Ламбда изрази

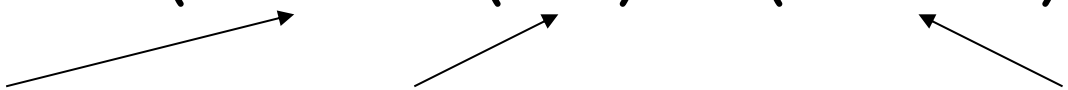
Идея. Често е досадно и ненужно да се определят (задават) имена на някои елементарни процедури, например

```
(define (next x) (+ x 4)).
```

В езика Racket съществуват средства, които позволяват да се дефинират процедури, без те да бъдат именувани, т.е. да се дефинират ***анонимни процедури***.

Процедурата от горния пример може да се запише още и така:

```
(lambda  ( x )   (+  x  4))
```



тази функция с аргумент x, която прибавя 4 към аргумента си

По-сложен пример. Дефиницията на процедура, която може да бъде използвана за приближено пресмятане на π/8 по формулата:

$$\frac{\pi}{8} = \frac{1}{1.3} + \frac{1}{5.7} + \frac{1}{9.11} + \ldots + \frac{1}{a(a+2)} + \frac{1}{(a+4)(a+6)} + \ldots ,$$

където *a* е цяло положително число от вида *4k+1*, може да се напише по следния начин с използване на ламбда дефиниции:

```
(define (sum-pi a b)
   (sum (lambda (x) (/ 1 (* x (+ x 2))))
        a
        (lambda (x) (+ x 4))
        b))
```

Общ вид на обръщението към специалната форма ***lambda***:

(lambda (<формални параметри>) <тяло>)

Семантика. В резултат на оценяването на обръщението към ***lambda*** се получава процедура, която обаче не се свързва с никакво име в средата. Тази процедура става оценка на обръщението към ***lambda***.

Връзка между дефинирането на процедура с помощта на ***define*** и ламбда дефиниция на процедура. Конструкциите

(define (<име> <формални параметри>) <тяло>) и
(define <име> (lambda (<формални параметри>)
<тяло>))

са еквивалентни.

Общ вид на обръщението към ***define***:

`(define <име> <израз>)`

При това ако **<израз>** е ламбда израз (ламбда дефиниция), ***define*** определя процедура с име **<име>**; иначе ***define*** определя променлива с име **<име>**.

Оценяване на комбинации, чиито оператори са ламбда изрази. В общия случай тези комбинации са от вида:

((lambda (x_1 x_2 ... x_n) <тяло>) a_1 a_2 ... a_n)

В процеса на оценяване на такава комбинация всички срещания (включвания) на x_i в **<тяло>** се заместват с [a_i] (оценката на a_i – вж. забележката по-долу), след което се оценява така полученият израз.

Пример

```
> ((lambda (x y z) (+ x y (* z z))) 1 2 3)
12
```

Забележка. С [<израз>] ще означаваме оценката на <израз>.

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:
a = 1+xy,
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$_i$>** се свързва с оценката на съответния израз **<изр$_i$>**. 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

((lambda (<пром$_1$> <пром$_2$> ... <пром$_n$>) <тяло>)
<изр$_1$> <изр$_2$> ... <изр$_n$>)

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

1. При използване на ***define*** областта на действие на съответните локални променливи съвпада с цялата среда, определена от блока на ***define***. Същевременно, при използване на ***let*** областта на действие на съответните локални 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 **<изр$_1$>** и **<пром$_1$>** се свързва с **[<изр$_1$>]**. Оценява се **<изр$_2$>** и **<пром$_2$>** се свързва с **[<изр$_2$>]**. При това, ако в **<изр$_2$>** се среща **<пром$_1$>**, при оценяването на **<изр$_2$>** се използва току-що свързаната с **[<изр$_1$>]** оценка на **<пром$_1$>**. По същия начин се продължава със свързване на следващите локални променливи (имена), докато се достигне до последната (последното). Тогава се оценява **<изр$_n$>** и **<пром$_n$>** се свързва с **[<изр$_n$>]**. Ако в **<изр$_n$>** се срещат някои от имената **<пром$_i$>**, **i<n**, при оценяването им се използват току-що свързаните с тях оценки. При така получените свързвания се оценява **<тяло>**.

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.

Обобщение. При използване на ***let**** областта на действие на локалната променлива **<пром$_i$>** е съвкупността от изрази **<изр$_{i+1}$>, … , <изр$_n$>** и тялото на ***let****.

