

Абстракция чрез структури от данни

Абстракция чрез данни и съставни типове данни в Racket – общи идеи и обосновки

Racket допуска работа не само с **примитивни типове данни** (например числа, константите ***#t*** и ***#f*** и др.), но така също и със **съставни типове данни**. Всяка стойност от такъв съставен тип се състои от различни елементи, които могат да бъдат извлечени (цитирани) с помощта на специални процедури, наречени ***селектори***. Обратно, ако са зададени отделните елементи, които изграждат дадена стойност от някакъв съставен тип, тази съставна стойност може да бъде построена от елементите ѝ с помощта на специални процедури – ***конструктори***.

Следователно, с всеки съставен тип данни по принцип са свързани специални (специфични) процедури за достъп, наречени конструктори и селектори.

Възможността за използване на съставни типове данни увеличава значително изразителната сила на езиците за програмиране - с тяхна помощ могат по естествен начин да се изразяват (означават) по-сложни обекти, които при необходимост могат да бъдат конструирани, а след това - цитирани и използвани като отделни концептуални единици.

Използването на съставни данни дава възможност за повишаване (подобряване) на модулността на съставяните програми.

Например, ако съставяме програмна система за работа с рационални числа, можем да отделим частта от системата, която работи с рационалните числа като такива (работи например с техните знак, числител и знаменател), от частта, която зависи от детайлите на конкретния избор на представяне на рационалните числа като двойки от цели числа. Тази методология, която позволява да бъдат отделени методите за използване на данните от методите за тяхното представяне, се нарича **абстракция чрез данни**.

Абстракцията чрез данни позволява съответните програми да се съставят, използват и модифицират значително по-лесно. Основната идея на абстракцията чрез данни е следната. Програмите се конструират така, че да работят с „абстрактни данни“ чиято структура евентуално може да не е (напълно) уточнена. След това представянето на данните се конкретизира с помощта на множество процедури, наречени конструктори и селектори, които реализират тези абстрактни данни по определения от автора конкретен начин.

Преимущества, до които води абстракцията чрез данни:

- както и абстракцията чрез процедури, абстракцията чрез данни позволява да се работи в термините на съответната предметна област или разглеждания модел на съответната област или задача (работи се в термините на областта, а не в термините на съответното представяне, следователно се работи на подходящо ниво на абстракция);

- при промяна на представянето на данните ще се променят малки части от програмната система - само тези процедури, които зависят от представянето на данните (конструкторите и селекторите).

Пример: програмна система за работа с рационални числа

Нека предположим, че разполагаме със следните процедури за достъп (един конструктор и два селектора):

(make-rat n d) —> рационалното число n/d ;

(numer x) —> числителя на рационалното число x ;

(denom x) —> знаменателя на рационалното число x .

Тогава ще дефинираме основните процедури за работа с рационални числа, като използваме следните равенства (които имат някои съществени недостатъци, но ще ги възприемем за удобство):

$$\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}, \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}, \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2}, \quad \frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{n_2 d_1}, \\
\frac{n_1}{d_1} &= \frac{n_2}{d_2} \text{ ТОГАВА И САМО ТОГАВА, КОГАТО } n_1 d_2 = n_2 d_1.
\end{aligned}$$

Дефиниции

```
(define (+rat x y)
  (make-rat
    (+ (* (numer x) (denom y))
       (* (denom x) (numer y)))
    (* (denom x) (denom y))))
```

```
(define (-rat x y)
  (make-rat
    (- (* (numer x) (denom y))
       (* (denom x) (numer y)))
    (* (denom x) (denom y))))
```



```
(define (*rat x y)
  (make-rat
    (* (numer x) (numer y))
    (* (denom x) (denom y))))
```

```
(define (/rat x y)
  (make-rat
    (* (numer x) (denom y))
    (* (denom x) (numer y))))
```

```
(define (=rat x y)
  (= (* (numer x) (denom y))
     (* (denom x) (numer y))))
```

Точкови двойки

Конкретизирането на представянето на рационалните числа (конкретизирането на структурата данни „рационално число“) ще извършим с помощта на вградената в езика Racket универсална структура, наречена **двойка** (**точкова двойка**, **cons-двойка**).

