ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

Магдалина Тодорова
magda@fmi.uni-sofia.bg
todorova_magda@hotmail.com
кабинет 517, ФМИ

Тема 11

Асоциативни списъци

1. Дефиниции и представяне

Дефиниция: Списък от вида

$((a_1 . e_1) (a_2 . e_2) \ldots (a_n . e_n))$

където $a_1, a_2, \ldots, a_n$ са различни атоми, а $e_1, e_2, \ldots, e_n$ са произволни S-изрази, се нарича ***асоциативен*** или ***A-списък***.

Първият елемент на всяка двойка се нарича ***ключ***, а вторият - ***асоциация***.

Примери: Списъците

((a . b) (c . (c d e)) (m . 1))

((x . 1) (y . 2) (z . 3) (m . 4) (m1 . 5))

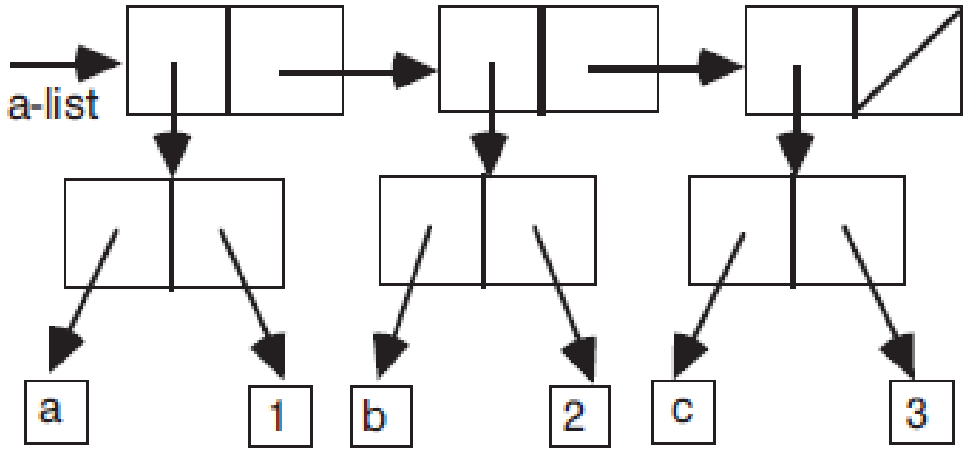
((x . (x . y)) (y . (y . z)) (k . (r s t n)))

са асоциативни.

Представяне: Асоциативният списък

((a . 1) (b . 2) (c . 3))

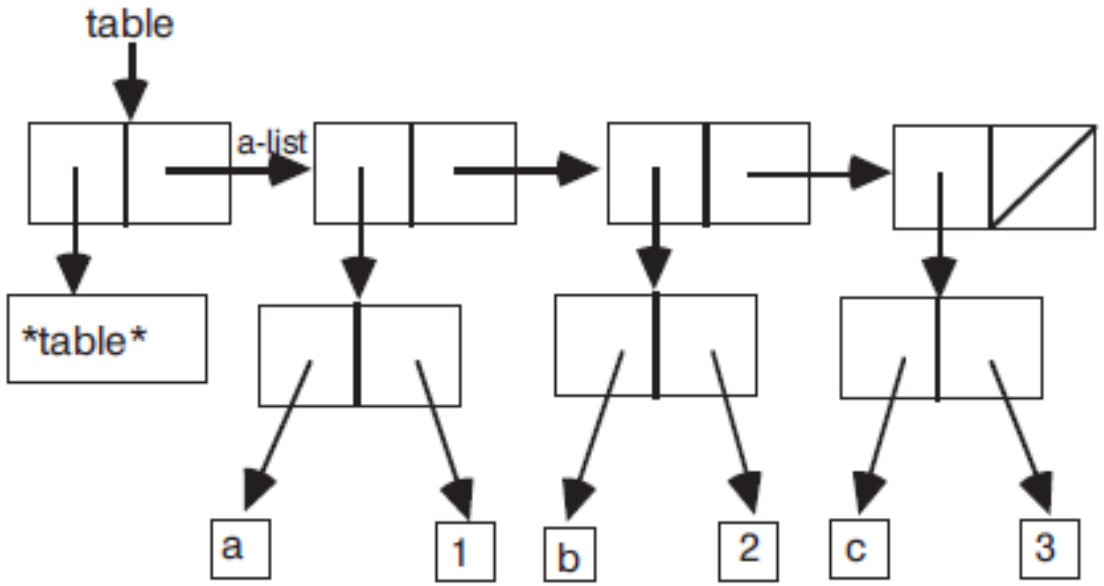
се представя графично по следния начин:



