Recursion and Iteration

September 18, 2018

Warm-up: Fizzbuzz

Count up from 0 to n in the following way:

- If the number is divisible by 3, print fizz
- If the number is divisible by 5, print buzz
- If the number is divisible by 3 **and** 5, print fizzbuzz
- Otherwise, print the number

Review: Lambda

Lambda: anonymous function

```
(lambda (x y) (+ x y))
```

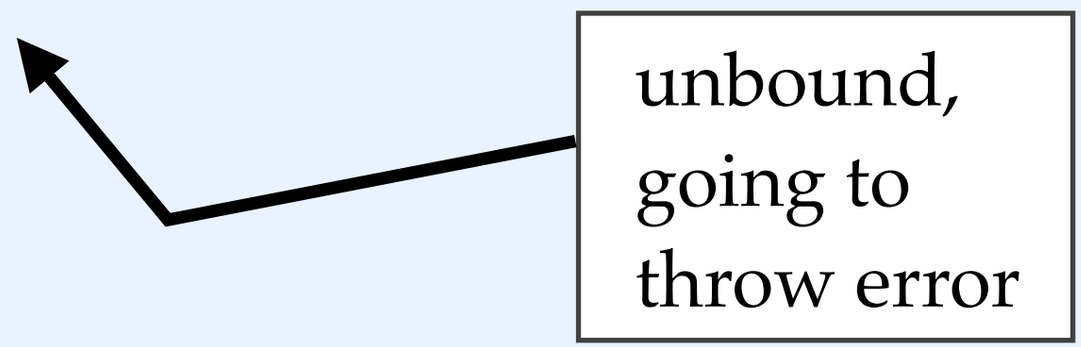
list of arguments **function body**

Practice: write an anonymous function that returns the second item in a list.

Review: Local Binding

Normal local binding: bindings are parallel (right-hand side is ignorant of left-hand side)

```
(let ((cat-speak   (printf “meow!”))
(dog-speak (printf “woof!”))
(unbound   (cat-speak)))
```



Review: Local Binding

Normal local binding: bindings are parallel (right-hand side is ignorant of left-hand side)

```
(let ((bound
       (lambda (x)
          (if (= x 0)
              (printf "zero!")
              (bound (- x 1))))))
```

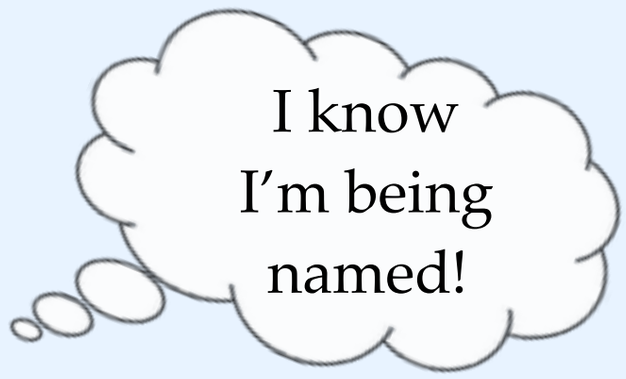
I don't know I'm being named!

uh oh! bound is undefined here, so we have no way to call the function in the recursive step!

Review: Local Binding

Recursive local binding:
(right-hand side knows that it's being named)

```
(letrec ((bound
          (lambda (x)
            (if (= x 0)
                (printf "zero!")
                (bound (- x 1))))))
```



bound by the left-hand side, can be called recursively

String-reverse using letrec

```
(define (reverse str)
  (letrec ((helper
            (lambda (str x)
              (if (= x (string-length str))
                  ""
                  (string-append
                    (helper str (+ x 1))
                    (string (string-ref str x)))))))
    (helper str 0)))
```

define helper function

helper function arguments

base case

recursive call

call helper function

Exercise

Rewrite count-up using letrec

```
(define (count-help x y)
        (printf (number->string x))
    (if (= x y)
        (void)
        (count-help (+ x 1) y)))

(define (count-up x)
    (count-help 1 x))
```

Recursion versus Iteration

How efficient is recursion anyway?

Recursion versus Iteration

How efficient is recursion anyway?

Iterative

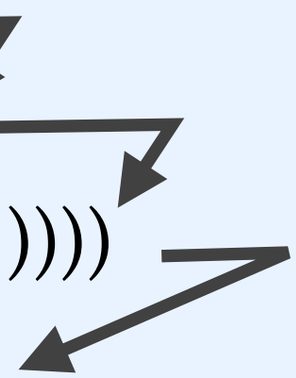
> (it-fac 4)

res = res*1
res = res*2
res = res*3
res = res*4

Recursive

> (fac 4)

(* 4 (fac 3))
(* 4 (* 3 (fac 2)))
(* 4 (* 3 (* 2 (fac 1))))
(* 4 (* 3 (* 2 1)))



Tail-recursion

In the tail-recursive version, the multiplication happens inside of the recursive call, not outside of it.

```
(define (tail-fac n)
  (letrec ((helper
            (lambda (n acc)
              (if (= 1 n)
                  acc
                  (helper (- n 1)
                          (* n acc))))))
    (helper n 1)))
```

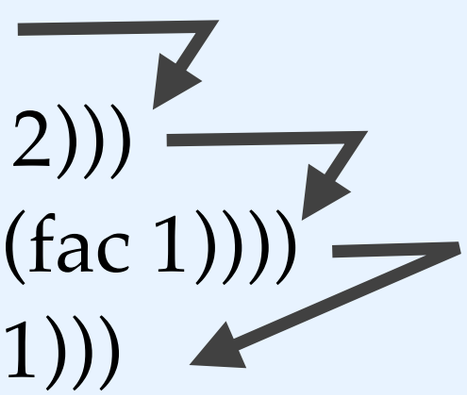
Tail-recursion

How efficient is recursion anyway?

Original version

> (fac 4)

(* 4 (fac 3))
(* 4 (* 3 (fac 2)))
(* 4 (* 3 (* 2 (fac 1))))
(* 4 (* 3 (* 2 1)))



Tail-recursive version

> (tail-fac 4)

(tail-fac 3 (* 4 1))
(tail-fac 2 (* 3 4))
(tail-fac 1 (* 2 12))
(24)

Exercise

Rewrite string-reverse to be tail-recursive