1) **Конструктор** на точковите двойки е примитивната процедура **cons**. Тя има два аргумента и връща съставен обект, който се състои от две части (два елемента), съвпадащи с оценките на аргументите на **cons**.

2) **Селектори** при точковите двойки са примитивните процедури **car** и **cdr**. Това са процедури с един аргумент, чиято оценка трябва да е точкова двойка. **car** служи за извличане на първия елемент на точковата двойка, а **cdr** - за извличане на втория елемент на точковата двойка.

Примери

```
(define x (cons 1 2))
```

```
> x
```

```
' (1.2)
```

```
> (car x)
```

```
1
```

```
> (cdr x)
```

```
2
```

```
(define y (cons 3 4))
```

```
(define z (cons x y))
```

```
> (car (car z))
```

```
1
```

```
> (car (cdr z))
```

```
3
```

Представяне на рационалните числа в примерната система за работа с рационални числа

Ще представяме рационалните числа като точкови двойки от числителите и знаменателите им, т.е. всяко рационално число ще бъде точкова двойка от неговите числител и знаменател. Тогава конструкторът **make-rat** и селекторите **numer** и **denom** ще имат следните дефиниции:


```
(define (make-rat n d) (cons n d))  
(define (numer x) (car x))  
(define (denom x) (cdr x))
```

Очевидно нашата реализация има някои недостатъци - например, в нея не е предвидено съкращаване (нормализиране) на съответните обикновени дроби. Можем да се справим с този недостатък, като дефинираме нов вариант на **make-rat**, например:

```
(define (make-rat n d)
  (let ([g (gcd n d)])
    (cons (/ n g) (/ d g))))
```

Сега вече е налице по-добър резултат.

Обща бележка. При програмирането на нашата система последователно преминахме през няколко нива на абстракция:

- 
- програми, използващи пакета за работа с рационални числа;
 - процедури, реализиращи действията с рационални числа;
 - конструиране на рационални числа и селекция на числителя/знаменателя на дадено рационално число;
 - конкретна реализация (избор на представяне) на рационалните числа чрез точкови двойки;
 - реализация на точковите двойки чрез конструктор (***cons***) и селектори (***car*** и ***cdr***).

Дефиниране на съставните типове данни

Съставните данни се задават чрез съвкупността от съответни конструктори и селектори заедно със специфичните условия, които те трябва да удовлетворяват, за да бъде представянето коректно.

Пример. Дефинираните от нас конструктори и селектори (в първия им вариант) удовлетворяват следните условия:

```
(numer (make-rat n d)) —> [n] ,  
(denom (make-rat n d)) —> [d] ,  
(make-rat (numer x) (denom x)) —> [x] .
```

В конкретния случай тези условия се изпълняват от нашите процедури, защото процедурите **car**, **cdr** и **cons** имат аналогични свойства:

$$\begin{aligned}(\text{car } (\text{cons } x \ y)) &\longrightarrow [x], \\(\text{cdr } (\text{cons } x \ y)) &\longrightarrow [y], \\(\text{cons } (\text{car } z) \ (\text{cdr } z)) &\longrightarrow [z].\end{aligned}$$

В Racket процедурите ***car***, ***cdr*** и ***cons*** са вградени поради съображения за ефективност. В действителност обаче всяка тройка от процедури, за които са верни първите две от изброените по-горе свойства, може да бъде използвана като основа за дефиниране (реализация) на точковите двойки в Racket.

Например, възможно е ***car***, ***cdr*** и ***cons*** да бъдат реализирани без използване на структури от данни, с използване само на процедури, както е направено в следната система от дефиниции:

```
(define (new-cons x y)
  (define (dispatch m)
    (cond [(= m 0) x]
          [(= m 1) y]
          [else (error "Argument not 0 or 1" m)]))
  dispatch)
```

```
(define (new-car z) (z 0))
```

```
(define (new-cdr z) (z 1))
```

Според първата дефиниция стойността, която връща (**new-cons x y**), е процедура - вътрешната процедура **dispatch**, която има един аргумент и връща **[x]** или **[y]** в зависимост от това, дали стойността на аргумента ѝ е 0 или 1. Ако стойността на аргумента е различна както от 0, така и от 1, специалната форма **error** предизвиква извеждане на "**Argument not 0 or 1**" и **[m]**, последвано от прекратяване на процеса на оценяване на обръщението към **dispatch**.

