

CS450

Structure of Higher Level Languages

Lecture 16: Challenges specifying Racket's define

Tiago Cogumbreiro

Press arrow keys   to change slides.

How do we add support for definitions?

- We extend the our language (λ_E) with `define`
- We introduce the AST
- We discuss parsing our language

Understanding definitions

Syntax

$$t ::= e \mid t; t \mid (\mathbf{define} \ x \ e)$$
$$e ::= v \mid x \mid (e_1 \ e_2) \mid \lambda x. t \quad v ::= n \mid \{E, \lambda x. t\} \mid \mathbf{void}$$

- New grammar rule: **terms**
- A program is now a non-empty sequence of terms
- Since we are describing the **abstract** syntax, there is no distinction between a basic and a function definition
- Since evaluating a definition returns a void, we need to update values

Values

■ We add `void` to values.

$$v ::= n \mid \{E, \lambda x.t\} \mid \text{void}$$

Racket implementation

```
;; Values  
(define (f:value? v) (or (f:number? v) (f:closure? v) (f:void? v)))  
(struct f:number (value))  
(struct f:closure (env decl))  
(struct f:void ())
```

Expressions

Expressions remain unchanged.

$$e ::= v \mid x \mid (e_1 e_2) \mid \lambda x.t$$

Racket implementation

```
(define (f:expression? e) (or (f:value? e) (f:variable? e) (f:apply? e) (f:lambda? e)))  
(struct f:variable (name))  
(struct f:apply (func args))  
(struct f:lambda (params body))
```

Terms

■ We implement terms below.

$$t ::= e \mid t; t \mid (\mathbf{define} \ x \ e)$$

Racket implementation

```
(define (f:term? t) (or (f:expression? t) (f:seq? t) (f:define? t)))  
(struct f:seq (fst snd))  
(struct f:define (var body))
```

The body of a function declaration is a single term

The body is no longer a list of terms!

■ A sequence is not present in the concrete syntax, but it simplifies the implementation and formalism (see reduction)

λ_F semantics

The incorrect way of implementing `define`

λ_F semantics

The incorrect way of implementing

Semantics $t \Downarrow_E \langle E, v \rangle$

$$\frac{e \Downarrow_E v}{e \Downarrow_E \langle E, v \rangle} \quad (\mathbf{E}\text{-exp})$$

- Evaluating a define **extends** the environment with a new binding
- Sequencing must thread the environments

$$\frac{e \Downarrow_E v}{(\mathbf{define} \ x \ e) \Downarrow_E \langle E[x \mapsto v], \mathbf{void} \rangle} \quad (\mathbf{E}\text{-def})$$

$$\frac{t_1 \Downarrow_{E_1} \langle E_2, v_1 \rangle \quad t_2 \Downarrow_{E_2} \langle E_3, v_2 \rangle}{t_1; t_2 \Downarrow_{E_1} \langle E_3, v_2 \rangle} \quad (\mathbf{E}\text{-seq})$$



Implementing defines with environments

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.t_b) \quad e_a \Downarrow_E v_a \quad t_b \Downarrow_{E_b[x \mapsto v_a]} v_b}{(e_f e_a) \Downarrow v_b} \quad (\mathbf{E}\text{-app})$$

$$\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \quad (\mathbf{E}\text{-exp})$$

$$\frac{e \Downarrow_E v}{(\mathbf{define} \ x \ e) \Downarrow_E (E[x \mapsto v], \mathbf{void})} \quad (\mathbf{E}\text{-def})$$

$$\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \quad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad (\mathbf{E}\text{-seq})$$

Why λ_F is incorrect?

Evaluating define

Example 1

Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

What is the output of this program?

Evaluating define

Example 1

Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

What is the output of this program? The output is: 20

Let us try and evaluate this program with our λ_F semantics!

