ЛЕКЦИЯ 5
Остаточные вычисления. Программирование в стиле передачи остаточных вычислений

Остаточные вычисления

- *Остаточные вычисления* (continuations) – что-то, что ждёт значение. С каждым промежуточным значением связано остаточное вычисление – вычисления, которые должны быть выполнены, как только значение станет известным.
- Например: (sqrt (+ 1 *(read)*)). Здесь всё, что вокруг вызова read ждёт, пока пользователь не введет число.
- Остаточные вычисления для вызова функции foo в выражении (* (+ 5 1) *(foo 1)*) можно описать через lambda: (lambda (x) (* (+ 5 1) x)) и переписать код как: ((lambda (x) (* x (+ 5 1))) *(foo 1)*). Заметим, что мы заставили интерпретатор вычислять в другом порядке: сначала lambda, потом 2й множитель, потом 1й множитель, потом произведение! Для (read) из примера выше: (lambda (x) (sqrt (+ 1 x))) ((lambda (x) (sqrt (+ 1 x))) (read)).

Остаточные вычисления можно передавать

- Вызывать составленную lamdba от своего результата можно заставить сами функции из подвыражения, если их «переформатировать».

```
(define (*-cps x y cc) (cc (* x y)))
; *-cps помимо 2х множителей получает функцию cc, являющуюся
; продолжением и вызывает её, передавая произведение
(define (+-cps x y cc) (cc (+ x y)))
(define (foo-cps x cc) (cc (foo x)))
; аналогично
; перепишем (* (+ 5 1) (foo 1)) с новыми функциями
(foo-cps 1 (lambda (foo-res)
        (+-cps 5 1 (lambda (sum)
                (*-cps sum foo-res (lambda (x) x))))))
; анонимная тождественная функция (lambda (x) x) ещё пригодится
```

Остаточные вычисления

- Остаточные вычисления – динамические объекты. На каждом шаге вычисления выражения есть текущие остаточные вычисления.
- Например, рассмотрим остаточные вычисления, связанные с рекурсивным вызовом при вычислении факториала:

```
(define (fact n) (if (= n 0) 1 ( * n (fact (- n 1)))))

> (fact 3) -->
 (* 3 (fact 2)) -->
 (* 3 (* 2 (fact 1))) -->
 (* 3 (* 2 (* 1 (fact 0)))) -->
…
-> 6
```

Факториал в стиле передачи продолжений

Рассмотрим реализацию факториала в виде функции с дополнительным параметром – функцией, представляющей собой остаточные вычисления в явном виде:

```
(define (fact-cps n cc)
        (if (= n 0) (cc 1)
                (fact-cps (- n 1) (lambda (x) (cc (* n x))))))
```

Пример: если нужен 2!, то n=2, cc = тождественная функция

(fact-cps 2 *(lambda (x) x)*)

по подстановочной модели:

> (fact-cps 1 *(lambda (x_1) ((lambda (x) x) (* 2 x_1)))*) -->

(fact-cps 0 *(lambda (x_2) ((lambda (x_1)((lambda (x) x)(* 2 x_1))) (* 1 x_2)))*) -->

(*((lambda (x_2) ((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 x_2)))* 1)

Факториал в стиле передачи продолжений
Окончание

> ((lambda (x_2) ((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 x_2))) 1) -->
((lambda (x_1) ((lambda (x) x) (* 2 x_1))) (* 1 1)) -->
((lambda (x) x) (* 2 1)) -->
((lambda (x) x) 2) ->
2

В цикле была вычислена функция, которая даст ответ, после чего она была применена.

fact-cps – итеративный, в нём хвостовая рекурсия!

На память, занимаемую доп. параметром cc, не будем обращать внимание. fact-cps описан в учебных целях, т. к. известен простой способ его реализовать хвостовой рекурсией.

