Functions in Racket

Racket Functions

Functions: the most important building block in Racket (and 251)
- Functions/procedures/methods/subroutines abstract over computations
- Like Java methods & Python functions, Racket functions have arguments and result
- But no classes, this, return, etc.
- The most basic Racket function are anonymous functions specified with lambda

Examples:

```
> ((lambda (x) (* x 2)) 5)
10
> (define dbl (lambda (x) (* x 2)))
> (dbl 21)
42
> (define quad (lambda (x) (dbl (dbl x))))
> (quad 10)
40
> (define avg (lambda (a b) (/ (+ a b) 2)))
> (avg 8 12)
10
```

lambda denotes an anonymous function

Syntax: `(lambda (Id1 ... Idn) Ebody)`
- `lambda`: keyword that introduces an anonymous function
  (the function itself has no name, but you’re welcome to name it using define)
- `Id1 ... Idn`: any identifiers, known as the parameters of the function.
- `Ebody`: any expression, known as the body of the function.
  It typically (but not always) uses the function parameters.

Evaluation rule:
- A lambda expression is just a value (like a number or boolean), so a lambda expression evaluates to itself!
- What about the function body expression? That’s not evaluated until later, when the function is called. (Synonyms for called are applied and invoked.)

Function applications (calls, invocations)

To use a function, you apply it to arguments (call it on arguments).
E.g. in Racket: `(dbl 3), (avg 8 12), (small? 17)`

Syntax: `(E0 E1 ... En)`
- A function application expression has no keyword. It is the only parenthesized expression that doesn’t begin with a keyword.
- `E0`: any expression, known as the rator of the function call (i.e., the function position).
- `E1 ... En`: any expressions, known as the rands of the call (i.e., the argument positions).

Evaluation rule:
1. Evaluate `E0 ... En` in the current environment to values `V0 ... Vn`.
2. If `V0` is not a lambda expression, raise an error.
3. If `V0` is a lambda expression, returned the result of applying it to the argument values `V1 ... Vn` (see following slides).
**Function application**

What does it mean to apply a function value (lambda expression) to argument values? E.g.

```
((lambda (x) (* x 2)) 3)
((lambda (a b) (/ (+ a b) 2)) 8 12)
```

We will explain function application using two models:

1. The **substitution model**: substitute the argument values for the parameter names in the function body.
2. The **environment model**: extend the environment of the function with bindings of the parameter names to the argument values.

---

**Substitution notation**

We will use the notation

\[ E[V_1, \ldots, V_n/\text{Id}_1, \ldots, \text{Id}_n] \]

to indicate the expression that results from substituting the values \( V_1, \ldots, V_n \) for the identifiers \( \text{Id}_1, \ldots, \text{Id}_n \) in the expression \( E \).

For example:

- \((* \ x \ 2)[3/x]\) stands for \((* \ 3 \ 2)\)
- \((/ (+ \ a \ b) \ 2)[8,12/a,b]\) stands for \((/ (+ \ 8 \ 12) \ 2)\)
- \((\text{if} \ (< \ x \ z) (+ (* \ x \ x) (* \ y \ y)) (/ \ x \ y)) [3,4/x,y]\)
  stands for \((\text{if} \ (< \ 3 \ z) (+ (* \ 3 \ 3) (* \ 4 \ 4)) (/ \ 3 \ 4))\)

It turns out that there are some very tricky aspects to doing substitution correctly. We’ll talk about these when we encounter them.

---

**Function application: substitution model**

**Example 1:**

```
((lambda (x) (* x 2)) 3)
```

Now evaluate \((* \ 3 \ 2)\) to \(6\)

**Example 2:**

```
((lambda (a b) (/ (+ a b) 2)) 8 12)
```

Now evaluate \((/ (+ \ 8 \ 12) \ 2)\) to \(10\)

---

**Avoid this common substitution bug**

Students sometimes **incorrectly** substitute the argument values into the parameter positions:

\[ ((\text{lambda} (a \ b) (/ (+ a b) 2)) 8 12) \]

Makes no sense

\[ (\text{lambda} (8 12) (/ (+ 8 12) 2)) \]