Подобрено представяне (едномерна таблица):
Асоциативният списък

((a . 1) (b . 2) (c . 3))

ще представяме графично по следния начин (чрез сентинел):



2. Реализация на асоциативен списък

Примитивни процедури в Scheme за търсене в асоциативен списък:

а) ***конструктор***

(make-table) - създава празна едномерна таблица

```
(define (make-table)
    (list '*table*))
```

2. Реализация на асоциативен списък

б) ***селектори***

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

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

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

2. Реализация на асоциативен списък

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

Забележка: Процедурата *assq* има *a-list* за свой параметър, т.е. (cdr table), а не *table*. Ако в нея **eq?** се замени с **eqv?**, получената процедура се означава с **assv**, а ако пък **eq?** се замени с **equal?**, получената процедура се означава с **assoc**.

Тези процедури са реализирани като примитивни в голяма част от реализациите на Scheme.

2. Реализация на асоциативен списък

(lookup key table) - намира асоциацията, свързана с ключ, идентичен с *key* в таблицата *table*. Ако *key* не е ключ в асоциативния списък на *table*, *lookup* връща *nil* (или '()).

```
(define (lookup key table)
    (let ((record (assq key (cdr table))))
        (if (null? record) '()
            (cdr record))))
```

2. Реализация на асоциативен списък

в) ***мутатори***

(insert! key value table) - включва двойката (*key* **.** *value*) в таблицата *table*.

```
(define (insert! key value table)
    (let ((record (assq key (cdr table))))
        (if (null? record)
            (begin
                (set-cdr! table (cons (cons key value) (cdr table)))
                table)
            (begin
                (set-cdr! record value)
                table))))
```

2. Реализация на асоциативен списък

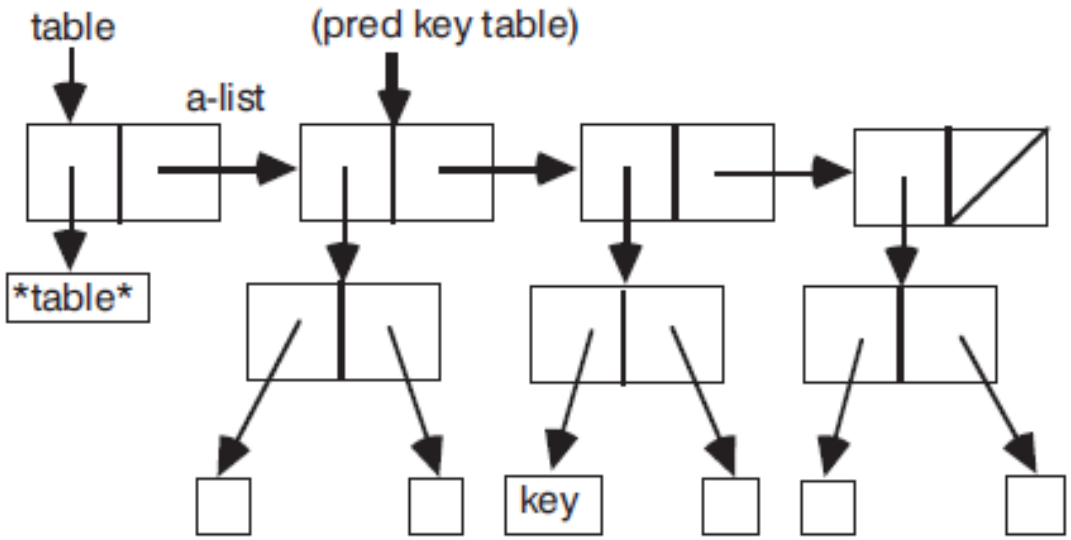
в) ***мутатори***

(remassq! key table) - изключва двойка с ключ *key* от таблицата *table*. **Предполагаме, че съществува двойка на *table*, която съдържа ключ, идентичен на *key*.** За реализирането на remassq! ще дефинираме помощната процедура *pred*:

(pred key table) - връща двойката, намираща се пред двойката, чийто *car* е двойка с ключ, идентичен на *key*.

