**Functions in Racket**

**CS251 Programming Languages**
Fall 2017, Lyn Turbak

Department of Computer Science
Wellesley College

---

**Racket Functions**

Functions: most important building block in Racket (and 251)
- Functions/procedures/methods/subroutines abstract over computations
- Like Java methods, Python functions have arguments and result
- But no classes, `this`, `return`, etc.

Examples:
```
(define dbl (lambda (x) (* x 2)))
(define quad (lambda (x) (dbl (dbl x))))
(define avg (lambda (a b) (/ (+ a b) 2)))
(define sqr (lambda (n) (* n n)))
(define n 10)
(define small? (lambda (num) (<= num n)))
```

---

**lambda** denotes a 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.

```lisp
((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

\[ E[V_1, \ldots, V_n / l_1, \ldots, l_n] \]

to indicate the expression that results from substituting the values \( V_1, \ldots, V_n \) for the identifiers \( l_1, \ldots, l_n \) in the expression \( E \).

For example:

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

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

**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

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

\[( \lambda (Id1 \ldots Idn) \ Ebody) \ V1 \ldots Vn \]  
\[\Rightarrow Ebody[V1 \ldots Vn/Id1 \ldots Idn]\] [function call (a.k.a.apply)]

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

Small-step semantics: function example

\[(\text{quad} \ 3)\]  
\[\Rightarrow \ ((\lambda (x) (\text{dbl} (\text{dbl} \ x))) \ 3)\] [varref]  
\[\Rightarrow (\text{dbl} (\text{dbl} \ 3))\] [function call]  
\[\Rightarrow ((\lambda (x) (* x 2)) (\text{dbl} \ 3))\] [varref]  
\[\Rightarrow ((\lambda (x) (* x 2)) (\lambda (x) (* x 2)) 3)\] [varref]  
\[\Rightarrow ((\lambda (x) (* x 2)) (* 3 2))\] [function call]  
\[\Rightarrow ((\lambda (x) (* x 2)) (* x 2)) 6\] [multiplication]  
\[\Rightarrow (* 6 2)\] [function call]  
\[\Rightarrow 12\] [multiplication]

Small-step substitution model semantics: your turn

Suppose \(env3 = n \rightarrow 10\),  
\(\text{small?} \rightarrow (\lambda (\text{num}) (\leq \text{num} \ n))\),  
\(\text{sqr} \rightarrow (\lambda (n) (* n n))\)

Give an evaluation derivation for \((\text{small?} \ (\text{sqr} \ n))\)# \(env3\)

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 parameter name \(n\) and 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`.

\[
(\lambda (x) (+ (* 4 5) x))
\]

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

Substitution model derivation

Suppose `env2` = `dbl` → `(lambda (x) (* x 2))`, `quad` → `(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

\[
E_0 \# \text{env} \downarrow (\lambda (Id_1 \ldots Id_n) \ E_{\text{body}})
\]

\[
E_1 \# \text{env} \downarrow V_1
\]

\[
E_n \# \text{env} \downarrow V_n
\]

\[
E_{\text{body}}[V_1 \ldots V_n/Id_1 \ldots Id_n] \# \text{env} \downarrow V_{\text{body}}
\]

(function call)

\[
(E_0 E_1 \ldots E_n) \# \text{env} \downarrow V_{\text{body}}
\]

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.

\[
\text{(define } \text{fact}
(\lambda (n)
  \text{(if } (= n 0)
    1
    (* n (\text{fact} (- n 1))))))
\]

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

Let’s use the abbreviation \( \lambda \_\text{fact} \) for the expression

\[ (\lambda \ n \ (if \ (= \ n \ 0) \ 1 \ (* \ n \ (\text{fact} \ (- \ n \ 1)))) \)

\( (\text{fact} \ 4) \)

\( \Rightarrow \ ((\lambda \_\text{fact} \ 4)) \)

\( \Rightarrow \ (if \ (= \ 4 \ 0) \ 1 \ (* \ 4 \ (\text{fact} \ (- \ 4 \ 1)))) \)

\( \Rightarrow \ ((\text{if} \ #f \ 1 \ (* \ 4 \ (\text{fact} \ (- \ 4 \ 1)))))) \)

\( \Rightarrow \ (* \ 4 \ ((\text{fact} \ (- \ 4 \ 1)))) \)

\( \Rightarrow \ (* \ 4 \ (\lambda \_\text{fact} \ (- \ 4 \ 1))) \)

\( \Rightarrow \ (* \ 4 \ ((\lambda \_\text{fact} \ 3)) \)

\( \Rightarrow \ (* \ 4 \ (if \ (= \ 3 \ 0) \ 1 \ (* \ 3 \ (\text{fact} \ (- \ 3 \ 1)))))) \)

\( \Rightarrow \ (* \ 4 \ ((\text{if} \ #f \ 1 \ (* \ 3 \ (\text{fact} \ (- \ 3 \ 1)))))) \)

\( \Rightarrow \ (* \ 4 \ (* \ 3 \ ((\text{fact} \ (- \ 3 \ 1)))) \)