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$_1$>**, **<пром$_2$>**, ... , **<пром$_n$>** може да се среща във всеки от изразите **<изр$_1$>**, **<изр$_2$>**, ... , **<изр$_n$>**.

При това тези променливи (имена) се разглеждат като локално дефинираните променливи (имена) **<пром$_1$>**, **<пром$_2$>**, ... , **<пром$_n$>** и следователно е възможно при дефинирането им да се използва рекурсия. С други думи, областта на действие на всяка от променливите **<пром$_1$>**, **<пром$_2$>**, ... , **<пром$_n$>** в случая на ***letrec*** съвпада с **<изр$_1$>**, **<изр$_2$>**, ... , **<изр$_n$>** 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

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

Ако искаме (с известно приближение, като игнорираме означението за граница) да запишем горната формула във вид на дефиниция на процедура на езика Racket, 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**.

По-сложни примери

Пример 1. Многократно прилагане на функция. Ако *f* е числова функция и *n* е естествено число, *n*-кратното прилагане на *f* се дефинира като функция, чиято стойност в дадена точка *x* е равна на $f(f(\ \ldots\ (f(x))\ \ldots\ ))$.

n

Да се дефинира процедура от по-висок ред с параметри функция *f* и естествено число *n*, която връща като резултат *n*-кратното прилагане на *f*.

```
(define (repeated f n)
   (lambda (x)
      (if (= n 1)
      (f x)
      (f ((repeated f (- n 1)) x)))))

> ((repeated (lambda (x) (+ x 1)) 5) 1)
6
```

Пример 2. Дефиниране на функция от по-висок ред, която връща като резултат функция, представляваща приближение на *n*-тата производна на функцията *f* при дадено нарастване на аргумента *dx*.

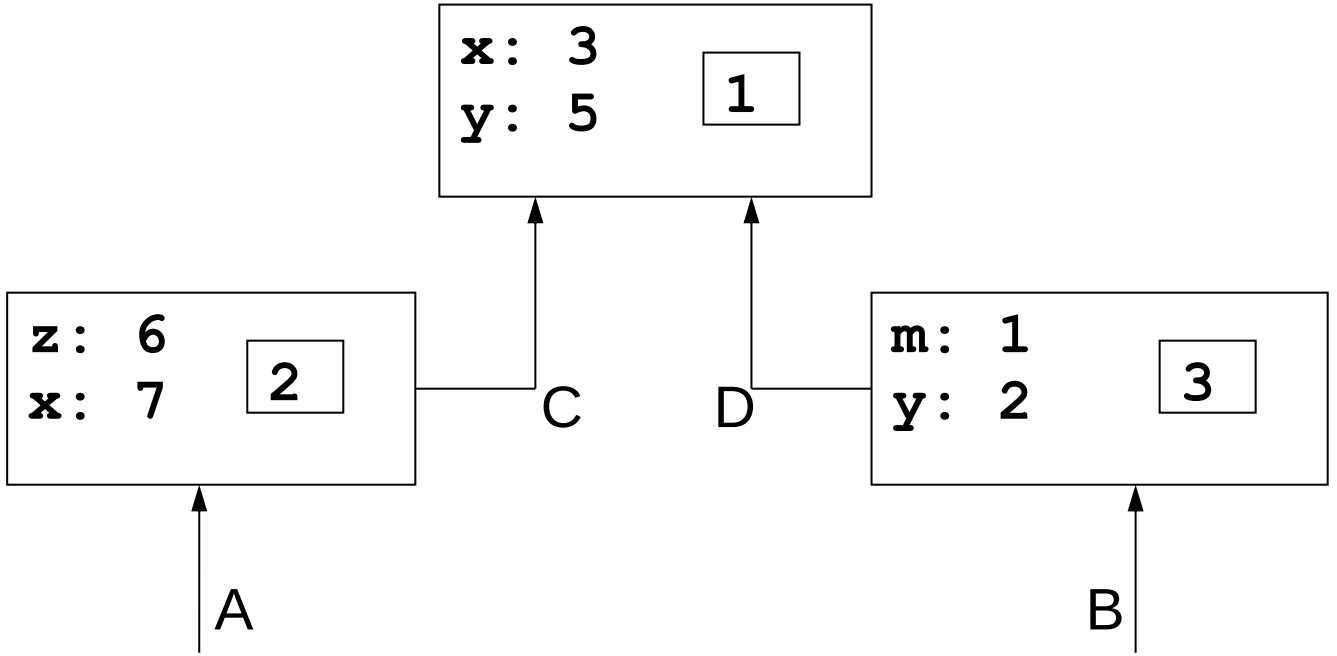
```
(define (derive f dx)
   (lambda (x) (/ (- (f (+ x dx)) (f x))dx)))

(define (Df f n dx)
   (if (= n 1)
       (derive f dx)
       (Df (derive f dx) (- n 1) dx)))
```

Модел на средите за оценяване на обръщения към съставни процедури

Всяка променлива в програма на езика Racket би трябвало да се разглежда като означение на определено „място“, в което се съхранява свързаната с нея стойност. В така наречения ***модел на средите*** тези „места“ са елементи на специални структури (по-точно, редове на специални таблици), наречени среди.