2. Реализация на асоциативен списък

```
(define (pred key table)
    (let ((record (assq key (cdr table))))
        (if (eq? (cadr table) record) table
            (pred key (cdr table)))))
```



2. Реализация на асоциативен списък

Тогава мутаторът remassq! има вида:

```
(define (remassq! key table)
    (let ((record (pred key table)))
        (set-cdr! record (cddr record))
        table))
```

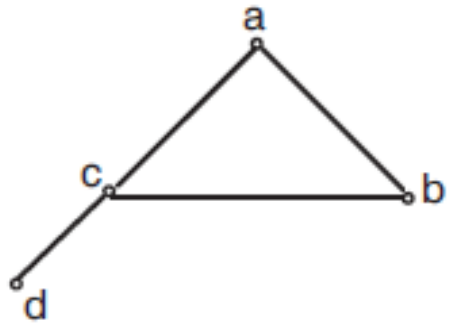
3. Приложение на едномерните таблици за реализиране на граф

За дефиниране на ориентиран граф ще използваме едномерна таблица, представяща асоциативен списък, ключовете на който са върховете на графа, а асоциациите - списъци, съдържащи върховете, с които върхът в съответния ключ е свързан.

Ще разглеждаме неориентирания граф като частен случай на ориентиран граф.

3. Приложение на едномерните таблици за реализиране на граф

Пример: Неориентираният граф



може да се представи чрез таблица, представяща асоциативния списък:

((a . (b c)) (b . (a c)) (c . (a b d)) (d . (c)))

или

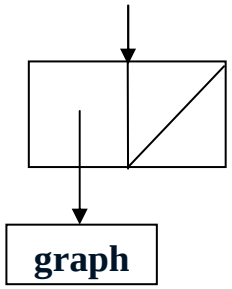
((a b c) (b a c) (c a b d) (d c))

3. Приложение на едномерните таблици за реализиране на граф

Ще определим следните конструктор, предикати и мутатори:

а) ***конструктор***

(make-graph) - създава празен граф




```
(define (make-graph)
        (list 'graph))
```

3. Приложение на едномерните таблици за реализиране на граф

б) ***предикати***

(top? a g) - проверява дали *a* е връх на графа *g*

```
(define (top? a g)
    (let ((L (assq a (cdr g))))
        (not (null? L))))
```

3. Приложение на едномерните таблици за реализиране на граф

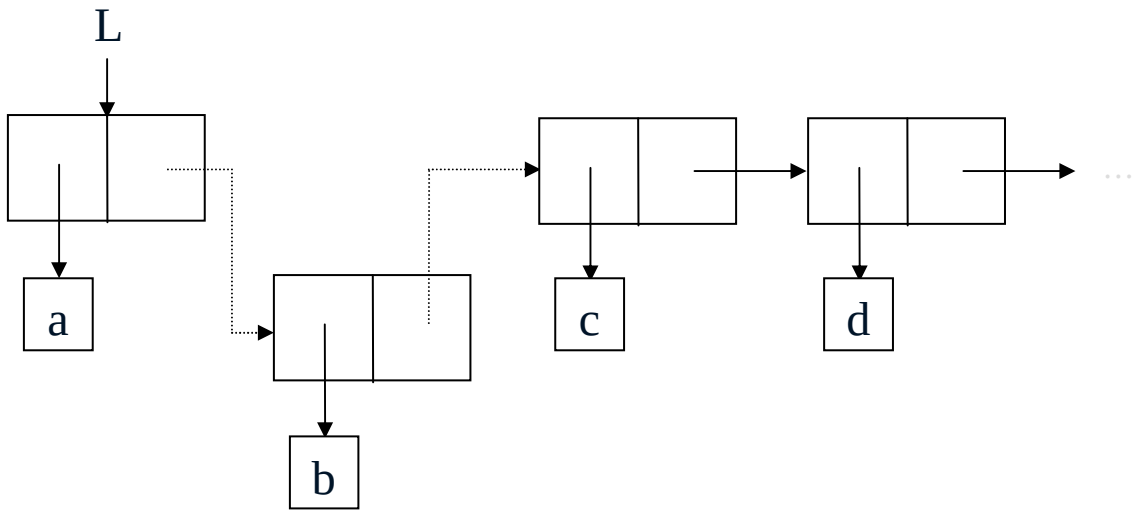
(rib? a b g) - проверява дали има ребро от върха *a* до върха *b* на графа *g*

```
(define (rib? a b g)
    (let ((L (assq a (cdr g))))
        (memq b (cdr L))))
```

