Programming Languages

CS 251

Fall 2021

Carolyn Anderson

Recap

Language components

- **Values**: expressions that cannot be reduced any further
- **Expressions**: bits of the language
- **Declarations**: bind variables to values

Our big step semantics for Racket

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

  Value rule: $v \downarrow v$

- **Expressions**: bits of the language

  Addition rule:

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

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

Big step semantics: practice

Let's write the big step semantics for **and**

And:

$$\frac{????}{(\text{and } e1\ e2) \downarrow v}$$

Big step semantics: practice

Let's write the big step semantics for **and**

And:

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

If v1 and v2 are Booleans, either v = v1 = v2 = #t or v is #f. Otherwise, a dynamic error is produced.

Big step semantics: practice

Let's write the big step semantics for **and:**

And-True:

$$\frac{e1 \downarrow \#t \qquad e2 \downarrow \#t}{(\text{and } e1\ e2) \downarrow \#t}$$

Otherwise, a dynamic error is produced.

And-False:

$$\frac{e1 \downarrow v1 \qquad e2 \downarrow v2}{(\text{and } e1\ e2) \downarrow \#f}$$

If v1 or v2 is #f and both are Booleans, (and e1 e2) evaluates to #f. Otherwise, a dynamic error is produced.

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*.

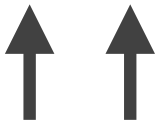
Syntax: (**lambda** (id_1, … id_n) e)

parameters **function body**

Semantics: v ↓ v

Function application

Syntax: (e_1 e_2)



function **argument**

Semantics: ????

What happens when a function is **applied**?

Function application

When a function is applied to a value, that value gets **bound** to the function's **parameter** inside the **scope** of the function.

(**define** (id e) e)
(id 5)

Function application

When a function is applied to a value, that value gets **bound** to the function's **parameter** inside the **scope** of the function.

But what does that mean?

(**define** (id e) e)
(id 5)

Variables and binding

What are variables?

What is variable binding?

Variables and binding

What are variables?

- Variables **store values**

Variable binding

What is variable binding?

- Variable binding links the value to all occurrences of the variable within the **binder's scope**

Variable binding

What is variable binding?

- Variable binding links the value to all occurrences of the variable within the **binder's scope**

Ok, but what does it mean to “link” the value and variable?

Variable binding

What is variable binding?

- Variable binding links the value to all occurrences of the variable within the **binder's scope**

Ok, but what does it mean to "link" the value and variable?

- There are different ways of thinking about this! For now, we will use:

The Substitution Model of Variable Binding: when a value v is bound to an expression e, substitute the value v for every occurrence of e in the scope of the binder.

Binding as substitution

The Substitution Model of Variable Binding: when a value ***v*** is bound to an expression ***e***, substitute the value ***v*** for every occurrence of ***e*** in the scope of the binder.

(id 5) = ((lambda (e) e) 5) = 5

Scope

What is the scope of a variable binding?
That depends on the binding construct!

Discovering variable scope

What is the scope of a variable binding?
That depends on the binding construct!

We can figure out the scope of a variable through experimentation.

Starting premise:

- Referencing an **unbound** variable will throw an error
- Referencing a **bound variable** will return the value it is bound to

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

Tracking our discoveries

Goal: figure out the binding scope of **let**

We should keep track of what we are learning about Racket's behavior!

Context	Variable value
before **let**	
within **let**	
after **let**	

Tracking our discoveries

Goal: figure out the binding scope of **let**

We should keep track of what we are learning about Racket's behavior!

Context	Variable value
before **let**	undefined
within **let**	
after **let**	

Tracking our discoveries

Goal: figure out the binding scope of **let**

We should keep track of what we are learning about Racket's behavior!

Context	Variable value
before **let**	undefined
within **let**	5
after **let**	

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

Hypothesis 1: let binds all occurrences of the variable in its body.

Context	Variable value
before **let**	undefined
within **let**	5
after **let**	undefined

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

Hypothesis 1: let binds all occurrences of the variable in its body.

What about nested **let**?

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

What about nested **let**?

Context	Variable value
before first **let**	
within first **let**	
within second **let**	
after both **lets**	

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

What about nested **let**?

Context	Variable value
before first **let**	undefined
within first **let**	
within second **let**	
after both **lets**	

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

What about nested **let**?

Context	Variable value
before first **let**	undefined
within first **let**	"Outer"
within second **let**	
after both **lets**	

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

What about nested **let**?

Context	Variable value
before first **let**	undefined
within first **let**	"Outer"
within second **let**	"Inner"
after both **lets**	

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

What about nested **let**?

Context	Variable value
before first **let**	undefined
within first **let**	"Outer"
within second **let**	"Inner"
after both **lets**	undefined

Variable shadowing

We've run into a case of **variable shadowing**: there is another binder for **x** nested within the first binder.