Всяка среда може да бъде разглеждана като поредица от таблици, при това всяка таблица съдържа някакъв (неотрицателен) брой свързвания (bindings), които асоциират (свързват) променливите, валидни в средата, със съответните им стойности. Всяка такава таблица съдържа не повече от едно свързване за дадена променлива. Всяка таблица (освен тази, която съответства на глобалната среда) съдържа също и указател към съдържащата (обхващащата) я по-обща (родителска) среда. Стойността на дадена променлива в (по отношение на) дадена среда съвпада с тази стойност, която се намира в първото срещнато свързване на променливата (т.е. съвпада със стойността, свързана с тази променлива в първата таблица от средата, която съдържа някакво свързване за тази променлива). Ако никоя от таблиците от дадената среда не съдържа свързване на (за) търсената променлива, то тази променлива се нарича ***несвързана*** (***unbound***) или ***неопределена*** във въпросната среда.

Пример. Нека е дадена (валидна) следната система от среди, съставена от три таблици, които са номерирани съответно с 1, 2 и 3:



Тук **A**, **B**, **C** и **D** са указатели към някакви среди, при това **C** и **D** сочат към една и съща среда. При очертаната ситуация (система от среди и свързвания на променливите в тях) използваните по-горе променливи имат следните стойности в посочените среди:

x в средата **D** - 3,
x в средата **B** - 3,
x в средата **A** - 7,
y в средата **A** - 5,
m в средата **A** - несвързана (неопределена) и т.н.

Принципи на оценяването на комбинации в рамките на модела на средите

И при модела на средите общото правило за получаване на оценката на комбинация остава непроменено, т.е. изглежда както следва:

- оценяват се подизразите на комбинацията;
- оценката на първия подизраз се прилага върху оценките на останалите подизрази.

По същество разликите между двата модела са свързани с ***начина, по който се определя понятието „прилагане на съставна (дефинирана) процедура към съответните аргументи“***.

В модела на средите всяка процедура се разглежда като двойка от съответен код и указател към подходяща среда. Процедури се създават само по един начин - с използване на (чрез оценяване на) ***lambda*** изрази. По този начин се създава процедура, чийто код се получава от текста (тялото) на съответната ***lambda*** дефиниция и чиято среда съвпада със средата, в която е била оценена тази ***lambda*** дефиниция с цел получаване на разглежданата процедура.

Пример. Нека разгледаме следната дефиниция на процедура:

