Defining Racket: Functions

https://cs.wellesley.edu/~cs251/
Topics

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

Syntax: 

\[
\text{\texttt{(lambda (x1 \ldots xn) e)}}
\]

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

Evaluation:

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

\[
\text{[closure]}
\]

\[
E \vdash (\text{\texttt{lambda (x1 \ldots xn) e}}) \downarrow (E, (\text{\texttt{lambda (x1 \ldots xn) e}}))
\]

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.
– \texttt{lambda} from the **\(\lambda\)-calculus**.
Function application (call)

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, \(\langle E', (\lambda (x_1 \ldots x_n) \ e)\rangle\), then

   The result is the result of evaluating the closure body, \(e\), under the closure environment, \(E'\), extended with argument bindings:
   \(x_1 \leftrightarrow v_1, \ldots, x_n \leftrightarrow v_n\).

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

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

Evaluation:

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

```
(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

- Kernel
  - Primitive values, data types
- Syntactic sugar
- Standard libraries
- User libraries

Functions
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 \((+ \ e_1 \ e_2)\), \((< \ e_1 \ e_2)\), and \((\text{not} \ e)\)are really just function calls!

Initial top-level environment binds built-in functions:

\[
\begin{align*}
+ & \mapsto \text{addition function}, \\
- & \mapsto \text{subtraction function}, \\
* & \mapsto \text{multiplication function}, \\
< & \mapsto \text{less-than function}, \\
\text{not} & \mapsto \text{boolean negation function}, \\
\ldots
\end{align*}
\]
(\text{where some built-in functions do primitive things})
Racket so far

Racket declaration bindings:

\[
\text{(define x e)}
\]

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
\[ e ::= v \mid x \mid (if \ e \ e \ e) \]
\[ \quad \mid (\lambda \ (x^*) \ e) \mid (e \ e^*) \]

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

Identifiers (variable names)
\[ x \] (see valid identifier explanation)
Meta-syntax so far

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

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

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

[var]

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

[if nonfalse]
\[
\begin{align*}
E & \vdash e1 \downarrow v1 \\
E & \vdash e2 \downarrow v2 \\
v1 & \text{is not } \#f \\
\frac{}{E \vdash (\text{if } e1 \ e2 \ e3) \downarrow v2}
\end{align*}
\]

[if false]
\[
\begin{align*}
E & \vdash e1 \downarrow \#f \\
E & \vdash e3 \downarrow v3 \\
E & \vdash (\text{if } e1 \ e2 \ e3) \downarrow v3
\end{align*}
\]