream 1
                (add-streams ones ints)))
```



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

Идея. Всеки елемент на потока (от третия включително нататък) е сума на предишните два.

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



Допълнителни бележки върху реализацията на процедурите delay и force в езика Scheme

По-горе отбелязахме, че ***delay*** е специална форма, която не оценява аргумента си и за която

(delay <израз>) ⟶ (lambda () <израз>).

"Обратната" й функция ***force*** може да се дефинира по следния начин:

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

Тази реализация на ***delay*** и ***force*** е задоволителна и работи коректно, но може да бъде оптимизирана. В много случаи се налага един и същ отложен обект да се форсира повече от един път. Това може да доведе до голяма неефективност например при рекурсивни програми, които работят с потоци. Едно възможно решение е съответният отложен обект да се построи така, че в резултат на първото прилагане на ***force*** да се съхрани получената при форсирането стойност. При по-късни прилагания на ***force*** просто ще се връща съхранената стойност, без да се повтарят всички извършени изчисления. Следователно, новата идея е да се реализира ***delay*** като процедура със специална памет, предназначена за съхраняване на посочените стойности. Един от начините да се осъществи тази идея е да се използва дефинираната по-долу процедура **memo-proc**, която има като аргумент процедура без аргументи и връща като резултат съответната процедура с памет.

При първото си стартиране тази процедура запомня намерения резултат и при следващите извиквания просто го връща:

```
(define (memo-proc proc)
   (let ((already-run? #f) (result nil))
      (lambda ()
         (if (not already-run?)
             (begin (set! result (proc))
                    (set! already-run? #t)
                    result)
             result))))
```

Тогава ***delay*** е специална форма, която се дефинира така, че **(delay <израз>)** е еквивалентно на:

```
(memo-proc (lambda () <израз>)),
```

а дефиницията на ***force*** може да остане непроменена.

Някои допълнителни възможности на езика Scheme

Дефиниране на процедури с променлив брой параметри

Дефинирането на процедури с произволен (т.е. променлив) брой параметри може да се извърши с помощта на т. нар. ***unrestricted lambda*** дефиниция, чийто синтаксис е:

```
(lambda <променлива> <тяло>)
```

Тук променливата **<променлива>**, която не се загражда в скоби, както при традиционната ***lambda*** дефиниция, играе ролята на формален параметър. При прилагане на процедурата параметърът **<променлива>** се свързва със списъка от съответните фактически параметри и в новополучената среда се оценява тялото **<тяло>**.

Като пример за използване на ***unrestricted lambda*** дефиниция ще дефинираме процедура **add**, която има произволен брой аргументи (чиито оценки трябва да са числа) и връща като резултат сумата от оценките на аргументите си:

```
(define add
   (lambda x (apply + x)))
```

Примери за използване на процедурата **add:**

```
> (add 1 2 3 4 5)
15
> (add (+ 2 3) 5 10)
20
> (add)
0
```

Като по-сложен пример ще дефинираме процедура, която е аналог на вградената процедура ***writeln***:

```
(define new-writeln
   (lambda args
      (for-each display args)
      (newline)))
```

```
(define (for-each proc lst)
;;; Прилага последователно дадена процедура към
;;; елементите на даден списък. Интересни са
;;; случаите, когато процедурата proc предизвиква
;;; съществени странични ефекти
   (if (not (null? lst))
       (begin
          (proc (car lst))
          (for-each proc (cdr lst)))))
```

Дефиниране на нови специални форми в езика Scheme

Езикът Scheme не предлага стандартни средства за дефиниране на нови специални форми. Независимо от това, в повечето реализации на Scheme са предвидени определени възможности за тази цел.

Специална форма macro

В много реализации на езика Scheme нови специални форми могат да се дефинират под формата на макроси. За целта се използва специалната форма ***macro***:

`(macro <нова-спец-форма> <преобр-правило>)`

В процеса на оценяването на даден ***macro*** израз (на дадено обръщение към специалната форма ***macro***) в текущата среда се създава нова специална форма с име **<нова-спец-форма>**. Винаги когато интерпретаторът на Scheme прочете израз, който започва с името **<нова-спец-форма>**, той най-напред изпълнява процедурата **[<преобр-правило>]** върху прочетения израз. Като резултат се получава нов, преобразуван израз; след това интерпретаторът на Scheme се държи по същия начин, както би се държал, ако беше прочел директно преобразувания израз.

Като пример ще се опитаме да дефинираме нова специална форма с името **name!**, която ще искаме да действа подобно на специалната форма ***set!***, но аргументите ѝ ще бъдат подредени в обратен ред. По-точно, ще искаме

