

## Локални дефиниции с помощта на специалните форми **let**, **let\*** и **letrec**

Друг типичен начин на употреба на специалната форма ***lambda*** е свързан с дефинирането на локални променливи.

Пример. Да предположим, че искаме да дефинираме процедура, с чиято помощ се пресмята функцията

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y).$$

Ако положим:

$$\mathbf{a = 1+xy},$$

$$\mathbf{b = 1-y},$$

то тази функция може да бъде записана като

$$f(x,y) = xa^2 + yb + ab$$

Решение – първи начин:

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a)) (* y b) (* a b)))
```

Решение – втори начин:

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a)) (* y b) (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

## Специална форма *let*

Решение – трети начин:

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a)) (* y b) (* a b))))
```

Общ вид на обръщението към *let*:

```
(let ((<пром1> <изр1>)
      (<пром2> <изр2>)
      . . .
      (<промn> <изрn>))
  <тяло>)
```

## Семантика:

Всяка променлива (всяко име) **<пром<sub>i</sub>>** се свързва с оценката на съответния израз **<изр<sub>i</sub>>**. След това изразите от **<тяло>** се оценяват последователно в локалната среда, получена чрез допълване на текущата среда (съществуващата до този момент среда) с новите свързвания. Оценката на последния израз от **<тяло>** се връща като оценка на обръщението към **let**.

**Забележка.** Обръщението към **let** от посочения общ вид е еквивалентно на

$$((\text{lambda } (<\text{пром}_1> <\text{пром}_2> \dots <\text{пром}_n>) <\text{тяло}>) \\ <\text{изр}_1> <\text{изр}_2> \dots <\text{изр}_n>)$$

## Сравнение между дефинирането на локални променливи чрез специалните форми ***define*** и ***let***

1. При използване на ***define*** областта на действие на съответните променливи съвпада с цялата среда, в която се изпълнява ***define***. Същевременно, при използване на ***let*** областта на действие на съответните променливи е само тялото на ***let***.

Например,

```
(define (f ... )  
  (define (g ... ) ( ... ))  
  (define (h ... ) ( ... g ... ))  
  ( ... ))
```

е допустима дефиниция, докато

```
(define (f ... )  
  (let ((g ... ) ... ) ( ... ))  
  (let ((h ... ) ... ) ( ... g ... ))  
  ( ... ))
```

не се допуска, освен ако няма дефинирано име **g** и в съответната обхващаща (съдържаща обръщението към **let**) среда.

2. Свързването в ***let*** се извършва **едновременно (псевдопаралелно)** за всички променливи (имена), докато при ***define*** (при последователни обръщения към ***define***) то става последователно. Тази особеност има значение, ако едни и същи имена (променливи) се срещат на няколко места в свързванията на ***let***.

Например, оценката на

```
(define x 2)
(let ((x 3) (y (+ x 2))) (* x y))
```

е 12, а не 15.

## Специална форма *let\**

За реализация на вложени *let* изрази може да се използва специалната форма *let\**.

Общ вид на обръщението към *let\**:

```
(let* ( (<пром1> <изр1>)  
        (<пром2> <изр2>)  
        . . .  
        (<промn> <изрn> )  
      <тяло> )
```



**Семантика.** Оценява се  $\langle \text{изр}_1 \rangle$  и  $\langle \text{пром}_1 \rangle$  се свързва с  $[\langle \text{изр}_1 \rangle]$ . Оценява се  $\langle \text{изр}_2 \rangle$  и  $\langle \text{пром}_2 \rangle$  се свързва с  $[\langle \text{изр}_2 \rangle]$ . При това, ако в  $\langle \text{изр}_2 \rangle$  се среща  $\langle \text{пром}_1 \rangle$ , при оценяването на  $\langle \text{изр}_2 \rangle$  се използва току-що свързаната с  $[\langle \text{изр}_1 \rangle]$  оценка на  $\langle \text{пром}_1 \rangle$ . По същия начин се продължава със свързване на следващите локални променливи (имена), докато се достигне до последната (последното). Тогава се оценява  $\langle \text{изр}_n \rangle$  и  $\langle \text{пром}_n \rangle$  се свързва с  $[\langle \text{изр}_n \rangle]$ . Ако в  $\langle \text{изр}_n \rangle$  се срещат някои от имената  $\langle \text{пром}_i \rangle$ ,  $i < n$ , при оценяването им се използват току-що свързаните с тях оценки. При така получените свързвания се оценява  $\langle \text{тяло} \rangle$ .