Reverse в стиле передачи продолжений

- Сначала опишем reverse в обычном стиле *рекурсивно*:

```
(define (reverse-simple lst)
  (if (null? lst) '()
   (append (reverse-simple (cdr lst)) (list (car lst)))))
```

- пример вычисления вызова:

```
> (reverse-simple '(1 2 3)) -->
```

; для наглядности в примере сразу подставлены результаты cdr и проч.

```
(append (reverse-simple '(2 3)) '(1)) -->
(append (append (reverse-simple '(3)) '(2)) '(1)) --> …
```

Reverse в стиле передачи продолжений

- Reverse в continuation-passing style

```
(define (reverse-cps lst cc)
  (if (null? lst) (cc '())
  (reverse-cps (cdr lst) (lambda (x) (cc (append x (list (car lst)))))))
```

- что общего с обычным описанием?

```
  (if (null? lst) '()
  (append (reverse-simple (cdr lst)) (list (car lst))))
```

- пример вычисления вызова:

```
> (reverse-cps '(1 2) (lambda (x) x)) -->
(reverse-cps '(2) (lambda (x1) ((lambda (x) x) (append x1 '(1)))) -->
(reverse-cps '() (lambda (x2) ((lambda (x1) ((lambda (x) x) (append x1 '(1))))
   (append x2 '(2))))) --> ...
```

Reverse в стиле передачи продолжений

((lambda(x_2)((lambda(x_1)((lambda(x) x)(append x_1 '(1))))(append x_2 '(2)))) *'()*) -->

((lambda (x_1) ((lambda (x) x) (append x_1 '(1)))) (append *'()* '(2))) -->

((lambda (x_1) ((lambda (x) x) (append x_1 '(1)))) *'(2)*) -->

((lambda (x) x) (append *'(2)* '(1))) -->

((lambda (x) x) *'(2 1)*)

-> *(2 1)*

reverse-cps описан в учебных целях, т. к. известен простой способ его реализовать хвостовой рекурсией.

Как reverse-simple стал reverse-cps

- шаг 1) переименовываем функцию и добавляем параметр cc

```
(define (reverse-cps lst cc)
(if (null? lst) '()
(append (reverse-cps (cdr lst)) (list (car lst)))))
```

- шаг 2) применяем cc к обеим веткам if:

```
(define (reverse-cps lst cc)
(if (null? lst) (cc '())
(cc (append (reverse-cps (cdr lst)) (list (car lst))))))
```

- шаг 3) по нижней строке выписываем остаточное вычисление относительно вызова reverse-cps как lambda:

```
(lambda (x) (cc (append x (list (car lst))))) – это cc для рекурсивного вызова
```

- шаг 4) переписываем рекурсивный вызов, добавляя 2й параметр

```
(reverse-cps (cdr lst) (lambda (x) (cc (append x (list (car lst))))))
```

Вставка в хвост в стиле передачи продолжений

- пишем вставку в хвост app-simple в обычном стиле *без хвостовой рекурсии*:

```
(define (app-simple lst e)
   (if (null? lst) (list e)
       (cons (car lst) (app-simple (cdr lst) e))))
```

- пример вычисления вызова

```
> (app-simple '(1 2) 3) -->
(cons 1 (app-simple '(2) 3)) -->
(cons 1 (cons 2 (app-simple '() 3))) --> …
-> (1 2 3)
```

Вставка в хвост в стиле передачи продолжений

- шаг 1) переименовываем функцию и добавляем параметр cc:

```
(define (app-cps lst e cc)
(if (null? lst) (list e)
(cons (car lst) (app-cps (cdr lst) e))))
```

- шаг 2) применяем cc ко всем веткам:

```
(define (app-cps lst e cc)
(if (null? lst) (cc (list e))
(cc (cons (car lst) (app-cps (cdr lst) e)))))
```

- шаг 3) выписываем остаточные вычисления в рекурсивном вызове:

```
(lambda (x) (cc (cons (car lst) x)))
```

- шаг 4) переписываем 3-ью строчку:

```
(app-cps (cdr lst) e (lambda (x) (cc (cons (car lst) x))))
```

Вставка в хвост в cps

- в итоге:

```
(define (app-cps lst e cc)
(if (null? lst) (cc (list e))
    (app-cps (cdr lst) e (lambda (x) (cc (cons (car lst) x))))))
```

- пример: (app-cps *'(1 2) 3 (lambda (x) x)*) ->

> (app-cps *'(2) 3* (lambda (x_1) (*(lambda (x) x)* (cons *1* x_1)))) -->

(app-cps *'() 3* (lambda(x_2)(*(lambda (x_1) ((lambda(x) x) (cons 1 x_1)))* (cons *2* x_2))))

--> (*(lambda (x_2) ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 x_2)))* '(*3*))

--> ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) (cons 2 *'(3)*))

--> ((lambda (x_1) ((lambda (x) x) (cons 1 x_1))) *'(2 3)*)

--> ((lambda (x) x) (cons 1 *'(2 3)*))

--> ((lambda (x) x) *'(1 2 3)*)

-> (1 2 3)

Всегда осмысленно переписывать в cps?

