

λ -исчисление

Кевролетин В.В. 236гр.

11 мая 2011 г.

Задание31

Привести пример функции $f : N \rightarrow N$, которая обладает свойством $f(1) + f(2) \neq f(2) + f(1)$. Объяснить причину такого поведения.

Условие

Решение

```
(define count 0)

(define (f x)
  (set! count (* (+ count x) x))
  count)
```

Результат возвращаемый описанной выше функции $f(x)$ зависит от глобальной переменной `count`, значение которой меняется в теле этой же функции. Поэтому результат зависит не только от переданного ей аргумента, но и от последовательности предыдущих вызовов $f(x)$. $f(1) + f(2) \neq f(2) + f(1)$ потому что в первом случае сначала вычисляется $f(1)$ а потом $f(2)$, а во втором случае наоборот: разный порядок выполнения - разные значения.

Задание32

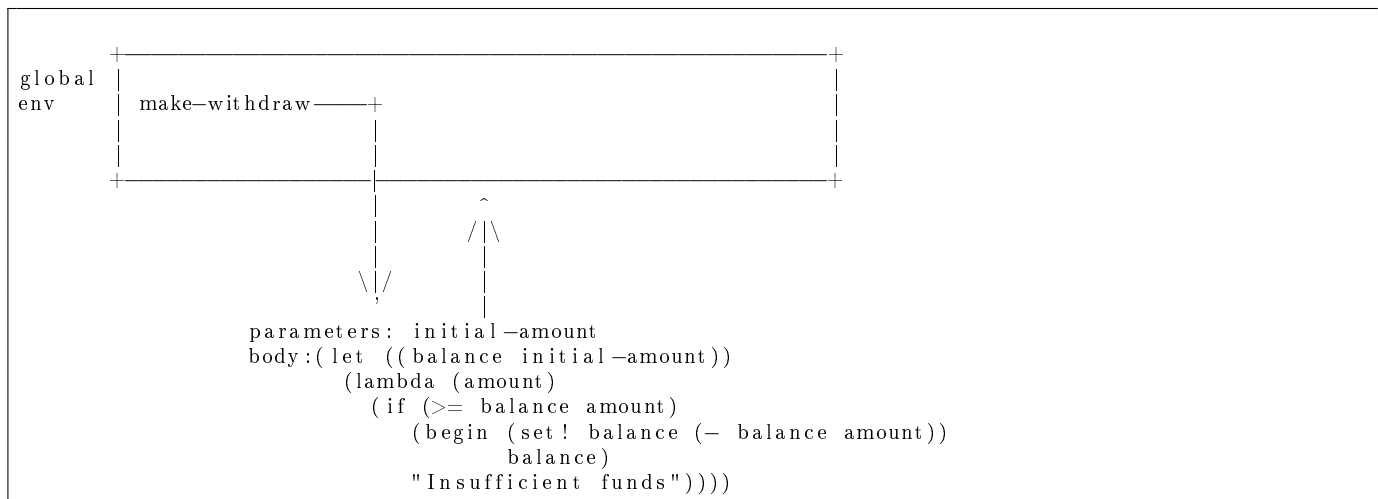
Условие

In the make-withdraw procedure, the local variable `balance` is created as a parameter of `make-withdraw`. We could also create the local state variable explicitly, using `let`, as follows:

