Работа със символни низове в езика Scheme

Вградени процедури за работа със символни низове

Символен низ в Scheme е всяка поредица от знакове (keyboard characters), заградени в двойни кавички. Оценката на даден символен низ съвпада с този низ.

Списък на някои често употребявани вградени функции
за работа с низове

`(string? <израз>)`

Проверява дали **[<израз>]** е символен низ (връща оценка ***#t***, ако **[<израз>]** е символен низ, или оценка ***#f*** – в противния случай).

`(string-length <низ>)`

Намира броя на знаковете в низа **[<низ>]** (дължината на низа **[<низ>]**).

(string-append <низ$_1$> <низ$_2$> ... <низ$_n$>)

Конкатенира низовете **[<низ$_1$>]**, **[<низ$_2$>]**, … , **[<низ$_n$>]** и връща получения резултат.

(substring <низ> <начало> <край>)

Предполага се, че **[<низ>]** е низ, а **[<начало>]** и **[<край>]** са цели числа, които удовлетворяват неравенствата **0≤[<начало>]≤[<край>]≤L**, където **L** е дължината на **[<низ>]**. Като резултат се връща низ, който е поднизът на **[<низ>]**, съставен от знаковете на **[<низ>]** с индекси от **[<начало>]** до **[<край>]-1** включително (смята се, че индексите на знаковете на низа се менят от **0** до **L-1**). Следователно, дължината на низа - резултат е равна на **[<край>]-[<начало>]**.

(symbol->string <символ>)

Конвертира символен атом (символ) в символен низ.

(string=? <низ$_1$> <низ$_2$>)

Проверява дали низовете **[<низ$_1$>]** и **[<низ$_2$>]** съвпадат (връща оценка ***#t***, ако двата низа съвпадат, или оценка ***#f*** – в противния случай), като различава главните и малките букви.

(string-ci=? <низ$_1$> <низ$_2$>)

Проверява дали низовете **[<низ$_1$>]** и **[<низ$_2$>]** съвпадат (връща оценка ***#t***, ако двата низа съвпадат, или оценка ***#f*** – в противния случай), като смята, че съответните главни и малки букви са неразличими (представляват едни и същи знакове). Частта “ci” в името на функцията означава “case insensitive”.

Примери

```
> (string-length "This is a string")
16
> (string-length "")
0
> (string-append "This is" " a string")
"This is a string"
> (string-append "12" "34" "56")
"123456"
> (substring "This is a string" 0 4)
"This"
> (substring "This is a string" 4 6)
" i"
> (substring "This is a string" 5 13)
"is a str"
> (symbol->string 'hello)
"hello"
```

```
> (string=? "This is a string"
     "This is a string")
#T
> (string=? "This is a string"
     "This is a STRING")
()
> (string-ci=? "This is a string"
     "This is a STRING")
#T
```

Дефиниции на процедури за работа със символни низове

Пример 1. Вмъкване на даден низ в друг низ (низът **insrt** се вмъква в низа **strng** така, че първият знак на **insrt** да има индекс **n** в низа – резултат).

Например **(string-insert "45" "123678" 3) ⟶ "12345678"** .

```
(define (string-insert insrt strng n)
   (string-append
      (substring strng 0 n)
      insrt
      (substring strng n (string-length strng))))
```

Пример 2. Обръщане на реда на знаковете на даден низ.

```
(define (string-reverse strng)
;;; Връща низ, който съдържа знаковете на [strng]
;;; в обратен ред
   (if (string=? strng "")
       ""
       (string-append
          (string-reverse
             (substring strng 1
                (string-length strng)))
          (substring-ref strng 0))))
(define (substring-ref strng n)
;;; Връща низ, съставен от n-тия елемент
;;; на даден низ
   (substring strng n (+ n 1)))
```

Пример 3. Проверка дали даден низ е подниз на друг низ.

```
(define (substring? sstr strng)
;;; Проверява дали низът [sstr] е подниз
;;; на низа [strng]
   (let ((l (string-length strng))
         (ls (string-length sstr)))
      (letrec ((substr?
                (lambda (i)
                   (if (> (+ i ls) l)
                       #f
                       (or (string=? sstr
                                     (substring strng i
                                        (+ i ls)))
                           (substr? (+ i 1))))))))
         (substr? 0))))
```

Вход и изход в Scheme

Вградени процедури за вход на данни в езика Scheme

Най-често за въвеждане на данни се използва вградената процедура ***read***. Обръщението към нея има следния вид:

```
(read)
```

Действие. В процеса на оценяване на обръщението към вградената процедура ***read*** интерпретаторът спира работата си и изчаква потребителя да въведе синтактично коректен S-израз (по-точно, коректната ***външна форма*** на някакъв S-израз). Оценката на обръщението към ***read*** съвпада с въведения S-израз.