Според втората дефиниция (**new-car z**) предизвиква прилагане на **z** върху 0 и, следователно, ако **z** е процедурата, формирана от (**new-cons x y**), то **z**, приложена към 0, дава оценката на **x**. Следователно, (**new-car (new-cons x y)**) връща **[x]** и аналогично (**new-cdr (new-cons x y)**) връща **[y]**. Следователно, тази процедурна реализация на **car**, **cdr** и **cons** е коректна.

Обща бележка. Демонстрираните по-горе дефиниции показват, че с помощта на процедури от по-висок ред могат да се моделират и структури от данни, т.е. могат да се използват процедурни представяния на структурите от данни.

Следователно, **няма принципна разлика между данните и програмите в Racket.**

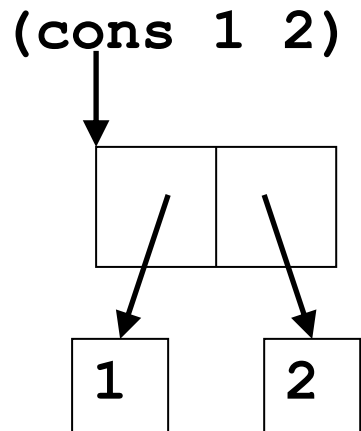
Списъци в езика Racket

Представяне на данните в езика Racket

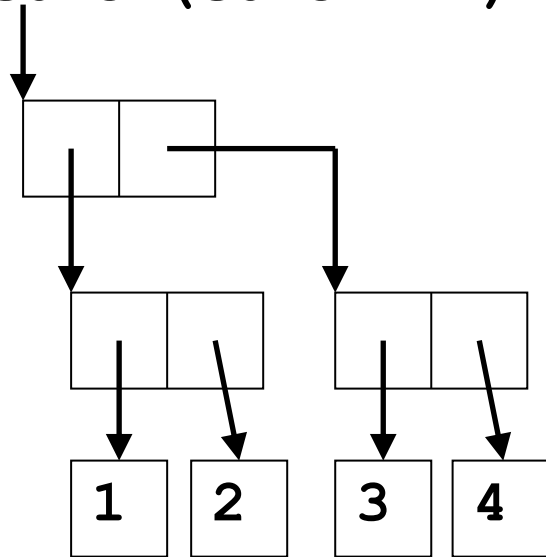
Ще разгледаме едно междинно логическо ниво, което се намира между нивото на текстовия запис на изразите и нивото на конкретното физическо представяне, зависещо от решенията на автора на конкретната реализация. Поддържането на това ниво се смята за задължително за авторите на реализации. За описанието му се използват диаграми, съдържащи клетки (кутии, boxes) и указатели.

Всеки обект се представя като указател към някаква клетка. Клетката, съответна на даден примитивен обект (каквито са например числата), съдържа представянето на този обект. Клетката, съответна на дадена точкова двойка, съдържа двойка указатели към представянията на ***car*** и ***cdr*** на тази точкова двойка.