If we want to be able to refer to the first value, we need to use two different variable names.

```
(let (x 7)
     (let (x 2)
          (+ 0 x)
     )

     x
)
```

Discovering Scope: **let**

Goal: figure out the binding scope of **let**

Hypothesis 2: let binds all **unbound** occurrences of the variable in its body.

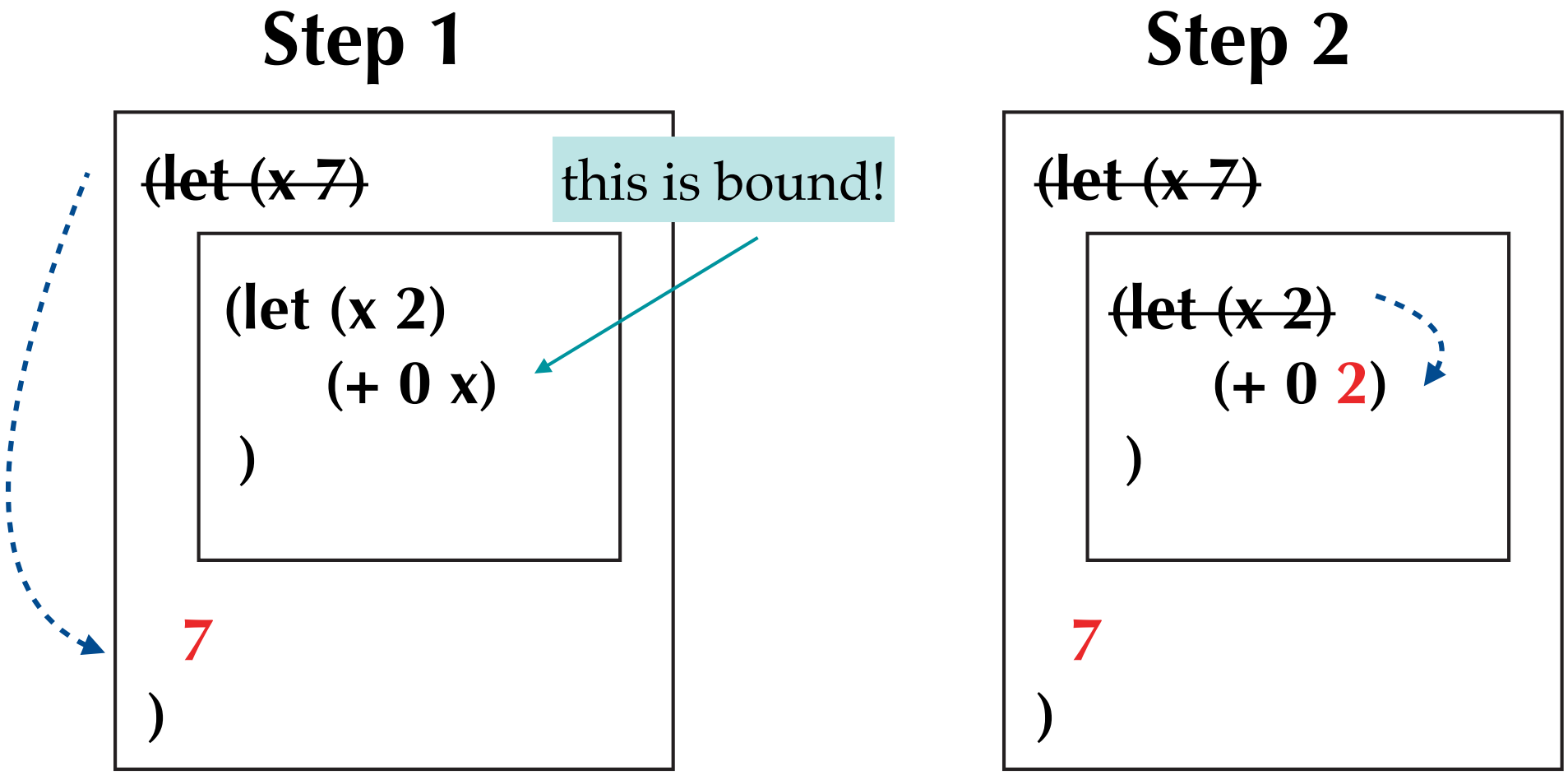
Context	Variable value
before first **let**	undefined
within first **let**	"Outer"
within second **let**	"Inner"
after both **lets**	undefined

Binding as substitution

The Substitution Model of Variable Binding:
When a value v is bound to an expression e, substitute the value v for every **unbound** occurrence of e in the scope of the binder.

Understanding variable shadowing

Variable shadowing may look like it mutates the variable. But doesn't. If we apply the substitution model of variable binding, we see:



Variable binding and scope

What is variable binding?

- Variable binding links the value to all **unbound** occurrences of the variable within the **binder's scope**

What is the scope of a variable binding?

