

Работа с асоциативни списъци в езика Scheme

Същност на асоциативните списъци

Дефиниция. *Асоциативен списък (А-списък)* е всеки списък, чийто елементи имат вида (<ключ>.<асоциация>). В много диалекти на Lisp се изисква <ключ> да бъде атом (дори само символен атом). В Scheme такова изискване няма - <ключ> тук може да бъде произволен S-израз. <асоциация> също може да бъде произволен S-израз.

Общ вид на асоциативните списъци:

$$((\langle \text{key}_1 \rangle . \langle \text{value}_1 \rangle) (\langle \text{key}_2 \rangle . \langle \text{value}_2 \rangle) \dots (\langle \text{key}_n \rangle . \langle \text{value}_n \rangle))$$

Забележка. Тъй като всеки непразен списък е точкова двойка, то от горната дефиниция следва, че всеки списък, чийто елементи са непразни списъци, също е А-списък.

Примитивни процедури за работа с асоциативни списъци

В Scheme съществуват примитивни (вградени) процедури само за търсене по зададен ключ в даден асоциативен списък. Най-общо тези процедури действат по следния начин: по дадени S-израз и A-списък те връщат първия от елементите на A-списъка, чийто ключ (т.е. чийто първи елемент) е равен (в смисъл на ***eq?***, ***eqv?*** или ***equal?***) на дадения S-израз; ако никой от елементите на A-списъка няма ключ, равен (в съответния смисъл) на дадения S-израз, то процедурите за търсене връщат резултат ***()***.

Примитивни процедури в Scheme за търсене в асоциативен списък:

(assq key a-list) —> първия елемент на **[a-list]**, който има ключ, равен (в смисъл на **eq?**) на **[key]**, или **()**, ако такъв елемент не съществува.

(assv key a-list) —> първия елемент на **[a-list]**, който има ключ, равен (в смисъл на **eqv?**) на **[key]**, или **()**, ако такъв елемент не съществува.

(assoc key a-list) —> първия елемент на **[a-list]**, който има ключ, равен (в смисъл на **equal?**) на **[key]**, или **()**, ако такъв елемент не съществува.

Забележка. В DrScheme функциите ***assq***, ***assv*** и ***assoc*** връщат стойност **#f**, ако в съответния асоциативен списък няма елемент със зададения ключ.

Примерна дефиниция на процедура, чието действие съвпада с това на примитивната процедура ***assq***:

```
(define (lookup key a-list)
  (cond ((null? a-list) '())
        ((eq? (caar a-list) key) (car a-list))
        (else (lookup key (cdr a-list)))))
```

В Scheme, а също и в останалите диалекти на Lisp, не са предвидени примитивни процедури за добавяне и изтриване на елементи в асоциативен списък. Затова има смисъл да бъдат дефинирани такива процедури.

Дефиниция на процедура за изтриване на елемент от асоциативен списък (с използване на **eq?** като процедура за сравнение):

```
(define (rem-assoc key a-list)
  (cond ((null? a-list) '())
        ((eq? key (caar a-list)) (cdr a-list))
        (else (cons (car a-list)
                      (rem-assoc key
                                (cdr a-list))))))
```

Дефиниция на процедура за добавяне на елемент към асоциативен списък:

```
(define (put-assoc pair a-list)
  (cons pair (rem-assoc (car pair) a-list)))
```

Забележка. Горните дефиниции са смислени при спазване на естественото ограничение в един асоциативен списък да няма повече от един елемент с даден ключ (т.е. да няма елементи с еднакви ключове).

Работа със списъци от свойства (Р-списъци) в езика Scheme

Същност на списъците от свойства

Досега разглеждахме символните атоми (символите) в Scheme като примитивни данни, които се характеризират с техните имена (които са идентификатори). Те могат да бъдат използвани като променливи, на които се присвояват стойности. Дефинирането на дадена променлива и присвояването на стойността ѝ могат да се извършват с помощта на специалната форма ***define***. В този смисъл се казва, че всеки символен атом се характеризира с името си и евентуално - със стойността си.

