Работа с асоциативни списъци в езика Racket

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 Racket такова изискване няма - **<ключ>** тук може да бъде произволен S-израз. **<асоциация>** също може да бъде произволен S-израз.

Общ вид на асоциативните списъци:

((<key$_1$>.<value$_1$>) (<key$_2$>.<value$_2$>) ...
(<key$_n$>.<value$_n$>))

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 Racket 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 А-списък те връщат първия от елементите на А-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 А-списъка няма ключ, равен (в съответния смисъл) на дадения S-израз, то процедурите за търсене връщат резултат ***#f***.

Примитивни процедури в Racket 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 #***f***, 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 #***f***, 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 #***f***, ако такъв елемент не съществува.

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

```
(define (lookup key a-list)
   (cond [(null? a-list) #f]
         [(eq? (caar a-list) key) (car a-list)]
         [else (lookup key (cdr a-list))]))
```

В Racket, 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).

Характерни приложения на асоциативните списъци

Работа с дървета в езика Racket

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]**.

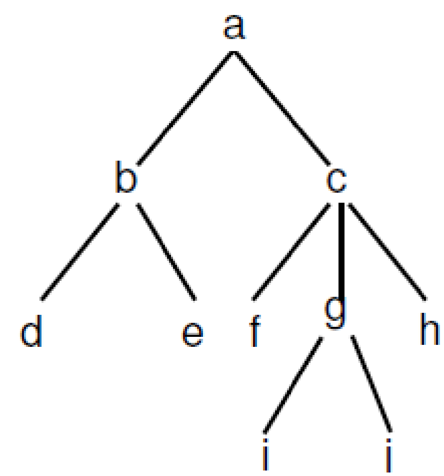
Решение

Представяне - чрез апарата на А-списъците

Нека **tree** —> **((<възел$_1$>.(<насл$_{11}$> <насл$_{12}$> ...))**
(<възел$_2$>.(<насл$_{21}$> <насл$_{22}$> ...))
. . .) =
= ((<възел$_1$> <насл$_{11}$> <насл$_{12}$> ...)
(<възел$_2$> <насл$_{21}$> <насл$_{22}$> ...)
. . .)

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

Примерно дърво



Това дърво ще има следното представяне:

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

Дефиниции

```
(define (succs node)
   (let ([lst (assq node tree)])
      (if lst (cdr lst) '())))

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

```
        c   node
            succs

     (f g h)
            map list-leaves

((f) (i j) (h))
            apply append

    (f i j h)
```

```
(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)))
```

```
            c      node
            ↓      succs
         (f g h)
            ↓      map list-of-paths
  (((f)) ((g i) (g j)) ((h)))
            ↓      apply append
     ((f) (g i) (g j) (h))
            ↓      map (lambda (x) (cons node x))
((c f) (c g i) (c g j) (c h))
```

```scheme
(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)))) ))
```

Забележка

Дискутираното по-горе представяне се характеризира с това, че за всеки възел, който не е лист в дървото, са 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. Такова представяне е удобно, когато се извършва търсене в посока от листата към корена на дървото.

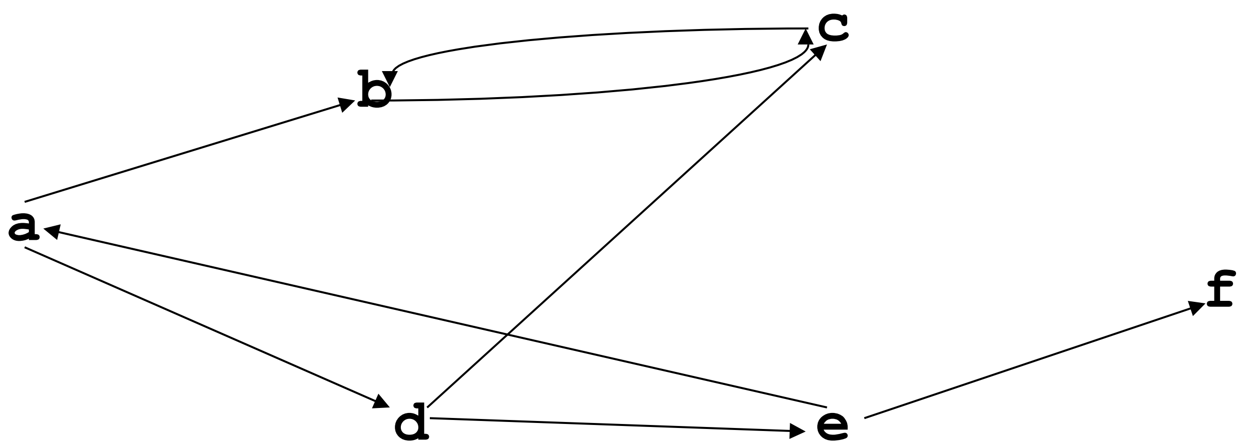
Работа с графи в Racket

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.

Примерен граф



Решение

Представяне - чрез апарата на А-списъците

Имената на възлите в графа са уникални символни атоми и

```
G ——> ((<възел1>.(<насл11> <насл12> ... ))
       (<възел2>.(<насл21> <насл22> ... ))
         . . . ) =
    = ((<възел1> <насл11> <насл12> ... )
       (<възел2> <насл21> <насл22> ... )
         . . . )
```

Тук **<възел$_i$>** са имената на възлите, от които започва поне една дъга в графа, а **<насл$_{ij}$>** са имената на краищата на дъгите, започващи от **<възел$_i$>**.

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 функция **succs**, която по даден възел **[node]** връща списък от имената на всички възли – преки наследници на **[node]**, т.е. които са краища на дъги в графа, започващи от **[node]**.

Дефиниция на функцията **succs**:

```
(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))]))
```

```scheme
(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 (assq node g) #t))    ; модифицираното
                             ; представяне
```

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
;;; продължават този път с една дъга
   (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 продължения на
;;; дадените с по една дъга
   (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))))]))
```

Обяснение на предложеното решение за т. 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 нови отсечки до всички "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.