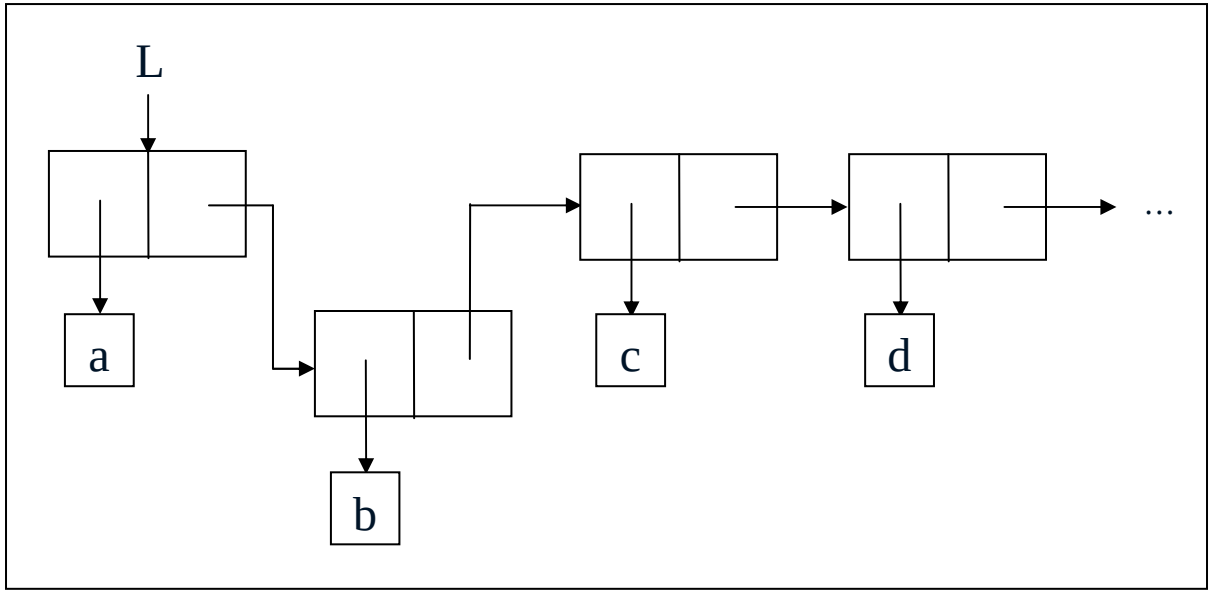
3. Приложение на едномерните таблици за реализиране на граф

в) ***мутатори***

(insert-rib! a b g) - включва ориентирано ребро от върха *a* до върха *b* на графа *g*



3. Приложение на едномерните таблици за реализиране на граф



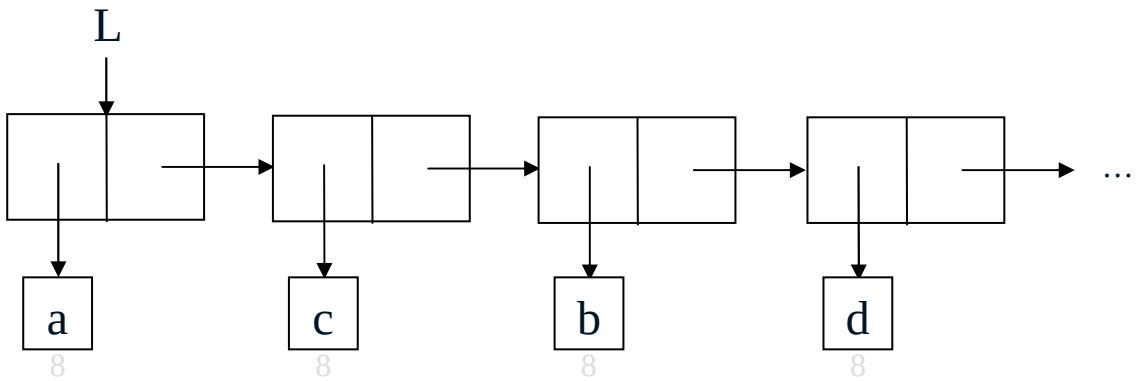

```
(define (insert-rib! a b g)
    (let ((L (assq a (cdr g))))
        (set-cdr! L (cons b (cdr L)))
        g))
```

3. Приложение на едномерните таблици за реализиране на граф

(del-rib! a b g) - изключва ориентирано ребро от върха *a* до върха *b* на графа *g*

```
(define (del-rib! a b g)
    (let ((L (assq a (cdr g))))
        (delete! b L)
        g))
```

където (delete! b L) изключва елемента *b* от (cdr L).