Примери

```
> (read)
123
123
> (read)
"abcd"
"abcd"
> (read)
a
A
> (read)
(a b (c) d)
(A B (C) D)
> (read)
(a.b)
(A.B)
```

Вградени процедури за изход на данни в езика Scheme

Най-често при извеждане на данни се използват вградените процедури ***display***, ***write***, ***newline*** и ***writeln***. Първите три процедури са стандартни, а последната не е включена в стандарта на езика, но се поддържа в повечето среди за програмиране на Scheme.

Общ вид на обръщението към ***display***:

`(display <израз>)`

Действие. Извежда **[<израз>]** на текущото изходно устройство. Символните низове се извеждат без заграждащите ги двойни кавички. Оценката на обръщението към ***display*** по стандарт е неопределена. В използваната реализация на PC Scheme не се извежда оценка на обръщението към ***display***.

Примери

```
> (display (+ 1 2))
3
> (display "alabala")
alabala
> (display "ala\\bala")
ala\bala
```

Общ вид на обръщението към ***write***:

`(write <израз>)`

Действие. Извежда **[<израз>]** на текущото изходно устройство. Символните низове се извеждат със заграждащите ги двойни кавички. Оценката на обръщението към ***write*** по стандарт е неопределена. В използваната реализация на PC Scheme не се извежда оценка на обръщението към ***write***.

Примери

```
> (write (+ 1 2))
3
> (write "alabala")
"alabala"
> (write "ala\\bala")
"ala\\bala"
```

Общ вид на обръщението към ***newline***:

```
(newline)
```

Действие. Премества курсора в началото на следващия ред. В използваната реализация оценката на обръщението към ***newline*** е ***()***.

Пример

```
> (newline)

()
```

Общ вид на обръщението към ***writeln***:

(writeln <израз$_1$> <израз$_2$> ... <израз$_n$>)

Действие. Извежда на текущото изходно устройство последователно **[<израз$_1$>], [<израз$_2$>], ... , [<израз$_n$>]**, след което изпълнява обръщение към ***newline***. Символните низове се извеждат без заграждащите ги двойни кавички. В използваната реализация оценката на обръщението към ***writeln*** е ***()***.

Примери

```
> (writeln "The sum of 1 and 2 is " (+ 1 2))
The sum of 1 and 2 is 3
()
> (writeln "alabala")
alabala
()
> (writeln "ala\\bala")
ala\bala
()
```

Входно-изходни операции с файлове в езика Scheme

Входно-изходните операции с файлове се описват подобно на традиционните входно-изходни операции, като предварително се дефинират съответните портове. ***Портът*** е обект, към който са насочени в определен момент входните или изходните операции (от който се извършва четенето на данни или в който се извършва записването на данни).

Записване във файл

За да се пренасочат изходните операции от стандартния изход (потребителския екран) към определен файл, най-напред се оценява обръщение към функцията ***open-output-file***, която получава като аргумент символен низ – името на файла, в който ще се извършва записването (името на изходния файл), и връща като резултат изходния порт, асоцииран с посочения файл. При следващи обръщения към функциите ***display***, ***write*** и т.н. може да се зададе и втори аргумент, чиято оценка съвпада с новия изходен порт, и тогава извеждането на съответните данни ще се извършва в асоциирания с него изходен файл. След като извеждането на данни в изходния файл приключи, се оценява обръщение към процедурата ***close-output-port***, която получава като аргумент определен изходен порт и затваря асоциирания с него файл.

Пример

```
(define (output-to-file)
   (let ((port-out (open-output-file
                        "output1.txt")))
      (display "This is an output test."
         port-out)
      (newline port-out)
      (close-output-port port-out)))
```

Четене от файл

За да се пренасочат входните операции от стандартния вход (клавиатурата) към определен файл, най-напред се оценява обръщение към процедурата ***open-input-file***, която получава като аргумент символен низ – името на съществуващ файл, от който ще се извършва четенето (името на входния файл), и връща като резултат създадения входен порт, асоцииран с посочения файл. При следващи обръщения към функцията ***read*** може да се зададе аргумент, чиято оценка съвпада с новия входен порт, и тогава въвеждането на данни ще се извършва от асоциирания с него входен файл. Най-накрая при оценяване на обръщение към процедурата ***close-input-port*** се затваря входният файл, асоцииран с порта, който е оценка на аргумента на обръщението.