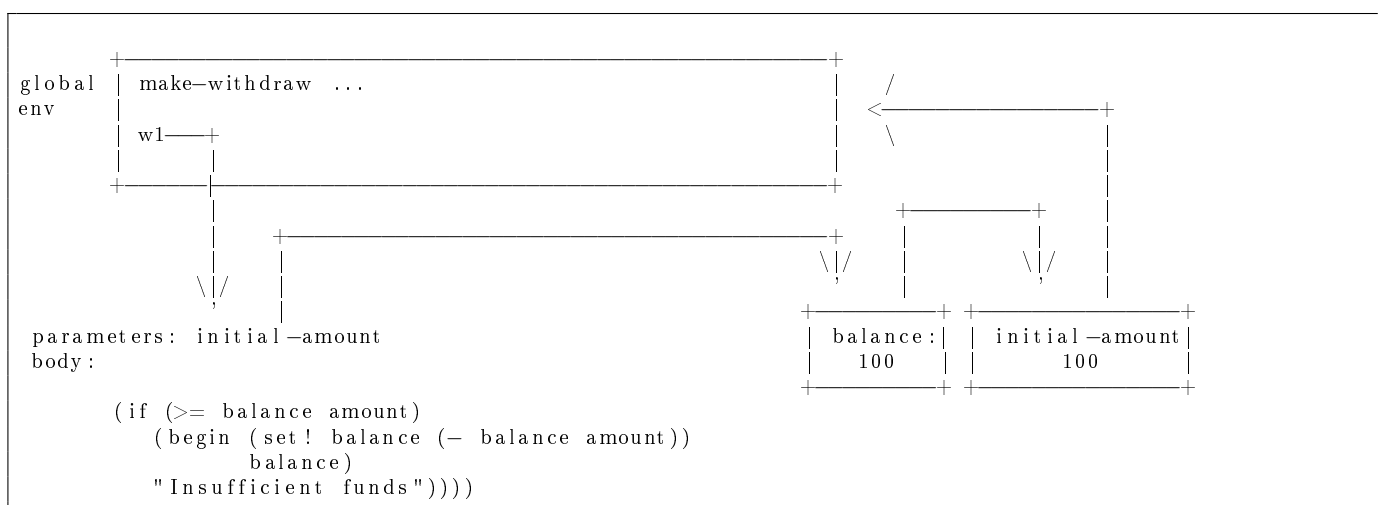
```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                  balance)
          "Insufficient funds")))))
```

Show that the two versions of `make-withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

Решение



После выполнения (define W1 (make-withdraw 100)) будет создан объект w1. Ниже на схеме изображено состояние окружения после создания w1. На схеме видно, что полученный объект не отличается от созданного в 1 версии.



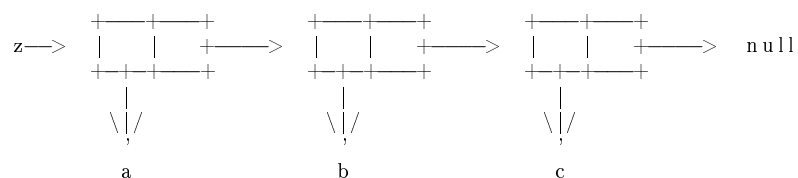
Задание33

Условие

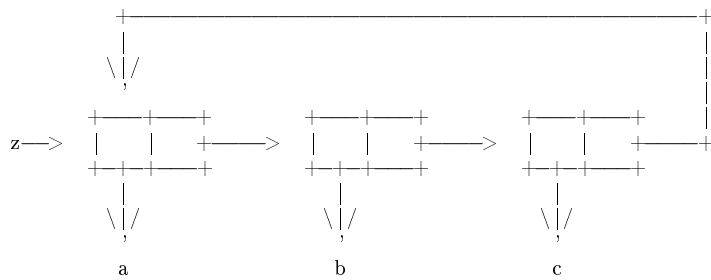
make-cycle

Решение

БЫЛО:



Станет:



Вызов (last-pair z) приведёт к заикливлению выполнения программы, т.к. условие (null? x) не выполняется не для одного элемента списка.

Задание34

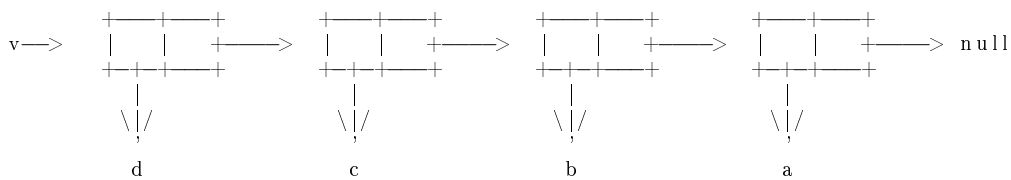
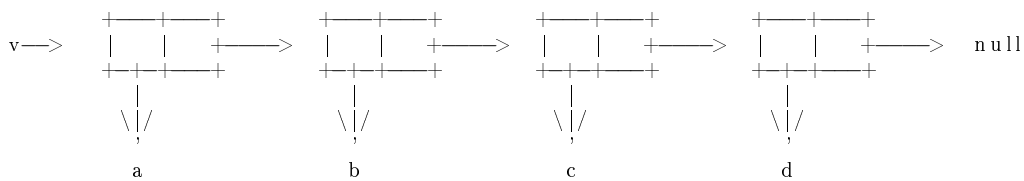
Условие