- That depends on the binding construct!

The binding scope of a **let** is its body.

Practice:

Use the same technique to figure out the binding scope of **function application**.

Discovering Scope: **Application**

Goal: figure out the binding scope of **function application**

Context	Variable value
before function application	undefined
within function body	"Outer"
outside of function call	undefined

Variable binding and scope

What is variable binding?

- Variable binding links the value to all **unbound** occurrences of the variable within the **binder's scope**

What is the scope of a variable binding?

- That depends on the binding construct!

The binding scope of a **let** is its body.
The binding scope of a **function** is its body.

What is the binding scope of **define**?

Practice:

Use the same technique to figure out the binding scope of **define**.

Discovering Scope: **Application**

Goal: figure out the binding scope of **define**

Context	Variable value
before define	
within function body	
within function body when name is shadowed	
after define	

Discovering Scope: **Application**

Goal: figure out the binding scope of **define**

Context	Variable value
before define	undefined
within function body	
within function body when name is shadowed	
after define	

Discovering Scope: **Application**

Goal: figure out the binding scope of **define**

Context	Variable value
before define	undefined
within function body	"donut"
within function body when name is shadowed	
after define	

Discovering Scope: **Application**

Goal: figure out the binding scope of **define**

Context	Variable value
before define	undefined
within function body	"donut"
within function body when name is shadowed	"mocha"
after define	

Discovering Scope: **Application**

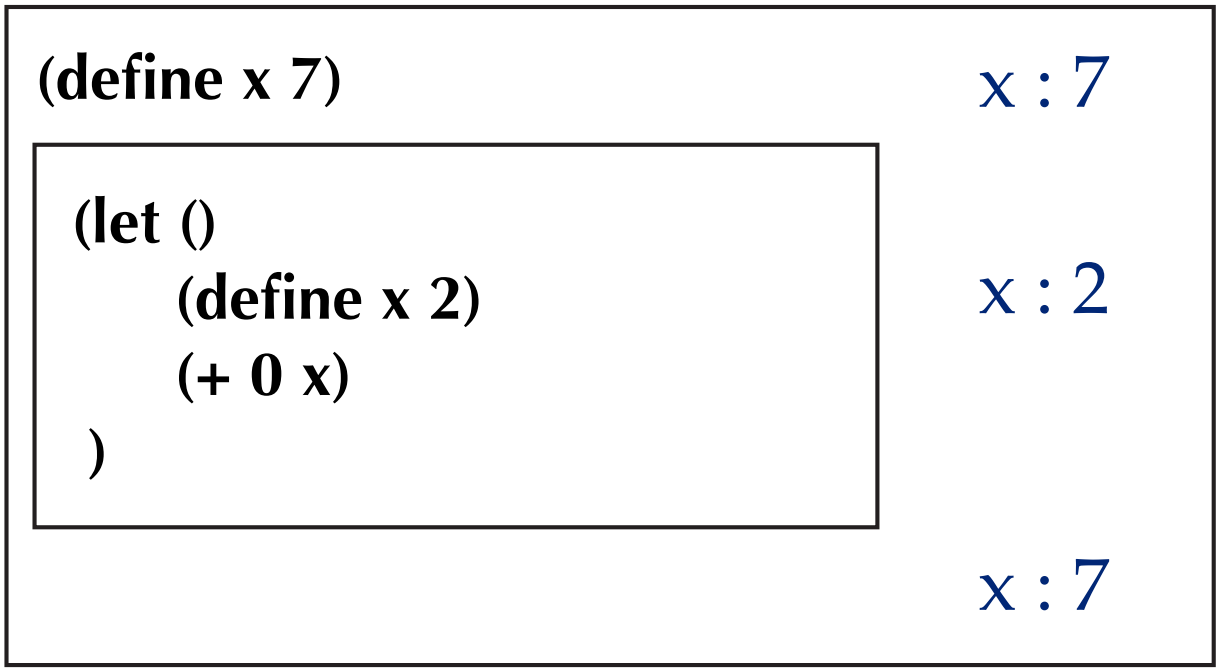
Goal: figure out the binding scope of **define**

Context	Variable value
before define	undefined
within function body	"donut"
within function body when name is shadowed	"mocha"
after define	"donut"

Define

What is the binding scope of **define**?

Similar to **let**, but there's no body. (**define** e x) scopes over all subsequent unbound occurrences of e within the current scope (the scope it is called in).



Define

What is the binding scope of **define**?

Similar to **let**, but there's no body. (**define** e x) scopes over all subsequent unbound occurrences of e within the current scope.

define can be used within functions and local binding constructs like **let**. But this is considered poor style.

What have we discovered?

- Checked our big step semantics for functions
- Tried to think about the semantics of function application
- Explored scope and variable binding
- Learned the substitution model of variable binding

Next class: big step semantics for function application and variable binding constructs!