CSC324 Principles of Programming Languages

Lecture 5

October 9, 2019

Exercise 4 and Assignment 1

Majority vote to:

- Optionally resubmit exercise 4 by Thursday 9pm (with -20% penalty)
- Move assignment 1 deadline to next Thursday 9pm

The midterm cannot be moved.

Exercise 4 Debrief (Task 1)

Add two new cases to the "eval-calc" function

- Function definition, i.e. `(lambda (x) x)`
- Function call, i.e. `(f 1)` or `((lambda (x) x) 1)`

There should be **no other cases**!

- There should not be a case to handle `(closure...)` expressions, because there is no such thing!

Exercise 4 Expressions vs Values

```
expression --->interpreter--->  value
```

- `'(closure (lambda (x) (x)) (hash))` is *not* a valid expression in the calculator grammar
- `'(closure (lambda (x) (x)) (hash))` is a possible *value* returned by the interpreter

Exercise 4 Debrief (Task 1: Function Definition)

Function definitions evaluate to closures

Why? To save the environment bindings.

Example:

```
(let* ((n 10)
       (f (lambda (x) (* x n))))
  (f 5))
```

Exercise 4 Debrief (Task 1: Why we need closures)

Function definitions evaluate to closures

Why? To save the environment bindings.

Example:

```
(let* ((Point (lambda (x y)
                (lambda (msg)
                    (if (= msg 0) x y))))
       (p1    (Point 1 2))
       (p2    (Point 3 4)))
  (p2 0))
```

Exercise 4 Debrief (Task 1: Function call)

```
(let* ((n 10)
       (f (lambda (x y) (* (+ x y) n))))
  (f (+ 1 1) 3))
```

To evaluate `(f 5)` first:

1. Evaluate the function subexpression: i.e. call `eval-calc` on `f`, get a closure
2. Evaluate the argument subexpressions: i.e. call `eval-calc` on both `(+ 1 1)` and 3, get 2 and 3
3. Evaluate the body of the function, using the environment **from the closure**, extended with the argument values
   - Evaluate `(* (+ x y) n)`
   - Extend the environment `{n: 10}` with `{x: 2, y: 3}`

Exercise 4 Debrief (Task 1: Lexical Scoping)

3. Evaluate the body of the function, using the environment **from the closure** . . .

If you don't use the environment from the closure, and instead use the environment from `eval-calc`, you won't have lexical scoping!

Exercise 4 Debrief (Task 1: the y-combinator!)

Your `calculator` should be able to evaluate this factorial function!

```
(let* ((fix (lambda (f)
              ((lambda (d) (d d))
               (lambda (x) (f (lambda (a) ((x x) a)))))))
       (fac (fix (lambda (fac)
                 (lambda (n)
                   (if (= n 0) 1 (* n (fac (- n 1)))))))))
  (fac 5))
```

Your calculator is Turing Complete!

Exercise 4 Debrief (Task 2)

- accumulate the `s-map`, starting from an empty hash
- for each definition, iterate through the parameters, and call `strict-in?`

Assignment 1

- Start early!
- Write tests before you write code, as a way to verify your understanding
- This assignment is as much about testing as it is about writing code
- Repurpose examples from the Exercise 4 FAQ

Assignment 1 Grading

- Correctness: 80% of your grade
  - Autograded
- Code Quality: 20% of your grade
  - Graded by a TA

Assignment 1 Office Hours

- Thursday Oct 10 2:30-4pm CC2110
- Thursday Oct 10 5:30-7pm CC2110

Midterm

See https://www.cs.toronto.edu/~lczhang/324/midterm.html

- Length: 50 minutes
- Written on paper. No computers, calculators, notes, etc.
- Closed book
- Aid sheet:
  `https://www.cs.toronto.edu/~lczhang/324/files/aid_sheet`

Past midterms are posted.

I won't ask you to *write* Haskell code, but you will be asked to evaluate and reason about Haskell code.