Пример. Следващата процедура работи при предположение, че във файла “input2.txt” е записана поредица от числа, и намира сумата на тези числа.

```
(define (sum-file)
   (let ((p (open-input-file "input2.txt")))
      (letrec ((add-items (lambda (sum)
                    (let ((item (read p)))
                       (cond ((eof-object? item)
                              (close-input-port p)
                              sum)
                             (else (add-items
                                (+ item sum)))))))))
         (add-items 0))))
```

Деструктивни процедури и присвояване на стойности в езика Scheme

Същност на деструктивните процедури и присвояването

Дефиниция. Процедура, която променя (част от) аргументите си, се нарича ***деструктивна процедура***.

Всички разглеждани до този момент процедури (примитивни и съставни, т.е. вградени и дефинирани) не бяха деструктивни, т.е. не променяха своите аргументи. Наличието и използването на деструктивни процедури създава условия за поява на процедури със странични ефекти, които причиняват промени в стойностите на нелокални променливи. Тези процедури от своя страна нарушават строго функционалния стил на програмиране, следван до този момент.

Пример. Реализация на проста система за теглене на пари от банкова сметка, която при първоначално налични в сметката 100 парични единици работи по следния начин:

```
> (withdraw 25)
75
> (withdraw 25)
50
>(withdraw 60)
"Not possible"
> (withdraw 35)
15
```

Примерна дефиниция

```
(define balance 100)
(define (withdraw amount)
   (if (>= balance amount)
       (begin (set! balance (- balance amount))
              balance)
       "Not possible"))
```

Обяснение на действието на използваните в горната дефиниция процедури и специални форми

1) Примитивна процедура за присвояване ***set!***

Общ вид на обръщението:

(set! <име на променлива> <израз>)

Действие. Оценява се **<израз>** и на променливата **<име на променлива>** (която предварително трябва да бъде дефинирана - например с помощта на ***define***) се присвоява нова стойност, равна на **[<израз>]**.

Оценката на обръщението към ***set!*** по стандарт е неопределена. В използваната реализация на PC Scheme съвпада с **[<израз>]**.

Забележка. Действието на процедурата ***set!*** се свежда до създаване на нова връзка в текущата среда между името на променливата и нейната нова стойност, като при това съществуващата до този момент връзка (а такава непременно съществува, тъй като използваната променлива трябва да бъде дефинирана предварително) се разрушава.

Следователно, процедурата ***set!*** е деструктивна. Имената на всички примитивни деструктивни процедури в Scheme завършват с "!".

2) Специална форма ***begin***

Общ вид на обръщението:

(begin <израз$_1$> <израз$_2$> ... <израз$_N$>)

Действие (семантика). Оценяват се последователно (от ляво на дясно) изразите **<израз$_i$>**.

Оценката на обръщението към ***begin*** съвпада с **[<израз$_N$>]**.

Анализ на предложеното решение на примерната задача. Предложеното решение е коректно, но в него се използва глобалната променлива **balance** (**balance** е променлива, която е дефинирана в глобалната среда). Тази променлива е достъпна както за процедурата **withdraw**, така и за всички останали процедури, които евентуално биха били дефинирани по-нататък, и следователно тя е изложена на опасност от некоректен достъп. Затова е целесъобразно да се потърси решение, в което променливата **balance** е достъпна само за процедурата **withdraw** и същевременно промените в стойността на **balance** имат достатъчно дълготраен характер (остават валидни до следващото използване на **withdraw**).

Пример за решение, което преодолява посочения проблем и удовлетворява поставените от нас изисквания към променливата **balance**:

```
(define new-withdraw
   (let ((balance 100))
      (lambda (amount)
         (if (>= balance amount)
             (begin (set! balance
                          (- balance amount))
                    balance)
             "Not possible"))))
```

Примери, илюстриращи работата на новата процедура

```
> (new-withdraw 30)
70
> (new-withdraw 30)
40
> (new-withdraw 50)
"Not possible"
```

Кратко обяснение на действието на процедурата **new-withdraw**. С помощта на обръщението към ***let*** в горната дефиниция се определя (установява) нова среда, в която е дефинирана локалната променлива **balance**, и тази променлива се свързва с начална стойност 100. В рамките на тази нова среда се използва ***lambda*** дефиниция, с помощта на която се дефинира (създава) процедура с аргумент **amount**, която има поведение, аналогично на това на дефинираната преди процедура **withdraw**. Тази процедура се връща като оценка на обръщението към ***let*** и, следователно, с нея се свързва името **new-withdraw**. С други думи, **new-withdraw** е процедура, която действа по същия начин, както дефинираната преди това процедура **withdraw**, но при нея променливата **balance** е недостъпна за други процедури.

