Functions in Racket

CS251 Programming Languages
Spring 2016, 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) e)`

- ***`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.
- `e`: 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**.

Function calls (applications)

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

Syntax: `(e0 e1 … en)`

- A function call expression has no keyword. A function call because it's 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 function 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.

Function application: substitution model

Example 1:

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

Substitute `3` for `x` in `(* x 2)` and evaluate the result:

`(* 3 2)` ↓ `6` (environment doesn't matter in this case)

Example 2:

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

Substitute `3` for `x` in `(* x 2)` and evaluate the result:

`(/ (+ 8 12) 2)` ↓ `10` (environment doesn't matter in this case)

Substitution notation

We will use the notation

e[***v1***, …, ***vn***/***id1***, …, ***idn***]

to indicate the expression that results from substituting the values ***v1***, …, ***vn*** for the identifiers ***id1***, …, ***idn*** 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.

Function call rule: substitution model

e0 **# env** ↓ `(lambda (`***id1*** … ***idn***`)` ***e_body***`)`

e1 **# env** ↓ ***v1***

⋮

en **# env** ↓ ***vn***

e_body[***v1*** … ***vn***/***id1*** … ***idn***] **# env** ↓ ***v_body***

`(`***e0 e1*** … ***en***`)` **# env** ↓ ***v_body*** (function call)

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

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
```


Substitution model derivation: 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***

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?

Small-step vs. big-step semantics

The evaluation derivations we've seen so far are called a **big-step semantics** because the derivation ***e* # *env2* ↓ *v*** explains the evaluation of ***e*** to ***v*** as one "big step" justified by the evaluation of its subexpressions.

An alternative way to express evaluation is a **small-step semantics** in which an expression is simplified to a value in a sequence of steps that simplifies subexpressions. You do this all the time when simplifying math expressions, and we can do it in Racket, too. E.g;

```
(- (* (+ 2 3) 9) (/ 18 6))
⇒ (- (* 5 9) (/ 18 6))
⇒ (- 45 (/ 18 6))
⇒ (- 45 3)
⇒ 42
```

Small-step semantics: intuition

Scan left to right to find the first **redex** (nonvalue subexpression that can be reduced to a value) and reduce it:

```
(- (* (+ 2 3) 9) (/ 18 6))
⇒ (- (* 5 9) (/ 18 6))
⇒ (- 45 (/ 18 6))
⇒ (- 45 3)
⇒ 42
```


Small-step semantics: reduction rules

There are a small number of reduction rules for Racket. These specify the redexes of the language and how to reduce them.

The rules often require certain subparts of a redex to be **values** in order to be applicable.

id ⇒ ***v*** , where ***id*** → ***v*** in the current environment* (varref)

`(+` ***v1 v2*** `)` ⇒ ***v***, where ***v*** is the sum of ***v1*** and ***v2*** (addition)

There are similar rules for other arithmetic operators

`(if #t` ***e_then e_else*** `)` ⇒ ***e_then*** (if true)

`(if #f` ***e_then e_else*** `)` ⇒ ***e_false*** (if false)

`((lambda (`***id1*** … ***idn***`)` ***e_body***`)` ***v1*** … ***vn*** `)`
⇒ ***e_body***[***v1*** … ***vn***/***id1*** … ***idn***] (function call)

* In a more formal approach, the notation would make the environment explicit. E.g., ***e # env*** ⇒ ***v***

Small-step semantics: conditional example

```
(+ (if (< 1 2) (* 3 4) (/ 5 6)) 7)
⇒ (+ (if #t (* 3 4) (/ 5 6)) 7)
⇒ (+ (* 3 4) 7)
⇒ (+ 12 7)
⇒ 19
```


Small-step semantics: errors as stuck expressions

Similar to big-step semantics, we model errors (dynamic type errors, divide by zero, etc.) in small-step semantics as expressions in which the evaluation process is **stuck** because no reduction rule is matched. For example

```
(- (* (+ 2 3) #t) (/ 18 6))
⇒ (- (* 5 #t) (/ 18 6))
```

```
(if (= 2 (/ (+ 3 4) (- 5 5))) 8 9)
⇒ (if (= 2 (/ 7 (- 5 5))) 8 9)
⇒ (if (= 2 (/ 7 0)) 8 9)
```

Small-step semantics: function example

```
(quad 3)
⇒ ((lambda (x) (dbl (dbl x))) 3)
⇒ (dbl (dbl 3))
⇒ ((lambda (x) (* x 2)) (dbl 3))
⇒ ((lambda (x) (* x 2))
    ((lambda (x) (* x 2)) 3))
⇒ ((lambda (x) (* x 2)) (* 3 2))
⇒ ((lambda (x) (* x 2)) 6)
⇒ (* 6 2)
⇒ 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: `(* 4 5)`

this is the first redex: `(+ 1 2)`

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

Small-step semantics: your turn

Use small-step semantics to evaluate `(small? (sqr n))`

Assume this is evaluated with respect to the same global environment used earlier.

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

What is the value of `(pow 5 2)`?

Recursion: your turn

Define and test the following recursive functions in Racket:

`(fact n)`: Return the factorial of the nonnegative integer `n`

`(fib n)`: Return the `n`th Fibonacci number

`(sum-between lo hi)`: return the sum of the integers between integers `lo` and `hi` (inclusive)

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)))))
```


Racket Operators are Actually Functions!

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

There is an initial top-level environment that contains bindings like:

- `+` → *addition function,*
- `-` → *subtraction function,*
- `*` → *multiplication function,*
- `<` → *less-than function,*
- `not` → *boolean negation function,*
- …

Summary So Far

Racket declarations:

- definitions: `(define` ***id e***`)`

Racket expressions:

- conditionals: `(if` ***e_test e_then e_else***`)`
- function values: `(lambda (`***id1 … idn***`)` ***e_body***`)`
- Function calls: (***e_rator e_rand1 … e_randn***)
  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)