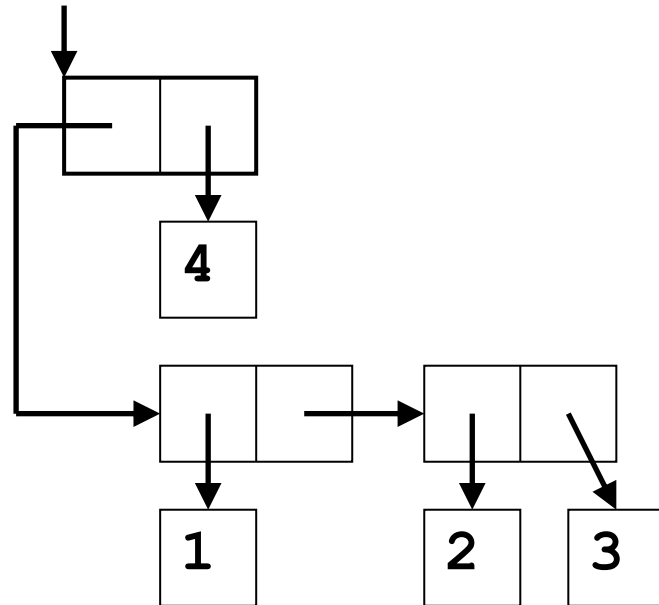
Примери



`(cons (cons 1 2) (cons 3 4))`



`(cons (cons 1 (cons 2 3)) 4)`



Важни следствия и бележки по отношение на данните в езика Racket

1. Елементите на точковите двойки могат да бъдат както числа (т.е. примитивни обекти), така и други точкови двойки. Следователно, точковите двойки дават възможност с тяхна помощ да бъдат представяни йерархични данни - данни, съставени от части, които също са съставени от различни части и т.н.

2. Разгледаните досега типове данни (числа и точкови двойки) са частни случаи на т. нар. **символни изрази (S-изрази)**. Това са обекти, които могат да бъдат произволни символи (а не само числа) или да се състоят от произволни символи.

S-изрази в Racket. Символни атоми

Дефиниция на понятието S-израз

S-изразите в езика Lisp (в частност, в Racket) се дефинират както следва:

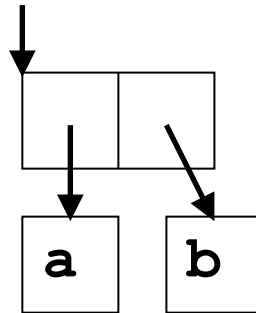
- 1) Атомите (числата, булевите константи **#t** и **#f**, символните низове и т. нар. символни атоми или символи) са S-изрази.
- 2) Ако **s1** и **s2** са S-изрази, то **(s1.s2)** също е S-израз.
- 3) Няма други S-изрази освен тези, описани в т. 1) и 2).

Допълнения и бележки:

- атомите (числата, символните низове и символните атоми) са примитивните типове данни в Lisp (Racket);
- символните атоми (символите) от синтактична гледна точка са идентификатори;
- списъците са частен случай на S-изрази.

Действие на специалната форма quote

Пример. Как се конструира точковата двойка, представена чрез следната диаграма:



Очевидно записът **(cons a b)** не е подходящ, тъй като при него се предполага оценяване на **a** и **b** и формиране на точкова двойка от получените оценки. В случая е необходима процедура (точно, специална форма), която не оценява аргумента си и го връща като оценка такъв, какъвто е, без да го променя.

Такава е примитивната процедура (специалната форма) ***quote***.

Общ вид на обръщението към ***quote***:

(quote <S-израз>) или ***' <S-израз>***

Семантика:

(quote <S-израз>) \longrightarrow ***<S-израз>***

Примери

' a \longrightarrow ***a***

' 25 \longrightarrow ***25***

' (a.b) \longrightarrow ***(a.b)***

' ' a \longrightarrow ***' a***

Примитивни предикати за проверка на типа на даден S-израз

В езика Racket са предвидени някои вградени (примитивни) процедури - предикати, които проверяват какъв е типът на оценката на аргумента им.

