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***, ⟨`E`, `(lambda (x1 … xn) e)`⟩, holding the current environment, `E`, and the function.

[closure]

$$\frac{}{\texttt{E} \vdash \texttt{(lambda (x1 … xn) e)} \Downarrow \langle \texttt{E}, \texttt{(lambda (x1 … xn) e)} \rangle}$$

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, …, vn`.
2. If `v0` is a function closure of n arguments, ⟨`E', (lambda (x1 … xn) e)`⟩ then

   The result is the result of evaluating the closure body, `e`, under the closure environment, `E'`, extended with argument bindings: `x1` ⟼ `v1`, …, `xn` ⟼ `vn`.

   Otherwise, there is a type error.

Function application (call) expressions

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

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

Evaluation:

$$\frac{\begin{array}{l} E \vdash e0 \Downarrow \langle E', (\texttt{lambda}\ (x1 \ldots xn)\ e)\rangle \\ E \vdash e1 \Downarrow v1 \\ \quad\ldots \\ E \vdash en \Downarrow vn \\ x1 \mapsto v1, \ldots, xn \mapsto vn, E' \vdash e \Downarrow v \end{array}}{E \vdash (e0\ e1\ \ldots\ en) \Downarrow v}\ [\text{apply}]$$

Function application derivation example

[closure]

$\cdot \vdash$ (lambda (x) (* x x)) $\downarrow \langle \cdot,$ (lambda (x) (* x x))$\rangle$

$\cdot \vdash 12 \downarrow 12$ [value]

$\cdot \vdash 8 \downarrow 8$ [value]

$4 = 12 - 8$

$\cdot \vdash$ (- 12 8) $\downarrow 4$ [sub]

$x \mapsto 4, \cdot \vdash x \downarrow 4$ [var]

$x \mapsto 4, \cdot \vdash x \downarrow 4$ [var]

$16 = 4 * 4$

$x \mapsto 4, \cdot \vdash$ (* x x) $\downarrow 16$ [mul]

$\cdot \vdash$ ((lambda (x) (* x x)) (- 12 8)) $\downarrow 16$ [apply]