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Evaluating

Output

```
Environment: [ (a . 20) ]  
Value: #<void>
```

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Output

```
Environment: [ (a . 20) ]  
Value: #<void>
```

Evaluating

$$\frac{20 \Downarrow\{\} 20 \quad (\text{E-val})}{(\text{define } a \ 20) \Downarrow\{\} (\{a : 20\}, \text{void})} \quad \text{E-def}$$

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: #<void>
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: #<void>
```

Evaluating

$$\frac{\lambda y.a \Downarrow_{\{a:20\}} (\{a : 20\}, \lambda y.a) \quad (\text{E-lam})}{(\text{define } b \lambda y.a) \Downarrow_{\{a:20\}} (\{a : 20, b : (\{a : 20\}, \lambda y.a)\}, \text{void})} \text{E-def}$$

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: 20
```

Evaluation

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: 20
```

Evaluation

$$\frac{\frac{\frac{E(b) = (\{a : 20\}, \lambda y.a)}{b \Downarrow_E (\{a : 20\}, \lambda y.a)} \text{E-var} \quad \frac{1 \Downarrow_E 1}{1 \Downarrow_E 1} \text{E-val} \quad \frac{\frac{F(a) = 20}{a \Downarrow_F 20} \text{E-var}}{\text{E-app}}}{(b \ 1) \Downarrow_E 20} \text{E-app}}{(b \ 1) \Downarrow_E (E, 20)} \text{E-exp}$$

where

$$E = \{a : 20, b : (\{a : 20\}, \lambda y.a)\}$$

$$F = \{a : 20\}[y \mapsto 1] = \{a : 20, y : 1\}$$



Evaluating define

Example 2

Evaluating define

Example 2

Consider the following program

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

What is the output of this program?

Evaluating define

Example 2

Consider the following program

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

What is the output of this program? The output is: 20

Let us try and evaluate this program with our λ_F semantics!

Example 2: step 1

Input

```
Environment: []  
Term: (define b (lambda (y) a))
```

Example 2: step 1

Input

```
Environment: []  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
)]  
Value: #<void>
```

Evaluation

Example 2: step 1

Input

```
Environment: []  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (b . (closure [] (lambda (y) a)))  
]  
Value: #<void>
```

Evaluation

$$\frac{\lambda y.a \Downarrow_{\{\}} (\{\}, \lambda y.a) \quad (\text{E-lam})}{(\text{define } b \lambda y.a) \Downarrow_{\{\}} (\{b : (\{\}, \lambda y.a)\}, \text{void})} \text{E-def}$$

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
]  
Term: (define a 20)
```

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
]  
Term: (define a 20)
```

Evaluation

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Value: #<void>
```

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
  ]  
Term: (define a 20)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
  ]  
Value: #<void>
```

Evaluation

$$\frac{20 \Downarrow_{\{b: (\{\}, \lambda y. a)\}} 20 \quad (\text{E-val})}{(\text{define } a \ 20) \Downarrow_{\{b: (\{\}, \lambda y. a)\}} (\{b : (\{\}, \lambda y. a), a : 20\}, \text{void})} \text{E-def}$$

Example 2: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Term: (b 1)
```