Списък на по-важните предикати за проверка на типа на даден S-израз

(този списък ще бъде допълнен по-късно при въвеждането на понятието „списък“)

$$(\text{pair? } \text{obj}) \longrightarrow \begin{cases} - \text{\textit{\#t}}, \text{ ако } [\text{obj}] \text{ е точкова двойка;} \\ - \text{\textit{\#f}}, \text{ в противния случай.} \end{cases}$$

`(list? obj)` \longrightarrow $\left\{ \begin{array}{l} - \text{\textit{\#t}}, \text{ ако } [\text{\textbf{obj}}] \text{ е списък;} \\ - \text{\textit{\#f}}, \text{ в противния случай.} \end{array} \right.$

`(number? obj)` \longrightarrow $\left\{ \begin{array}{l} - \text{\textit{\#t}}, \text{ ако } [\text{\textbf{obj}}] \text{ е число;} \\ - \text{\textit{\#f}}, \text{ в противния случай.} \end{array} \right.$

$(\text{string? } \text{obj}) \longrightarrow \begin{cases} - \text{\#t}, \text{ ако } [\text{obj}] \text{ е символен низ;} \\ - \text{\#f}, \text{ в противния случай.} \end{cases}$

$(\text{symbol? } \text{obj}) \longrightarrow \begin{cases} - \text{\#t}, \text{ ако } [\text{obj}] \text{ е символен атом;} \\ - \text{\#f}, \text{ в противния случай.} \end{cases}$

Примери

`(pair? ' (a.b)) —> #t`

`(pair? 'a) —> #f`

`(list? ' (a s d)) —> #t`

`(number? 58) —> #t`

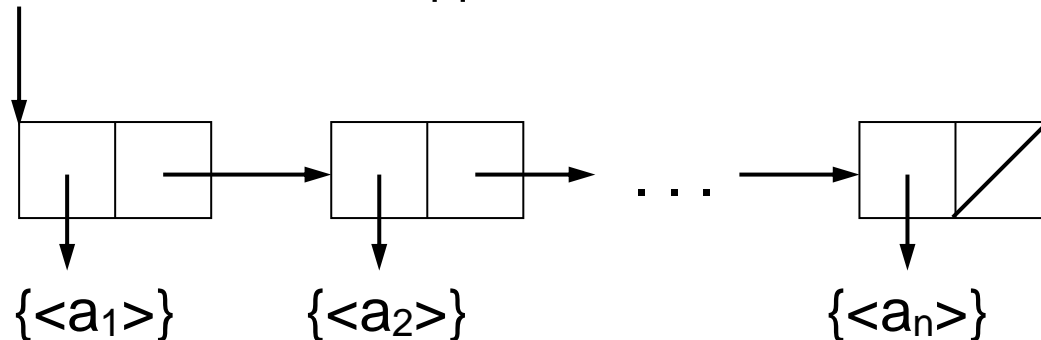
`(symbol? 'qwerty) —> #t`

`(string? "a string") —> #t`

`(string? 'no-string) —> #f`

Дефиниция на списъците като точкови двойки. Конструирание на списъци и цитиране на елементи на даден списък

Списъците са точкови двойки, които представят крайни редици от елементи по следния начин:



Забележка. С $\{<S\text{-израз}>\}$ тук означаваме представянето на $<S\text{-израз}>$.

Записът във вид на точкова двойка на S-израза (списъка), представен с помощта на горната диаграма, е следният:

$$(<a_1>. (<a_2>. (\dots . (<a_n>. ()) \dots)))$$

или също:

$$(\text{cons } '<a_1> (\text{cons } '<a_2> \\ (\text{cons } \dots (\text{cons } '<a_n> ' ()) \dots)))$$

Тук $()$ е означение на т. нар. **празен списък**. Той е еквивалентен на вградения (примитивния) атом **null**, който служи за означаване на края на всеки списък – в графичните диаграми обикновено **null** се означава като диагонална линия в съответната клетка. Горният списък се записва още по следния начин:

$$(<a_1> <a_2> \dots <a_n>)$$

Тук **<a_i>** се наричат **елементи на списъка**. Елементите на един списък могат да бъдат произволни S-изрази (в частност, символни атоми или други списъци).

Следователно, понятието „списък“ в езика Racket се дефинира по следния начин:

- (1) празният списък **()** е списък;
- (2) ако **lst** е списък, то точковата двойка **(a.lst)**, където **a** е произволен S-израз, също е списък.

Тази дефиниция в частност означава, че всеки непразен списък е точкова двойка. Празният списък няма елементи и **не е** точкова двойка.