Фактически, освен с име и стойност символните атоми (символите) могат да се характеризират и с т. нар. **свойства** (*properties*), които се обединяват в т. нар. **списък от свойства** или **Р-списък** (*property list, P-list*) на съответния символен атом. Всяко свойство на даден символен атом се характеризира със свои **име** (което задължително е символен атом) и **стойност** (която може да бъде произволен S-израз). За работа с Р-списъка на даден символен атом са предвидени специални примитивни процедури: **getprop** (за извличане на стойност на свойство), **putprop** (за задаване на стойност на свойство), **remprop** (за премахване на свойство), **proplist** (за извличане на Р-списъка на даден символен атом).

Примитивни процедури за работа с Р-списъци

Задаване на (стойност на) свойство - putprop

Общ вид на обръщението:

```
(putprop <симв. атом> <стойност на свойство>  
  <име на свойство>)
```

Действие. Ако в Р-списъка на символния атом, който е оценка на първия аргумент, няма свойство с име [**<име на свойство>**], то в Р-списъка на този атом се създава свойство с посоченото име и стойност, равна на [**<стойност на свойство>**]. Ако в Р-списъка на дадения символен атом вече има свойство със зададеното име, то неговата стойност се променя, като старата се изтрива и се заменя с [**<стойност на свойство>**].

Оценката на обръщението към ***putprop*** по стандарт е неопределена.

Извличане на стойност на свойство - `getprop`

Общ вид на обръщението:

`(getprop <симв. атом> <име на свойство>)`

Оценката на обръщението към ***getprop*** е стойността на свойството с име **`[<име на свойство>]`** за атома **`[<симв. атом>]`** или **`()`**, ако няма такова свойство в Р-списъка на дадения символен атом.

Премахване на свойство - *remprop*

Общ вид на обръщението:

(*remprop* <симв. атом> <име на свойство>)

Действие. Премахва свойството с име [<име на свойство>] от Р-списъка на символния атом [<симв. атом>]. Ако в Р-списъка на дадения атом няма свойство с посоченото име, то ***remprop*** не предизвиква никакво действие.

Оценката на обръщението към ***remprop*** по стандарт е неопределена.

Извличане на Р-списъка на даден символен атом - `proplist`

Общ вид на обръщението:

`(proplist <симв. атом>)`

Оценката на обръщението към ***proplist*** е списък, чийто нечетни елементи съвпадат с имената на свойствата, а четните - със стойностите на съответните свойства от Р-списъка на [***симв. атом***]. Ако в Р-списъка на дадения атом не са зададени никакви свойства, то ***proplist*** връща стойност `()`.

Примери

```
> (putprop 'patrick ' (robert dorothy) 'parents)
(robert dorothy)
> (putprop 'patrick ' (sarah) 'children)
(sarah)
> (getprop 'patrick 'children)
(sarah)
> (getprop 'patrick 'age)
()
> (proplist 'patrick)
(children (sarah) parents (robert dorothy))
```

```
> (putprop 'patrick  
      (cons 'john  
            (getprop 'patrick 'children))  
      'children)  
(john sarah)  
> (remprop 'patrick 'parents)  
(patrick children (john sarah))
```

Забележка. В DrScheme функциите за работа с Р-списъци не са вградени. Част от тях (***putprop*** и ***getprop***) са дефинирани в системната библиотека ***compat.ss*** и за да могат да бъдат използвани, посочената библиотека трябва да бъде предварително заредена. Същото се отнася и за други “стандартни” функции, които не са вградени в DrScheme (например за предиката ***atom?***).

Функцията ***getprop*** в DrScheme връща оценка #f, когато в Р-списъка на дадения символен атом не е дефинирано свойство със зададеното име. Редът на последните два аргумента в обръщението към функцията ***putprop*** в DrScheme е обратен на стандартния.

Пример за използване на функции за работа с Р-списъци в DrScheme:

```
> (require (lib "compat.ss"))  
> (putprop 'a 'pr1 '(q w e r t y))  
> (getprop 'a 'pr1)  
(q w e r t y)  
> (getprop 'a 'pr2)  
#f
```

Примерна задача

Нека **[gl]** е списък от символни атоми от вида **N<цяло без знак>**, които представят учениците от даден клас. Всеки от тези атоми има по три свойства:

ime – със стойност списък от трите имена на ученика;

srusp – със стойност реално число, равно на средния успех на ученика;

brots – със стойност цяло число, равно на броя на отсъствията на ученика.