Комбинирането на ***let*** (като средство за дефиниране на локални променливи) и ***set!*** е най-общата техника за програмиране, която ще използваме за конструиране на обекти, които имат поведение, аналогично на това на променливата **balance** от процедурата **withdraw**. За съжаление, използването на тази техника води до следния сериозен проблем. При разглеждането на средствата за дефиниране на процедури в езика Scheme ние въведохме модела на оценяване чрез заместване като механизъм, по който интерпретаторът действа, когато оценява обръщение към някаква съставна (дефинирана) процедура. Същността на този модел е следната: прилагането на дадена дефинирана процедура може да бъде интерпретирано като оценяване на тялото на тази процедура, при което формалните параметри са заместени със съответните фактически.

Въвеждането на средства за присвояване (каквото средство в случая е процедурата ***set!***) обаче води до неадекватност на модела на оценяване чрез заместване, който до този момент беше валиден и вършеше добра работа. Защо това е така, ще покажем малко по-нататък. Най-важно обаче в случая е това, че в момента не сме в състояние да обясним по-дълбоките механизми на действието на процедурата **new-withdraw**, тъй като вече не разполагаме с подходящ за целта модел. За да можем да дадем достатъчно задълбочено обяснение на действието на тази процедура, имаме нужда от по-съвършен модел на оценяване на обръщенията към дефинирани процедури. Такъв модел е т. нар. ***модел на средите*** и той ще бъде въведен малко по-нататък. Преди това обаче ще разгледаме някои възможни развития на идеите, илюстрирани в дефиницията на процедурата **new-withdraw**.

Дефиниране на процедура - генератор на банкови сметки, предназначени само за теглене на пари

Дефиниция

```
(define (make-withdraw balance)
   (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance
                       (- balance amount))
                 balance)
          "Not possible")))
```

Коментар на дефиницията

Дефинираната току-що процедура **make-withdraw** е процедура от по-висок ред (като резултат тя връща процедура).

Формалният параметър **balance** задава началното количество пари (в някакви единици), с което се създава съответната сметка, предназначена за теглене на пари.

Примери, илюстриращи работата на процедурата **make-withdraw**

```
> (define w1 (make-withdraw 100))
w1
> (define w2 (make-withdraw 100))
w2
> (w1 50)
50
> (w2 70)
30
> (w2 40)
"Not possible"
> (w1 40)
10
```

И т.н.

Коментар на примерите

Дефинираните по-горе променливи **w1** и **w2** имат смисъл на две различни сметки за теглене на пари, в които началната сума е една и съща - 100 единици. Те описват два различни, напълно независими обекта. Тегленето на пари от едната сметка по никакъв начин не влияе на наличната сума в другата, т.е. тегленето на пари от едната сметка не влияе на възможностите за теглене на пари от другата сметка.

Дефиниране на процедура - генератор на банкови сметки, предназначени за теглене и внасяне на пари

Дефиниция

```
(define (make-account balance)
   (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance
                       (- balance amount))
                 balance)
          "Not possible"))
   (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
```

```
(define (dispatch m)
   (cond ((eq? m 'withdraw) withdraw)
         ((eq? m 'deposit) deposit)
         (else (error "Unknown request!" m))))
dispatch)
```

Коментар на дефиницията

Дефинираната процедура **make-account** също е процедура от по-висок ред. При всяко обръщение към **make-account** се създава нова среда, която е старата (съществуващата) плюс добавената към нея променлива **balance**. В тази среда се дефинират три нови процедури, две от които променят стойността на **balance**, а третата анализира заявката на потребителя и връща като резултат една от другите две.

Примери, илюстриращи работата на процедурата **make-account**

```
> (define acc (make-account 100))
acc
> ((acc 'withdraw) 50)
50
> ((acc 'withdraw) 60)
"Not possible"
> ((acc 'deposit) 40)
90
> ((acc 'withdraw) 60)
30
```

И т.н.

Коментар на примерите

При всяко обръщение към **acc** се получава като резултат една от двете локални процедури **deposit** или **withdraw**, след което получената процедура се прилага върху конкретно зададена парична сума **amount**. Както беше и при процедурата **make-withdraw**, при всяко ново обръщение към **make-account** се създава напълно нов обект - нова банкова сметка със зададена начална парична сума в нея.

Проблеми, до които води присвояването на стойности в езика Scheme

Нека разгледаме като пример следния опростен вариант на въведената по-горе процедура - генератор на банкови сметки, предназначени за теглене на пари:

```
(define (make-w balance)
   (lambda (amount)
      (set! balance (- balance amount))))
```

Както вече видяхме, **make-w** връща като резултат процедура, която изважда стойността на своя аргумент от стойността на променливата **balance**, като при това променя стойността на **balance**.