When substituting argument values for parameters, **only the modified body should remain. The lambda and params disappear!**

\[ ((\text{lambda} (a \ b) (/ (+ a b) 2)) 8 12) \]

\[ (/ (+ 8 12) 2) \]
Small-step function application rule: substitution model

\[
(\text{lambda } (I_1 \ldots I_n) \ E_{\text{body}}) \ V_1 \ldots V_n
\]

\[\Rightarrow E_{\text{body}}[V_1, \ldots, V_n/I_1, \ldots, I_n]\] [function call (a.k.a. apply)]

Note: could extend this with notion of “current environment”

Small-step semantics: function example

Suppose \(env_2 = \text{quad} \mapsto (\text{lambda } (x) (\text{dbl } (\text{dbl } x))),\)

\[
dbl \mapsto (\text{lambda } (x) (* x 2))
\]

\[
(\text{quad} 3) \# env_2
\]

\[
(\text{dbl } (\text{dbl } 3)) \# env_2 \quad [\text{function call}]
\]

\[
((\text{lambda } (x) (* x 2)) (\text{dbl } 3)) \# env_2 \quad [\text{varref}]
\]

\[
((\text{lambda } (x) (* x 2)) ((\text{lambda } (x) (* x 2)) 3)) \# env_2 \quad [\text{function call}]
\]

\[
((\text{lambda } (x) (* x 2)) ([* \text{three} 2])) \# env_2 \quad [\text{function call}]
\]

\[
((\text{lambda } (x) (* x 2)) 6) \# env_2 \quad [\text{multiplication}]
\]

\[
(* 6 2) \# env_2 \quad [\text{function call}]
\]

\[
12 \# env_2 \quad [\text{multiplication}]
\]

Small-step substitution model semantics: your turn

Suppose \(env_3 = n \mapsto 10,\)

\[\text{small}\? \mapsto (\text{lambda } (\text{num}) (\leq \text{num} n)),\]

\[\text{sqr} \mapsto (\text{lambda } (n) (* n n))\]

Give an evaluation derivation for \((\text{small}\? (\text{sqr } n))\# env_3\)

Stepping back: name issues

Do the particular choices of function parameter names matter?

Is there any confusion caused by the fact that \(\text{dbl}\) and \(\text{quad}\) both use \(x\) as a parameter?

Are there any parameter names that we can’t change \(x\) to in \(\text{quad}\)?

In \((\text{small}\? (\text{sqr } n))\), is there any confusion between the global variable named \(n\) and the parameter \(n\) in \(\text{sqr}\)?

Is there any parameter name we can’t use instead of \(\text{num}\) in \(\text{small}\)?
Evaluation Contexts

Although we will not do so here, it is possible to formalize exactly how to find the next redex in an expression using so-called evaluation contexts.

For example, in Racket, we never try to reduce an expression within the body of a lambda.

\[
\begin{align*}
(\lambda x \, (+ (* 4 5) x))
\end{align*}
\]

We’ll see later in the course that other choices are possible (and sensible).

Substitution model derivation

Suppose \(env2 = dbl \rightarrow (\lambda x \, (* x 2))\),
\(quad \rightarrow (\lambda x \, (dbl (dbl x)))\)

```
quad \# env2 ↓ (lambda (x) (dbl (dbl x)))
3 \# env2 ↓ 3
  dbl \# env2 ↓ (lambda (x) (* x 2))
  dbl \# env2 ↓ (lambda (x) (* x 2))
  3 \# env2 ↓ 3
  (* 3 2) \# env2 ↓ 6 [multiplication rule, subparts omitted]
    (function call)
  (dbl 3) \# env2 ↓ 6
  (* 6 2) \# env2 ↓ 12 (multiplication rule, subparts omitted)
    (function call)
  (dbl (dbl 3)) \# env2 ↓ 12 (function call)
(quad 3) \# env2 ↓ 12
```

Big step function call rule: substitution model

\[
\begin{align*}
 E0 \# env ↓ (\text{lambda} \quad (ld1 \ldots ldn) \quad Ebody) \\
 E1 \# env ↓ V1 \\
 \vdots \\
 En \# env ↓ Vn \\
 Ebody[V1 \ldots Vn/ld1 \ldots ldn] \# env ↓ Vbody \quad \text{(function call)} \\
 (E0 \ E1 \ldots \ En) \# env ↓ Vbody
\end{align*}
\]

Note: no need for function application frames like those you’ve seen in Python, Java, C, ...

Recursion

Recursion works as expected in Racket using the substitution model (both in big-step and small-step semantics).