Да се състави процедура, която формира списък от фамилните имена на тези ученици, които имат успех, по-висок от 4.50, и са направили по-малко от 10 отсъствия.

Решение

Първи начин (чрез рекурсия)

```
(define (spis gl)
  (cond ((null? gl) '())
        ((and (> (getprop (car gl) 'srusp) 4.50)
              (< (getprop (car gl) 'brots) 10))
         (cons (caddr (getprop (car gl) 'ime))
               (spis (cdr gl))))
        (else (spis (cdr gl)))))
```

Втори начин (чрез изобразяване)

```
(define (spis gl)
  (apply append
    (map (lambda (x)
           (if (and (> (getprop x 'srusp)
                     4.50)
                 (< (getprop x 'brots)
                   10))
             (cddr (getprop x 'ime))
             ' ()))
    gl) ))
```

Характерни приложения на асоциативните списъци и списъците от свойства

Работа с дървета в езика Scheme

Примерна задача. Нека **[tree]** е списък, който описва някакво дърво. Да се избере подходящо представяне на дървото, зададено чрез **[tree]**, и да се дефинират следните процедури:

(succs node) —> списък от преките наследници на възела **[node]** в дървото **[tree]**;

(leaf? node) —> **#t**, ако възелът **[node]** е лист в дървото **[tree]**, и **#f** - в противния случай;

(list-of-leaves node) —> списък от листата на дървото **[tree]**, които са наследници на възела **[node]**;

(list-of-paths node) —> списък от пътищата в дървото **[tree]** от възела **[node]** до всички листа на дървото, които са наследници на **[node]**.

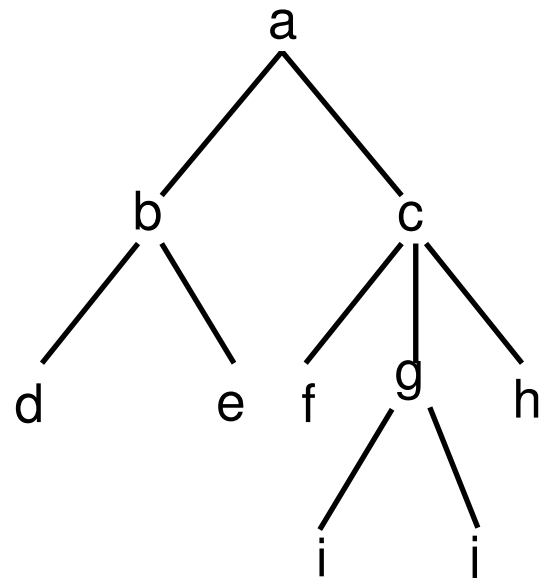
Решение

Представяне - първи начин (чрез апарата на Р-списъците)

Нека **[tree]** е списък от символни атоми, които са имена на възлите от дървото (тези имена трябва да са уникални). Всеки от тези атоми, който не е име на лист в дървото, има свойство с име **next**, чиято стойност е списък от имената на преките наследници на съответния възел.

Пример

```
(define tree '(a b c d e f g h i j))  
(putprop 'a '(b c) 'next)  
(putprop 'b '(d e) 'next)  
(putprop 'c '(f g h) 'next)  
(putprop 'g '(i j) 'next)
```



Представяне - втори начин (чрез апарата на А-списъците)

$$\begin{aligned} \text{Нека } tree \longrightarrow & \left(\left(\langle \text{възел}_1 \rangle . \left(\langle \text{насл}_{11} \rangle \langle \text{насл}_{12} \rangle \dots \right) \right) \right. \\ & \left. \left(\langle \text{възел}_2 \rangle . \left(\langle \text{насл}_{21} \rangle \langle \text{насл}_{22} \rangle \dots \right) \right) \right. \\ & \left. \dots \right) = \\ = & \left(\left(\langle \text{възел}_1 \rangle \langle \text{насл}_{11} \rangle \langle \text{насл}_{12} \rangle \dots \right) \right. \\ & \left(\langle \text{възел}_2 \rangle \langle \text{насл}_{21} \rangle \langle \text{насл}_{22} \rangle \dots \right) \\ & \left. \dots \right) \end{aligned}$$

Тук $\langle \text{възел}_i \rangle$ са имената на възлите в дървото, които не са листа (и тук имената на възлите трябва да са уникални), а $\langle \text{насл}_{ij} \rangle$ са имената на преките наследници на $\langle \text{възел}_i \rangle$.