mystery

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

Решение

Процедура делает то же самое что и reverse



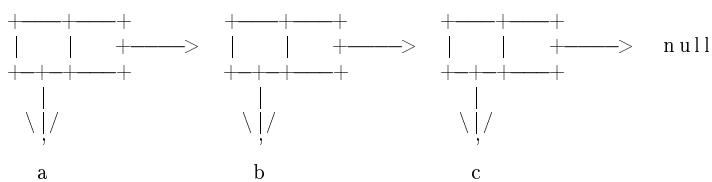
Задание35

Условие

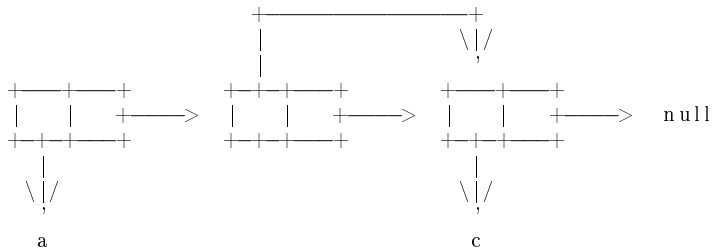
count-pairs

Решение

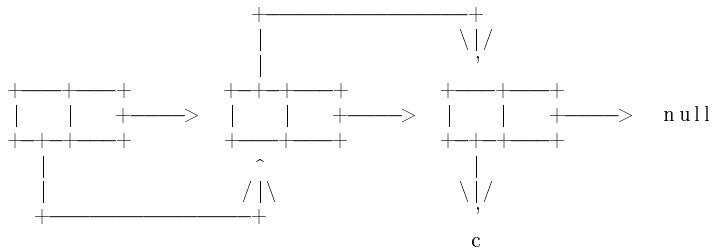
3



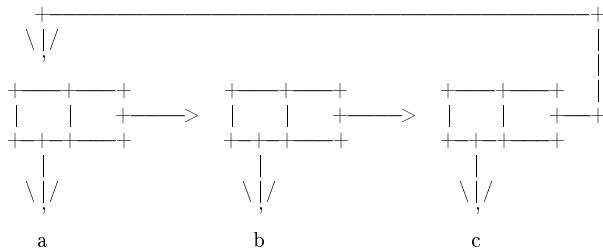
4



7



Никогда не завершится



Задание36

Условие

A correct version of the count-pairs.

Решение

В моей версии пары, которые мы уже посчитали помещаются в список counted. Перед тем как учесть очередную пару проверяется есть ли она в списке counted. Если есть то не считаем её. Если нет то учитываем её и помещаем её в список counted.

```
(define (in-list? l x)
  (cond ((null? l) 0)
        ((eq? (car l) x) 1)
        (else (in-list? (cdr l) x))))
```

```
(define (push l x)
  (set! l (cons x l)))
```

```
(define counted '())
```