Example 2: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Value: error! a is undefined
```

Insight

When creating a closure we copied the existing environment, and therefore any future updates are forgotten.

The semantics of λ_F is not enough! We need to introduce a notion of **mutation**.



CS450

Structure of Higher Level Languages

Lecture 16: Challenges specifying Racket's define

Tiago Cogumbreiro

Press arrow keys   to change slides.

Introducing the λ_D

Language λ_D : Terms

We highlight in **red** an operation that produces a side effect: *mutating an environment*.

$$\frac{e \Downarrow_E v \quad \mathbf{E} \leftarrow [x := v]}{(\mathbf{define} \ x \ e) \Downarrow_E \mathbf{void}} \quad (\mathbf{E-def})$$

$$\frac{t_1 \Downarrow_E v_1 \quad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2} \quad (\mathbf{E-seq})$$

Language λ_D : Expressions

Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \quad e_a \Downarrow_E v_a \quad \mathbf{E}_b \leftarrow \mathbf{E}_f + [x := v_a] \quad t_b \Downarrow_{E_b} v_b}{(e_f e_a) \Downarrow_E v_b} \quad (\mathbf{E}\text{-app})$$

Can you explain why the order is important?

Language λ_D : Expressions

Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \quad e_a \Downarrow_E v_a \quad \mathbf{E}_b \leftarrow \mathbf{E}_f + [x := v_a] \quad t_b \Downarrow_{E_b} v_b}{(e_f e_a) \Downarrow_E v_b} \quad (\mathbf{E}\text{-app})$$

Can you explain why the order is important? Otherwise, we might evaluate the body of the function e_b without observing the assignment $x := v_a$ in E_b .



Mutable operations on environments

Mutable operations on environments

Put

$$E \leftarrow [x := v]$$

Take a reference to an environment E and mutate its contents, by adding a new binding.

Push

$$E \leftarrow E' + [x := v]$$

Create a new environment referenced by E which copies the elements of E' and also adds a new binding.

Making side-effects explicit

Mutation as a side-effect

Let us use a triangle \blacktriangleright to represent the order of side-effects.

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\text{define } x \ e) \Downarrow_E \text{ void}} \text{ (E-def)}$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2} \text{ (E-seq)}$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x. t_b) \quad \blacktriangleright \quad e_a \Downarrow_E v_a \quad \blacktriangleright \quad E_b \leftarrow E_f + [x := v_a] \quad \blacktriangleright \quad t_b \Downarrow_{E_b} v_b}{(e_f \ e_a) \Downarrow_E v_b} \text{ (E-app)}$$

Implementing side-effect mutation

Making the heap explicit

We can annotate each triangle with a heap, to make explicit which how the global heap should be passed from one operation to the next. In this example, defining a variable takes an input global heap H and produces an output global heap H_2 .

$$\frac{\begin{array}{ccc} \blacktriangleright_H & e \Downarrow_E v & \blacktriangleright_{H_1} \end{array} \quad \begin{array}{c} \color{red} E \leftarrow [x := v] \\ \blacktriangleright_{H_2} \end{array}}{\blacktriangleright_H (\text{define } x \ e) \Downarrow_E \text{void} \blacktriangleright_{H_2}} \quad (\text{E-def})$$

Let us use our rule sheet!

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\text{define } x \ e) \Downarrow_E \text{void}} \text{ (E-def)}$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2} \text{ (E-seq)}$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x. t_b) \quad \blacktriangleright \quad e_a \Downarrow_E v_a \quad \blacktriangleright \quad E_b \leftarrow E_f + [x := v_a] \quad \blacktriangleright \quad t_b \Downarrow_{E_b} v_b}{(e_f \ e_a) \Downarrow_E v_b} \text{ (E-app)}$$

$$v \Downarrow_E v \quad \text{ (E-val)}$$

$$x \Downarrow_E E(x) \quad \text{ (E-var)}$$

$$\lambda x. t \Downarrow_E (E, \lambda x. t) \quad \text{ (E-lam)}$$

Examples

Evaluating Example 2

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

Input

```
E0: []  
---  
Env: E0  
Term: (define b (lambda (y) a))
```

Evaluating Example 2

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

Input

```
E0: []  
---  
Env: E0  
Term: (define b (lambda (y) a))
```

Output

```
E0: [  
  (b . (closure E0 (lambda (y) a)))  
]  
Value: #<void>
```

$$\frac{\lambda y.a \Downarrow_{E_0} (E_0, \lambda y.a)}{(\text{define } b \lambda y.a) \Downarrow_{E_0} \text{void}}$$

▶

$$\frac{}{E_0 \leftarrow [b := (E_0, \lambda y.a)]}$$

Example 2: step 2

Input

```
E0: [  
  (b . (closure E0 (lambda (y) a)))  
]  
---  
Env: E0  
Term: (define a 20)
```


Example 2: step 2

Input

```
E0: [  
  (b . (closure E0 (lambda (y) a)))  
]  
---  
Env: E0  
Term: (define a 20)
```

Output