- Предыдущий reverse-cps квадратичный. ☹
- Сначала опишем линейный reverse в обычном стиле:

```
(define (reverse2-simple lst)
  (let loop ((lst lst) (res null))
    (if (null? lst) res
      (loop (cdr lst) (cons (car lst) res)))))
```

- шаг 1) переименовываем и добавляем cc

```
(define (reverse2-cps lst cc)
  (let loop-cps ((lst lst) (res null) (cc cc))
    (if (null? lst) res
      (loop-cps (cdr lst) (cons (car lst) res)))))
```

Линейный reverse-cps

- шаг 2) применяем cc:

```
(define (reverse2-cps lst cc)
  (let loop-cps ((lst lst) (res null) (cc cc))
    (if (null? lst) (cc res)
      (cc (loop-cps (cdr lst) (cons (car lst) res))))))
```

- шаг 3) пишем lambda:

(lambda (x) (cc x)) можно упростить до cc

- шаг 4) подставляем cc в вызов loop-cps:

```
(loop-cps (cdr lst) (cons (car lst) res) cc)
```

Линейный reverse-cps

- итог:

```
(define (reverse2-cps lst cc)
  (let loop-cps ((lst lst) (res null) (cc cc))
    (if (null? lst) (cc res)
      (loop-cps (cdr lst) (cons (car lst) res) cc))))
```

- Всё верно! Остаточных вычислений внутри loop *не было!* Проще ☺:

```
(define (reverse2-cps lst cc)
 (let loop ((lst lst) (res null))
    (if (null? lst) (cc res)
      (loop (cdr lst) (cons (car lst) res)))))
```

Переписывать хвостовую рекурсию в стиле cps – *нонсенс!*

Линейный reverse-cps

- зато теперь можно

```
> (reverse2-cps '(1 2 3) (lambda (x) (append x '(2 3))))
-> (3 2 1 2 3)
```

но такая «овчинка» не стоит выделки!

Переписывать в cps имеет смысл тогда, когда код порождает рекурсивный процесс и не получается с ходу написать код, порождающий итеративный процесс. В учебных целях.

Flatten-cps – убирание внутренних скобок

- пример (flatten '((1) 2 ((3 4) 5) ((()))) (((6))) 7 8 ())) -> (1 2 3 4 5 6 7 8)
- пишем рекурсивное решение в обычном стиле:

```
(define (flatten x)
  (cond ((null? x) '())
        ((not (pair? x)) (list x))
        (else (append (flatten (car x)) (flatten (cdr x))))))
```

- видим, что написать без остаточных вычислений сложно
- действуем по методике:

1) переименовываем, добавляем cc

```
(define (flatten-cps x cc)
  (cond ((null? x) '())
        ((not (pair? x)) (list x))
        (else (append (flatten-cps (car x)) (flatten-cps (cdr x))))))
```

Flatten-cps. Продолжение

2) применяем ко всем веткам cond’а cc

```
(define (flatten-cps x cc)
 (cond ((null? x) (cc '()))
   ((not (pair? x)) (cc (list x)))
   (else (cc (append (flatten-cps (car x)) (flatten-cps (cdr x)))))))
```

3) выписываем lambda относительно 1го рек. вызова:

```
(lambda (y) (cc (append y (flatten-cps (cdr x)))))
```

4) выписываем lambda относительно 2го рек. вызова:

```
(lambda (z) (cc (append y z)))
```

5) переписываем верхнюю lambda (для 1го рек. вызова):

```
(lambda (y) (flatten-cps (cdr x) (lambda (z) (cc (append y z)))))
```

Flatten-cps. Окончание

6)готовы писать итоговый код

```
(define (flatten-cps x cc)
 (cond ((null? x) (cc '()))
   ((not (pair? x)) (cc (list x)))
   (else (flatten-cps (car x)
     (lambda (y) (flatten-cps (cdr x) (lambda (z) (cc (append y z)))))
    ))))
```

Обратим внимание, что мы явно задали порядок вычислений

в (append (flatten (car x)) (flatten (cdr x))))

сначала (flatten (car x)), потом (flatten (cdr x)) и append

могло быть (flatten (cdr x)), потом (flatten (car x)) и append

```
(flatten-cps (cdr x) (lambda (y) (flatten-cps (car x) (lambda (z)
   (cc (append z y))))) )
```

