Programming Languages

CS 251
Fall 2021

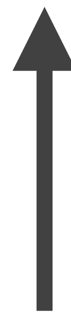
Carolyn Anderson

Recap

Anonymous Functions

A lambda expression is an anonymous function.
(define (fn)) is really short for (define fn (lambda))

(define (hello-world) (display “hello world!”))
(define hello-world (lambda ( ) (display “hello world!)))

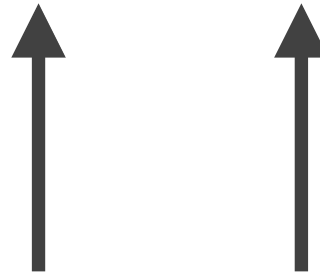


parameters **function body**

Lambdas

Lambda: anonymous function

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



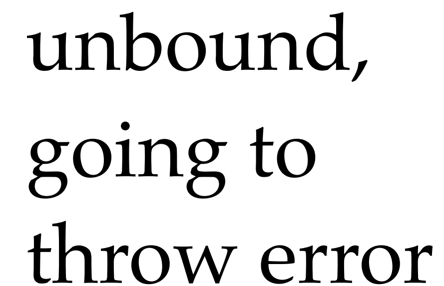
list of arguments **function body**

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)

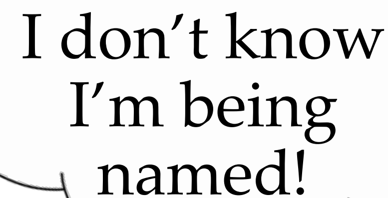
```
(let ((cat-speak   (printf “meow!”))
      (dog-speak (printf “woof!”))
      (unbound   (cat-speak)))
```



Review: Local Binding

Normal local binding: bindings are parallel

(right-hand side is ignorant of left-hand side)

