Class 16

anonymous procedures, recurrence proofs, trees
Anonymous procedures
Lambda (version 1)

(lambda name-list body)

when evaluated, produces a user-defined procedure (i.e., a closure), with the name-list as the argument list, and body as the body.

(define (f x) (+ x 1))
(define f (lambda (x) (+ x 1))

do exactly the same thing!
Rule of evaluation

A lambda-expression has the form
(lambda (n1 ... nk) exp)
where each of n1, n2, ... nk is a name, and exp is an expression.

(version 1) To evaluate this expression, we create a closure
with (possibly empty) formal argument list n1 ... nk, and
body exp. This closure is the value of the lambda-expression.
Lambda practice

Evaluate:
(((lambda (x) (+ x 1)) 4)

Solution:
1. The first expression evaluates to a closure [ x | (+ x 1)], so it's a proc-app expression.
2. Evaluate the second, 4, to get the number value 4.
3. Apply the proc to the args:
   1. Bind formal arg x to actual arg 4
   2. Evaluate body (+ x 1) to get 5, which is the value of the proc-app-expr
   3. Forget the bindings from step 1.
Lambda (v. 2)

- I lied about closures.
- A closure right now consists of an argument list and a body.
- The truth...
- A closure consists of an arg list, a body, and a set of bindings called the "local environment"
- During evaluation of a proc-app-expression where the procedure is a user-defined procedure
  - We look up names in $T + E$, where $T$ is the top level environment, and $E$ is the local environment of the closure
  - We look for bindings in $E$ first, "shadowing" prior bindings in $T$
Creating a closure

• A closure consists of an arg list, a body, and a set of bindings called the "local environment"

• Where does the "local environment" come from?
  • All bindings in the current environment except those in the TLE

• What's the current environment?
  • It's the TLE plus any bindings resulting from other procedure applications, etc.

• What does "etc." mean?
  • "let" expressions...
  • which are just syntactic sugar for lambdas
Final piece of Racket syntax: let-expressions

(let ((x 1)
     (y 2))
  (+ x y))

- Temporarily extend current environment by bindings like x -> 1 and y -> 2
- Evaluate the body
- Forget those extended bindings

- Note: if the body is a lambda-expression, those extended bindings become part of the "local environment" of the resulting closure
"let" is syntactic sugar

(let ((x 5))
  (+ x 3))

is exactly the same as

((lambda (x) (+ x 3)) 5)

- Multi-binding let expressions correspond to multi-argument lambda expressions.
Using "let"

• "numbers" from last week: (numbers (list true false true)) => (ilst 3 2 1)
• Second strategy: take "first" of recursive result, increment it, and cons it onto recursive result.
• For length-n list, took time approximately $2^n$!
• Managed to speed this up with a helper.
Revised version

(define (numbers alod)
  (cond
    [(empty? alod) empty]
    [(empty? (rest alod)) (list 1)]
    [(cons? alod) (let ((rr (numbers (rest alod))))
      (cons (+ 1 (first rr) rr)))]))
Another application

• Perhaps in bignum, you're adding together some bigits, and need to compute both a quotient and remainder of this sum.

• approach 1: compute the sum, compute quotient; recompute sum, compute remainder.

• approach 2: (let ((sum ...))
  
  ...(quotient sum ...) ...

• Please *don't* go rewrite your bignum to do this if it already works well!
Lambda's power

- We can now write a procedure that produces...procedures!

```
(define (raise-maker amt) (lambda (x) (+ x amt)))
;raise-maker: int -> (int -> int)
(raise-maker 3) => ???
```

```
(define ta-salaries (list 10 3 5 9 2))

(map (raise-maker 3) ta-salaries) => (13 6 8 12 5)
```
Another HOP: filter!
Filtering

;; Filter
;; filter: ('a -> bool) * ('a list) -> ('a list)
;; input: a predicate p operating on type 'a
;; a list, alod, of items of type 'a
;; output: all items of alod, in order, for which
;; p produces a true result
;; example: (filter odd? (list 1 2 3)) => (list 1 3)
;;
define (filter p alod)
  (cond
   [(empty? alod] empty)
   [(cons? alod) (if (p (first alod))
     (cons (first alod) (filter p (rest alod)))
     (filter p (rest alod))))]}
• You'll see lots more filtering on this week's homework.
Well-ordering proofs of plug-n-chug conjectures
Well-ordering principle
Well-ordering principle

"A nonempty set of nonnegative integers has a smallest element."

Do you believe this? Try a few examples.

Find counterexamples to each of these statements:

A set of nonnegative integers has a least element
A nonempty set of integers has a least element
A nonempty set of nonnegative numbers has a least element
"A nonempty set of natural numbers has a smallest element."
Called the *Well-ordering principle*.
An axiom of mathematics
  - Equivalent to something called the "axiom of induction"
  - If you've encountered proofs by induction, well-ordering proofs are just a different flavor of those
  - If not...a little less confusion awaits you.
Proof by contradiction

• We assume some statement is true. (The "reason", in a 2-column proof, is "contradiction hypothesis")
• We reason from this to arrive at some false statement.
• We conclude that the hypothesis is wrong, and that the statement is therefore false.
• Agatha Christie example: "If Joey was on that 5:32 commuter train, then *someone* from Tullybridge would have seen him, and we've interviewed hundreds of people, and no one saw him on that train. So he must not have been on it."
• "contradiction hypotheses": *Joey was on the train*
• Reasoning: *If that was true, someone would have seen him. No one did.*
• Conclusion: "*Joey was on the train*" must be false!
Our proofs by contradiction will always have the same structure!
Proving that a plug-n-chug conjecture is correct.

This is the contradiction hypothesis.
Remember introduction to "map"?

• Every procedure looked almost the same as the others, right?
• Same thing happens with these proofs
Parts that change are in red
Parts that might change are
Let's try it again
Let's try it again