Следователно, изразът

```
(let* ((<пром1> <изр1>)
      (<пром2> <изр2>)
      . . .
      (<промn> <изрn>))
  <тяло>)
```

е еквивалентен на

```
(let ((<пром1> <изр1>))
  (let ((<пром2> <изр2>))
    . . .
    (let ((<промn> <изрn>))
      <тяло> . . . ) ) )
```

Разликата между ***let*** и ***let\**** е в начина (реда) на свързване на имената и стойностите на локалните променливи. Докато при ***let*** свързването е едновременно, при ***let\**** то е последователно.

В предишния пример: оценката на

```
(let* ((x 3) (y (+ x 2))) (* x y))
```

вече ще бъде 15. Нещо повече, ако в предишния пример липсваше първоначалната дефиниция (***define x 2***), то при изпълнението на следващото обръщение към ***let*** щеше да се получи грешка, докато обръщението към ***let\**** и в този случай щеше да бъде коректно.

## Специална форма *letrec*

В *let* израза

```
(let ( (<пром> <изр> ) ) <тяло>)
```

всички променливи, които се срещат в израза **<изр>** и не са свързани в самия израз **<изр>**, трябва да бъдат свързани извън *let* израза, т.е. в обхващащата (съдържащата) го среда. При оценяването на **<изр>** интерпретаторът търси извън *let* израза свързванията за всички свободни променливи, които се срещат в **<изр>**.

Например, изразът

```
(let ((fact (lambda (n)
               (if (= n 0) 1 (* n (fact (- n 1)))))))
  (fact 4))
```

е **некоректен**, тъй като подчертаното включване на **fact** в него не е свързано извън **let** израза.

Ако искаме да използваме рекурсивна дефиниция в **<изр>** частта на обръщение към специална форма, подобна на **let**, трябва да сме в състояние да преодолеем проблема с несвързаните променливи (имена) от горния пример. Това може да стане с помощта на специалната форма **letrec**. При обръщения към тази специална форма може да се извършват локални свързвания, при които рекурсията е допустима.

Синтаксис на **letrec**:

```
(letrec ( (<пром1> <изр1>)  
         (<пром2> <изр2>)  
         . . .  
         (<промn> <изрn> ) )  
  <тяло> )
```

Тук обаче всяка от променливите **<пром<sub>1</sub>>**, **<пром<sub>2</sub>>**, ... , **<пром<sub>n</sub>>** може да се среща във всеки от изразите **<изр<sub>1</sub>>**, **<изр<sub>2</sub>>**, ... , **<изр<sub>n</sub>>**. При това тези променливи (имена) се разглеждат като локално дефинираните променливи (имена) **<пром<sub>1</sub>>**, **<пром<sub>2</sub>>**, ... , **<пром<sub>n</sub>>** и следователно е възможно при дефинирането им да се използва рекурсия. С други думи, областта на действие на променливите **<пром<sub>1</sub>>**, **<пром<sub>2</sub>>**, ... , **<пром<sub>n</sub>>** в случая на ***letrec*** съвпада с **<изр<sub>1</sub>>**, **<изр<sub>2</sub>>**, ... , **<изр<sub>n</sub>>** и **<тяло>**.