За дървото от горния пример представянето от разглеждания тип ще бъде следното:

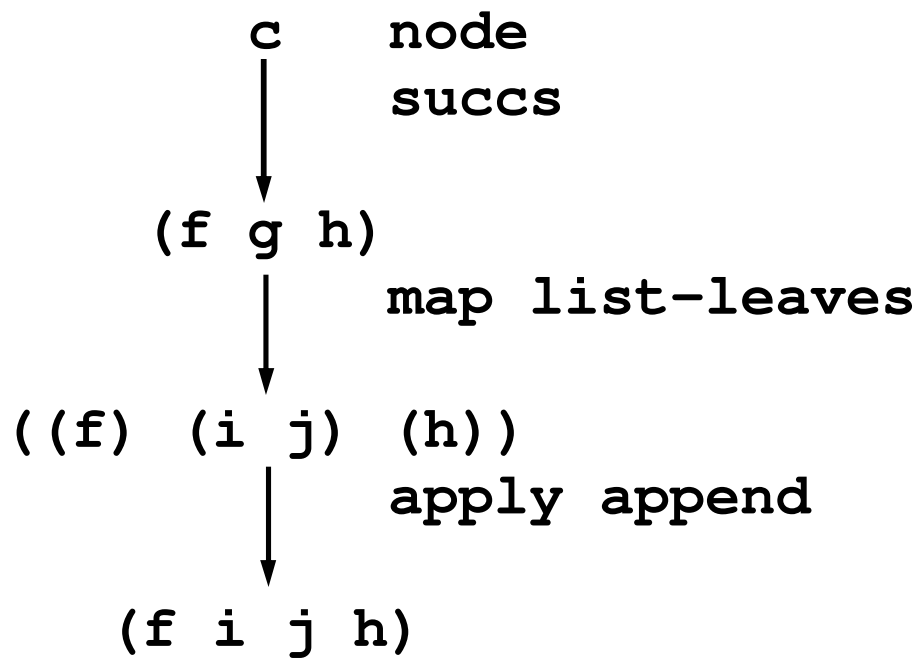
```
(define tree ' ((a b c)
                 (b d e)
                 (c f g h)
                 (g i j)))
```

Дефиниции

```
(define (succs node)
;;; Дефиниция при първото представяне
  (let ((lst (getprop node 'next)))
    (if lst lst ' ())))
```

```
(define (succs node)
;;; Дефиниция при второто представяне
  (let ((lst (assq node tree)))
    (if lst (cdr lst) ' ())))
```

```
(define (leaf? node)
  (null? (succs node)))
```