```
E0: [  
  (a . 20)  
  (b . (closure E0 (lambda (y) a)))  
]  
Value: #<void>
```

$$\frac{20 \Downarrow_{E_0} 20 \quad \blacktriangleright \quad E_0 \leftarrow [a := 20]}{(\text{define } a \ 20) \Downarrow_{E_0} \text{void}}$$

Example 2: step 3

Input

```
E0: [  
  (a . 20)  
  (b . (closure E0 (lambda (y) a)))  
]  
---  
Env: E0  
Term: (b 1)
```

Example 2: step 3

Input

```
E0: [  
  (a . 20)  
  (b . (closure E0 (lambda (y) a)))  
]  
---  
Env: E0  
Term: (b 1)
```

Output

```
E0: [  
  (a . 20)  
  (b . (closure E0 (lambda (y) a)))  
]  
E1: [ E0  
      (y . 1)  
]  
Value: 20
```

$$\frac{b \Downarrow_{E_0} (E_0, \lambda y.a) \blacktriangleright 1 \Downarrow_{E_0} 1 \blacktriangleright E_1 \leftarrow E_0 + [y := 1] \blacktriangleright a \Downarrow_{E_1} 20}{(b\ 1) \Downarrow_{E_0} 20}$$

Example 3

```
(define (f x) (lambda (y) x))  
(f 10)
```

Input

```
E0: []
```

```
---
```

```
Env: E0
```

```
Term: (define (f x) (lambda (y) x))
```

Example 3

```
(define (f x) (lambda (y) x))  
(f 10)
```

Input

```
E0: []  
---  
Env: E0  
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
Value: void
```

Example 3

```
(define (f x) (lambda (y) x))  
(f 10)
```

Input

```
E0: []  
---  
Env: E0  
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [  
  (f . (closure E0  
         (lambda (x) (lambda (y) x))))  
]  
Value: void
```

$$\frac{\lambda x. \lambda y. x \Downarrow_{E_0} (E_0, \lambda x. \lambda y. x)}{(\text{define } f \lambda x. \lambda y. x) \Downarrow_{E_0} \text{void}}$$

Example 3

```
(define (f x) (lambda (y) x))  
(f 10)
```

Input

```
E0: []  
---  
Env: E0  
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
Value: void
```

$$\frac{\lambda x.\lambda y.x \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x) \quad \blacktriangleright \quad E_0 \leftarrow [f := (E_0, \lambda x.\lambda y.x)]}{(\text{define } f \lambda x.\lambda y.x) \Downarrow_{E_0} \text{void}}$$

Example 3

Input

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
---  
Env: E0  
Term: (f 10)
```


Example 3

Input

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
---  
Env: E0  
Term: (f 10)
```

Output

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
E1: [ E0 (x . 10) ]  
Value: (closure E1 (lambda (y) x))
```

Example 3

Input

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
---  
Env: E0  
Term: (f 10)
```

Output

```
E0: [  
  (f . (closure E0  
        (lambda (x) (lambda (y) x))))  
]  
E1: [ E0 (x . 10) ]  
Value: (closure E1 (lambda (y) x))
```

$$E_0(f) = (E_0, \lambda x. \lambda y. x)$$

$$\frac{f \Downarrow_{E_0} (E_0, \lambda x. \lambda y. x)}{\frac{\frac{10 \Downarrow_{E_0} 10 \quad E_1 \leftarrow E_0 + [x := 10]}{\lambda y. x \Downarrow_{E_1} (E_1, \lambda y. x)}}{(f \ 10) \Downarrow_{E_0} (E_1, \lambda y. x)}}$$

How to implement mutation
without mutable constructs?

Motivating example

- Calling function `b` must somehow access variable `a` which is defined after its creation.

```
; Env: []  
(define b (lambda (x) a))  
; Env: [(b . (closure ?? (lambda (x) a)))]  
(define a 20)  
; Env: [(b . (closure ?? (lambda (x) a)) (a . 20))]  
(b 1)
```