Пример 1. Дефиниция на процедура за пресмятане на  $n!$  с използване на итеративна помощна (локална) процедура

```
(define (fact n)
  (letrec ((fact-iter
            (lambda (arg res)
              (if (= arg 0)
                  res
                  (fact-iter
                    (- arg 1) (* arg res))))))
    (fact-iter n 1)))
```



Пример 2. Описание на косвена рекурсия с помощта на ***letrec***

```
(define (even-odd? n)
  (letrec ((even?
            (lambda (x)
              (or (= x 0) (odd? (- x 1)))))
    (odd?
      (lambda (x)
        (and (not (= x 0))
              (even? (- x 1))))))
    (odd? n)))
```

## Процедурите като оценки на обръщения към процедури

Пример. Нека разгледаме изречението: "Производната на функцията  $x^3$  е функцията  $3x^2$ ". То означава, че производната на функцията, чиято стойност в  $x$  е  $x^3$ , е друга функция - тази, чиято стойност в  $x$  е  $3x^2$ . Следователно, понятието "производна" може да бъде разглеждано като определен оператор (в математически смисъл), който за дадена функция  $f$  връща друга функция  $Df$ . В този смисъл, за да опишем понятието "производна", можем да кажем, че ако  $f$  е някаква функция, то производната  $Df$  на  $f$  е функцията, чиято стойност за всяко число  $x$  се получава като

$$Df(x) = \lim_{dx \rightarrow 0} \frac{f(x + dx) - f(x)}{dx}.$$

Ако искаме (с известно приближение, като игнорираме означението за граница) да запишем горната формула във вид на дефиниция на процедура на езика Scheme, това може да стане по следния начин:

- дясната страна на формулата може да се запише като

```
(lambda (x)
  (/ (- (f (+ x dx)) (f x)) dx)) ,
```

където **dx** е име на променлива, която означава (чиято стойност по идея е) някакво малко положително число;

- цялата формула може да се запише като

```
(define (deriv f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx)))
```

Коментар. Така **deriv** е процедура, която има като аргумент процедура (има и още един аргумент - числото **dx**) и която връща като резултат нова процедура - тази, която е дефинирана чрез използвания ламбда израз. Последната процедура, приложена към някакво число  $x$ , връща стойността на производната  $f'(x)$ .

Процедурата **deriv** може да бъде дефинирана също и с използване на локална именувана процедура по следния начин:

```
(define (deriv f dx)
  (define (right_hand_side x)
    (/ (- (f (+ x dx)) (f x)) dx))
  right_hand_side)
```

Пример за използване на процедурата **deriv**

```
> ((deriv cube 0.001) 5)
75.0150010000255
> ((deriv cube 0.00001) 5)
75.0001499966402
```

В този пример операторът на използваната комбинация също е комбинация, тъй като процедурата, която се прилага към аргумента **5**, е стойността на процедурата **deriv** при аргумент **cube**.

# Абстракция чрез структури от данни

## Абстракция чрез данни и съставни типове данни в Scheme – общи идеи и обосновки

Scheme 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):

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} ,$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} ,$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2} , \quad \frac{n_1 / d_1}{n_2 / d_2} = \frac{n_1 d_2}{n_2 d_1} ,$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \text{ ТОГАВА И САМО ТОГАВА, КОГАТО } n_1 d_2 = n_2 d_1 .$$

## 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 Scheme 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
```

```
> x
```

```
(1.2)
```

```
> (car x)
```

```
1
```

```
> (cdr x)
```

```
2
```

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

```
y
```

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

```
z
```

```
> (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))
```

Ако искаме да дефинираме и процедура за извеждане на рационални числа, тя ще изглежда примерно така:

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/" )
  (display (denom x)))
```

## Примери

```
> (define one-third (make-rat 1 3))
```

```
ONE-THIRD
```

```
> (define one-half (make-rat 1 2))
```

```
ONE-HALF
```

```
> (print-rat (+rat one-half one-third))
```

```
5/6
```

```
> (print-rat (+rat one-third one-third))
```

```
6/9
```


Както показва последният пример, 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:

```
> (print-rat (+rat one-third one-third))
2/3
```

**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 Scheme 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 Scheme.

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 **y** 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 Scheme.**