CSCI-C311 Programming Languages

Racket: Function Calls and lambda Functions

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- [Tutorial] Quick: An Introduction to Racket with Pictures
  - https://docs.racket-lang.org/quick/index.html
  - Part 6 Functions are Values
- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - 2.2.4 Function Calls (Procedure Applications)
  - 2.2.6 Function Calls, Again
  - 2.2.7 Anonymous Functions with lambda
  - 4.4 Functions (Procedures): lambda; 4.4.2 Declaring Optional Arguments

2

Function Calls as Procedure Applications

- Function calls we have seen are *procedure applications*
- Its syntax is

(‹*function_name*› ‹*argument_expr*›*)

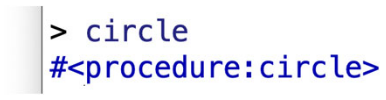
- Examples:

```
(circle 10)  ;this is a built-in function in #lang slideshow.
(hc-append (circle 10) (rectangle 15 25))  ; also in #lang slideshow

> (string-append "rope" "twine" "yarn")  ; append strings
"ropetwineyarn"
```

3

Functions Are Values

- Instead of calling circle as a function, try evaluating it as an expression



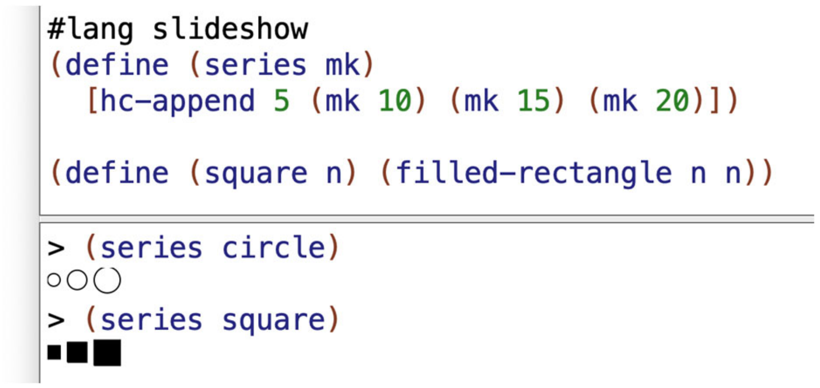
```
> circle
#<procedure:circle>
```

  - The identifier circle is bound to a function (a.k.a. “procedure”)
  - There’s no way of printing the function, so DrRacket just prints #<procedure:circle>
- This shows functions are values, just like numbers and pictures.

4

Functions Are Values

- Can define functions that accept other functions as arguments



```
#lang slideshow
(define (series mk)
  [hc-append 5 (mk 10) (mk 15) (mk 20)])

(define (square n) (filled-rectangle n n))
```

```
> (series circle)
∘○◯
> (series square)
▪■■
```

5

General Function Calls

- The 1st expression in a function call doesn't need to be an identifier for function name, but any expression that evaluates to a function value

```
(define (double v)
  ((if (string? v) string-append +) v v))

> (double "mnah")
"mnahmnah"
> (double 5)
10
```

This conditional expression evaluates to function `string-append` or function `+`

6

General Function Calls

- Syntactically, the 1st expression in a function call could be a number
- But that leads to an error, since a number is not a function.

```
> (1 2 3 4)
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 1
```

7

Anonymous `lambda` Function

- It would be tedious if you had to name all of your functions.
  - When calling a function that accepts a function argument, the argument function often isn't needed anywhere else.
  - Having to write down the function via define would be a hassle, because you have to make up a name and find a place to put the function definition.

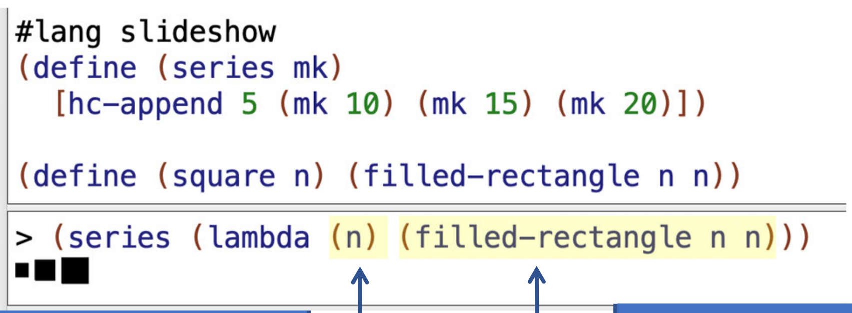
```
(define (series mk)
  [hc-append 5 (mk 10) (mk 15) (mk 20)])

(define (square n) (filled-rectangle n n))
```

```
> (series square)
```

8

Anonymous `lambda` Function

- Instead, can use `lambda`, which creates an anonymous function



```
#lang slideshow
(define (series mk)
  [hc-append 5 (mk 10) (mk 15) (mk 20)])

(define (square n) (filled-rectangle n n))
```

```
> (series (lambda (n) (filled-rectangle n n)))
```

The parenthesized names after a `lambda` are the arguments to the function

and the expression after the argument names is the function body

9

Anonymous `lambda` Function

- Evaluating a lambda form by itself produces a function:

```
> (lambda (n) (filled-rectangle n n))
#<procedure>
```

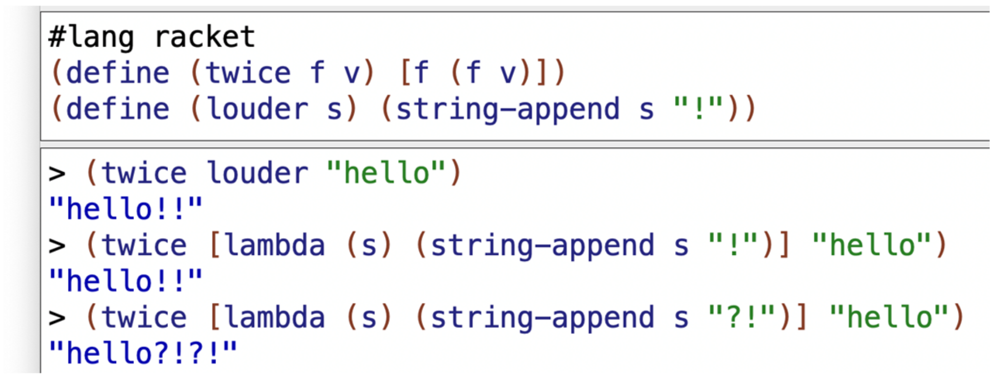
- A define form for a function is really a shorthand for a simple define using lambda as the value.
  - For example, the series definition could be written as

```
(define series
  (lambda (mk)
    (hc-append 4 (mk 5) (mk 10) (mk 20))))
```

10

Anonymous lambda Function

- More Examples in #lang racket



```
#lang racket
(define (twice f v) [f (f v)])
(define (louder s) (string-append s "!"))
```

```
> (twice louder "hello")
"hello!!"
> (twice [lambda (s) (string-append s "!")] "hello")
"hello!!"
> (twice [lambda (s) (string-append s "?!")] "hello")
"hello?!?!"
```

11

Anonymous lambda Function

- Use of lambda as a result for a function that generates functions:

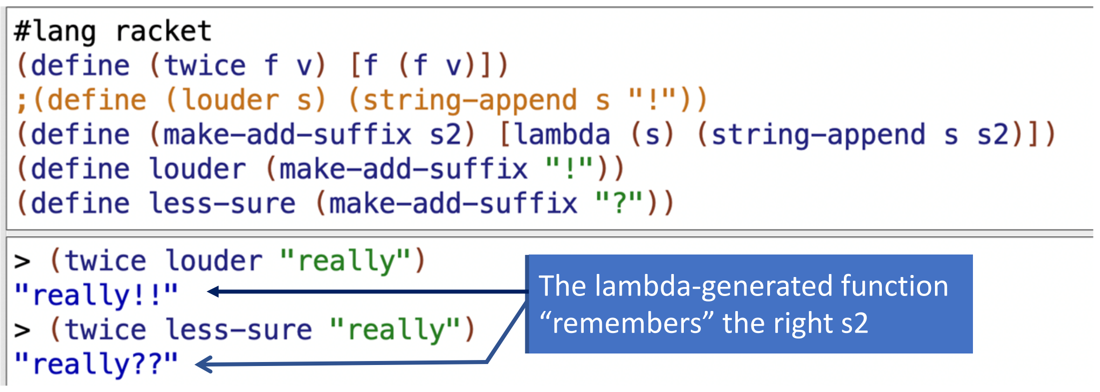
```
(define (make-add-suffix s2)
  (lambda (s) (string-append s s2)))

> (twice (make-add-suffix "!") "hello")
"hello!!"
> (twice (make-add-suffix "?!") "hello")
"hello?!?!"
> (twice (make-add-suffix "...") "hello")
"hello......"
```

The function make-add-suffix generates and returns an anonymous function.

12

Lexical Scope with `lambda` Function

- Racket is a *lexically scoped* language, which means that s2 in the function returned by make-add-suffix always refers to the argument for the call that created the function.



```
#lang racket
(define (twice f v) [f (f v)])
;(define (louder s) (string-append s "!"))
(define (make-add-suffix s2) [lambda (s) (string-append s s2)])
(define louder (make-add-suffix "!"))
(define less-sure (make-add-suffix "?"))
```

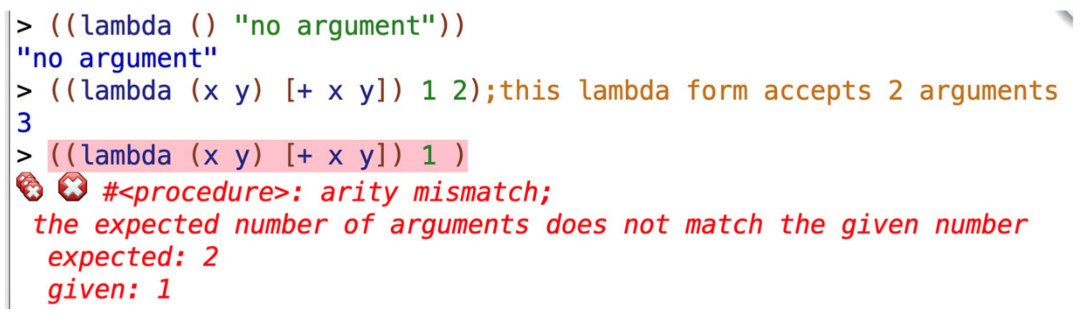
```
> (twice louder "really")
"really!!"
> (twice less-sure "really")
"really??"
```

The lambda-generated function "remembers" the right s2

13

Anonymous `lambda` Function

- A lambda form can also accept zero or more than one arguments

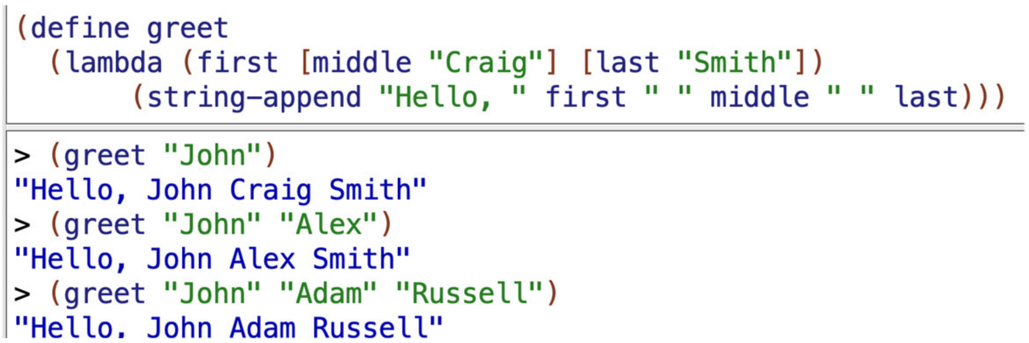


```
> ((lambda () "no argument"))
"no argument"
> ((lambda (x y) [+ x y]) 1 2);this lambda form accepts 2 arguments
3
> ((lambda (x y) [+ x y]) 1 )
#<procedure>: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 2
  given: 1
```

14

Optional Arguments in lambda Form

- An argument of the form `[arg-id default-expr]` is optional
  - When the argument is not supplied in a call, *default-expr* is the default value.
  - The *default-expr* can refer to any preceding *arg-id*, and every following *arg-id* must have a default as well.



```
(define greet
  (lambda (first [middle "Craig"] [last "Smith"])
       (string-append "Hello, " first " " middle " " last)))
```

```
> (greet "John")
"Hello, John Craig Smith"
> (greet "John" "Alex")
"Hello, John Alex Smith"
> (greet "John" "Adam" "Russell")
"Hello, John Adam Russell"
```

15