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 (atom? a) (not (pair? a)))
;;; Функцията atom? ще бъде използвана
;;; наготово по-нататък в курса
(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 (в рекурсивен стил), 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 lst)
   (if (null? lst)
       0
       (+ (car lst) (sum-list (cdr lst)))))
```

Умножаване на елементите на даден списък

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

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

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

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

`(accum + 0 lst)` **`—>`** сумата от елементите на **[lst]** (действа като **sum-list**);

`(accum * 1 lst)` **`—>`** произведението на елементите на **[lst]** (действа като **product-list**);

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

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 Racket, 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 опростения 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
```