Flatten-cps. Окончание

- При другом порядке рек. вызовов другой код:

```
(define (flatten2-cps x cc)
 (cond ((null? x) (cc '()))
   ((not (pair? x)) (cc (list x)))
   (else (flatten2-cps (cdr x)
     (lambda (y) (flatten2-cps (car x) (lambda (z) (cc (append z y)))))
    ))))
```

- Важно, что flatten-cps – первая из рассмотренных нами функций, для которой написание cps-версии оправдано *прагматически*. Для остальных примеров CPS применён в учебных целях, т. к. написать линейное итеративное решение (без CPS) не составляет труда.

fringe-cps – обход дерева

- рекурсивное описание в обычном стиле:

```
(define (fringe-tree t)
  (cond ((tree-empty? t) '())
((and (tree-empty? (tree-left t))(tree-empty? (tree-right t)))(list (tree-data t)))
    (else (append (fringe-tree (tree-left t)) (fringe-tree (tree-right t))))))
```

видим, что написать без остаточных вычислений сложно

- действуем по методике:

1) переименовываем, добавляем cc

```
(define (fringe-cps t cc)
  (cond ((tree-empty? t) '())
((and (tree-empty? (tree-left t))(tree-empty? (tree-right t)))(list (tree-data t)))
    (else (append (fringe-cps (tree-left t)) (fringe-cps (tree-right t))))))
```

fringe-cps. Продолжение

2) применяем cc

```
(define (fringe-cps t cc)
 (cond ((tree-empty? t) (cc '()))
       ((and (tree-empty? (tree-left t))(tree-empty? (tree-right t)))
             (cc (list (tree-data t))))
  (else (cc (append (fringe-cps (tree-left t)) (fringe-cps (tree-right t)))))))
```

3) выписываем lambda относительно 1го рек. вызова:

```
(lambda (y) (cc (append y (fringe-cps (tree-right t)))))
```

4) выписываем lambda относительно 2го рек. вызова:

```
(lambda (z) (cc (append y z))))
```

5) переписываем верхнюю lambda (для 1го рек. вызова):

```
(lambda (y) (fringe-cps p (tree-right t) (lambda (z) (cc (append y z)))))
```

fringe-cps. Окончание

6)готовы писать итоговый код

```
(define (fringe-cps t cc)
  (cond ((tree-empty? t) (cc '())) ((and (tree-empty? (tree-left t))
                                         (tree-empty? (tree-right t))) (cc (list (tree-data t))))
  (else (fringe-cps (tree-left t) (lambda (y) (fringe-cps (tree-right t)
                                                          (lambda (z) (cc (append y z)))))))))
```

Обратим внимание, что мы явно задали порядок обхода дерева:
сначала влево, потом вправо.
могло быть наоборот:

```
(fringe-cps p (tree-right t) (lambda (z) (fringe-cps (tree-left t)
                                                     (lambda (y) (cc (append y z))))))
```

fringe-cps. Окончание

- При другом порядке рек. вызовов другой код:

```
(define (fringe-cps2 t cc)
  (cond ((tree-empty? t) (cc '())) ((and (tree-empty? (tree-left t))
                                         (tree-empty? (tree-right t))) (cc (list (tree-data t))))
  (else (fringe-cps2 (tree-right t) (lambda (z) (fringe-cps2 (tree-left t)
                                                             (lambda (y) (cc (append y z)))))))))
```

```
> (fringe-tree #(1 #(2 #() #()) #(3 #() #()))) -> (2 3)
> (fringe-cps #(1 #(2 #() #()) #(3 #() #())) (lambda(x) x)) -> (2 3)
> (fringe-cps2 #(1 #(2 #() #()) #(3 #() #())) (lambda(x) x)) -> (2 3)
```

Если присмотреться, то flatten – это тоже обход древовидной структуры.
То есть, CPS прагматически «годен» для обработки деревьев.

CPS – не единственное применение продолжений

- В Scheme есть call-with-current-continuation или call/cc, с помощью которой можно задействовать продолжение для управления вычислением.
- Вызов call/cc имеет вид: (call/cc f), где f – функция от одного аргумента, которую call/cc вызовет, передав ей в этот аргумент продолжение вокруг своего вызова.
- Вспомним (* (+ 5 1) (foo 1)) и перепишем с call/cc
  > (* (+ 5 1) (call/cc (lambda (cc) (foo-cps 1 cc))))
- То есть, call/cc составит продолжение
  (lambda (x) (* (+ 5 1) x))
  а затем вызовет свой аргумент с этим lambda в качестве аргумента
  > ((lambda (cc) (foo-cps 1 cc)) (lambda (x) (* (+ 5 1) x)))

