

## Functions in Racket



### CS251 Programming Languages Spring 2018, 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)))
```

Functions 2

## 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**.)

Functions 3

## 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: `(EO E1 ... En)`

- A function application expression has no keyword. It is the only parenthesized expression that **doesn't** begin with a keyword.
- **EO**: 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 **EO ... En** in the current environment to values **VO ... Vn**.
2. If **VO** is not a `lambda` expression, raise an error.
3. If **VO** is a `lambda` expression, returned the result of applying it to the argument values **V1 ... Vn** (see following slides).

Functions 4

## 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.

This lecture

Later

Functions 5

## Function application: substitution model

Example 1:

```
((lambda (x) (* x 2)) 3)
↓ Substitute 3 for x in (* x 2)
(* 3 2)
```

Now evaluate (\* 3 2) to 6

Example 2:

```
((lambda (a b) (/ (+ a b) 2)) 8 12)
↓ Substitute 8 for a and 12 for b
in (/ (+ a b) 2)
(/ (+ 8 12) 2)
```

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

Functions 6

## Substitution notation

We will use the notation

$E[V_1, \dots, V_n / id_1, \dots, id_n]$

to indicate the expression that results from substituting the values  $V_1, \dots, V_n$  for the identifiers  $id_1, \dots, id_n$  in the expression  $E$ .

For example:

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

Functions 7

## Avoid this common substitution bug

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

Makes no sense

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

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

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

Functions 8

## Small-step function application rule: substitution model

```
( (lambda (Id1 ... Idn) Ebody) V1 ... Vn )  
⇒ Ebody[V1, ..., Vn/Id1, ..., Idn] [function call (a.k.a. apply)]
```

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

Functions 9

## Small-step semantics: function example

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

```
(quad 3) # env2  
⇒ ((lambda (x) (dbl (dbl x))) 3) # env2 [varref]  
⇒ (dbl (dbl 3)) # env2 [function call]  
⇒ ((lambda (x) (* x 2)) (dbl 3)) # env2 [varref]  
⇒ ((lambda (x) (* x 2))  
    ((lambda (x) (* x 2)) 3)) # env2 [varref]  
⇒ ((lambda (x) (* x 2)) (* 3 2)) # env2 [function call]  
⇒ ((lambda (x) (* x 2)) 6) # env2 [multiplication]  
⇒ (* 6 2) # env2 [function call]  
⇒ 12 # env2 [multiplication]
```

Functions 10

## Small-step substitution model semantics: your turn

```
Suppose env3 = n → 10,  
small? → (lambda (num) (<= num n)),  
sqr → (lambda (n) (* n n))
```

Give an evaluation derivation for (small? (sqr n)) # env3

Functions 11

## Stepping back: name issues

Do the particular choices of function parameter names matter?

Is there any confusion caused by the fact that dbl and quad both use x as a parameter?

Are there any parameter names that we can't change x to in quad?

In (small? (sqr n)), is there any confusion between the global parameter name n and parameter n in sqr?

Is there any parameter name we can't use instead of num in small?

Functions 12

## 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)) (+ 1 2) )
```

↑ not this                      ↑ this is the first redex

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

Functions 13

## Big step function call rule: substitution model

```

E0 # env ↓ (lambda (Id1 ... Idn) Ebody)
E1 # env ↓ V1
  ⋮
En # env ↓ Vn
Ebody[V1 ... Vn/Id1 ... Idn] # env ↓ Vbody (function call)
(E0 E1 ... En) # env ↓ Vbody
    
```

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

Functions 14

## Substitution model derivation

Suppose  $env2 = db1 \rightarrow (lambda (x) (* x 2))$ ,  
 $quad \rightarrow (lambda (x) (dbl (dbl x)))$

```

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
    
```

Functions 15

## 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 `(fact 3)`?

Functions 16

## Small-step recursion derivation for (fact 4) [1]

Let's use the abbreviation `λ_fact` for the expression

```
(λ (n) (if (= n 0) 1 (* n (fact (- n 1)))))
```

```
{fact} 4)
⇒ {(λ_fact 4)}
⇒ (if {= 4 0} 1 (* 4 (fact (- 4 1))))
⇒ {(if #f 1 (* 4 (fact (- 4 1))))}
⇒ (* 4 ({fact} (- 4 1)))
⇒ (* 4 (λ_fact {(- 4 1)}))
⇒ (* 4 {(λ_fact 3)})
⇒ (* 4 (if {= 3 0} 1 (* 3 (fact (- 3 1)))))
⇒ (* 4 {(if #f 1 (* 3 (fact (- 3 1))))})
⇒ (* 4 (* 3 ({fact} (- 3 1))))
⇒ (* 4 (* 3 (λ_fact {(- 3 1)}))
⇒ (* 4 (* 3 {(λ_fact 2)}))
⇒ (* 4 (* 3 (if {= 2 0} 1 (* 2 (fact (- 2 1)))))
⇒ (* 4 (* 3 {(if #f 1 (* 2 (fact (- 2 1))))})
... continued on next slide ...
```