За проверка на това, дали един S-израз е еквивалентен на празния списък, се използва примитивната процедура (вграденият предикат) ***null?***:

$$(\text{null? } \text{obj}) \longrightarrow \begin{cases} \text{\#t, ако } [\text{obj}] \text{ е } (); \\ \text{\#f, в противния случай.} \end{cases}$$

Конструирането на списъци се извършва най-често с помощта на примитивните процедури *list* и *cons*.

Процедурата *list* има произволен брой аргументи. Тя конструира и връща като резултат списък от оценките на аргументите си:

$$(\text{list } \langle a_1 \rangle \langle a_2 \rangle \dots \langle a_n \rangle) \longrightarrow$$
$$([\langle a_1 \rangle] [\langle a_2 \rangle] \dots [\langle a_n \rangle])$$

Горният запис е еквивалентен на

```
(cons <a1> (cons <a2>  
  (cons ... (cons <an> '()) ... ))) ,
```

което означава, че **cons** може да се използва за конструиране на списък, а също и за добавяне на елемент в началото (преди първия елемент) на списък:

```
(cons 'a ' (b c d)) —> (a b c d)
```

Извличане на елементи на списък

От записа на списъците във вид на точкови двойки следва, че

$$\begin{aligned}(\text{car } '(<a_1> <a_2> \dots <a_n>)) &\longrightarrow <a_1> \text{ и} \\(\text{cdr } '(<a_1> <a_2> \dots <a_n>)) &\longrightarrow (<a_2> \dots <a_n>)\end{aligned}$$

Следователно, примитивната процедура **car** може да се използва за извличане (цитиране) на първия елемент на даден списък, а процедурата **cdr** може да се използва за получаване на списъка без неговия първи елемент (за получаване на опашката на дадения списък).

Забележка. Обръщения от вида `(car '())` и `(cdr '())` са некоректни, тъй като празният списък не е точкова двойка.

Следователно поредните елементи на един списък могат да бъдат извлечени (цитирани) както следва:

- първият елемент на списъка: **(car <списък>);**

- вторият елемент:

(car (cdr <списък>)) ≡ (cadr <списък>);

- третият елемент:

(car (cdr (cdr <списък>))) ≡ (caddr <списък>);

- четвъртият елемент:

(car (cdr (cdr (cdr <списък>)))) ≡ (cadddr <списък>) и т.н.

Извличането на n-тия пореден елемент ($n > 0$) на даден списък l може да стане с помощта на следната процедура:

```
(define (nth n l)
  (if (= n 1)
      (car l)
      (nth (- n 1) (cdr l))))
```

Основни примитивни процедури за работа със списъци в езика Racket

Намиране на броя на елементите (дължината) на даден списък
- примитивна процедура `length`

`(length l)` —> число, равно на броя на елементите
на `l` (`l` трябва да е списък)

Забележка. Процедурата ***length*** е вградена. Ако това не е
така, тя може да бъде дефинирана например по следните начини:

- в рекурсивен стил

```
(define (length l)
  (if (null? l)
      0
      (+ 1 (length (cdr l))))))
```

- в итеративен стил

```
(define (length l)
  (define (length-iter arg count)
    (if (null? arg)
        count
        (length-iter (cdr arg)
                       (+ 1 count))))
  (length-iter l 0))
```

Обединяване на елементите на произволен брой списъци –
примитивна процедура `append`

`(append l1 l2 ... ln)` \longrightarrow списък, който съдържа
елементите на `[l1]`, следвани от елементите на `[l2]`, ... ,
елементите на `[ln]`

Примери

`(append ' (a b) ' (c d))` \longrightarrow `(a b c d)`
`(append ' ((a b) (c d)) ' (x y) ' (k l))` \longrightarrow
`((a b) (c d) x y k l)`
`(append ' (a b) ' ())` \longrightarrow `(a b)`