There is no need for any special rules involving recursion! The existing rules for definitions, functions, and conditionals explain everything.

```
(define fact
  (lambda (n)
    (if (= n 0)
      1
      (* n (fact (- n 1))))))
```

What is the value of \(\text{fact} \ 3\)?
Small-step recursion derivation for (fact 4) [1]

Let's use the abbreviation \( \lambda \text{fact} \) for the expression
\((\lambda \text{n} \ (\text{if (= \text{n} 0) 1 (* \text{n} (\text{fact (- \text{n} 1))}))})\)

\((\text{fact} \ 4) \Rightarrow ((\lambda \text{fact} \ 4)) \Rightarrow (\text{if ((= 4 0) 1 (* 4 (\text{fact (- 4 1))))}) \Rightarrow ((\text{if #f 1 (* 4 (\text{fact (- 4 1))))}) \Rightarrow (* 4 ((\text{fact} (- 4 1)))) \Rightarrow (* 4 (\lambda \text{fact} ((- 4 1)))) \Rightarrow (* 4 (((\lambda \text{fact} 3]))) \Rightarrow (* 4 (* 3 ((\lambda \text{fact} 2)))) \Rightarrow (* 4 (* 3 (if ((= 2 0) 1 (* 2 (\text{fact (- 2 1)))))))) \Rightarrow (* 4 (* 3 ((if #f 1 (* 2 (\text{fact (- 2 1))))))))… continued on next slide …

Abbreviating derivations with \( \Rightarrow \)

\( E1 \Rightarrow* E2 \) means \( E1 \) reduces to \( E2 \) in zero or more steps

\((\text{fact} \ 4) \Rightarrow ((\lambda \text{fact} \ 4)) \Rightarrow* (* 4 (((\lambda \text{fact} 3))) \Rightarrow* (* 4 (* 3 ((\lambda \text{fact} 2)))) \Rightarrow* (* 4 (* 3 (* 2 ((\lambda \text{fact} 1))))) \Rightarrow* (* 4 (* 3 (* 2 (* 1 ((\lambda \text{fact} 0))))) \Rightarrow* (* 4 (* 3 (* 2 ((- 1 1))))) \Rightarrow* (* 4 (* 3 ((( 2 1))))) \Rightarrow* (* 4 ((( 3 2)))) \Rightarrow* ((( 4 6))) \Rightarrow 24

Small-step recursion derivation for (fact 4) [2]

... continued from previous slide ...

\((\text{fact} \ 4) \Rightarrow ((\lambda \text{fact} \ 4)) \Rightarrow* (* 4 (((\lambda \text{fact} 3))) \Rightarrow* (* 4 (* 3 ((\lambda \text{fact} 2)))) \Rightarrow* (* 4 (* 3 (* 2 ((\lambda \text{fact} 1))))) \Rightarrow* (* 4 (* 3 (* 2 (* 1 ((\lambda \text{fact} 0))))) \Rightarrow* (* 4 (* 3 (* 2 ((- 1 1))))) \Rightarrow* (* 4 (* 3 ((( 2 1))))) \Rightarrow* (* 4 ((( 3 2)))) \Rightarrow* ((( 4 6))) \Rightarrow 24

Recursion: your turn

Show an abbreviated small-step evaluation of \((\text{pow} \ 5 \ 3)\) where \text{pow} is defined as:

\(\begin{align*}
\text{(define pow} \\
\quad \text{lambda} \ (\text{base exp}) \\
\quad \quad \text{if (= exp 0)} \\
\quad \quad \quad 1 \\
\quad \quad \quad (* \text{ base (pow base (- exp 1))}))
\end{align*}\)

How many multiplications are performed in
\((\text{pow} \ 2 \ 10)\)?
\((\text{pow} \ 2 \ 100)\)?
\((\text{pow} \ 2 \ 1000)\)?

What is the stack depth (number of pending multiplies) in these cases?
Recursion: your turn 2

Show an abbreviated small-step evaluation of \((\text{fast-pow} \ 2 \ 10)\) with the following definitions:

\[
\begin{align*}
\text{define square} & \ (\lambda n \ (* \ n \ n)) \\
\text{define even?} & \ (\lambda n \ (= 0 \ \text{remainder} \ n \ 2)) \\
\text{define fast-pow} & \ (\lambda (\text{base \ exp}) \ \begin{cases} 
1 & \text{if} \ (\text{even? \ exp}) \\
(\text{fast-pow} \ (\text{square \ base}) \ (\text{/ \ exp \ 2})) & \text{else}
\end{cases} \\
& \ (* \ \text{base} \ (\text{fast-pow \ base} \ (\text{- \ exp \ 1})))
\end{align*}
\]

How many multiplications are performed in
\((\text{fast-pow} \ 2 \ 10)\)?
\((\text{fast-pow} \ 2 \ 100)\)
\((\text{fast-pow} \ 2 \ 1000)\)

What is the stack depth (# pending multipies) in these cases?