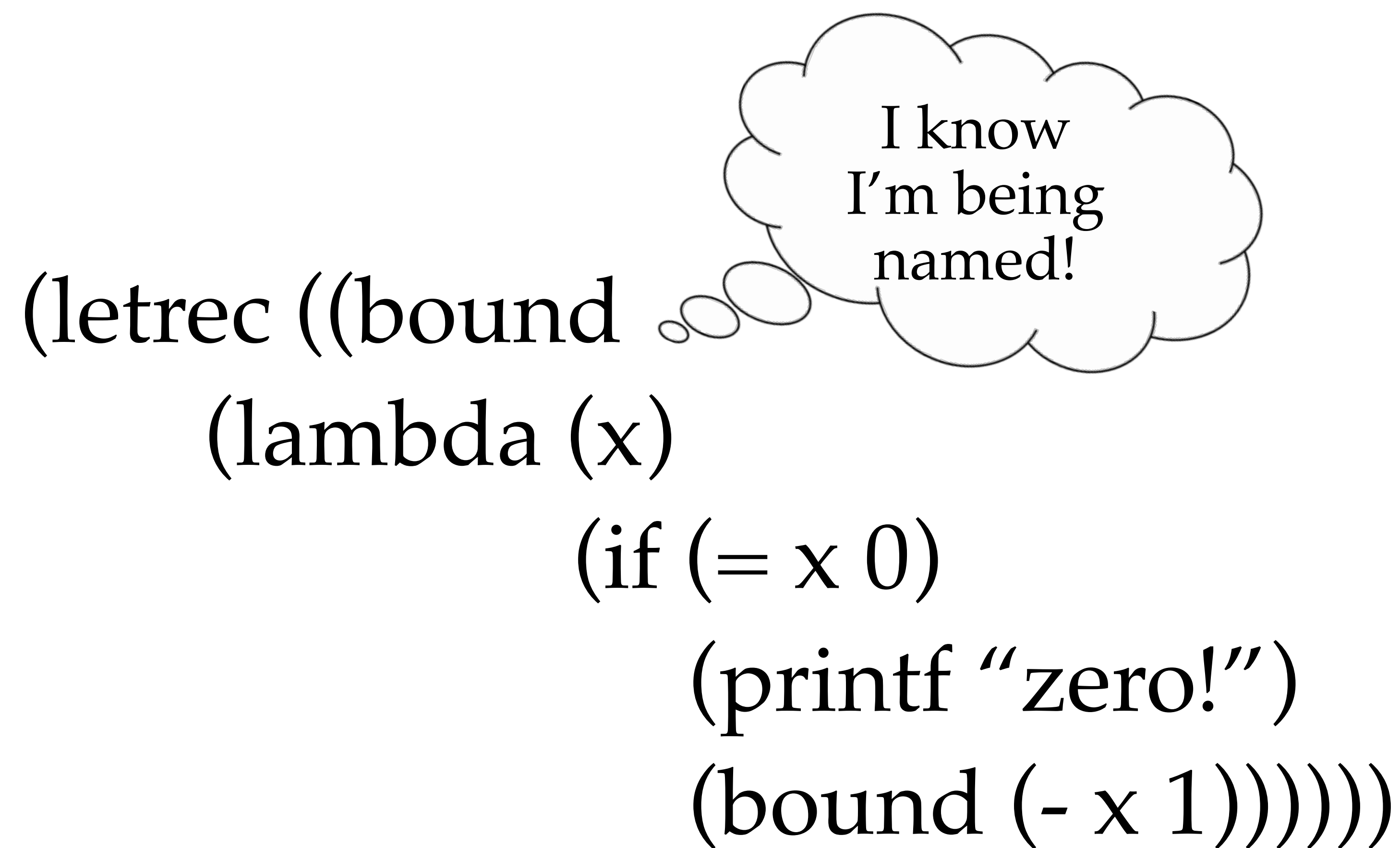


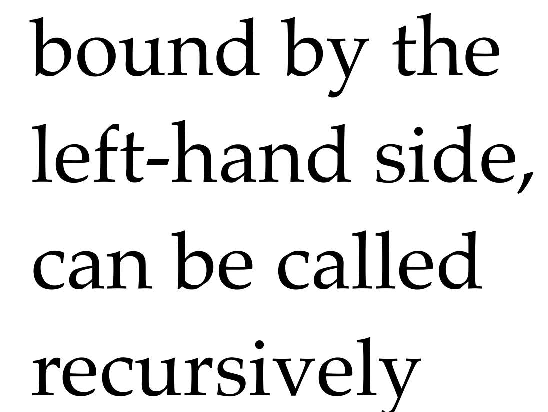

```
(let ((bound
      (lambda (x)
              (if (= x 0)
                  (printf "zero!")
                  (bound (- x 1))))))
```

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

Review: Local Binding

Letrec: names given on the left are known on the right





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

Practice:

Rewrite string-reverse to be tail-recursive.

Formal Semantics

Formal Semantics

- Gives a way of representing the semantics of the language
- Helps us reason mathematically about program behavior
- Basic idea: write rules for how expressions get evaluated
- Two main styles:
  - **Big step semantics**: models the evaluation of each expression
  - **Small step semantics**: models the **step-by-step** evaluation of each expression

Language components

- **Expressions**: bits of the language

(+ 1 2) "cat" (define (foo n) n)

- **Values**: expressions that cannot be reduced any further

"cat" (define (foo n) n)

- **Declarations**: bind variables to values

(let

Big step semantics: values

- Syntax:
  - Numbers: 0, 1, …
  - Strings: “donut”, “cookie”, …
  - Functions: (define (foo x) x), …
  - Booleans: #t, #f
- Evaluation:
  - Values evaluate to themselves.

Big step semantics: addition

- Syntax: (+ e1 e2)
  - e1 and e2 stand for arbitrary expressions
- Evaluation:
  1. Evaluate e1 to a value v1.
  2. Evaluate e2 to a value v2.
  3. Return the arithmetic sum of v1 and v2.

Big step semantics: addition

- Syntax: (+ e1 e2)
  - e1 and e2 stand for arbitrary expressions
- Evaluation:
  1. Evaluate e1 to a value v1.
  2. Evaluate e2 to a value v2.
  3. Return the arithmetic sum of v1 and v2.

Let's test it out!

Big step semantics: addition

- Syntax: (+ e1 e2)
  - e1 and e2 stand for arbitrary expressions
- Evaluation:

1. Evaluate e1 to a value v1.
2. Evaluate e2 to a value v2.
3. **If v1 and v2 are numbers**, return the arithmetic sum of v1 and v2.
4. Otherwise, a dynamic type-checking error occurs.

Practice:

Write the big step semantics for string concatenation.

Big step semantics: string concatenation

- Syntax: (string-append e1 e2)
- Evaluation:

1. Evaluate e1 to a value v1.
2. Evaluate e2 to a value v2.
3. **If v1 and v2 are strings**, return the concatenation of v1 and v2.
4. Otherwise, a dynamic type-checking error occurs.

Let's test it out!

Approach 1: change the big step semantics

- Syntax: (string-append e1 … e_n)
- Evaluation:

1. For all e from e1 to e_n, evaluate e to a value v.
2. **If v1 … v_n are strings**, return the concatenation of v1 … v_n.
3. Otherwise, a dynamic type-checking error occurs.

Approach 2: treat it as "syntactic sugar"

- Syntax: (string-append e1 e2 e3)
- Evaluation:

  (string-append e1 e2 e3) evaluates to

  (string-append (string-append e1 e2) e3).

Big step semantics: notation

- The ↓ symbol is used to express evaluation assertions:

  e ↓ v is pronounced "e evaluates to v".

- Translating our rules:

  Values: v ↓ v

  Addition: If:

  - e1 ↓ v1,
  - e2 ↓ v2, and
  - v1 and v2 are numbers that sum to v,

  Then (+ e1 e2) ↓ v

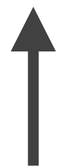
Big step semantics: notation

We can use this notation to write our semantics even more concisely as a "proof tree.":

Addition:

$$\frac{e1 \downarrow v1 \quad e2 \downarrow v2}{(+\ e1\ e2) \downarrow v}$$

where v1 and v2 are numbers and v is the sum of v1 and v2



This part is important!

Practice:

Model the derivation of (+ 5 (+ 10 9)) in the proof tree format.

First class functions

In Racket, functions are values. This is because Racket has **first class functions**: functions have all the rights and privileges of other values.

Function Bill of Rights:

We the Racketeers hereby declare that functions:

- Do not need to be named (lambdas)
- Can be returned by functions
- Can be arguments to functions

Functions as values

Do we need to do anything special for functions?

No! Like other values, functions can be evaluated any further… *until they are applied*.

Syntax: (**lambda** (id_1, … id_n) e)

parameters **function body**

Semantics: v ↓ v

Functions as values

Since functions are values, they need to be handled by our big step semantics for values.
Do we need to do anything special for functions?

Syntax: (**lambda** (id_1, … id_n) e)

parameters **function body**

Semantics: ????