

CSC324 Principles of Programming Languages

Lecture 4

October 2, 2019

Exercise 3

- ▶ Well done! (lower mean than e2, but higher median)
- ▶ **Do not use mutation.** In the future, any code that uses mutation will be assigned a grade of 0.
- ▶ If you don't know how to solve a problem without mutation **come to office hours!**

Today

This next part of the course uses Racket macros to explore two programming paradigms:

- ▶ Object-oriented programming (familiar from Python/Java)
- ▶ Logic programming (likely new to you)

To build a language on top of Racket that supports OOP and Logic Programming, we'll use **macros** to **introduce new syntax**.

Object-Oriented Programming

Concepts of **Object-Oriented Programming** (OOP) in languages like Java and Python:

- ▶ **Object**: collection of attributes (instance variables), and methods (functions) that operate on the attributes
- ▶ **Class**: determines the attributes and methods available to an object
- ▶ **Constructor**: method that is called when a new object is created

Python Example (point.py)

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def from_origin(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def same_distance(self, other):
        return self.from_origin() == other.
            from_origin()
```

Objects. Classes. Attributes. Methods. Constructors. self. Can we really model all of this in Racket?

Implementing Attributes (point1.rkt)

- ▶ We can model an object as a function
- ▶ The function takes a symbol representing an attribute, and returns the value of that attribute
- ▶ “Message” is a common term for a request to an object

```
#lang racket
```

```
(define (my-point msg)  
  (cond [(equal? msg 'x) 10]  
        [(equal? msg 'y) -5]  
        [else "Unrecognized_message"]]))
```

Implementing Attributes...

```
#lang racket
(define (my-point msg)
  (cond [(equal? msg 'x) 10]
        [(equal? msg 'y) -5]
        [else "Unrecognized_message"]))
```

Concerns

- ▶ How do we make more than one “point” object?
- ▶ Can we support methods?

Let's first figure out how to make multiple objects!

Implementing Classes

- ▶ We'll stick to our idea of modeling an object as a function
- ▶ We will **model a class as a higher-order function**
 - ▶ Parameters: values of attributes
 - ▶ Return value: object
 - ▶ The object can access the attributes through the closure!

A First Implementation of Classes

```
#lang racket
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized_message"])))
```

- ▶ Now, we can create as many Points as we like
- ▶ Each Point object has its own attributes
- ▶ We get a (Point ...) “constructor” for free

Lexical Scoping

Lexical scoping really saves the day here:

```
(define p (Point 2 3))  
(let ([x 10])  
  (p 'x))
```

When `p` looks up its `x` attribute, it had better be 2 (not 10)!

Implementing Methods

- ▶ Add each method to the `cond`, like for instance variables
- ▶ But methods are *functions* that take 0 or more parameters

```
#lang racket
```

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'from-origin)
           (lambda ()
              (sqrt (+ (* x x) (* y y))))])
          [(equal? msg 'distance)
           (lambda (other-point)
              (let ([dx (- x (other-point 'x))]
                    [dy (- y (other-point 'y))])
                (sqrt (+ (* dx dx) (* dy dy))))])
          [else "Unrecognized_message"])))
```

OOP Features

What do we have so far?

- ▶ Classes
- ▶ Objects
- ▶ Constructors
- ▶ Ability to create multiple objects from a class
- ▶ Attributes (values are in the closure; looked-up by the `cond`)
- ▶ Methods (looked-up by the `cond`)

Missing Feature

OOP & Macros Worksheet: Question 1

Missing Feature: “self”

- ▶ We'll come back to the implementation of “self”
- ▶ But first. . .