```
(name! x y)
```

да е еквивалентно на

```
(set! y x)
```

Дефиницията на новата специална форма може да бъде конструирана по следния начин:

```
(macro name!
   (lambda (expr)
      (list 'set! (caddr expr) (cadr expr))))
```

Примери, които илюстрират работата на новата специална форма

Пример 1

```
>(define a 1)
a
> (name! 3 a)
3
> a
3
```

Пример 2

```
> (define *test-variable* 0)
*test-variable*
> (define (increment-test-variable)
     (name! (1+ *test-variable*) *test-variable*)
     *test-variable*)
increment-test-variable
> (increment-test-variable)
1
> (increment-test-variable)
2
```

Специална форма quasiquote

За забрана на оценяването в езика Scheme се използва специалната форма ***quote*** (означава се за краткост с апостроф **'**). В частност, тази специална форма забранява оценяването на всички елементи на даден списък. Понякога обаче е полезно да се използва друг тип маркер, който означава селективно оценяване на определени порции от даден списък. Такова е предназначението на специалната форма ***quasiquote*** (обикновено за краткост се означава с обратен апостроф (backquote) `).

Когато интерпретаторът на Scheme оценява списък, предшестван от знака backquote, той се държи приблизително по същия начин, както когато списъкът е предшестван от апостроф (знака quote). Изключение прави само поведението му по отношение на тези от изразите, участващи в списъка, които са предшествани от запетая. Всички изрази, предшествани от запетая, се оценяват и оценките им се включват вместо тях в оценката на обръщението към специалната форма ***quasiquote***.

Примери

```
> `(a b c)
(a b c)
> `(a b ,(* 3 5) d)
(a b 15 d)
> (define c 3)
c
> `(a b ,c)
(a b 3)
> `(a b ,(car '(1 2)))
(a b 1)
> `(a ,* b)
(a #<PROCEDURE *> b)
> `((+ 2 3) ,(+ 2 3))
((+ 2 3) 5)
```

```
> (define a 1)
a
> `(a ,a ,(+ a 3) (+ ,a 3) ,(list 'a a))
(a 1 4 (+ 1 3) (a 1))
> (car `(,(* 8 5) ,(* 4 5)))
40
```

Понякога в обръщения към специалната форма ***quasiquote*** се среща запетая, последвана от знака **@** (the comma-at sign), както например в израза

```
`(1 ,@(list (+ 3 4) 8)).
```

Знакът **,@** означава, че следващият го израз се оценява и оценката му трябва да е списък. Този списък се включва в "разграден" вид (без най-външните скоби) в съответния списък от по-високо ниво, т.е. той бива заместен от поредицата от елементите си в оценката на обръщението към специалната форма ***quasiquote***.

Примери

```
> `(1 ,(list (+ 3 4) 8))
(1 (7 8))
> `(1 ,@(list (+ 3 4) 8))
(1 7 8)
> `(1 ,(list (* 8 5)) ,@(list (* 8 5)))
(1 (40) 40)
> `(list 1 ,@(map 1+ '(1 2 3)))
(list 1 2 3 4)
> `((1 ,@(list 2 3)) 4)
((1 2 3) 4)
> (list 1 '(2 3) `(4 ,(* 5 6) ,@(cons 7 '(8 9))))
(1 (2 3) (4 30 7 8 9))
```

```
> (define b '(1 2))
b
> `(b ,b ,(cdr b) ,@(cdr b))
(b (1 2) (2) 2)
> `(,@(append b '(3 4)))
(1 2 3 4)
```

Дефиниране на нови специални форми чрез macro и quasiquote

Най-често правилата за преобразуване (преобразуващите правила, rewriting rules) в ***macro*** дефинициите се конструират с помощта на специалната форма ***quasiquote*** (с помощта на backquote нотацията).

Така например, специалната форма **name!**, която дефинирахме по-горе, може да бъде дефинирана по-лесно и разбираемо по следния начин:

```
(macro name!
   (lambda (expr) `(set! ,(caddr expr)
                         ,(cadr expr))))
```

Като по-сложен пример ще дефинираме нова специална форма, която ще наречем **unless**. Специалната форма **unless** ще има два аргумента. Ако оценката на първия аргумент е ***false***, ще се оцени вторият аргумент и неговата оценка ще се върне като оценка на обръщението към **unless**; в противен случай оценката на обръщението към **unless** ще бъде ***false***.

Следват два примера, които илюстрират очакваното поведение на **unless**:

```
> (unless (= 0 0) 'hello)
()
> (unless (= 0 1) 'hello)
hello
```

Дефиницията на специалната форма **unless** може да бъде конструирана по следния начин:

```
(macro unless
   (lambda (expr)
      `(if (not ,(cadr expr)) ,(caddr expr) #f)))
```

Специални форми с действие, аналогично на това на формите ***delay*** и ***cons-stream***, които стоят в основата на реализацията на потоците в Scheme, могат да бъдат дефинирани както следва:

```
(macro delay
   (lambda (expr)
      `(lambda () ,(cadr expr))))

(macro cons-stream
   (lambda (expr)
      `(cons ,(cadr expr) (delay ,(caddr expr)))))
```

В заключение ще отбележим, че специалната форма ***macro*** действа по дискутирания досега начин не във всички реализации на езика Scheme. В повечето среди за програмиране на Scheme обаче се поддържат средства, подобни на разгледаните, които позволяват да се конструират нови специални форми с помощта на правила за преобразуване.

Допълнителни сведения за средите в езика Scheme

В много реализации на езика Scheme средите са не само средство за описание на модела на оценяване, но също и системни обекти, с които потребителят може да работи.

За (двукратно) оценяване на даден израз в езика Scheme се използва примитивната процедура ***eval***:

`(eval <израз>) ⟶ [[<израз>]],`

като оценяването се извършва в текущата среда.

В действителност примитивната процедура ***eval*** може да бъде изпълнявана и с два аргумента, като вторият аргумент определя специфичната среда, в която да се извърши оценяването:

(eval <израз> <среда>).

Остава да покажем как може да се цитира конкретна среда в обръщение към процедурата ***eval***.

В PC Scheme променливата ***user-initial-environment*** е свързана с глобалната (началната потребителска) среда. С други думи, тази променлива може да се използва като втори аргумент на ***eval*** за цитиране на глобалната среда.

Например:

```
> (eval '+ user-initial-environment)
#<Procedure +>
```

Нещата стават по-интересни, когато като втори аргумент на ***eval*** се използва среда, различна от глобалната. Една възможност да се цитира друга среда е свързана с използването на примитивната процедура ***the-environment***.

Като пример нека разгледаме следната дефиниция на процедура:

```
(define (sample-env-maker)
   (let ((a 2) (b 7))
      (the-environment)))
```

В тялото на тази процедура се прави обръщение към ***the-environment*** без аргументи. Оценката на обръщението към процедурата **sample-env-maker** е локалната среда, в която името **a** е свързано със стойност 2, а името **b** е свързано със стойност 7. След оценката на горната дефиниция можем да създадем нова среда с име **env-1**:

(define env-1 (sample-env-maker))

В тази нова среда можем да оценяваме изрази по известния вече начин:

```
> (eval 'a env-1)
2
> (eval '(* a b) env-1)
14
```

Друга свързана с темата примитивна процедура в PC Scheme е ***procedure-environment***. Процедурата ***procedure-environment*** има един аргумент. Този аргумент се оценява и оценката му трябва да е процедура (процедурен обект). Оценката на обръщението към ***procedure-environment*** е средата, асоциирана с този процедурен обект.

В повечето случаи средата, асоциирана с даден процедурен обект, съвпада с глобалната среда. Интерес по отношение на работата на процедурата ***procedure-environment*** представляват процедурните обекти, които са дефинирани в среда, представляваща разширение на глобалната.

Пример. Нека дефинираме следната процедура:

```
(define sample-procedure
   (let ((a 5)) (lambda (number) (* a number))))
```

Тук името **sample-procedure** се свързва с процедурен обект, с който е асоциирана среда, съдържаща свързване за името **a**. Следващият пример илюстрира възможността за оценяване на изрази в средата, асоциирана с името **sample-procedure**:

```
> (eval 'a (procedure-environment sample-procedure))
5
```

Като заключителен пример ще разгледаме поведението на процедурата **withdraw**, дефинирана по следния начин:

```
(define withdraw
   (let ((balance 100))
      (lambda (amount)
         (if (>= balance amount)
             (begin (set! balance
                (- balance amount)) balance)
             "Not possible!"))))
```

Работата на тази процедура беше анализирана детайлно при разглеждането на модела на средите, затова сега ще покажем само как може да се проследи промяната на стойността на променливата **balance** при поредица от обръщения към **withdraw**:

```
> (eval 'balance (procedure-environment withdraw))
100
> (withdraw 30)
70
> (eval 'balance (procedure-environment withdraw))
70
> (withdraw 20)
50
> (eval 'balance (procedure-environment withdraw))
50
```