\( \Rightarrow \ (* \ 4 \ (* \ 3 \ (\lambda \_\text{fact} \ (- \ 3 \ 1))) \)

\( \Rightarrow \ (* \ 4 \ (* \ 3 \ ((\lambda \_\text{fact} \ 2)))) \)

\( \Rightarrow \ (* \ 4 \ (* \ 3 \ (if \ (= \ 2 \ 0) \ 1 \ (* \ 2 \ (\text{fact} \ (- \ 2 \ 1)))))) \)

\( \Rightarrow \ (* \ 4 \ (* \ 3 \ ((if \ #f \ 1 \ (* \ 2 \ (\text{fact} \ (- \ 2 \ 1))))))) \)

… continued on next slide …

Abbreviating derivations with \( \Rightarrow \)

\( E_1 \ \Rightarrow \ E_2 \) means \( E_1 \) reduces to \( E_2 \) in zero or more steps

\[
\begin{align*}
(\{\text{fact} \ 4\}) & \Rightarrow ((\lambda \_\text{fact} \ 4)) \\
& \Rightarrow (* \ 4 \ ((\lambda \_\text{fact} \ 3))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ ((\lambda \_\text{fact} \ 2)))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ ((\lambda \_\text{fact} \ 1)))))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ ((\lambda \_\text{fact} \ 0))))))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ ((\lambda \_\text{fact} \ 0))))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ ((\lambda \_\text{fact} \ 0))))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ 1)))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ 1)))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ 1))) \\
& \Rightarrow (* \ 4 \ (* \ 3 \ 2)) \\
& \Rightarrow (* \ 4 \ 6) \\
& \Rightarrow 24
\end{align*}
\]

Recursion: your turn

Show an abbreviated small-step evaluation of \( (\text{pow} \ 5 \ 3) \)

where \( \text{pow} \) is defined as:

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

How many multiplications are performed in

\( (\text{pow} \ 2 \ 10) ? \)

\( (\text{pow} \ 2 \ 100) ? \)

\( (\text{pow} \ 2 \ 1000) ? \)

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

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

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

How many multiplications are performed in

\[
\text{(pow 2 10)?}
\]
\[
\text{(pow 2 100)?}
\]
\[
\text{(pow 2 1000)?}
\]

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

Tree Recursion: fibonacci

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

\[
\text{(\text{fib} 4)}
\]
\[
\Rightarrow ((\_fib 4))
\]
\[
\Rightarrow (+ ((\lambda \_fib 3)) (\text{fib} (- 4 2)))
\]
\[
\Rightarrow (+ (+ ((\lambda \_fib 2)) (\text{fib} (- 3 2))) (\text{fib} (- 4 2)))
\]
\[
\Rightarrow (+ (+ (+ (((\_fib 1))) (\text{fib} (- 3 2))) (\text{fib} (- 4 2))))
\]
\[
\Rightarrow (+ (+ 1 (((\_fib 0))) (\text{fib} (- 3 2))) (\text{fib} (- 4 2)))
\]
\[
\Rightarrow (+ (+ 1 (((\_fib 0))) (\text{fib} (- 4 2))))
\]
\[
\Rightarrow (+ 2 (((\_fib 1))) (\text{fib} (- 4 2)))
\]
\[
\Rightarrow (+ 2 (((\_fib 0))))
\]
\[
\Rightarrow (+ 2 ((+ 1 0)))
\]
\[
\Rightarrow {{+} (2 1)}
\]
\[
\Rightarrow 3
\]

Syntactic sugar: function definitions

\text{Syntactic sugar}: simpler syntax for common pattern.

\text{– Implemented via textual translation to existing features.}

\text{– i.e., not a new feature.}

Example: Alternative function definition syntax in Racket:

\[
\text{(define (Id\_funName Id1 ... Idn) E\_body)}
\]

desugars to

\[
\text{(define Id\_funName (lambda (Id1 ... Idn) E\_body))}
\]

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

\[
\text{(define (quad x) (dbl (dbl x)))}
\]

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

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:

\[
\text{+ \rightarrow addition function,}
\]
\[
\text{– \rightarrow subtraction function,}
\]
\[
\text{* \rightarrow multiplication function,}
\]
\[
\text{< \rightarrow less-than function,}
\]
\[
\text{not \rightarrow boolean negation function,}
\]
\[
\text{...}
\]

(\text{where some built-in functions can do special primitive things that regular users normally can’t do --- e.g. add two numbers})
Summary So Far

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

Racket expressions:
- conditionals: `(if Etest Ethen Eelse)`
- function values: `(lambda (Id1 ... Idn) Ebody)`
- Function calls: `(Erator Erand1 ... Erandn)`
  Note: arithmetic and relation operations are 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)