Function bindings and recursion

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

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

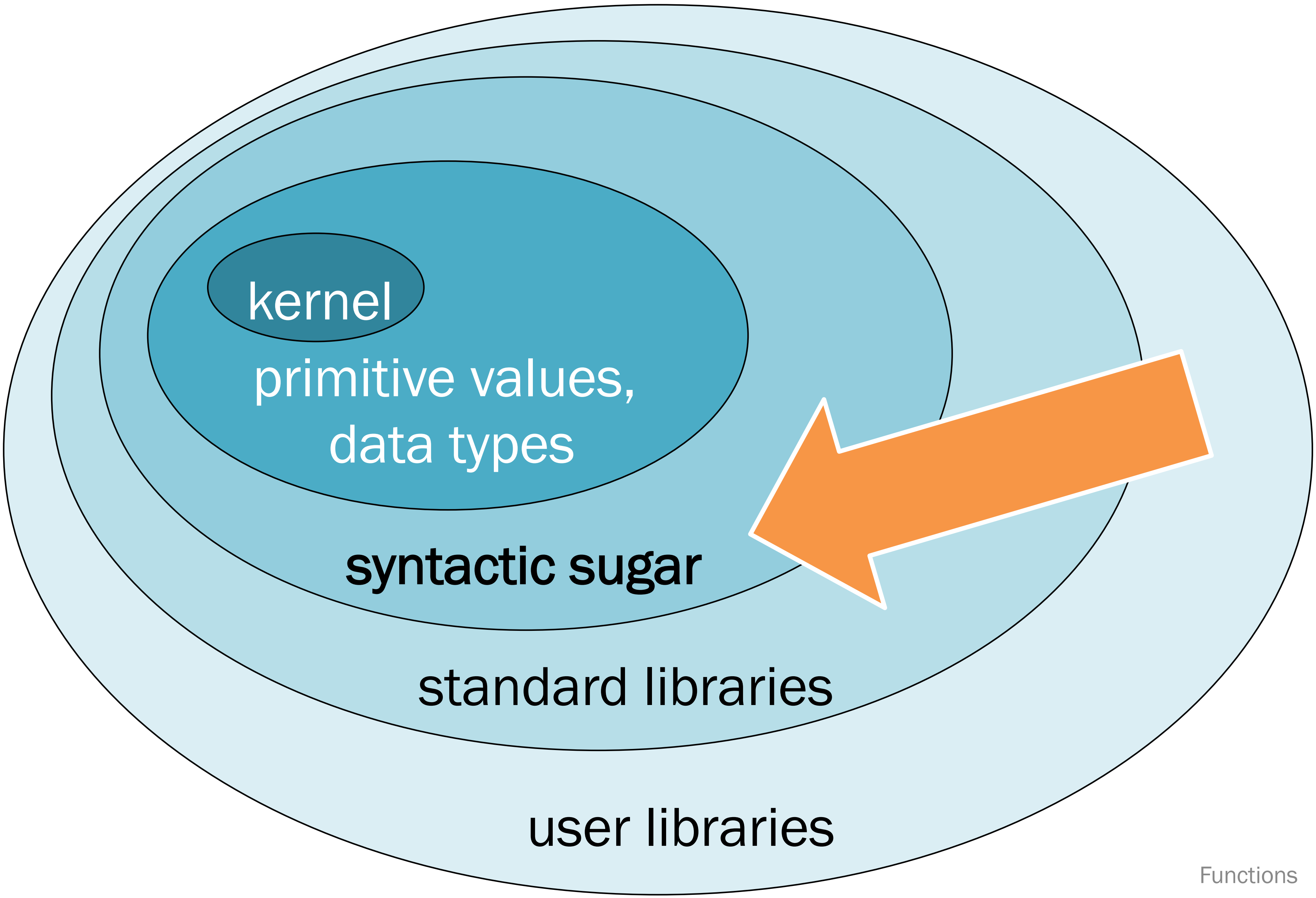
`define` also adds self-binding to function's environment*, supporting **recursion.**

```
(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))
```

More syntactic sugar

What else looks like a function application?

What looks like a function application but really is not?

So sweet

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

Initial top-level environment binds built-in functions:

`+` ⟼ *addition function,*

`-` ⟼ *subtraction function,*

`*` ⟼ *multiplication function,*

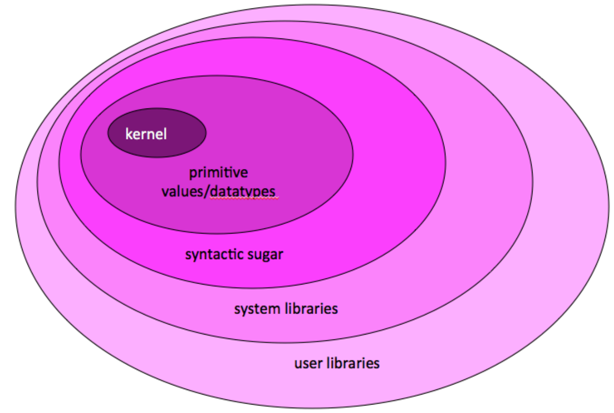
`<` ⟼ *less-than function,*

`not` ⟼ *boolean negation function,*

...

(where some built-in functions do primitive things)

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.

Racket kernel syntax so far

Bindings

```
b ::= (define x e)
```

Expressions

```
e ::= v | x | (if e e e)
    | (lambda (x*) e) | (e e*)
```

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

Literal Values (booleans, numbers, strings)

```
v ::= #f | #t | n | 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 ::= ... | $\langle E$, `(lambda (`x*`)` e`)`$\rangle$

Environments

E ::= . | $x \longmapsto v$, E

Racket kernel dynamic semantics so far

Binding evaluation: $\boxed{E \vdash b \Downarrow E'}$

[define]

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

Expression evaluation: $\boxed{E \vdash e \downarrow v}$

[value]

$$E \vdash v \downarrow v$$

[var]

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

[if nonfalse]

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

[apply]

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

[if false]

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