Defining Racket: Functions
Topics

• Function definitions
• Function application
• Functions are first-class values.
Anonymous function definition expressions

Syntax: \((\text{lambda } (x_1 \ldots x_n) \; e)\)

- **parameters:** \(x_1\) through \(x_n\) are identifiers
- **body:** \(e\) is any expression

Evaluation:

1. The result is a **function closure**, \(\langle E, (\text{lambda } (x_1 \ldots x_n) \; e)\rangle\), holding the current environment, \(E\), and the function.

\[\text{[closure]} E \vdash (\text{lambda } (x_1 \ldots x_n) \; e) \downarrow \langle E, (\text{lambda } (x_1 \ldots x_n) \; e)\rangle\]

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 **\(\lambda\)-calculus**.
Function **application (call) expressions**

**Syntax:** \((e_0 \ e_1 \ldots \ e_n)\)

**Evaluation:**

1. Under the current dynamic environment, \(E\), evaluate \(e_0\) through \(e_n\) to values \(v_0, \ldots, v_n\).
2. If \(v_0\) is a function closure of \(n\) arguments, \(\langle E', (\text{lambda} \ (x_1 \ldots x_n) \ e) \rangle\) then
   
   The result is the result of evaluating the closure body, \(e\), under the closure environment, \(E'\), extended with argument bindings:
   
   \[ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n. \]

   Otherwise, there is a type error.
**Function application (call) expressions**

**Syntax:** 
\[(e_0 \ e_1 \ldots \ e_n)\]

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

**Evaluation:**

\[
\begin{align*}
E \vdash e_0 & \downarrow \langle E', (\lambda (x_1 \ldots x_n) \ e) \rangle \\
E \vdash e_1 & \downarrow v_1 \\
& \ldots \\
E \vdash e_n & \downarrow v_n \\
x_1 & \mapsto v_1, \ldots, x_n \mapsto v_n, E' \vdash e \downarrow v \\
E \vdash (e_0 \ e_1 \ldots \ e_n) & \downarrow v
\end{align*}
\]

[apply]
Function application derivation example

\[
\begin{align*}
&\vdash (\text{lambda} \, (x) \, (\ast \, x \, x)) \downarrow \langle \, ., \, (\text{lambda} \, (x) \, (\ast \, x \, x)) \rangle \\
&\vdash 12 \downarrow 12 \quad \text{[value]} \\
&\vdash 8 \downarrow 8 \quad \text{[value]} \\
&4 = 12 - 8 \\
&\vdash (\text{-}12 \, 8) \downarrow 4 \quad \text{[sub]} \\
&x \mapsto 4, \quad \vdash x \downarrow 4 \quad \text{[var]} \\
&x \mapsto 4, \quad \vdash x \downarrow 4 \quad \text{[var]} \\
&16 = 4 \ast 4 \\
&x \mapsto 4, \quad \vdash (\ast \, x \, x) \downarrow 16 \quad \text{[mul]} \\
&\vdash (\text{lambda} \, (x) \, (\ast \, x \, x))((-12 \, 8)) \downarrow 16 \quad \text{[apply]} 
\end{align*}
\]
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)))))
```

* Magic for now. We will be precise later.

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

* * *
PL design/implementation: layers

- kernel
- primitive values, data types
- syntactic sugar
- standard libraries
- user libraries
Syntactic sugar for function bindings

(\texttt{define (pow base exp)}
 \begin{align*}
 (\texttt{if (< exp 1)} & \\
 1 & \\
 (* \texttt{base (pow base (- exp 1)))}))
\end{align*}

\textbf{Syntactic sugar}: simpler syntax for common idiom.
- Static textual translation to existing features.
- \textit{i.e., not a new feature}.

\textbf{Desugar}
(\texttt{define (x0 x1 \ldots xn) e})
\texttt{to}
(\texttt{define x0 (lambda (x1 \ldots 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 \((+ \ e_1 \ e_2)\), \((< \ e_1 \ e_2)\), and \((\text{not} \ e)\) are really just function calls!

Initial top-level environment binds built-in functions:
- \(+ \mapsto \text{addition function},\)
- \(- \mapsto \text{subtraction function},\)
- \(* \mapsto \text{multiplication function},\)
- \(< \mapsto \text{less-than function},\)
- \(\text{not} \mapsto \text{boolean negation function},\)
- ...

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

Expressions
\[ e ::= v | x | (\text{if } e \ e \ e) \]
\[ | (\text{lambda } (x^*) \ e) | (e \ e^*) \]

Literal Values (booleans, numbers, strings)
\[ v ::= \#f | \#t | n | s \]

Identifiers (variable names)
\[ x \text{ (see valid identifier explanation)} \]

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

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

Values (+closures)

\[ v ::= \ldots \mid \langle E, (\lambda x^* e) \rangle \]

Environments

\[ E ::= . \mid x \mapsto v, E \]
Racket kernel dynamic semantics so far

Binding evaluation:  \[ E \vdash b \downarrow E' \]

\[[\text{define}]\]
\[
\frac{E \vdash e \downarrow v}{E \vdash (\text{define } x e) \downarrow x \mapsto v, E'}
\]

Expression evaluation:  \[ E \vdash e \downarrow v \]

\[[\text{value}]\]
\[
E \vdash v \downarrow v
\]

\[[\text{var}]\]
\[
\frac{E(x) = v}{E \vdash x \downarrow v}
\]

\[[\text{apply}]\]
\[
E \vdash e0 \downarrow \langle E', (\lambda(x1 \ldots xn)\ e) \rangle
E \vdash e1 \downarrow v1 \quad \ldots \quad E \vdash en \downarrow vn
x1 \mapsto v1, \ldots, xn \mapsto vn, E' \vdash e \downarrow v
E \vdash (e0 \ e1 \ \ldots \ en) \downarrow v
\]

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

\[[\text{if false}]\]
\[
\frac{E \vdash e1 \downarrow \#f}{E \vdash e3 \downarrow v3}
\]

\[
E \vdash (\text{if } e1\ e2\ e3) \downarrow v3
\]