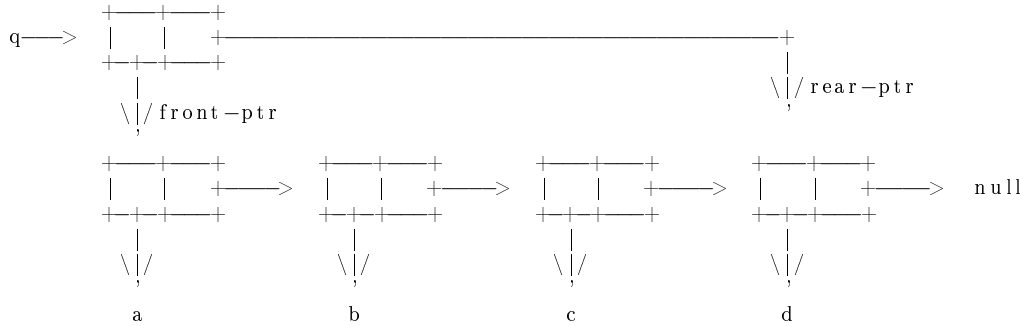
```
(define (count-pairs x)
  (cond ((not (pair? x)) 0)
        ((in-list? counted x) 0)
        (else
         (push counted x)
         (+ (count-pairs (car x))
            (count-pairs (cdr x))
            1))))
```

Задание37

Условие

print-queue

Решение



Видно, что до последнего элемента можно дойти 2мя путями, поэтому интерпретатор печатает его дважды. Чтобы распечатать все элементы очереди без повторения достаточно просто распечатать список, на который указывает `front-ptr`, и не печатать хвост, на который указывает `rear-ptr`:

```
(define (print-queue q) (front-ptr q))
```

Задание38

Условие

queue как процедура с локальными состояниями

Решение

```
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '()))

    (define (empty-queue?) (null? front-ptr))
    (define (front-queue)
      (if (empty-queue?)
          (error "FRONT called with an empty queue")
          (car front-ptr)))
    (define (insert-queue!)
      (lambda (item)
        (let ((new-pair (cons item '())))
          (cond ((empty-queue?)
                 (set! front-ptr new-pair)
                 (set! rear-ptr new-pair))
                (else
                 (set-cdr! rear-ptr new-pair)
                 (set! rear-ptr new-pair))))))
    (define (delete-queue!)
      (cond ((empty-queue?)
            (error "DELETE! called with an empty queue"))
            (else
             (set! front-ptr (cdr front-ptr)))))

    (define (dispatch m)
      (cond
        ((eq? m 'front-ptr) front-ptr)
        ((eq? m 'rear-ptr) rear-ptr)
        ((eq? m 'empty-queue?) (empty-queue?))
        ((eq? m 'front-queue) (front-queue))
        ((eq? m 'insert-queue!) (insert-queue!))
        ((eq? m 'delete-queue!) (delete-queue!))
        (else (error "Undefined operation — QUEUE" m))))
    dispatch))

(define (front-ptr q) (q 'front-ptr))
(define (rear-ptr q) (q 'rear-ptr))
(define (empty-queue? q) (q 'empty-queue?))
(define (front-queue q) (q 'front-queue))
(define (insert-queue! q v) ((q 'insert-queue!) v))
(define (delete-queue! q) (q 'delete-queue!))
```

