Деструктивни процедури и присвояване на стойности в езика Racket

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 стойностите на нелокални променливи и в частност 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 (#<***void***>).

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 примитивните деструктивни процедури в Racket 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 Racket 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. Защо това е така, ще покажем малко по-нататък. Най-важно обаче в случая е това, че за да можем да дадем достатъчно задълбочено обяснение на механизма на действието на процедурата **new-withdraw**, имаме нужда от по-съвършен модел на оценяване на обръщенията към дефинирани процедури. Такъв модел е т. нар. ***модел на средите***, който вече беше разгледан накратко. Преди да го анализираме по-подробно, 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))
(define w2 (make-withdraw 100))

> (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 '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 Racket

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:

- моделът на оценяване чрез заместване не е валиден и, следователно, имаме нужда от нов, по-съвършен модел на оценяване, за да можем да проследяваме изпълнението на нашите програми. Такъв модел на оценяване е разгледаният вече ***модел на средите***;

- 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.

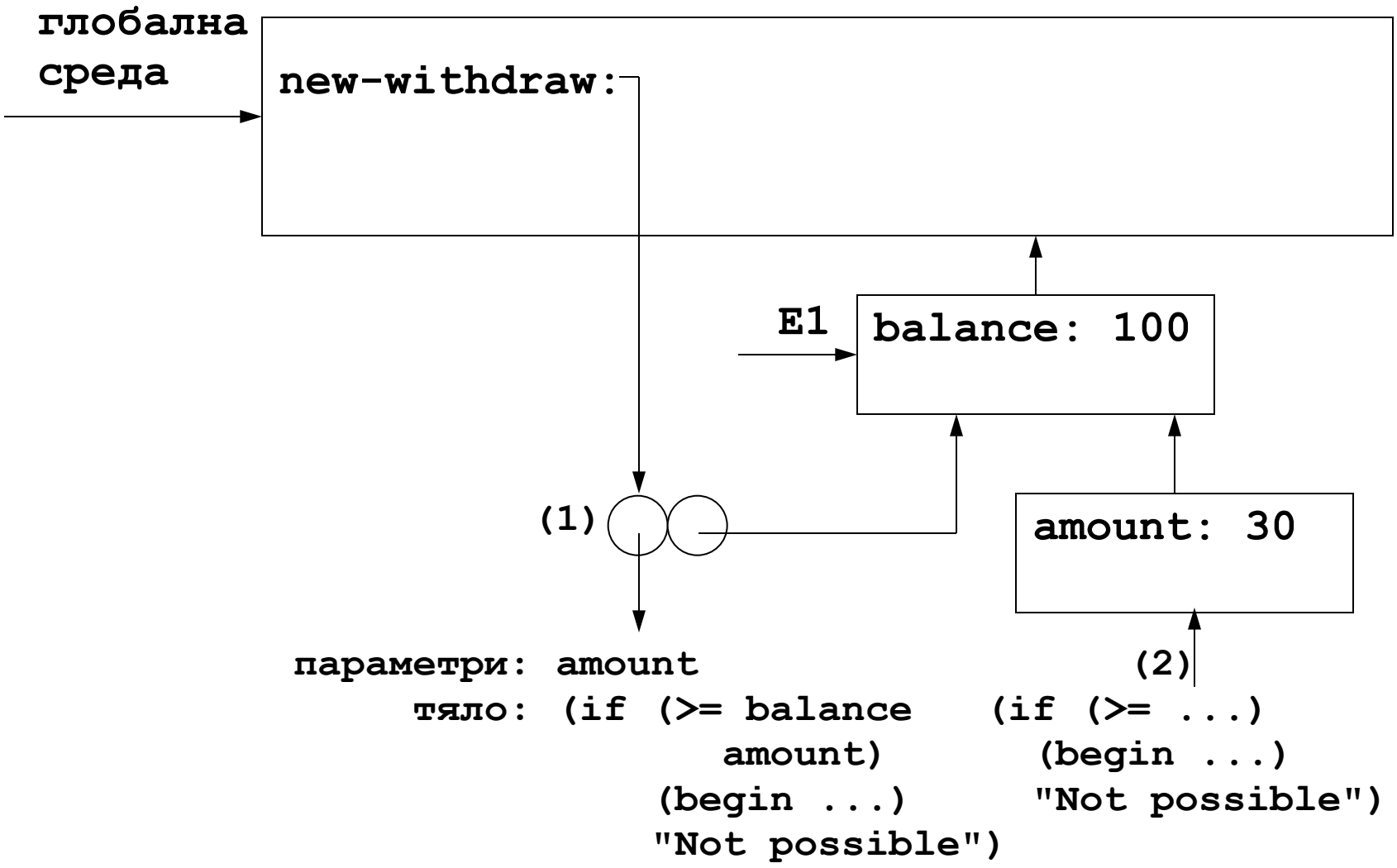
Ще проследим процеса на оценяване в рамките на модела на средите на следните изрази:

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

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

Оценяване на обръщение към специалната форма ***set!*** в рамките на модела на средите:

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.


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

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