Improving the Syntax

```
#lang racket
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'from-origin)
           (lambda ()
             (sqrt (+ (* x x) (* y y))))])
          [(equal? msg 'distance)
           (lambda (other-point)
             (let ([dx (- x (other-point 'x))]
                   [dy (- y (other-point 'y))])
               (sqrt (+ (* dx dx) (* dy dy))))])
          [else "Unrecognized_message"])))
```

- ▶ The syntax doesn't look **object-oriented** — we're just defining a bunch of functions
- ▶ Lots of boilerplate code for each attribute or method

We would like a syntax that looks like this:

```
(my-class Point (x y)
  (method (from-origin)
    (sqrt (+ (* x x) (* y y)))))
```

How do we add **new syntax** like this to Racket?

Macros

Macros

- ▶ A **macro** is a “function” that operates not on values, but on *source code*
- ▶ A macro takes code as its parameters and returns new code as its result
- ▶ Before a program is run, **macro expansion** is carried out to transform code according to the defined macros

Text-Based Macros

What does this C program print?

```
#include <stdio.h>

#define swap(x, y) { int tmp; tmp = x; x = y; y = tmp; }

int main(void) {
    int a = 4, b = 99;
    printf("a %d, b %d\n", a, b);
    swap(a, b);
    printf("a %d, b %d\n", a, b);
    return 0;
}
```

Text-Based Macros...

How about now? What does this C program print?

```
#include <stdio.h>

#define swap(x, y) { int tmp; tmp = x; x = y; y = tmp; }

int main(void) {
    int a = 4, tmp = 99;
    printf("a %d, tmp %d\n", a, tmp);
    swap(a, tmp);
    printf("a %d, tmp %d\n", a, tmp);
    return 0;
}
```

Racket Macros are Hygienic

Compared to C macros:

- ▶ Racket macros work on the level of *Abstract Syntax Trees* (AST), not text
 - ▶ AST: Tree (nested list) representation of the syntactic structure of code
- ▶ Racket macros are **hygienic**: they are statically-scoped, not dynamically-scoped
 - ▶ They do *not* capture surrounding identifiers (like `tmp` in the C examples)
- ▶ Identifiers defined within a macro are *not* accessible outside of the macro

List Comprehensions

We will write a macro to add list comprehensions to Racket!

Here's how a list comprehension works in Haskell (similar in Python):

```
Prelude> [x + 2 | x <- [1, 2, 3, 10]]  
[3,4,5,12]
```

As a grammar rule:

```
<list-comp> =  
  "[" <out-expr> "|" <id> "<-" <list-expr> "]"
```

We would like to add this grammar rule to Racket.

List Comprehensions...

Question: using what we already have in Racket, how can we implement the functionality of a list comprehension?

```
Prelude> [x + 2 | x <- [1, 2, 3, 10]]
```

List Comprehensions...

Question: using what we already have in Racket, how can we implement the functionality of a list comprehension?

```
Prelude> [x + 2 | x <- [1, 2, 3, 10]]
```

Using map!

```
> (map (lambda (x) (+ x 2)) (list 1 2 3 10))  
'(3 4 5 12)
```

Our macro will *rewrite* a list comprehension using map!

Transforming a List Comprehension

We want to transform an expression that looks like:

```
(list-comp <out-expr> : <id> <- <list-expr>)
```

... into ...

```
(map (lambda (<id>) <out-expr>) <list-expr>)
```

Macro, List Comps (lc.rkt)

```
#lang racket
(define-syntax list-comp
  (syntax-rules (: <-)
    [(list-comp <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>)
          <list-expr>)])])
```

Macro, List Comps (lc.rkt)

```
#lang racket
(define-syntax list-comp
  (syntax-rules (: <-)
    [(list-comp <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>)
          <list-expr>)])])
```

- ▶ `syntax-rules`: defines a **pattern-based** macro
- ▶ `(: <-)`: defines the keywords used by the macro
- ▶ `(list-comp <out-expr> : <id> <- <list-expr>)`: this is a **pattern**
- ▶ `(map (lambda (<id>) <out-expr>) <list-expr>)`: this is a **template**
- ▶ A pattern and template together form one **pattern rule**
- ▶ We can have multiple pattern rules!