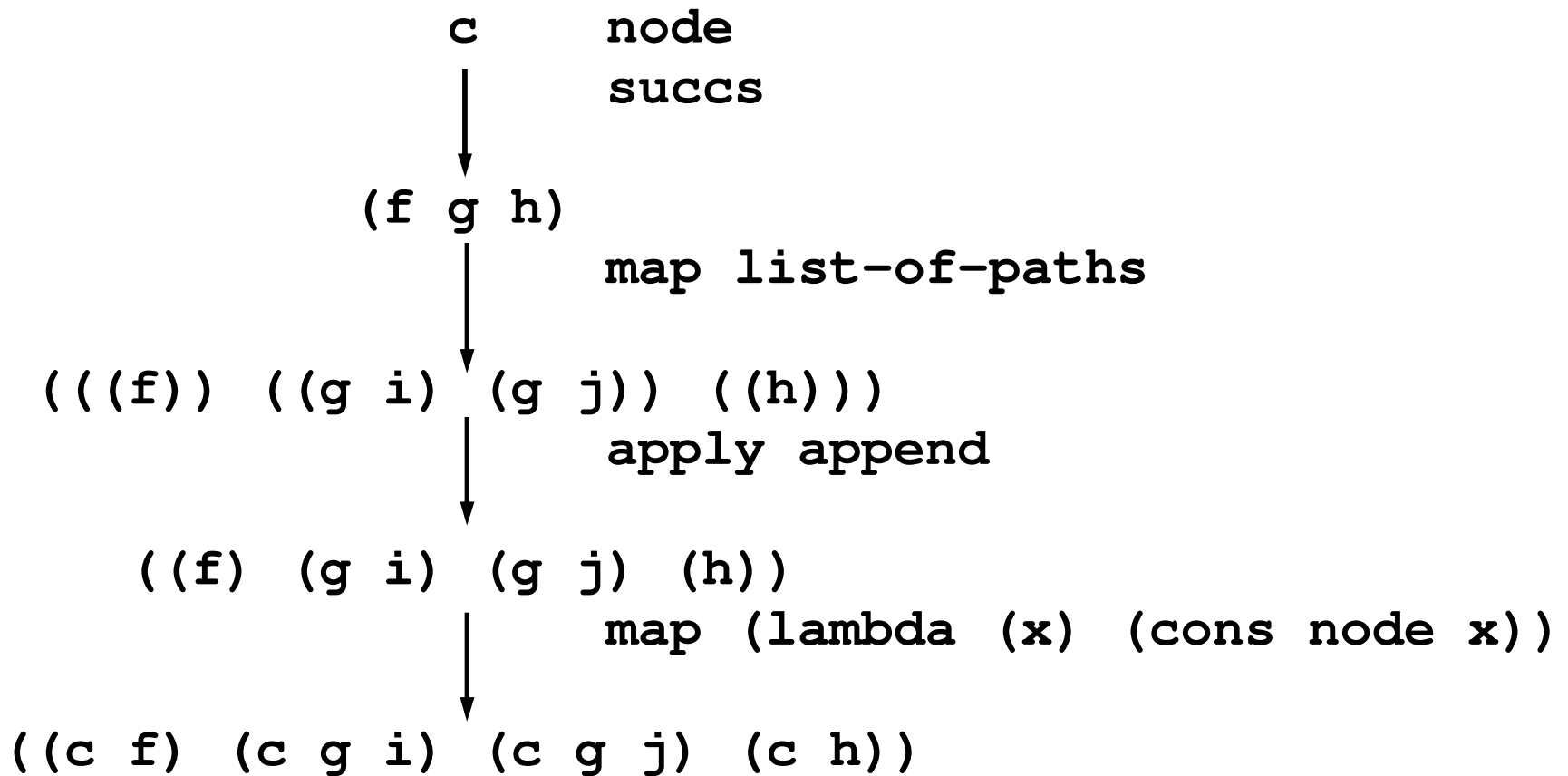


```

(define (list-leaves node)
  ;; Връща ([node]), когато [node] е лист от
  ;; дървото
  (if (leaf? node)
      (list node)
      (apply append (map list-leaves
                          (succs node))))))

(define (list-of-leaves node)
  ;; Връща (), когато [node] е лист от дървото
  (if (leaf? node)
      '()
      (list-leaves node)))

```



```
(define (list-of-paths node)
  ;; Връща ([node]), когато [node] е лист от
  ;; дървото
  (if (leaf? node)
      (list (list node))
      (map (lambda (x) (cons node x))
           (apply append
                   (map list-of-paths
                        (succs node))))))
```

Забележки

1) Както се вижда от горните дефиниции, само една от тях (тази на процедурата **succs**) зависи от избраното представяне на дървото.

2) Дискутираните по-горе представяния имаха една обща характерна черта - за всеки възел, който не е лист в дървото, бяха зададени неговите преки наследници. Такъв тип представяне е удобно, когато се извършва търсене в посока от корена към листата на дървото. Възможен е и друг тип представяне, при което за всеки възел, който не съвпада с корена на дървото, е даден неговият родител. Например, при използване на механизма на Р-списъците **[tree]** отново е списък от имената на възлите на дървото, а всеки атом, който е име на възел (с изключение на корена на дървото), има свойство с име **pred** и стойност - името на родителя (прекия предшественик) на този възел. Такова представяне е удобно, когато се извършва търсене в посока от листата към корена на дървото.