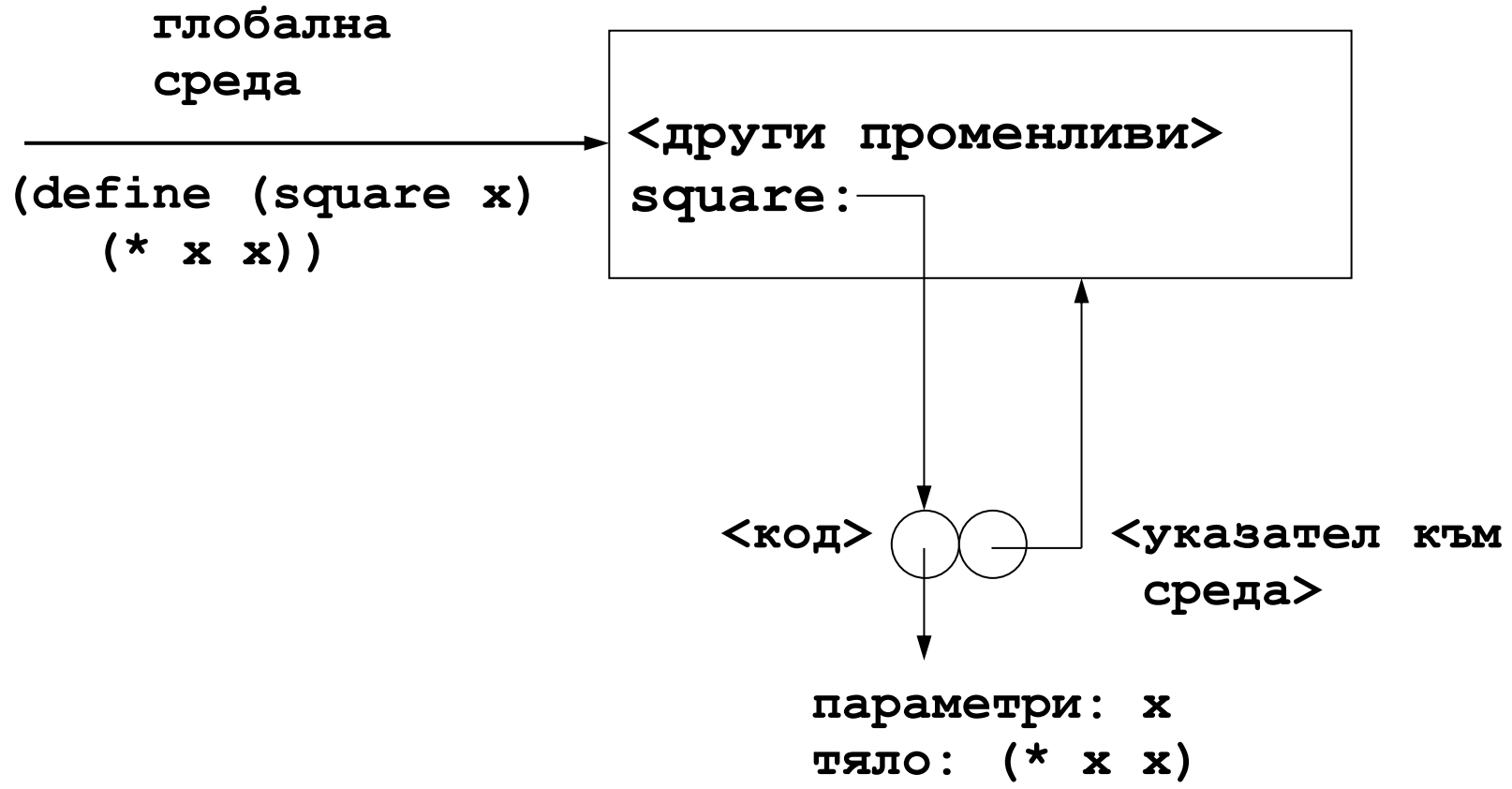
```
(define (square x) (* x x))
```

и да предположим, че тя се оценява в глобалната среда. Тази дефиниция е еквивалентна на следната дефиниция, която използва подходящ ***lambda*** израз:

```
(define square (lambda (x) (* x x)))
```

Последната дефиниция води до оценяване на израза **(lambda (x) (* x x))** и до съответно свързване на променливата (символа) **square** с получената стойност в глобалната среда.

Графична илюстрация на резултата от оценяването на горната дефиниция в глобалната среда:

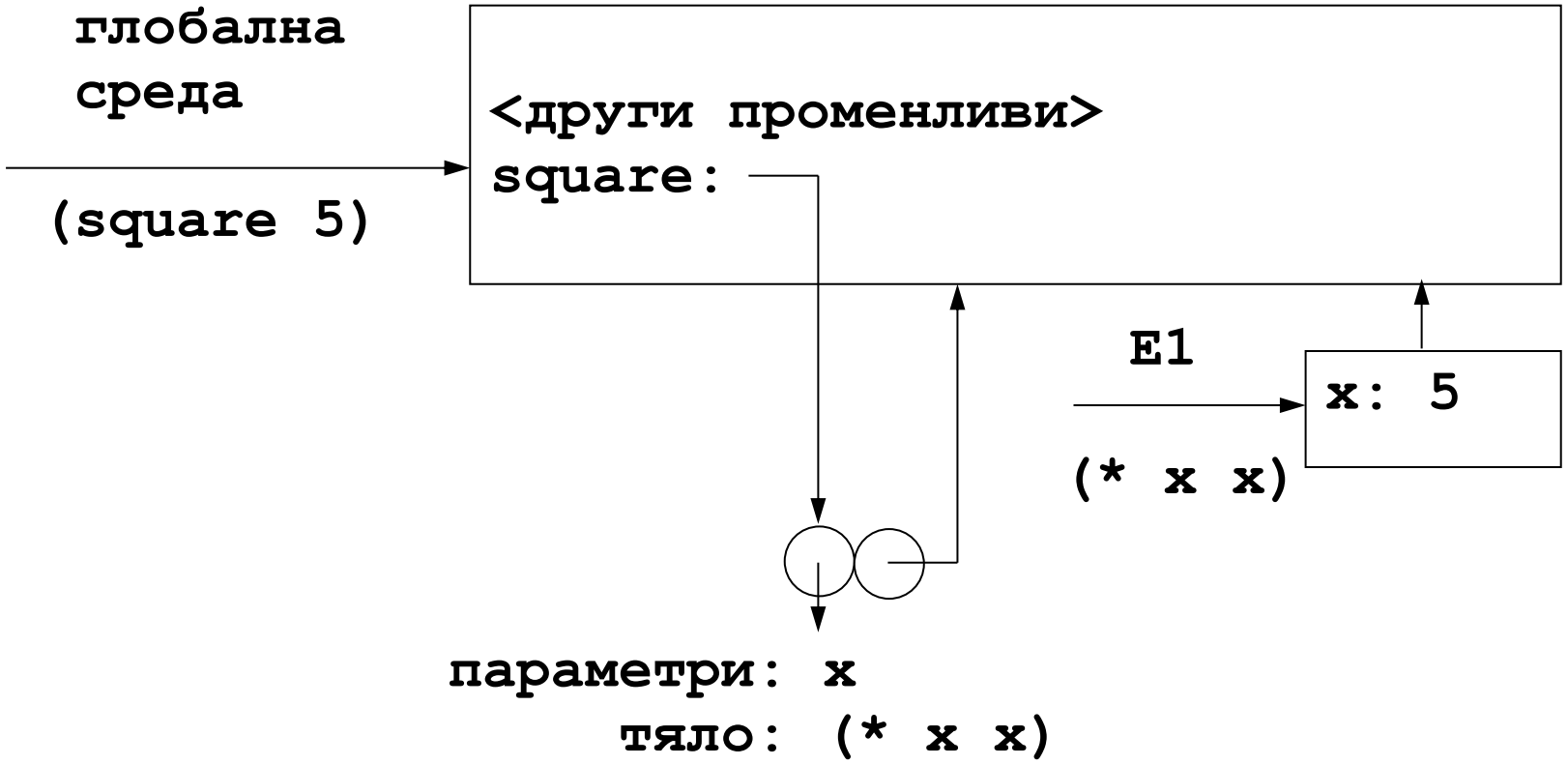


След като вече знаем как се дефинират процедури, можем да покажем и как се прилагат дефинирани процедури.

Пример. Ще покажем как се оценява обръщението **(square 5)** в глобалната среда.

Правило. В резултат на прилагането на процедурата **square** се създава нова среда, означена по-долу като **E1**, която започва с таблица, в която формалният параметър **x** на **square** е свързан със стойността на фактическия параметър (аргумента) 5. От тази таблица излиза указател, който сочи към глобалната среда, т.е. средата, която е родителска (обхващаща) за тази среда, съвпада с глобалната среда. Това е така, защото глобалната среда беше тази среда, в която беше оценена дефиницията на **square** (т.е. указателят към средата в представянето на **square** сочи към глобалната среда). В **E1** се оценява тялото на процедурата **(* x x)** и се връща полученият резултат 25.

Графична илюстрация:



Обобщение. Моделът на средите за оценяване на обръщения към съставни процедури може да бъде обобщен (формулиран) както следва:

1) Дадена процедура се прилага към определено множество от аргументи чрез конструиране на специална таблица, която съдържа свързвания на формалните параметри на процедурата със съответните фактически параметри (аргументите) на обръщението, последвано от оценяване на тялото на (дефиницията на) процедурата в контекста на новопостроената среда. При това новопостроената таблица има за родителска (обхващаща) среда същата среда, която се сочи от указателя към средата в представянето на (дефиницията на) прилаганата процедура.

2) Процедура се създава чрез оценяване на определен ***lambda*** израз в съответната среда. В резултат се създава специален обект (процедура), който може да се разглежда като двойка, съдържаща текста на ***lambda*** израза и указател към средата, в която е създадена процедурата.

Забележки:

1) При дефинирането на дадена променлива с помощта на ***define*** се създава ново свързване в текущата среда (на променливата със съответната ѝ стойност);

2) Оценяването на обръщението **(let ([<пром$_1$> <изр$_1$>] ...) <тяло>)** се извършва по следния начин. В текущата среда се оценяват изразите **<изр$_i$>**. Създава се нова среда, която е разширение на текущата (за която текущата среда е родителска), и в нея **<пром$_i$>** се свързват с **[<изр$_i$>]**. В новата среда се оценява **<тяло>**.