



Defining Racket: Functions

Topics

- Function definitions
- Function application
- Functions are first-class values.

Anonymous function **definition** expressions

Syntax: `(lambda (x1 ... xn) e)`

- **parameters:** x1 through xn are identifiers
- **body:** e is any expression

Evaluation:

1. The result is a **function closure**, $\langle E, (\text{lambda } (x1 \dots xn) e) \rangle$, holding the current environment, E , and the function.

[\[closure\]](#)

$E \vdash (\text{lambda } (x1 \dots xn) e) \downarrow \langle E, (\text{lambda } (x1 \dots xn) e) \rangle$

Note:

- An anonymous function definition is an expression.
- A function closure is a new kind of value. Closures are not expressions.
- This is a **definition**, not a call. The body, e , is **not** evaluated now.
- `lambda` from the λ -calculus.

Function **application (call)** expressions

Syntax: `(e0 e1 ... en)`

function expression (1) argument expressions (0 or more)

Evaluation:

1. Under the current dynamic environment, E , evaluate $e0$ through en to values $v0, \dots, vn$.
2. If $v0$ is a function closure of n arguments, $\langle E', (\text{lambda } (x1 \dots xn) e) \rangle$ then

The result is the result of evaluating the closure body, e , under the closure environment, E' , extended with argument bindings:

$x1 \mapsto v1, \dots, xn \mapsto vn$.

Otherwise, there is a type error.

Function application (call) expressions

Syntax: $(e_0 e_1 \dots e_n)$

function expression (1) argument expressions (0 or more)

Evaluation:

$$\frac{\begin{array}{l} E \vdash e_0 \downarrow \langle E', (\text{lambda } (x_1 \dots x_n) e) \rangle \\ E \vdash e_1 \downarrow v_1 \\ \dots \\ E \vdash e_n \downarrow v_n \\ x_1 \mapsto v_1, \dots, x_n \mapsto v_n, E' \vdash e \downarrow v \end{array}}{E \vdash (e_0 e_1 \dots e_n) \downarrow v} \text{ [apply]}$$

Function application derivation example

$$\begin{array}{l} \cdot \vdash ((\text{lambda } (x) (* x x))) \downarrow \langle \cdot, (\text{lambda } (x) (* x x)) \rangle \text{ [closure]} \\ \cdot \vdash 12 \downarrow 12 \text{ [value]} \\ \cdot \vdash 8 \downarrow 8 \text{ [value]} \\ 4 = 12 - 8 \\ \cdot \vdash (- 12 8) \downarrow 4 \text{ [sub]} \\ \begin{array}{l} x \mapsto 4, \cdot \vdash x \downarrow 4 \text{ [var]} \\ x \mapsto 4, \cdot \vdash x \downarrow 4 \text{ [var]} \\ 16 = 4 * 4 \end{array} \\ x \mapsto 4, \cdot \vdash (* x x) \downarrow 16 \text{ [mul]} \\ \cdot \vdash (((\text{lambda } (x) (* x x)) (- 12 8))) \downarrow 16 \text{ [apply]} \end{array}$$

Function bindings and recursion

A function is an expression, so `define` works:

```
(define square
  (lambda (x) (* x x)))
```

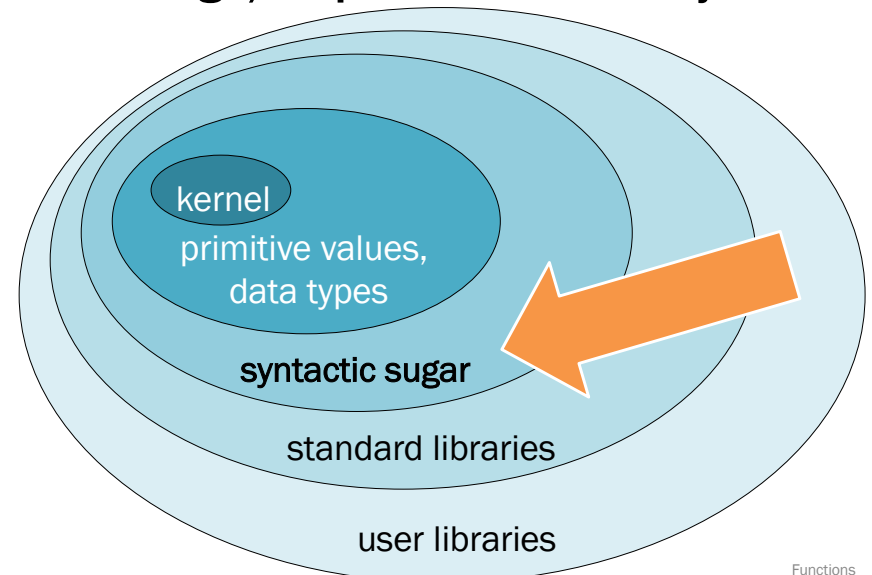
`define` also adds self-binding to function's environment*, supporting recursion.

```
(define pow
  (lambda (base exp)
    (if (< exp 1)
        1
        (* base (pow base (- exp 1))))))
```

During an application of this function, `pow` will be bound to this function.