Нека освен това символът **w** е дефиниран както следва:

```
(define w (make-w 25))
```

Примери за работата на **w**:

```
> (w 10)
15
> (w 10)
5
```

Опит за обяснение на работата на **w**. Ако се опитаме да приложим модела на оценяване чрез заместване, който използвахме досега, се получава следният резултат:

```
(w 10) ⟶ ((make-w 25) 10) ⟶
((lambda (amount) (set! 25 (- 25 amount))) 10)
⟶
(set! 25 (- 25 10)) ⟶ (set! 25 15) ⟶ ???
```

Следователно, като използвахме модела на оценяване чрез заместване, се получи безсмислен резултат.

Извод. Методът (моделът) на оценяване чрез заместване не е валиден в този случай (и изобщо в случаите на процедури, в които се извършва присвояване на стойност).

Причина. Моделът на оценяване чрез заместване се основава на идеята, че символите (променливите) са имена на стойности (в някаква среда). С въвеждането на ***set!*** и идеята, че стойността на дадена променлива може да се променя, променливата вече не е само име на стойност. Сега всяка променлива се свързва по определен начин със специално място (с указател към специално място), където се съхранява съответната ѝ стойност, и тази стойност (т.е. стойността, която се съхранява на това място) може да се променя.

Следствия:

- използваният досега модел на оценяване не е валиден и, следователно, имаме нужда от нов, по-съвършен модел на оценяване, за да можем да проследяваме изпълнението на нашите програми;

- не можем да установяваме и доказваме еквивалентност на програми и по-общо, не можем да установяваме кои обекти са еднакви (или еквивалентни в определен смисъл).

Пример. Нека **w1** и **w2** са дефинирани както следва:

```
(define w1 (make-w 25))
(define w2 (make-w 25))
```

Въпрос: еднакви (еквивалентни) ли са **w1** и **w2**?

На този въпрос би трябвало да се отговори с ***не***, тъй като **w1** и **w2** имат различно поведение във времето (промените, които се извършват върху елементите на единия от двата обекта, не се отразяват автоматично върху другия обект).

Ако обаче разгледаме обектите **acc1** и **acc2**, дефинирани както следва:

```
(define acc1 (make-w 25)),
(define acc2 acc1),
```

то обектите **acc1** и **acc2** са еднакви (еквивалентни), или по-точно **acc1** и **acc2** са две имена на един и същ обект.

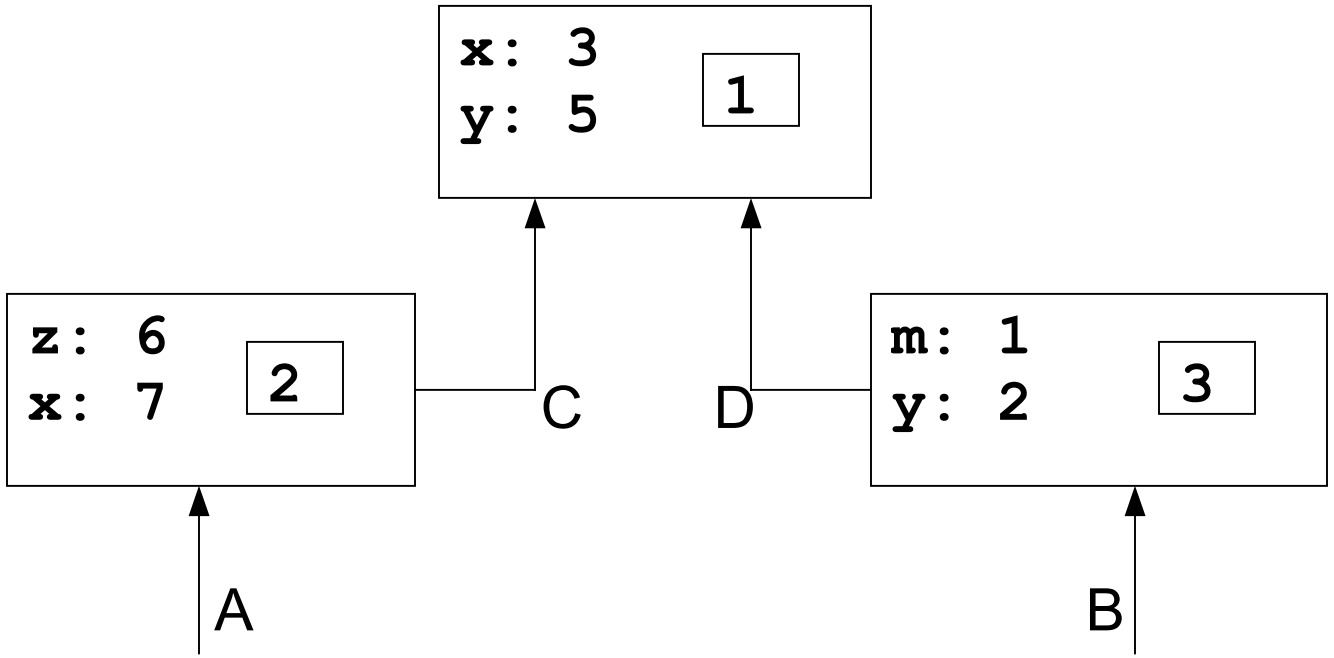
Обща бележка. Както вече беше показано, използването на деструктивни процедури води до много проблеми. Все пак в много езици те се използват, тъй като водят до създаването на програми, които работят много по-ефективно на масовите фон Нойманови компютри. Някои от съвременните езици за функционално програмиране не включват деструктивни процедури (операции), а разчитат на принципно различни (не Нойманови) методи за ефективна реализация.

