Defining Racket: Functions

Topics

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

Anonymous function definition expressions

Syntax: \( \texttt{(lambda (x1 \ldots xn) e)} \)

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

Evaluation:
1. The result is a \textit{function closure}, \( \langle E, \texttt{(lambda (x1 \ldots xn) e)} \rangle \), holding the current environment, E, and the function.

Function application (call)

Syntax: \( \texttt{(e0 e1 \ldots en)} \)

Evaluation:
1. Under the current dynamic environment, E, evaluate e0 through en to values v0, \ldots, vn.
2. If v0 is a function closure of n arguments, \( \langle E', \texttt{(lambda (x1 \ldots xn) e)} \rangle \) then
   The result is the result of evaluating the closure body, e, under the closure environment, E', extended with argument bindings: x1 \( \mapsto \) v1, \ldots, xn \( \mapsto \) vn.
   Otherwise, there is a type error.
Function application (call)

Syntax: \((e_0 \ e_1 \ … \ e_n)\)

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

Function application derivation example

Assume initial environment is empty.
\[
((\lambda (x) (* x x)) (- 12 8))
\]

Function bindings and recursion

A function is an expression, so define works:

\[
\begin{align*}
& (\text{define } \text{square} \\
& \quad (\lambda (x) (* x x)))
\end{align*}
\]

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

\[
\begin{align*}
& (\text{define } \text{pow} \\
& \quad (\lambda (\text{base} \ \text{exp}) \\
& \quad \quad (\text{if} (< \ \text{exp} \ 1) \\
& \quad \quad \quad 1 \\
& \quad \quad \quad (* \ \text{base} \ (\text{pow} \ \text{base} \ (- \ \text{exp} \ 1))))))
\end{align*}
\]

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

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

\[
\texttt{(define } \texttt{(pow base exp)}\n\texttt{(if } (< \texttt{exp 1)}\n\texttt{1}\n\texttt{(* base (pow base (\- \texttt{exp 1)})))})
\]

**Syntactic sugar**: simpler syntax for common idiom.
- Static textual translation to existing features.
- i.e., not a new feature.

**Desugar**
\[
\texttt{(define } (x0 x1 \ldots xn e)}\n\texttt{to}\n\texttt{(define x0 (lambda (x1 \ldots xn e))})
\]

---

More syntactic sugar

What else looks like a function application?

What looks like a function application but really is not?

---

So sweet

Expressions like \((+ e1 e2), (< e1 e2), \text{and} (\text{not } e)\) 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:**
\[
\texttt{(define x e)}
\]

**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 \ldots xn e))
- function application: (e0 e1 \ldots 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
\[ e ::= v \mid x \mid (\text{if } e\ e\ e) \mid (\text{lambda } (x^*) \ e) \mid (e\ e^*) \]

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

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

Racket kernel dynamic semantics so far

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

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

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

[value]
\[ E \vdash v \Downarrow v \]
\[ E(x) = v \]
\[ E \vdash x \Downarrow v \]

[var]
\[ E \vdash e \Downarrow v \]

[apply]
\[ E \vdash e_0 \Downarrow \langle E', (\text{lambda } (x_1 \ldots x_n) \ e) \rangle \]
\[ E \vdash e_1 \Downarrow v_1 \ldots E \vdash e_n \Downarrow v_n \]
\[ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, E' \vdash e \Downarrow v \]
\[ E \vdash (e_0\ e_1 \ldots\ e_n) \Downarrow v \]

Meta-syntax so far

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

Values (+closures)
\[ v ::= \ldots \mid \langle E, (\text{lambda } (x^*) \ e) \rangle \]

Environments
\[ E ::= \cdot \mid x \mapsto v, E \]