Functions in Racket

Design of Programming Languages

Modified Slides of Lyn Turbak

Racket Functions

Functions: the most important building block in Racket

- 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 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.

```
((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.

This lecture

2. The **environment model**: extend the environment of the function with bindings of the parameter names to the argument values.

Later

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

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.

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)
```

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”

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]
```

Small-step substitution model semantics: your turn

Suppose ***env3*** = `n` ⟼ `10`,
`small?` ⟼ `(λ (num) (<= num n))`,
`sqr` ⟼ `(λ (n) (* n n))`

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

Small-step substitution model semantics: your turn

Suppose ***env3*** = `n` ⟼ `10`,

`small?` ⟼ `(λ (num) (<= num n))`,

`sqr` ⟼ `(λ (n) (* n n))`

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

```
({small?} (sqr n)) # env3
⇒ ( (λ (num) (<= num n)) ({sqr} n) ) # env3 [varref]
⇒ ( (λ (num) (<= num n)) ( (λ (n) (* n n)) {n} ) ) # env3 [varref]
⇒ ( (λ (num) (<= num n)) {( (λ (n) (* n n)) 10 )} ) # env3 [varref]
⇒ ( (λ (num) (<= num n)) {(* 10 10)} ) # env3 [function call]
⇒ {( (λ (num) (<= num n)) 100 )} # env3 [multiplication]
⇒ (<= 100 {n}) # env3 [function call]
⇒ (<= 100 10) # env3 [varref]
⇒ #f # env3 [less-than]
```

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 variable named `n` and the parameter `n` in `sqr`?

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

Stepping back: name issues Answers

Do the particular choices of function parameter names matter?
No, the substitution model implies that as long as the parameter names are used consistently in the body and do not conflict with other names, they can be any names you like.

Is there any confusion caused by the fact that `dbl` and `quad` both use `x` as a parameter?
No, the substitution model shows that these two different `x`s do not interact in any way.

Are there any parameter names that we can't change `x` to in `quad`?
`x` can be any name except `dbl`; a `dbl` parameter would "capture" the references to the function `dbl` and change the meaning of the body.

In `(small? (sqr n))`, is there any confusion between the global variable named `n` and the parameter `n` in `sqr`?
No. The substitution model handles references to the the parameter `n` in the body of `sqr` and the [varref] rule handles references to the global variable `n`.

Is there any parameter name we can't use instead of `num` in small?
Yes: changing the paremeter `num` to `n` or `<=` would change the meaning of the function.

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).

Big step function call rule: substitution model

$$\frac{\begin{array}{l} E0 \;\#\; \text{env} \downarrow \texttt{(lambda (}Id1 \ldots Idn\texttt{)}\; Ebody\texttt{)} \\ E1 \;\#\; \text{env} \downarrow V1 \\ \quad\vdots \\ En \;\#\; \text{env} \downarrow Vn \\ Ebody[V1 \ldots Vn/Id1 \ldots Idn] \;\#\; \text{env} \downarrow Vbody \end{array}}{\texttt{(}E0\ E1 \ldots En\texttt{)} \;\#\; \text{env} \downarrow Vbody} \text{ (function call)}$$

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
```

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)`?

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 …

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 (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
```

Abbreviating derivations with ⇒*

E1* ⇒* *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
```

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?

Recursion: your turn **Answers**

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

```
({pow} 5 3)
⇒ {(λ_pow 5 3)}
⇒* (* 5 {(λ_pow 5 2)})
⇒* (* 5 (* 5 {(λ_pow 5 1)}))
⇒* (* 5 (* 5 (* 5 {(λ_pow 5 0)})))
⇒* (* 5 (* 5 {(* 5 1)}))
⇒ (* 5 {(* 5 5)})
⇒ {(* 5 25)}
⇒ 125
```

Call	# multiplications	stack depth
`(pow 2 10)`	10	10
`(pow 2 100)`	100	100
`(pow 2 1000)`	1000	1000

linear in exp, i.e. O(exp)

Recursion: your turn 2

it's your turn

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

`(fast-pow 2 10)`?

`(fast-pow 2 100)`?

`(fast-pow 2 1000)`?

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

Recursion: your turn 2 Answers

Show an **abbreviated** small-step evaluation of `(fast-pow 2 10)`

```
({fast-pow} 2 10)
⇒  {(λ_fast-pow 2 10)}
⇒*  {(λ_fast-pow 4 5)}
⇒*  (* 4 {(λ_fast-pow 4 4)})
⇒*  (* 4 {(λ_fast-pow 16 2)})
⇒*  (* 4 {(λ_fast-pow 256 1)})
⇒*  (* 4 (* 256 {(λ_fast-pow 256 0)}))
⇒*  (* 4 {(* 256 1)})
⇒  {(* 4 256)}
⇒  1024
```

1 + (number of bits in binary rep)

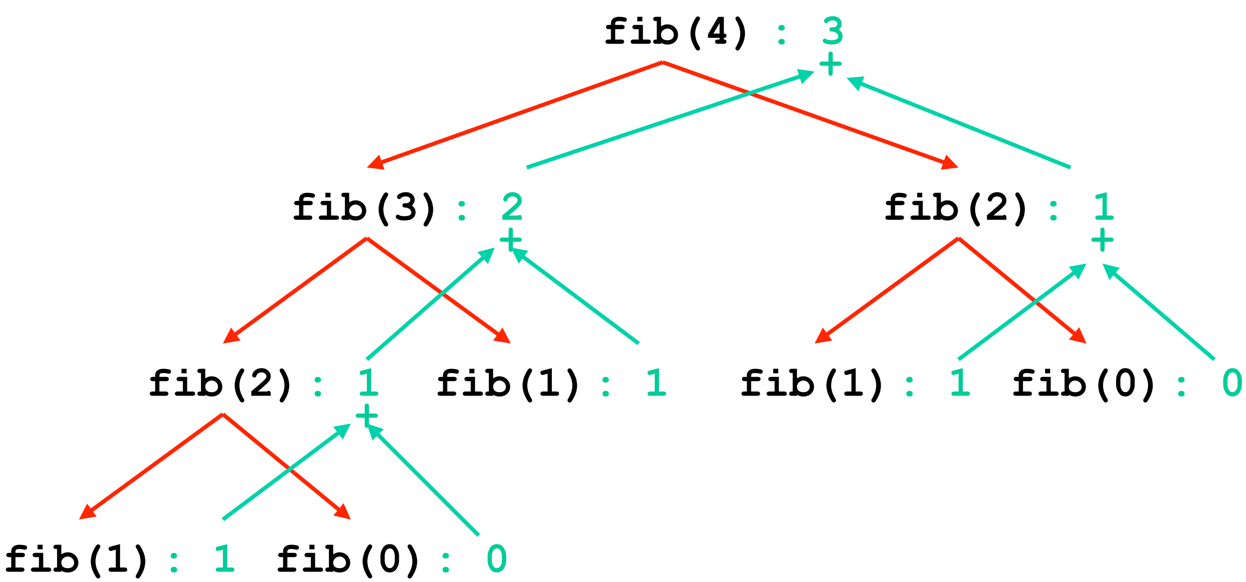
number of 1s in binary rep

Call	exp in binary	# multiplications	stack depth
`(pow 2 10)`	1010	5	2
`(pow 2 100)`	1100100	8	3
`(pow 2 1000)`	1111101000	11	6

logartihmic in exp, i.e. O(lg(exp))

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?

```
                                   fib ⟼ λ_fib      λ_fib
(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?

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***`)` and `(not `***e***`)` are really just function calls!

There is an initial top-level environment that contains bindings for built-in functions like:

$+ \mapsto$ *addition function,*

$- \mapsto$ *subtraction function,*

$* \mapsto$ *multiplication function,*

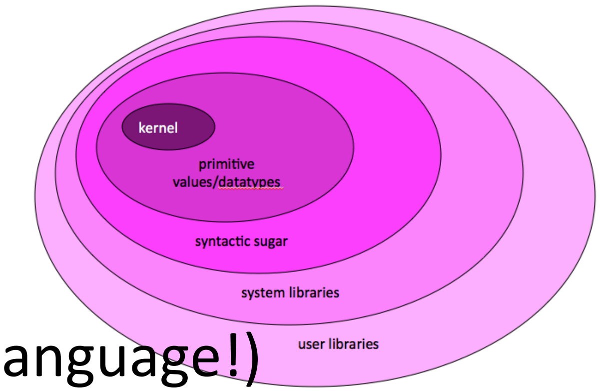
$< \mapsto$ *less-than function,*

`not` $\mapsto$ *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 Eelse*** `)`
- 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!