Работа с графи в Scheme

Примерна задача. Даден е ориентиран граф G . Да се избере и опише накратко подходящо представяне на G и да се състави програма, която:

а) проверява дали дадена крайна редица от възли P представлява коректен път без цикли в графа G ;

б) проверява дали съществува път между два дадени възела a и b от графа G , който е съставен от не повече от n дъги на графа;

в) намира дължината на най-късия прост цикъл в графа G и един от простите цикли в G , който има такава дължина. Прост цикъл в графа G ще наричаме всеки път в G , съставен от поне три възела, в който първият и последният възел съвпадат, а останалите са различни помежду си и са различни от първия възел.

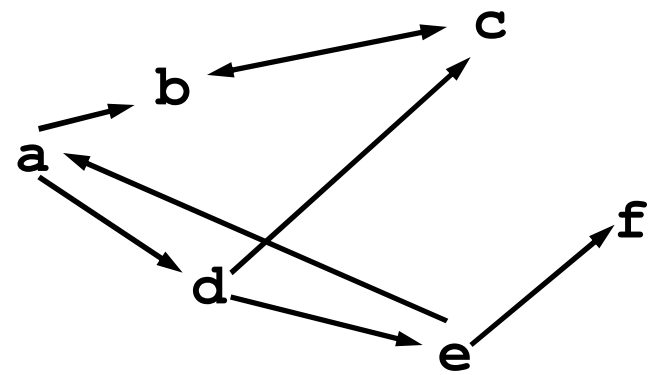
Решение

Представяне - първи начин (чрез апарата на Р-списъците)

[G] е списък от имената на всички възли в графа. Имената на възлите са уникални символни атоми и всеки от тези атоми, който съответства на възел, от който започва поне една дъга от графа (т.е. който има поне един наследник), има в Р-списъка си свойство с име **next**, чиято стойност е списък от имената на възлите, които са краища на дъгите в графа, започващи от въпросния възел.

Пример

```
(define g '(a b c d e f))  
(putprop 'a '(b d) 'next)  
(putprop 'b '(c) 'next)  
(putprop 'c '(b) 'next)  
(putprop 'd '(c e) 'next)  
(putprop 'e '(a f) 'next)
```



Представяне - втори начин (чрез апарата на А-списъците)

Отново имената на възлите в графа са уникални символни атоми и

$$\begin{aligned} G \longrightarrow & \left(\left(\langle \text{възел}_1 \rangle . \left(\langle \text{насл}_{11} \rangle \langle \text{насл}_{12} \rangle \dots \right) \right) \right. \\ & \left. \left(\langle \text{възел}_2 \rangle . \left(\langle \text{насл}_{21} \rangle \langle \text{насл}_{22} \rangle \dots \right) \right) \right. \\ & \left. \dots \right) = \\ = & \left(\left(\langle \text{възел}_1 \rangle \langle \text{насл}_{11} \rangle \langle \text{насл}_{12} \rangle \dots \right) \right. \\ & \left(\langle \text{възел}_2 \rangle \langle \text{насл}_{21} \rangle \langle \text{насл}_{22} \rangle \dots \right) \\ & \left. \dots \right) \end{aligned}$$

Тук $\langle \text{възел}_i \rangle$ са имената на възлите, от които започва поне една дъга в графа, а $\langle \text{насл}_{ij} \rangle$ са имената на краищата на дъгите, започващи от $\langle \text{възел}_i \rangle$.

В горния пример:

```
(define g ' ((a b d) (b c) (c b) (d c e) (e a f)))
```

Забележка. В някои случаи (а такава е и нашата задача) е по-добре това представяне да се модифицира така, че като ключове на елементите на **[G]** да участват всички възли. Тогава лесно може да се извлече броят на възлите в графа, а при необходимост могат да се извлекат и самите възли. В такъв случай дефиницията на **G** ще има следния вид:

```
(define g ' ((a b d) (b c) (c b) (d c e)
              (e a f) (f)))
```

Дефиниции

а) Отново има смисъл да бъде дефинирана помощна процедура **succs**, която по даден възел **[node]** връща списък от имената на всички възли, които са преки наследници на **[node]**, т.е. които са краища на дъги в графа, започващи от **[node]**.