Fibonacci with small-step semantics

Suppose the global env contains binding \(\text{fib} \mapsto \lambda\_\text{fib}\), where \(\lambda\_\text{fib}\) abbreviates
\((\lambda \ (n) \ (\text{if} \ (<= \ n \ 1) \ n \ (+ \ (\text{fib} \ (\text{- \ n} \ 1)) \ (\text{fib} \ (\text{- \ n} \ 2))))\))

\[
\begin{align*}
(\text{fib} \ 4) & \Rightarrow \ ((\lambda\_\text{fib} \ 4)) \\
& \Rightarrow* (+ \ ((\lambda\_\text{fib} \ 3) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 2)) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 1)) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 1)) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 1)) \ (\text{fib} \ (\text{- \ n} \ 2))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow* (+ \ (+ \ ((\lambda\_\text{fib} \ 0)))) \\
& \Rightarrow 3
\end{align*}
\]

Tree Recursion: Fibonacci

\[
\begin{align*}
\text{define (fib \ n) \ ; \ returns \ rabbit \ pairs \ at \ month \ n} \\
& \ \text{(if} \ (<= \ n \ 1) \ ; \ \text{assume} \ n \ >= \ 0 \\
& \ \text{n} \\
& \ \text{(+} \ (\text{fib} \ (\text{- \ n} \ 1)) \ ; \ \text{pairs \ alive \ last \ month} \\
& \ \text{((fib} \ (\text{- \ n} \ 2)) \ ; \ \text{newborn \ pairs} \\
& \ \text{))))}
\end{align*}
\]

How many additions as a function of \(n\)?
What is the stack depth as a function of \(n\)?

Syntactic sugar: function definitions

**Syntactic sugar**: simpler syntax for common pattern.

- Implemented via textual translation to existing features.
- \textit{i.e., not a new feature}.

Example: Alternative function definition syntax in Racket:

\[
\begin{align*}
\text{(define (Id\ funName \ ld1 \ ld2 ... ldn) E\_body)} \\text{desugars to} \\
& \text{(define \ funName \ (lambda (ld1 \ ld2 ... \ldn) E\_body))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (dbl \ x) \ (* \ x \ 2))} \\
\text{(define (quad \ x) \ (dbl \ (dbl \ x)))} \\
\text{(define (pow \ base \ exp)} \\
& \ \text{(if} \ (<= \ \text{exp} \ 1) \\
& \ \text{1} \\
& \ \text{(* \ base \ (pow \ base \ \text{- \ exp} \ 1))))
\end{align*}
\]
Racket Operators are Actually Functions!

Surprise! In Racket, operations like `( + e1 e2 )`, `( < e1 e2 )` and `( not e )` are really just function calls!

There is an initial top-level environment that contains bindings for built-in functions like:
- `+` (`addition function`),
- `-` (`subtraction function`),
- `*` (`multiplication function`),
- `<` (`less-than function`),
- `not` (`boolean negation function`),
- ...

(where some built-in functions can do special primitive things that regular users normally can’t do --- e.g. add two numbers)

Racket Language Summary So Far

Racket Declarations:
- definitions: `(define Id E)`

Racket Expressions (this is most of the kernel language!)
- literal values (numbers, boolean, strings): e.g. 251, 3.141, #t, "Lyn"
- variable references: e.g., x, fact, positive?, fib_n-1
- conditionals: `(if Etest Ethen Else )`
- function values: `(lambda (Id1 ... Idn) Ebody)`
- function calls: `(Erator Erand1 ... Erandn)`
  Note: arithmetic and relational operations are really just function calls!

What about:
- Assignment? Don’t need it!
- Loops? Don’t need them! Use tail recursion, coming soon.
- Data structures? Glue together two values with `cons` (next time).
  - Can even implement data structures with `lambda`! (See Wacky Lists on PS4, Functional Sets on PS8)
  - Motto: `lambda` is all you need!