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: \( e_1 \downarrow v_1 \)
  
  \( e_2 \downarrow v_2 \)
  
  \( (+ e_1 e_2) \downarrow v \)
  
  where \( v_1 \) and \( v_2 \) are numbers and \( v \) is the sum of \( v_1 \) and \( v_2 \)
Big step semantics: practice

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

**And:**

```
????
```

```
(and e1 e2) \downarrow v
```
Let's write the big step semantics for **and**

**And:**

\[
\begin{align*}
e_1 \downarrow v_1 \\
e_2 \downarrow v_2 \\
\hline
(\text{and } e_1 \text{ e}_2) \downarrow v
\end{align*}
\]

If \(v_1\) and \(v_2\) are Booleans, either \(v = v_1 = v_2 = \texttt{#t}\) or \(v\) is \(\texttt{#f}\). Otherwise, a dynamic error is produced.
Let's write the big step semantics for `and`:

**And-True:**

\[
\begin{align*}
e_1 & \downarrow \#t \\
e_2 & \downarrow \#t \\
\hline
(\text{and } e_1 e_2) & \downarrow \#t
\end{align*}
\]

Otherwise, a dynamic error is produced.

**And-False:**

\[
\begin{align*}
e_1 & \downarrow v_1 \\
e_2 & \downarrow v_2 \\
\hline
(\text{and } e_1 e_2) & \downarrow \#f
\end{align*}
\]

If `v_1` or `v_2` is `\#f` and both are Booleans, `(\text{and } e_1 e_2)` 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: \((\text{lambda} \ (id_1, \ldots \ id_n) \ e)\)

Semantics: \(v \downarrow v\)
Function application

Syntax: \((e_1 \; e_2)\)

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.

\[
\text{(define (id e) e)}
\]
\[
\text{(id 5)}
\]
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?

\[
\text{(define (id e) e)} \\
\text{(id 5)}
\]
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.
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.

$$(\text{id} \ 5) = ((\lambda (e) \ e) \ 5) = 5$$
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!

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before <code>let</code></td>
<td></td>
</tr>
<tr>
<td>within <code>let</code></td>
<td></td>
</tr>
<tr>
<td>after <code>let</code></td>
<td></td>
</tr>
</tbody>
</table>
# Tracking our discoveries

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

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within <code>let</code></td>
<td></td>
</tr>
<tr>
<td>after <code>let</code></td>
<td></td>
</tr>
</tbody>
</table>
Tracking our discoveries

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

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within <code>let</code></td>
<td>5</td>
</tr>
<tr>
<td>after <code>let</code></td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: `let`

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

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within <code>let</code></td>
<td>5</td>
</tr>
<tr>
<td>after <code>let</code></td>
<td>undefined</td>
</tr>
</tbody>
</table>
Discovering Scope: \texttt{let}

\textbf{Goal:} figure out the binding scope of \texttt{let}

\textbf{Hypothesis 1:} \texttt{let} binds all occurrences of the variable in its body.

What about nested \texttt{let}?
Discovering Scope: `let`

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

What about nested `let`?

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first <code>let</code></td>
<td></td>
</tr>
<tr>
<td>within first <code>let</code></td>
<td></td>
</tr>
<tr>
<td>within second <code>let</code></td>
<td></td>
</tr>
<tr>
<td>after both <code>lets</code></td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: `let`

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

What about nested `let`?

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first <code>let</code></td>
<td><code>undefined</code></td>
</tr>
<tr>
<td>within first <code>let</code></td>
<td></td>
</tr>
<tr>
<td>within second <code>let</code></td>
<td></td>
</tr>
<tr>
<td>after both <code>lets</code></td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: `let`

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

What about nested `let`?

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within first <code>let</code></td>
<td>&quot;Outer&quot;</td>
</tr>
<tr>
<td>within second <code>let</code></td>
<td></td>
</tr>
<tr>
<td>after both <code>lets</code></td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: `let`

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

What about nested `let`?

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within first <code>let</code></td>
<td>&quot;Outer&quot;</td>
</tr>
<tr>
<td>within second <code>let</code></td>
<td>&quot;Inner&quot;</td>
</tr>
<tr>
<td>after both <code>lets</code></td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: `let`

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

**What about nested `let`?**

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first <code>let</code></td>
<td>undefined</td>
</tr>
<tr>
<td>within first <code>let</code></td>
<td>&quot;Outer&quot;</td>
</tr>
<tr>
<td>within second <code>let</code></td>
<td>&quot;Inner&quot;</td>
</tr>
<tr>
<td>after both <code>lets</code></td>
<td>undefined</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before first let</td>
<td>undefined</td>
</tr>
<tr>
<td>within first let</td>
<td>&quot;Outer&quot;</td>
</tr>
<tr>
<td>within second let</td>
<td>&quot;Inner&quot;</td>
</tr>
<tr>
<td>after both lets</td>
<td>undefined</td>
</tr>
</tbody>
</table>
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:

Step 1

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

Step 2

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

this is bound!
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**

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before function application</td>
<td>undefined</td>
</tr>
<tr>
<td>within function body</td>
<td>&quot;Outer&quot;</td>
</tr>
<tr>
<td>outside of function call</td>
<td>undefined</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before define</td>
<td></td>
</tr>
<tr>
<td>within function body</td>
<td></td>
</tr>
<tr>
<td>within function body when name is shadowed</td>
<td></td>
</tr>
<tr>
<td>after define</td>
<td></td>
</tr>
</tbody>
</table>
### Discovering Scope: Application

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before define</td>
<td>undefined</td>
</tr>
<tr>
<td>within function body</td>
<td></td>
</tr>
<tr>
<td>within function body when name is shadowed</td>
<td></td>
</tr>
<tr>
<td>after define</td>
<td></td>
</tr>
</tbody>
</table>
**Discovering Scope: Application**

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before define</td>
<td>undefined</td>
</tr>
<tr>
<td>within function body</td>
<td>&quot;donut&quot;</td>
</tr>
<tr>
<td>within function body when name is shadowed</td>
<td></td>
</tr>
<tr>
<td>after define</td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: Application

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before define</td>
<td>undefined</td>
</tr>
<tr>
<td>within function body</td>
<td>&quot;donut&quot;</td>
</tr>
<tr>
<td>within function body when name is shadowed</td>
<td>&quot;mocha&quot;</td>
</tr>
<tr>
<td>after define</td>
<td></td>
</tr>
</tbody>
</table>
Discovering Scope: Application

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before define</td>
<td>undefined</td>
</tr>
<tr>
<td>within function body</td>
<td>&quot;donut&quot;</td>
</tr>
<tr>
<td>within function body when name is shadowed</td>
<td>&quot;mocha&quot;</td>
</tr>
<tr>
<td>after define</td>
<td>&quot;donut&quot;</td>
</tr>
</tbody>
</table>
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 x 7)  x : 7

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

```
  x : 2
  x : 7
```
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!