Functions 17

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

... continued from previous slide ...

```
⇒ (* 4 (* 3 (* 2 ({fact} (- 2 1))))
⇒ (* 4 (* 3 (* 2 (λ_fact {(- 2 1)})))
⇒ (* 4 (* 3 (* 2 {(λ_fact 1)})))
⇒ (* 4 (* 3 (* 2 {(λ_fact 1)})))
⇒ (* 4 (* 3 (* 2 (if {= 1 0} 1 (* 1 (fact (- 1 1)))))
⇒ (* 4 (* 3 (* 2 {(if #f 1 (* 1 (fact (- 1 1))))})
⇒ (* 4 (* 3 (* 2 (* 1 ({fact} (- 1 1)))))
⇒ (* 4 (* 3 (* 2 (* 1 (λ_fact {(- 1 1)})))
⇒ (* 4 (* 3 (* 2 (* 1 {(λ_fact 0)})))
⇒ (* 4 (* 3 (* 2 (* 1 (if {= 0 0} 1 (* 0 (fact (- 0 1)))))
⇒ (* 4 (* 3 (* 2 (* 1 {(if #t 1 (* 0 (fact (- 0 1))))})
⇒ (* 4 (* 3 (* 2 {( * 1 1)})))
⇒ (* 4 (* 3 {( * 2 1)}))
⇒ (* 4 {( * 3 2)})
⇒ {( * 4 6)}
⇒ 24
```

Functions 18

## Abbreviating derivations with $\Rightarrow^*$

$E1 \Rightarrow^* E2$  means  $E1$  reduces to  $E2$  in zero or more steps

```
{fact} 4)
⇒ {(λ_fact 4)}
⇒* (* 4 {(λ_fact 3)})
⇒* (* 4 (* 3 {(λ_fact 2)}))
⇒* (* 4 (* 3 (* 2 {(λ_fact 1)})))
⇒* (* 4 (* 3 (* 2 (* 1 {(λ_fact 0)})))
⇒* (* 4 (* 3 (* 2 {( * 1 1)}))
⇒ (* 4 (* 3 {( * 2 1)}))
⇒ (* 4 {( * 3 2)})
⇒ {( * 4 6)}
⇒ 24
```

Functions 19

## Recursion: your turn

Show an **abbreviated** small-step evaluation of `(pow 5 3)` where `pow` is defined as:

```
(define pow
  (lambda (base exp)
    (if (= exp 0)
        1
        (* base (pow base (- exp 1)))))
```

How many multiplications are performed in

`(pow 2 10)`?

`(pow 2 100)`?

`(pow 2 1000)`?

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

Functions 20

## Recursion: your turn 2

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

```
(define square (lambda (n) (* n n)))
(define even? (lambda (n) (= 0 (remainder n 2))))
(define fast-pow
  (lambda (base exp)
    (if (= exp 0)
        1
        (if (even? exp)
            (fast-pow (square base) (/ exp 2))
            (* base (fast-pow base (- exp 1)))))))
```

How many multiplications are performed in

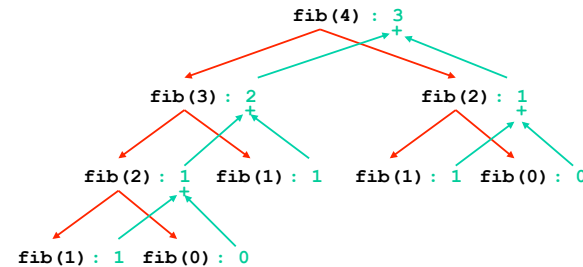
`(pow 2 10)?`  
`(pow 2 100)?`  
`(pow 2 1000)?`

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

Functions 21

## Tree Recursion: Fibonacci

```
(define (fib n) ; returns rabbit pairs at month n
  (if (<= n 1) ; assume n >= 0
      n
      (+ (fib (- n 1)) ; pairs alive last month
         (fib (- n 2)) ; newborn pairs
        )))
```



How many additions as a function of n?

What is the stack depth as a function of n?

Functions 22

## Fibonacci with small-step semantics

Suppose the global env contains binding `fib ↦ λ_fib`, where `λ_fib` abbreviates `(λ (n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))`

```
{fib} 4)
⇒ {(λ_fib 4)}
⇒* (+ {(λ_fib 3)} (fib (- 4 2)))
⇒* (+ (+ {(λ_fib 2)} (fib (- 3 2))) (fib (- 4 2)))
⇒* (+ (+ (+ {(λ_fib 1)} (fib (- 2 2))) (fib (- 3 2))) (fib (- 4 2)))
⇒* (+ (+ (+ 1 {(λ_fib 0)}) (fib (- 3 2))) (fib (- 4 2)))
⇒* (+ (+ {(+ 1 0)} (fib (- 3 2))) (fib (- 4 2)))
⇒* (+ (+ 1 {(λ_fib 1)} (fib (- 4 2)))
⇒* (+ {(+ 1 1)} (fib (- 4 2)))
⇒* (+ 2 {(λ_fib 2)})
⇒* (+ 2 (+ {(λ_fib 1)} (fib (- 2 2))))
⇒* (+ 2 (+ 1 {(λ_fib 0)}))
⇒* (+ 2 {(+ 1 0)})
⇒ {(+ 2 1)}
⇒ 3
```

How many additions?

What is the stack depth?

Functions 23

## Syntactic sugar: function definitions



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

- Implemented via textual translation to existing features.
- *i.e.*, **not a new feature**.

Example: Alternative function definition syntax in Racket:

```
(define (Id_funName Id1 ... Idn) E_body)
```

desugars to

```
(define Id_funName (lambda (Id1 ... Idn) E_body))
```

```
(define (dbl x) (* x 2))
```

```
(define (quad x) (dbl (dbl x)))
```

```
(define (pow base exp)
  (if (< exp 1)
      1
      (* base (pow base (- exp 1)))))
```

Functions 24

## 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)

Functions 25

## 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)

Functions 26