От представянето се вижда, че *b* не е първи елемент на *L*.

```
(define (delete! b L)
    (let ((f (pr b L)))
        (set-cdr! f (cddr f))
        L))
```

Тя използва процедурата (pr b L), която намира указател пред елемента *b* на списъка *L*. Известно е, че *b* не е първи елемент на списъка *L*.

```
(define (pr b L)
    (if (eq? (cadr L) b) L
        (pr b (cdr L))))
```

3. Приложение на едномерните таблици за реализиране на граф

(insert-top! a g) - включва върха *a* в графа *g*

```
(define (insert-top! a g)
    (insert! a '() g))
```

където ***insert!*** е мутаторът, дефиниран за едномерните таблици.

3. Приложение на едномерните таблици за реализиране на граф

(del-top! a g) - изключва върха a на графа g

Реализира се като се изтрият:

1) всички ребра от върхове на g към върха a на g;
2) върхът a и всички ребра, изходящи от a.

3. Приложение на едномерните таблици за реализиране на граф

```
(define (del-top! a g)
  ; изтрива всички ребра от върхове на g към върха b
  (define (del-top1! b f)
    (cond ((null? f) nil)
          ((rib? (caar f) b g) (del-rib! (caar f) b g)
                               (del-top1! b (cdr f)))
          (else (del-top1! b (cdr f)))))
  ; изтрива всички ребра от върхове на g към върха a
  (del-top1! a (cdr g))
  ; изтрива върха a и всички ребра изходящи от върха a
  (remassq! a g)
  g)
```

3. Приложение на едномерните таблици за реализиране на граф

г) ***селектори***

За селектори на граф ще използваме селекторите на асоциативен списък и едномерна таблица: *assq* и *lookup*.

3. Приложение на едномерните таблици за реализиране на граф

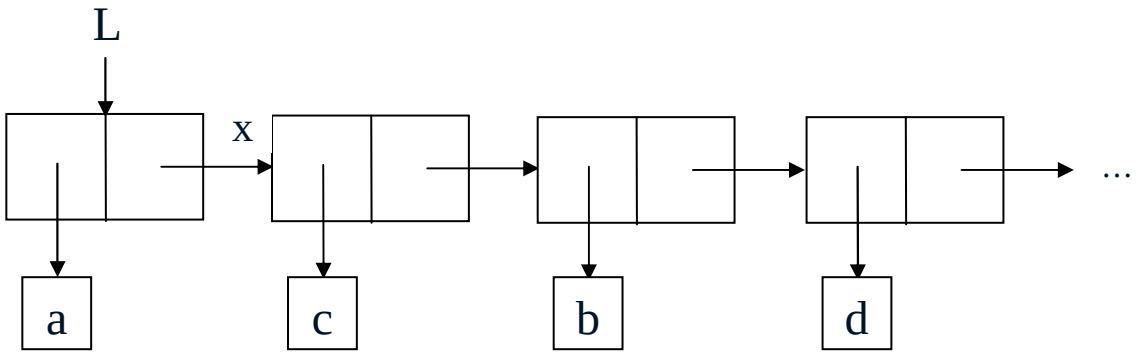
Задача.

Да се дефинира процедура, която проверява дали има път от връх *a* до връх *b* на графа *g*.

```
(define solution #f)

(define (way1? x b g)     ; x е списък от върхове на графа g
    (cond ((null? x) #f)
          ((way? (car x) b g) #t)
          (else (way1? (cdr x) b g))))
```

3. Приложение на едномерните таблици за реализиране на граф




```
(define (way? a b g)
    (let* ((L (assq a (cdr g)))
           (x (cdr L)))
      (cond ((null? x) #f)
            ((rib? a b g) (set! solution #t) solution)
            (else (set-cdr! L nil)
                  (set! solution (way1? x b g))
                  (set-cdr! L x)
                  solution))))
```