Продолжение как нелокальный возврат

- Ранее мы рассматривали функции foldl, foldr и т. д., особенностью которых было то, что они обрабатывали весь список целиком, доходя до конца. Что если нужный результат может быть получен без полного просмотра списка? Неужели тогда свёртка не может быть применена?
- Пустим в дело call/cc

```
((define (1st-negative lst)
(call/cc
 (lambda (cc-exit)
  (foldl (lambda (x y) (if (negative? x) (cc-exit x) y)) #f lst))))
> (1st-negative '(0 1 2 3 4 5)) -> #f
> (1st-negative '(0 -1 2 -3 4 5)) -> -1
```

Продолжение как нелокальный возврат

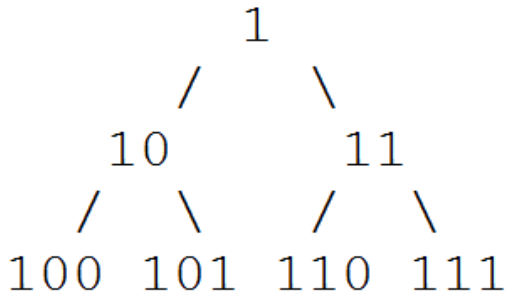
- Убедимся, что список обрабатывается не до конца

```
((define (1st-negative-p lst)
(call/cc
 (lambda (cc-exit)
  (foldl (lambda (x y) (begin (write x) ; добавим печать
                              (if (negative? x) (cc-exit x) y))) #f lst))))
> (1st-negative-p '(0 1 2 3 4 5)) -> 012345#f
> (1st-negative-p '(0 -1 2 -3 4 5)) -> 0-1-1
```

Нелокальные возвраты актуальны для деревьев

- При вычислении произведения чисел при вершинах дерева достаточно встретить первый 0, чтобы выдать 0 как результат.

```
(define (*-tree tree)
  (call/cc (lambda (cc-exit)
    (let helper ((tree tree))
      (cond ((empty-tree? tree) 1)
            ((= 0 (tree-data tree)) (cc-exit 0))
            (else (* (tree-data tree) (helper (tree-left tree))
                     (helper (tree-right tree)))))))))
```

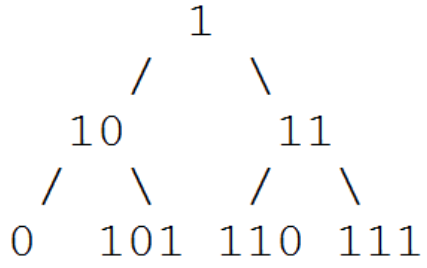



```
> (*-tree #(1 #(10 #(100 #() #()) #(101 #() #())) #(11 #(110 #() #()) #(111 #() #()))))) -> 13565310000
> (*-tree #(1 #(10 #(0 #() #()) #(101 #() #())) #(11 #(110 #() #()) #(111 #() #())))) -> 0
```

Нелокальные возвраты актуальны для деревьев

- Убедимся, что дерево обходится не до конца.

```
(define (*-tree-p tree)
  (call/cc (lambda (cc-exit)
    (let helper ((tree tree)) (begin (write '*) ; печать
      (cond ((empty-tree? tree) 1)
            ((= 0 (tree-data tree)) (cc-exit 0))
            (else (* (tree-data tree) (helper (tree-left tree))
                     (helper (tree-right tree))))))))))
```




```
> (*-tree-p #(1 #(10 #(100 #() #()) #(101 #() #())) #(11 #(110 #() #()) #(111 #() #()))))) -> **********13565310000
```

; 15 * выданы, так как в дереве 15 вершин, включая пустые поддеревья

```
> (*-tree-p #(1 #(10 #(0 #() #()) #(101 #() #())) #(11 #(110 #() #()) #(111 #() #()))))) -> ***0
```

Итоги лекции 5

- Преобразуя определение функции в стиле остаточных вычислений мы достигаем следующих целей:
  - Всю рекурсию делаем хвостовой.
  - Явно управляем порядком вычислений.
  - Тренируем мозг. ☺
- CPS демонстрирует возможности использования функций высших порядков.
- Накладные расходы (памяти), связанные с CPS, оставляем без внимания.
- Если в коде хвостовая рекурсия, то использовать cps бессмысленно.
- В примерах на список CPS неактуален, а для деревьев – *актуален*.
- Продолжения – «настоящая магия».