Дефиниция на **succs** при първото представяне:

```
(define (succs node)
  (let ((lst (getprop node 'next)))
    (if lst lst ' ())))
```

Дефиниция на **succs** при второто представяне:

```
(define (succs node)
  (let ((lst (assq node g)))
    (if lst (cdr lst) ' ())))
```

Дефиниция на процедурата от т. а):

```
(define (correct-path p)
  (and (is-a-path p) (not (cycled p))))
```

```
(define (cycled p)
  (cond ((null? p) #f)
        ((memq (car p) (cdr p)) #t)
        (else (cycled (cdr p))) ))
```

```
(define (is-a-path p)
  (cond ((null? p) #t)
        ((null? (cdr p)) (is-a-node (car p)))
        ((memq (cadr p) (succs (car p)))
         (is-a-path (cdr p))) ))
```

```
(define (is-a-node node)      ; Дефиниция при
  (if (memq node g) #t))      ; първото представяне
```

```
(define (is-a-node node)      ; Дефиниция при
  (if (assq node g) #t))      ; второто представяне
                                ; (модифициран
                                ; вариант)
```

б) Целта тук е дефинирането на процедура (**connected a b n**), която проверява дали съществува път между възлите **a** и **b**, съставен от не повече от **n** дъги. Най-напред обаче ще дефинираме някои помощни, но много важни процедури за генериране на пътища в графа.

```
(define (gen-next path)
;;; При даден път в графа, представен във вид на
;;; списък от съответните възли, взети в обратен
;;; ред, връща списък от всички пътища, които са
;;; негови преки наследници
  (map (lambda (x) (cons x path))
       (succs (car path))))
```

```
(define (generate-paths lp)
  ;; При даден списък от пътища връща списък от
  ;; нови пътища, които са всички преки наследници
  ;; на дадените
  (apply append (map gen-next lp)))

(define (connected a b n)
  (cond ((eq? a b) #t)
        ((<= n 0) #f)
        (else (connect1 (list (list a)) b n))))
```

```

(define (connect1 lp b n)
  ;; lp - списък от вече изследвани пътища,
  ;; представени във вид на списъци от съответните
  ;; възли, взети в обратен ред
  (cond ((<= n 0) #f)
        ((null? lp) #f)
        (else (let ((lnp (generate-paths lp)))
                  (if (assq b lnp)
                      #t
                      (connect1 lnp b
                                (- n 1)))))))

```

в) Тук ще работим, като имаме предвид само първото представяне (това, което използва механизма на Р-списъците). Освен това, ще предполагаме, че в графа няма (не се допускат) "примки" от вида $a \rightarrow a$.

Целта тук е дефинирането на процедура (**minc-graph**), която решава поставената задача.

```
(define (minc-graph)
  (min-cycle (map list g)))
```

```
(define (min-cycle chains)
  (if (null? chains)
      '()
      (let ((result (has-cycles chains)))
        (if (null? result)
            (min-cycle (generate-paths chains))
            (list (length result) result))))))
```

```
(define (has-cycles chains)
  (cond ((null? chains) '())
        ((cycle? (car chains))
         (reverse (car chains)))
        (else (has-cycles (cdr chains)))))

(define (cycle? chain)
  (and (>= (length chain) 3)
       (eq? (car chain)
             (car (last-pair chain)))))
```

Забележка. Процедурата *last-pair* е вградена според стандарта на езика Scheme. Тя има един аргумент, чиято оценка трябва да бъде списък. Оценката на обръщението към *last-pair* е списък от последния елемент на оценката на аргумента.

Обяснение на предложеното решение за т. б) и в)

б) Постепенно се генерират всички пътища (вериги), които започват от възела **a**. На първата стъпка се генерират всички пътища с дължина 1 (за дължина на пътя ще смятаме броя на дъгите, от които е съставен той); на втората стъпка се генерират всички пътища с дължина 2, които се получават, като към резултатите от първата стъпка се добавят новите отсечки - "наследници" на краищата на резултатите от тази стъпка и т.н. Работата приключва при три възможни случая:

- достига се **b**: успех;
- направени са повече от **n** стъпки, без да се достигне **b**: неуспех;
- не могат да се генерират нови пътища, а **b** не е достигнат: неуспех.

в) Тръгва се паралелно от всички възли на графа, като постъпково се генерират всички възможни пътища. Процесът приключва при два възможни случая:

- на дадена стъпка е генериран прост цикъл. Тогава първият намерен (генериран) такъв цикъл е решение на задачата;

- не могат да се генерират нови пътища. В този случай в графа няма цикли.