CSC324 Principles of Programming Languages

Lecture 6

October 19/20, 2020

Where we are

From week 1:

> *We will define/analyze/modify the **syntactic** and **semantic** features of programming languages, and create small languages of our own.*
>
> ***Ideal Result**: Recognize when a programming problem is a "programming languages" problem.*

So far, we have...

- created our own syntax for a lamdba-calculus-like language
- implemented left-to-right eager evaluation
- worked with types and algebraic data types
- used programming languages techniques to solve a problem about spreadsheets (project 1)

What's next?

- **macros**: adding syntax to a language like Racket
- **delimited continuations**: manipulating the control flow in a language like Racket
- **logic programming**: searching for answers given some constraints
- **parametic polymorphism**: a different way of manipulating data
- **states**: reasoning about mutable states in a functional language

Next ∼4 lectures

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**.

To circumvent Racket's left-to-right eager evaluation order and manipulate control flow, we'll introduce **streams** and **delimited continuations**.

Mid-Course Survey

- “It’s fine now”, “Nothing, keep on going!”
- Weekly exercises
  - Spaced practice is necessary when we learn new languages and new ideas
- Racket & Haskell at the same time
  - There are aspects of each language that we want to explore
  - Haskell in the first project lets us do more fun things with Racket in the second project
- Evening office hours: Andi might experiment

Mid-Course Survey: Breakout Rooms

Breakout rooms during lectures	5 respondents	13 %	✓
Piazza	16 respondents	40 %	
Office hours	8 respondents	20 %	
Notes	31 respondents	78 %	
Tutorial	15 respondents	38 %	
Quizzes	10 respondents	25 %	
Exercises	8 respondents	20 %	
No Answer	1 respondents	3 %	

Objects in Racket

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: Can we emulate this in Racket?

```
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 as functions

- 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

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

- What about creating more than one Point object?
- What about methods?

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 the Point class

```
(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

```
(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`)

Improving the Syntax

- 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

- Interpreter: Code -> Value
- Macro: Code -> Code

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 Hygenic

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 **hygenic**: 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

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

Macro, List Comps

```
(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

```
(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...

```
(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
- `(<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"

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!

Breakout Group

- Exercise 1-4
- 10 minutes

Which of the following expressions the pattern (macro <a> <b> <c> ...)?

```
(macro 1 2)

(macro 1 2 1)

(macro 1 2 3 4)

(macro 1 2 3 4 5 6)

(macro 1 (2 3 4) 5 6)
```

Which of the following expressions the pattern `(macro <a> (<b> <c>) ...)`?

```
(macro 1)

(macro 1 (1 1))

(macro 1 (() 3) 1 (2 3))

(macro 1 (2 3) (4 5) (6 7))

(macro 1 (2 3) (4 5 6))
```

Which of the following expressions the pattern `(macro (<a> ...) ...)`?

```
(macro)

(macro 1)

(macro (1))

(macro (1) (2 3))

(macro (1 2 3 4) 5)
```

Macro Expansion

```
(define-syntax my-macro
  (syntax-rules ()
    [(my-macro (<x> <y>) ...) (list '1st (- <x> <y>) ...)]
    [(my-macro (<a> ...) ...) (list '2nd (+ <a> ...) ...)]
    [(my-macro <e> ...)       (list '3rd)]))
```

```
(my-macro)
```

```
(my-macro 2 3)
```

```
(my-macro (3 2) (7 5))
```

```
(my-macro (2 3) (4 5 6) (2 2 6))
```

```
(my-macro (+ 2) (- 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

We want to add syntax to avoid boilerplating code

```
(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

```
(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!

```
(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

```
(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!"])))]))
```

Macro Practice

Syntactic Forms

Recall the following four syntactic forms in Racket: `or`, `and`, `if`, and `cond`.

- These cannot be implemented as functions due to short-circuiting

In fact... `or`, `and`, and `cond` can all be implemented using `if`!

Implementing `or` using `if`

What is wrong with this implementation?

```
(define (my-or p q)
  (if p
      #t
      q))
```

Using macros to rewrite `or`

```
(define-syntax my-or
  (syntax-rules ()
    [(my-or <p> <q>)
     (if <p> #t <q>)]))
```

Breakout Group

Implement `my-and` by rewriting to an `if` expression

Implement `my-cond` by rewriting to several `if` expressions

Bonus: Implement `let*` by rewriting to function application

Debrief (blank page)