Модел на средите за оценяване на обръщения към съставни процедури

Както вече беше посочено, всяка променлива 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:

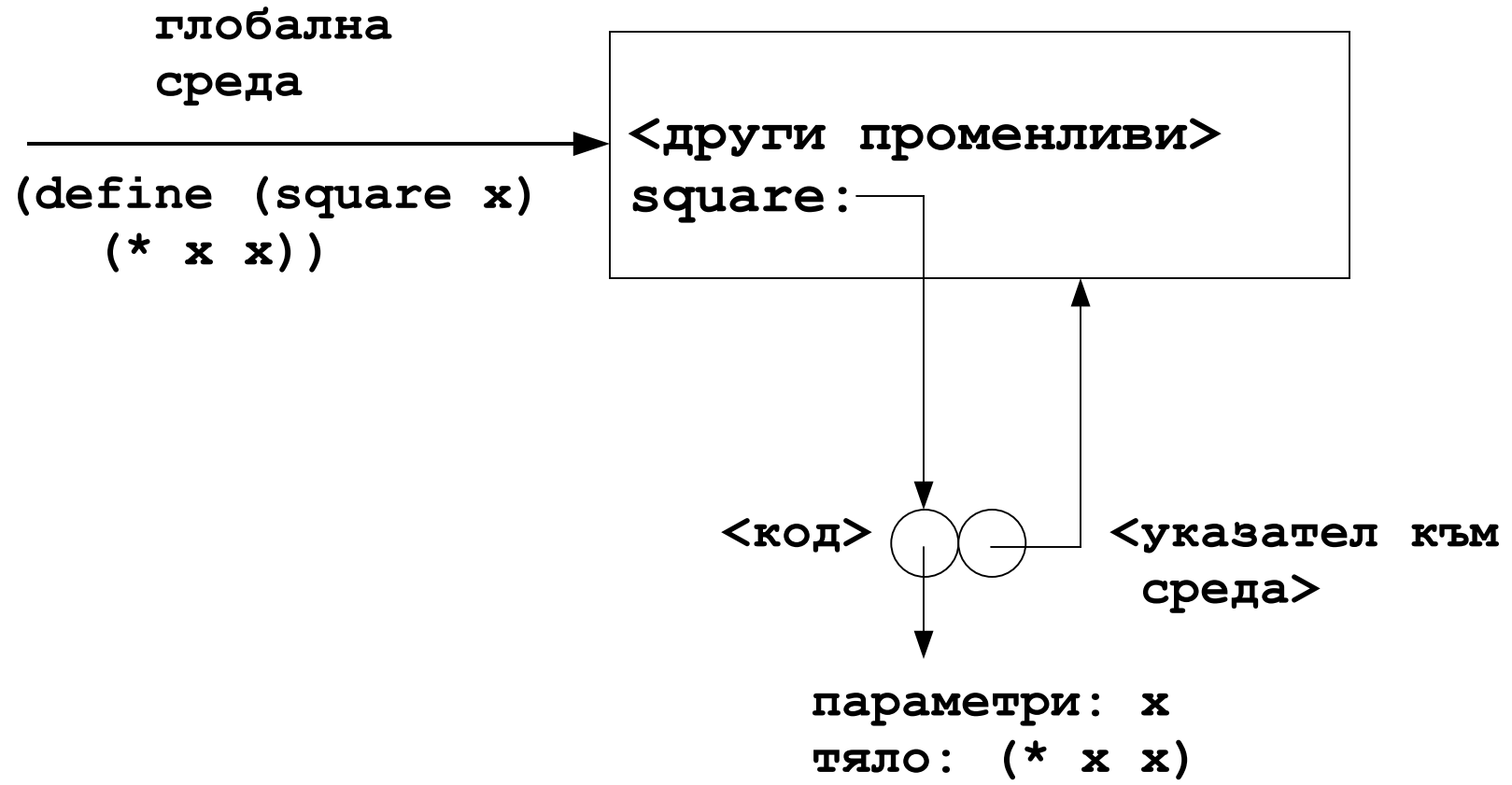
```
(define (square x) (* x x))
```

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:

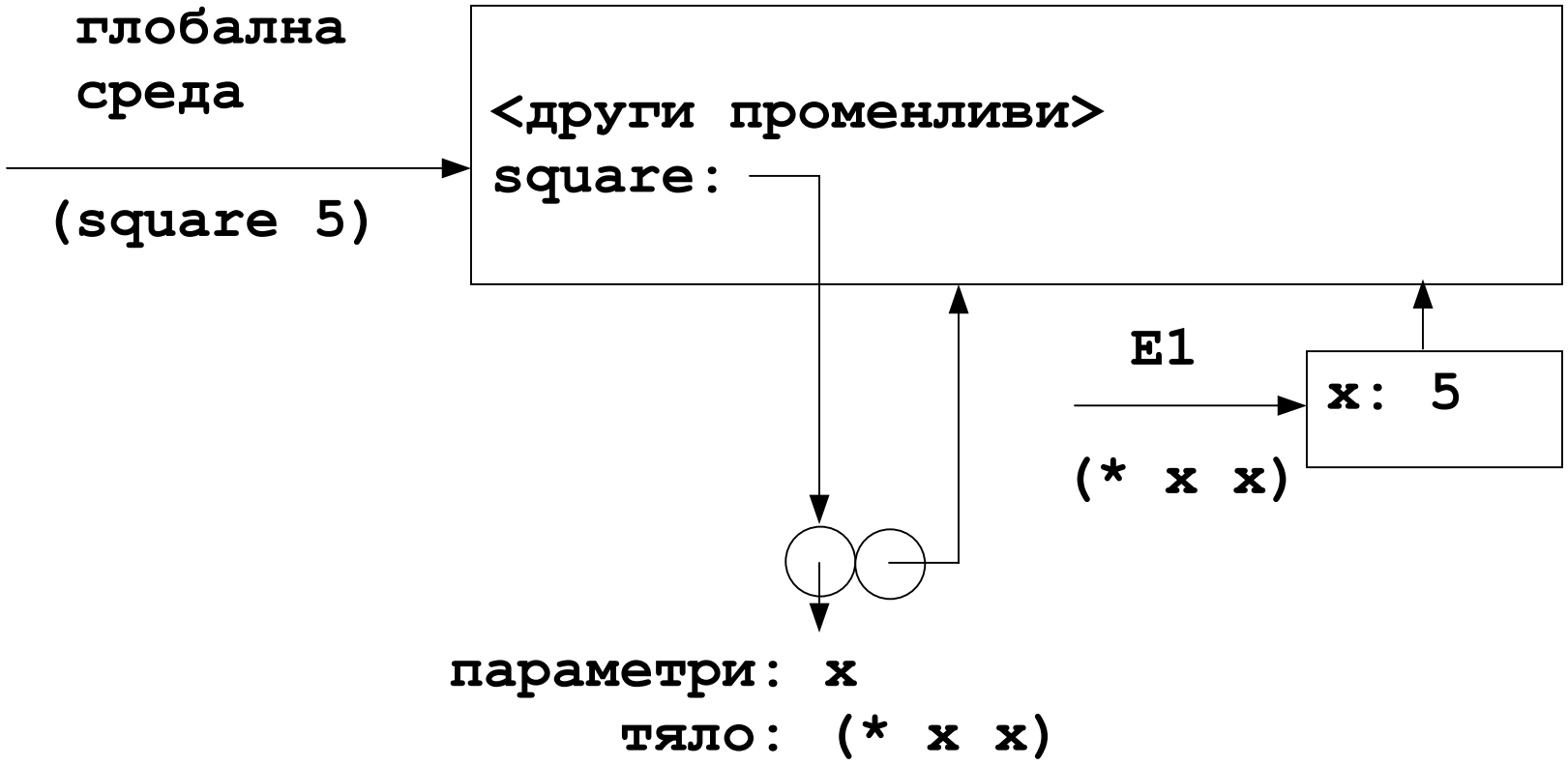


След като вече знаем как се дефинират процедури, можем да покажем и как се прилагат дефинирани процедури.

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).