Забележка. Процедурата ***append*** е вградена, при това има произволен (променлив) брой аргументи. Вариантът ѝ с два аргумента може да бъде дефиниран в рекурсивен стил както следва:

```
(define (append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

Обръщане на реда на елементите на даден списък – примитивна процедура `reverse`

(reverse l) —> списък, съставен от елементите на **[l]**, но взети в обратен ред (**[l]** трябва да бъде списък)

Примери

(reverse ' (a b c d)) —> **(d c b a)**

(reverse ' ()) —> **()**

(reverse ' ((a b) (c d) e)) —> **(e (c d) (a b))**

Примерна дефиниция на процедура (в итеративен стил), аналогична по действие на примитивната процедура ***reverse***:

```
(define (reverse lst)
  (define (reverse_it lst result)
    (if (null? lst)
        result
        (reverse_it (cdr lst)
                     (cons (car lst) result))))
  (reverse_it lst ' ()))
```


Предикати за проверка на равенство и проверка за принадлежност към списък

Проверка за равенство

Обща бележка. Точният механизъм на действие на тези предикати е свързан с изясняване на някои допълнителни подробности от представянето на обектите в Racket. Затова сега действието им ще бъде въведено не съвсем точно и ще бъде уточнено по-нататък.

$(eq? \ s1 \ s2) \longrightarrow \left\{ \begin{array}{l} \text{\textbf{\#t}}, \text{ ако } [s1] \text{ и } [s2] \text{ са идентични (ако } [s1] \\ \text{и } [s2] \text{ са означения на един и същ} \\ \text{обект, т.е. на един и участък от} \\ \text{паметта);} \\ \text{\textbf{\#f}}, \text{ в противен случай.} \end{array} \right.$

Забележки

1) Обикновено предикатът **eq?** се използва за проверка на това, дали **[s1]** и **[s2]** са еднакви символни атоми (има и други случаи, в които **eq?** дава резултат **#t**, но те са много малко).

2) За сравняване на числа е добре да се използва аритметичният предикат **=** или предикатът **eqv?** (той обединява в определен смисъл действието на **eq?** и **=**).

`(equal? s1 s2)` \longrightarrow $\left\{ \begin{array}{l} \text{\textbf{\#t}}, \text{ ако } [s1] \text{ и } [s2] \text{ са еквивалентни} \\ \text{S-изрази (в частност, списъци),} \\ \text{т.е. или са еднакви символни} \\ \text{атоми, или са еднакви низове,} \\ \text{или са равни числа от един и} \\ \text{същ тип, или са точкови двойки с} \\ \text{еквивалентни } \textbf{car} \text{ и } \textbf{cdr} \text{ части;} \\ \text{\textbf{\#f}}, \text{ в противния случай.} \end{array} \right.$

Проверка за принадлежност към списък

$(\text{memq } \text{item } l) \longrightarrow \begin{cases} \text{\#f, ако } [\text{item}] \text{ не съвпада (в смисъл на } \text{eq?}) \text{ с никой от елементите на списъка } [l]; \\ \text{тази част от списъка } [l], \text{ която започва с първото срещане на елемент, равен (в смисъл на } \text{eq?}) \text{ на } [\text{item}]. \end{cases}$

Примери

`(memq 'a '(c a b a d)) —> (a b a d)`

`(memq 'a '(b c d)) —> #f`

`(memq '(a b) '((a b) c d)) —> #f`

Примерна дефиниция на функционален аналог на процедурата *memq*:

```
(define (memq item l)
  (cond [(null? l) #f]
        [(eq? (car l) item) l]
        [else (memq item (cdr l))]))
```

Примитивната процедура (*member item l*) има същото действие като *memq*, но при нея сравнението се извършва с помощта на предиката *equal?* (а не с *eq?*, както е при *memq*).

Примери

```
(member 'a ' (b a c a d)) —> (a c a d)
(member ' (a b) ' ((a b) (c d))) —> ((a b) (c d))
(member ' (c d) ' ((a b) (c d))) —> ((c d))
(member 'a ' ((a b) (c d))) —> #f
```

Примерна дефиниция на функционален аналог на процедурата *member*.

```
(define (member item l)
  (cond [(null? l) #f]
        [(equal? (car l) item) l]
        [else (member item (cdr l))]))
```