The template specifies how the new syntax should be generated from the pattern.

List Comprehensions with Filtering

Haskell and Python list comprehensions support filtering:

```
> [x + 2 | x <- [8, 2, 5, 1], x > 2]  
[10,7]
```

List Comprehensions with Filtering...

We want to transform code that looks like:

```
<list-comp-if> =  
  "[" <out-expr> "|" <id> "<->" <list-expr>  
    "if" <condition> "]"
```

... into ...

```
(map (lambda (<id>) <out-expr>)  
     (filter (lambda (<id>) <condition>)  
             <list-expr>))
```

Macro, List Comps with Filtering (lc2.rkt)

```
(define-syntax list-comp-if
  (syntax-rules (: <- if)
    ; This is the old pattern.
    [(list-comp-if <out-expr> :
                   <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>)
          <list-expr>)]

    ; This is the new pattern.
    [(list-comp-if <out-expr> :
                   <id> <- <list-expr>
                   if <condition>)
     (map (lambda (<id>) <out-expr>)
          (filter (lambda (<id>)
                    <condition>)
                  <list-expr>)))]))
```

Macro Ellipses

How might we support multiple “if”-filters in a list comprehension?

```
> (list-comp-if (+ x 2) : x <- (list 1 2 3 4 5)
      if (> x 1) (even? x))
'(4 6)
```

- ▶ We don't know how many filters there will be, so we can't write a separate rule for each case
- ▶ Instead, we use the **macro ellipse** feature, which lets us bind zero or more expressions to a pattern variable

Macro Ellipses...

```
#lang racket
(define-syntax list-comp-if
  (syntax-rules (: <- if)
    [(list-comp-if <out-expr> :
                  <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>)
          <list-expr>)]

    [(list-comp-if <out-expr> :
                  <id> <- <list-expr>
                  if <condition> ...)
     (map (lambda (<id>) <out-expr>)
          (filter (lambda (<id>)
                   (and <condition> ...))
                 <list-expr>)))]))
```

The `<condition> ...` allows zero or more expressions to come after the `if`.

Macro Ellipses in Patterns

Works like a Kleene star in regular expressions

- ▶ `<attr> ...` matches zero or more expressions of any type

Macro Ellipses in Patterns

Works like a Kleene star in regular expressions

- ▶ `<attr> ...` matches zero or more expressions of any type
- ▶ `(<x> <y>) ...` matches zero or more “pairs” (note that these are not necessarily valid Racket expressions; `<x>` does not have to be a function!)

Macro Ellipses in Patterns

Works like a Kleene star in regular expressions

- ▶ `<attr> ...` matches zero or more expressions of any type
- ▶ `(<x> <y>) ...` matches zero or more “pairs” (note that these are not necessarily valid Racket expressions; `<x>` does not have to be a function!)
- ▶ `(<a> ...)` ... matches zero or more “lists”

Worksheet Questions 3-5

Macro Ellipses in Templates

Works like `map`: repeats the entire expression *before* the ellipsis, once for each match

- ▶ `(list <attr> ...)`
- ▶ `((+ <x> <y>) ...)`

Macro Ellipses in Templates

Works like `map`: repeats the entire expression *before* the ellipsis, once for each match

- ▶ `(list <attr> ...)`
- ▶ `((+ <x> <y>) ...)`

Pattern variables and ellipsis should be used together; do not separate the pattern variable from the ellipsis!

Worksheet Questions 6

Errors in macro expansion

Three cases:

1. No macro pattern matches. Results in **syntax error**
2. A macro pattern matches, but the resulting code is invalid.
Results in **runtime error**
3. A macro pattern matches, and the resulting code is valid. No errors. Yay!