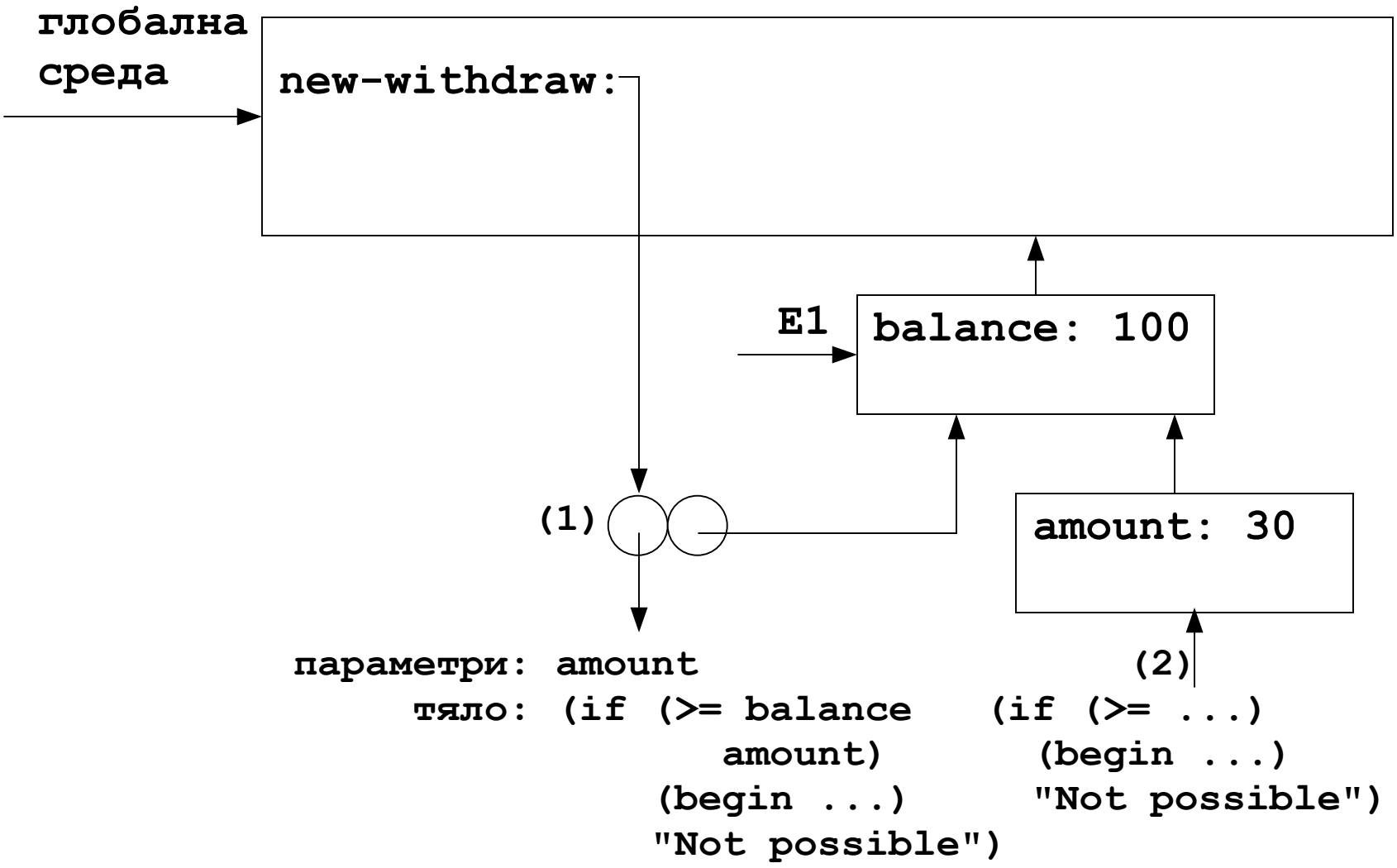
2) При оценяване на формата **(set! <променлива> <стойност>)** в някаква среда се локализира съществуващото свързване (в тази среда) за цитираната променлива и се променя стойността, свързана до този момент със същата променлива;

3) 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>**.

Ще проследим процеса на оценяване на следните изрази:

```
(1)  (define new-withdraw
       (let ((balance 100))
          (lambda (amount)
             (if (>= balance amount)
                 (begin (set! balance
                              (- balance amount))
                        balance)
                 "Not possible"))))

(2)  (new-withdraw 30)
```


глобална
среда
new-withdraw:
E1
balance: 100
(1)
amount: 30
(2)
параметри: amount
тяло: (if (>= balance
amount)
(begin ...)
"Not possible")
(if (>= ...)
(begin ...)
"Not possible")

След оценяването на **(2)** се получава следното състояние на глобалната среда и средата **E1**: