CSC324

lecture(#3)

Date: Sept, 21, 2016

Notes:

Defn: Linear Recursion

A recursive function is linear if it calls itself at most once at each level of recursion.

NonLinear Recursion (example):

Count -> Return # Symbols

```
(Count '(A B)) => 2
(Count '((A B) (C D))) => 4
(Count '(1 a 2 b)) => 2
(Count '(ab) b ))) => 2
(Count 1) => 0
(Count '()) => 0
```

```
(define (Count E)
        (Cond (null? E) 0)
        ((number? E) 0 )
        ( (Symbol? E) 1 )
        (else (+ (Count (Car E))
                 (Count (cdr E) )))))
```

Sample execution:

(Count '(1 a) b c)) => 3

$\underbrace{\text{(Count '(1 a) b c))}}_{=3}$

$\underbrace{\text{(Count '(1 a))}}_{=1}$ + $\underbrace{\text{(Count '(b c))}}_{=2}$

$\underbrace{\text{(Count 1)}}_{0}$ + $\underbrace{\text{(Count '(a))}}_{1}$

$\underbrace{\text{(Count a)}}_{=1}$ + $\underbrace{\text{(Count ())}}_{=0}$

$\underbrace{\text{(Count b)}}_{=1}$ + $\underbrace{\text{(Count '(c))}}_{=1}$

$\underbrace{\text{(Count c)}}_{=1}$ $\underbrace{\text{(Count ())}}_{=0}$

unwinding

~~Recursiode~~

Tail Recursion

recursion

Answer

Linear (non tail) recursion

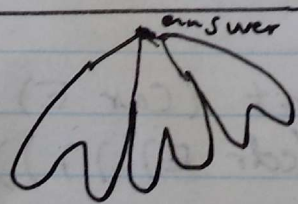
recursion

unwinding

Answer

non tail recursion

recursion



unwinding

Mutal Recursion

Function f_1 calls f_2, calls f_3, ...

....

calls f_k

calls f_1

$f_1, f_2, \ldots, f_k$

are mutualy recursive.

```
(even? L)
(odd? L)

(even? '(a b c d)) => #t

(even? '(a b c)) => #f

(odd? '(a b)) => #f

(odd? '(a b c)) => #t


(even? '()) => #t
(odd? '()) => #f
```

```
(define (even l)
        (if (null? l) #t
          (odd? (cdr l))) ))

(define (odd? l)
        (if (null? l) #f)
          (even? cdr l))))
```

Sample execution

```
(even? '(a b c d))

=> (odd? '(b c d))
=> (even? '(c d))
=> (odd? '(d))
=> (even? ())
=> #t
```

```
(rervese '(a b c d)) ⇒ (d c b a)

(reverse L)

(reverse L1 L2)

(reverse2 '(a b cd) '(1 2 3 4))
    ⇒ (d c b a 1 2 3 4)

(reverse L) ⇒ (reverse 2 L '())

(define reverse L)
    (reverse2 L '())

(reverse 2 '() L) → L

(reverse 2 '(a b c) '(1 2 3))
    ⇒ (reverse 2 '(c b) '(a 1 2 3))

(define (reverse 2 L1 L2)
    (if (null? L1) L2
        (reverse 2 (cdr L1) (cons (car L1) L2))))
```

Sample execution

```
(reverse (a b c d)) => (d c b a)

   => (reverse2 (a b c d) ())
   => (reverse2 (b c d) (a))
   => (reverse2 (c d) (b a))
   => (reverse2 (d) (c b a))
   => (reverse 2 ( ) (d c b a))
   => (d c b a)

(accumulated
      variable)
```

```
(3 Sum '(1 2 3 4 5)) => 3 + 4 + 5 = 12

(define (3sum L)
        (+ (first (reverse L))
           (second (revers L))
           (third (reverse L))))
```

```
(define (3sum L)
        (3sum_help (reverse L)))


(define (3sum-help RL )
        (+ (first RL )
           (second RL )
           (third RL )))
```

Let expressions:

```
(Let (var1 exp)
     (var2 exp2)
        :
   (varn expn))
   exp )
```

- First, create new variables $var_1, \ldots, var_n$
- Then evaluate $exp_1, \ldots, exp_n$
- then initalize each var_i to the value of exp_i

- evaluate exp and return its value.

```
(define (3 sum L)
        (let ((RL (reverse L))
            (+ (first RL))
            ( second RL)
            (third RL )))
```

$(x-y)^3 + (x+y)^3$

```
(define (dcube x y)
    (Let ((D (- x y))
          ( S (+ x y))
          (+ (+ D D D) (* S S S))))
```

<u>Scoping vales</u>

```
let ((x 1) (y 2))
    (+ y (let ((y 5) (z 3))
       2
         (+ x  y  z )))
            1  5  3
```

· Y takes the nearest enclosing value.

(Let ((x 3)) => 9

(* x

((let ((x 2)) (- x 1)) x))

refrentional transprancy

(Let ((F car) (G cdr)

(F (g '(a b c d))))

=> b

(car (cdr (a b c d)

'(b c d)

= b

(let ((sq (lamba (x) (* x x)))

(cube (lamda (x) (* x x x)))

(+ (sq 3) (cube 2))

=> $3^2 + 2^3 = 17$

Lexical Scoping

(let ((f (lamda (x) (* x 3))))

↑ (f 2)

6

2 * 3 = 6

(let ((z 3))

↑ (let ((f lamda (x) (* x z)))

6 ↑

6 (f 2)))

2x3 = 6

(let ((z 1)) (f 2)))

⇒ 2 X 3 = 6