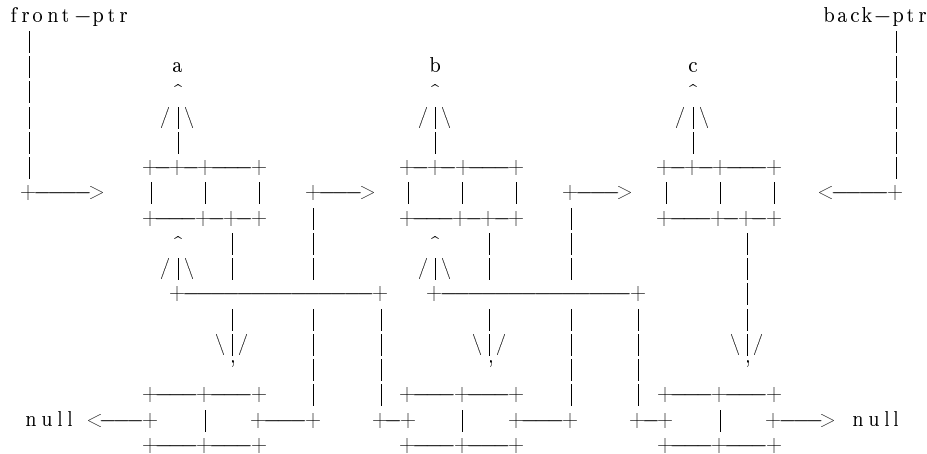
Задание39

Условие

deque

Решение

Для одного элемента надо хранить 2 указателя: на следующий и предыдущий элемент. Для этого будем в паре хранить данные и другую пару, в которой будут содержаться ссылки на предыдущий и следующий элемент:



```
(define (make-dequeue)
  (let ((front-ptr '())
        (back-ptr '()))

    (define (empty-dequeue?) (null? front-ptr))
    (define (front-dequeue)
      (if (empty-dequeue?)
          (error "FRONT called with an empty dequeue")
          (car (car front-ptr))))
    (define (back-dequeue)
      (if (empty-dequeue?)
          (error "FRONT called with an empty dequeue")
          (car (car back-ptr))))
    (define (insert-front-dequeue! item)
      (lambda (item)
        (let ((new-bottom-pair (cons '() '())))
          (let ((new-top-pair (cons item new-bottom-pair)))
            (cond ((empty-dequeue?)
                   (set! front-ptr new-top-pair)
                   (set! back-ptr new-top-pair))
                  (else
                   (set-car! (cdr front-ptr) new-top-pair)
                   (set-cdr! new-bottom-pair front-ptr)
                   (set! front-ptr new-top-pair)))))))
    (define (insert-back-dequeue! item)
      (lambda (item)
        (let ((new-bottom-pair (cons '() '())))
          (let ((new-top-pair (cons item new-bottom-pair)))
            (cond ((empty-dequeue?)
                   (set! front-ptr new-top-pair)
                   (set! back-ptr new-top-pair))
                  (else
                   (set-cdr! (cdr back-ptr) new-top-pair)
                   (set-car! new-bottom-pair back-ptr)
                   (set! back-ptr new-top-pair)))))))
    (define (delete-front-dequeue!)
      (cond ((empty-dequeue?)
             (error "DELETE! called with an empty queue"))
            (else
             (set! front-ptr (cdr (cdr front-ptr)))
             (if (not (null? front-ptr))
                 (set-car! (cdr front-ptr) '())))))
    (define (delete-back-dequeue!)
      (cond ((empty-dequeue?)
```

```

        (error "DELETE! called with an empty queue"))
      (else
       (set! back-ptr (car (cdr back-ptr)))
       (if (not (null? back-ptr))
           (set-cdr! (cdr back-ptr) '()))))
  (define (print-dequeue back-node result)
    (if (null? back-node) '(result)
        (let ((new-res (cons (car back-node) result)))
          (print-dequeue (car (cdr back-node)) new-res))))

  (define (dispatch m)
    (cond
      ((eq? m 'front-ptr) front-ptr)
      ((eq? m 'back-ptr) back-ptr)
      ((eq? m 'empty-dequeue?) (empty-dequeue?))
      ((eq? m 'front-dequeue) (front-dequeue))
      ((eq? m 'back-dequeue) (back-dequeue))
      ((eq? m 'insert-front-dequeue!) (insert-front-dequeue!))
      ((eq? m 'insert-back-dequeue!) (insert-back-dequeue!))
      ((eq? m 'delete-front-dequeue!) (delete-front-dequeue!))
      ((eq? m 'delete-back-dequeue!) (delete-back-dequeue!))
      ((eq? m 'print-dequeue) (print-dequeue back-ptr))
      (else (error "Undefined operation — QUEUE" m))))
  dispatch))

(define (front-ptr q) (q 'front-ptr))
(define (back-ptr q) (q 'back-ptr))
(define (empty-dequeue? q) (q 'empty-dequeue?))
(define (front-dequeue q) (q 'front-dequeue))
(define (insert-front-dequeue! q v) ((q 'insert-front-dequeue!) v))
(define (insert-back-dequeue! q v) ((q 'insert-back-dequeue!) v))
(define (delete-front-dequeue! q) (q 'delete-front-dequeue!))
(define (delete-back-dequeue! q) (q 'delete-back-dequeue!))
(define (print-dequeue q) (q 'print-dequeue))

```