* Magic for now. We will be precise later.

PL design/implementation: layers





Syntactic sugar for function bindings

```
(define (pow base exp)
  (if (< exp 1)
      1
      (* base (pow base (- exp 1)))))
```

Syntactic sugar: simpler syntax for common idiom.

- Static textual translation to existing features.
- *i.e.*, not a new feature.

Desugar

```
(define (x0 x1 ... xn) e)
to
(define x0 (lambda (x1 ... xn) e))
```

Functions 9

More syntactic sugar

What else looks like a function application?

What looks like a function application but really is not?

Functions 10

So sweet

Expressions like `(+ e1 e2)`, `(< e1 e2)`, and `(not e)` are really just function calls!

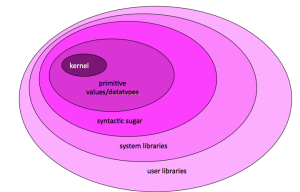
Initial top-level environment binds built-in functions:

- `+` \mapsto addition function,
- `-` \mapsto subtraction function,
- `*` \mapsto multiplication function,
- `<` \mapsto less-than function,
- `not` \mapsto boolean negation function,
- ...

(where some built-in functions do primitive things)

Functions 11

Racket so far



Racket declaration bindings:

```
(define x e)
```

Racket expressions (most of the kernel language!)

literal values (numbers, booleans, strings): `251`, `3.141`, `#t`, `"PL"`

variable references: `x`, `fact`, `positive?`, `fib@n-1`

conditionals: `(if e1 e2 e3)`

functions: `(lambda (x1 ... xn) e)`

function application: `(e0 e1 ... en)`

What about:

- Assignment? Unnecessary! Thread state through.
- Loops? Unnecessary! Use recursion.
- Data structures? `lambda` is all we need, but other options soon.

Functions 12

Racket kernel syntax so far

Bindings

$b ::= (\text{define } x \ e)$

Expressions

$e ::= v \mid x \mid (\text{if } e \ e \ e)$
 $\quad \mid (\text{lambda } (x^*) \ e) \mid (e \ e^*)$

"*" is grammar meta-syntax that means "zero or more of the preceding symbol."

Literal Values (booleans, numbers, strings)

$v ::= \#f \mid \#t \mid n \mid s$

Identifiers (variable names)

x (see valid identifier explanation)

Meta-syntax so far

- Syntax of our evaluation model.
- Not part of the Racket syntax.
- Cannot write in source programs.

Values (+closures)

$v ::= \dots \mid \langle E, (\text{lambda } (x^*) \ e) \rangle$

Environments

$E ::= \cdot \mid x \mapsto v, E$

Racket kernel dynamic semantics so far

Binding evaluation: $E \vdash b \Downarrow E'$

[define]

$$\frac{E \vdash e \Downarrow v}{E \vdash (\text{define } x \ e) \Downarrow x \mapsto v, E'}$$

Expression evaluation: $E \vdash e \Downarrow v$

[value]

$E \vdash v \Downarrow v$

[var]

$$\frac{E(x) = v}{E \vdash x \Downarrow v}$$

[if nonfalse]

$$\frac{E \vdash e1 \Downarrow v1 \quad E \vdash e2 \Downarrow v2 \quad v1 \text{ is not } \#f}{E \vdash (\text{if } e1 \ e2 \ e3) \Downarrow v2}$$

[apply]

$$\frac{E \vdash e0 \Downarrow \langle E', (\text{lambda } (x1 \ \dots \ xn) \ e) \rangle \quad E \vdash e1 \Downarrow v1 \quad \dots \quad E \vdash en \Downarrow vn \quad x1 \mapsto v1, \dots, xn \mapsto vn, E' \vdash e \Downarrow v}{E \vdash (e0 \ e1 \ \dots \ en) \Downarrow v}$$

[if false]

$$\frac{E \vdash e1 \Downarrow \#f \quad E \vdash e3 \Downarrow v3}{E \vdash (\text{if } e1 \ e2 \ e3) \Downarrow v3}$$