The `my-class` macro

Creating Objects

Here's where we ended up last time:

```
#lang racket
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'from-origin)
           (lambda ()
              (sqrt (+ (* x x) (* y y))))])
          [(equal? msg 'distance)
           (lambda (other-point)
              (let ([dx (- x (other-point 'x))]
                    [dy (- y (other-point 'y))])
                (sqrt (+ (* dx dx) (* dy dy))))])
          [else "Unrecognized_message"])))
```


Desired Syntax

We want to write syntax like this:

```
(my-class Point (x y)
  (method (from-origin)
    (sqrt (+ (* x x) (* y y))))))
```

Macros will help us with exactly this kind of task!

Step 1: Macro that supports attributes only

Example uses of macro:

- ▶ `(my-class Point (x y))`
- ▶ `(my-class Person (name age gender))`
- ▶ `(my-class Pet (species name cuteness))`

One Attribute

Starting with this:

```
(my-class Point  
  (x))
```

We want to rewrite to this:

```
(define (Point x)  
  (lambda (msg)  
    (cond [(equal? msg 'x) x]  
          [else "Unrecognized message!"])))
```

One Attribute...

```
[(equal? msg 'x) x]
```

- ▶ Name `x` is in the closure, so we can access it
- ▶ But we also require `'x` for the `cond` comparison

```
> (quote x)
```

```
'x
```

```
> (quote from-origin)
```

```
'from-origin
```

Macro that supports one attribute

```
#lang racket
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr>))
     (define (<class-name> <attr>)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               [else "Unrecognized_message!"]))))))
```

Multiple attributes

Starting with this:

```
(my-class Point
  (x y))
```

We want to rewrite to this:

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized message!"])))
```

Macro: Multiple Attributes

No problem for macro ellipses!

```
#lang racket
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr> ...))
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ...
               [else "Unrecognized_message!"]))))))
```

Macro Ellipses

This use of ... is a bit different from the others:

```
[(equal? msg (quote <attr>)) <attr>]
```

```
...
```

Racket repeats the *entire* bracketed expression before the ...

Step 2: Macro that supports methods!

- ▶ Attributes required only one use of `...`, to support an arbitrary number of attributes
- ▶ But supporting methods will require *two* uses of `...`!
 - ▶ arbitrary number of methods
 - ▶ arbitrary number of *parameters* for each method
- ▶ Note the keyword `method` before each method

The desired syntax looks like:

```
(my-class <class-name> (<attr> ...)
  (method (<method-name> <param> ...) <body>)
  ...)
```

Methods...

Starting with this:

```
(method (bigger-x other) (> x (other 'x)))
```

We want to rewrite to this:

```
[(equal? msg 'bigger-x)  
 (lambda (other) (> x (other 'x)))]
```

Macros for methods: method name

Starting with this:

```
(method (<method-name> other) (> x (other 'x)))
```

We want to rewrite to this:

```
[(equal? msg (quote <method-name>))  
 (lambda (other) (> x (other 'x)))]
```

Macros for methods: body

Starting with this:

```
(method (<method-name> other) <body>)
```

We want to rewrite to this:

```
[(equal? msg (quote <method-name>))  
 (lambda (other) <body>)]
```

Macros for methods: parameter

Starting with this:

```
(method (<method-name> <param>) <body>)
```

We want to rewrite to this:

```
[(equal? msg (quote <method-name>))  
 (lambda (<param>) <body>)]
```

Macros for methods: zero or more parameters

Starting with this:

```
(method (<method-name> <param> ...) <body>)
```

We want to rewrite to this:

```
[(equal? msg (quote <method-name>))  
 (lambda (<param> ...) <body>)]
```

Macro: Attributes and Methods

```
#lang racket
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name>
      (<attr> ...)
      (method (<method-name> <param> ...) <body>)
      ...)
     (define (<class-name> <attr> ...)
      (lambda (msg)
        (cond [(equal? msg (quote <attr>)) <attr>]
              ...
              [(equal? msg (quote <method-name>))
               (lambda (<param> ...) <body>)]
              ...
              [else "Unrecognized_message!"]))))))
```

Next time

- ▶ Fixing issue in Worksheet Question #1
 - ▶ **self**