Midterm Logistics

- Midterm will be during the **first 50 minutes**
- We'll have a lecture in the following hour

Today

- OOP: self
- Streams

Object Oriented Programming

Last time

```
#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!"])))]))
```

Hash Tables: Improving Performance

- A `cond` with one branch per attribute/method is slow!
  - Requires a linear search through the `cond` branches
- Instead: as you did in lab, use a hash table
  - Keys: attribute/method names
  - Values: associated values (primitives for attributes; *functions* for methods)

Macro: Hash-Table Lookup

```
#lang racket
; Arbitrary numbers of attributes and methods;
   hash-table lookup
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <Class> (<attr> ...)
       (method (<method-name> <param> ...) <body>)
           ...)
     (define (<Class> <attr> ...)
       (let* ([__dict__
               (make-immutable-hash
                (list (cons (quote <attr>) <attr>)
                      ...
                      (cons (quote <method-name>)
                            (lambda (<param> ...)
                              <body>))
                      ...))])
         (lambda (msg)
           (hash-ref __dict__ msg
                     "Unrecognized message!")))))]))
```

Python's Attribute Dictionary

The use of an "attribute dictionary" is not new:

- In Python, every object has an attribute named `__dict__`
- It's similar to our use of the Racket hash table

What is stored in there?

Python's Attribute Dictionary...

```
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()
```

```
>>> p1 = Point(3, 4)
>>> p1.__dict__
```

Python's Attribute Dictionary...

```
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()
```

```
>>> p1 = Point(3, 4)
>>> p1.__dict__
```

```
{'x': 3, 'y': 4}
```

What happened to `scale`?

Python's Method Dictionary

- Python stores methods in an attribute of the *class*, not the *object*

```
>>> Point.__dict__['scale']
<function Point.scale at ...>
```

- So, Python splits up the attributes and methods:
  - Attributes in a per-object dictionary
  - Methods in a per-class dictionary

Why Separate Dictionaries?

- Used to implement inheritance
  - On a method call, search the class dictionary, then the superclass dictionary, etc. until the method is found
- Helpful for implementing behaviour that is specific to attributes or methods . . .

Let's split our Racket dictionary in two!

Macro: Separate Attribute and Method Dictionaries

```
#lang racket
(define-syntax my-class (syntax-rules (method)
  [(my-class <Class> (<attr> ...)
     (method (<method-name> <params> ...)
             <body>) ...)
   (begin (define class__dict__
     (make-immutable-hash (list
       (cons (quote <method-name>)
             (lambda (<params> ...) <body>)) ...)))
     (define (<Class> <attr> ...)
       (let* ([self__dict__ (make-immutable-hash
                (list (cons (quote <attr>)
                            <attr>) ...))])
         (lambda (msg) (cond
           [(hash-has-key? self__dict__ msg)
            (hash-ref self__dict__ msg)]
           [(hash-has-key? class__dict__ msg)
            (hash-ref class__dict__ msg)]
           [else "Unrecognized message!"])))))]))
```

begin

- Syntax: `(begin <expr> ...)`
- Evaluates all the inner expressions and definitions
- The entire expression evaluates to the *last* expression

```
> (begin (define a 3) (define b 4) (+ a b))
7
> a
3
>
```

Separating Attribute and Method Dictionaries

- `class__dict__` stores the methods, and lives *outside* of the objects
- `self__dict__` stores the attributes, and lives *inside* each object
- When looking up an attribute/method, first we check `self__dict__`, then `class__dict__`

We've Lost Something!

```
(my-class Point
  (x y)
  (method (bigger-x other) (> x (other 'x))))
```

This no longer works.

Why?

self

- Methods no longer have access to attributes like `x` and `y`
- Python solves this through the `self` parameter of methods
- Each method has `self` as its first parameter, and Python automatically supplies the calling object as its argument

self in Python

```
    def scale(self, factor):
        return Point(self.x * factor, self.y * factor)
```

- To call `scale`, we pass only one argument . . . not two!

```
p1.scale(8) # self is automatically set to p1
p2.scale(10) # self is automatically set to p2
```

self in Racket

- Each of our methods will now take one *extra* parameter

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

- When a method is looked-up, we will automatically *fix* the *first* argument of the method to be the calling object
- *fix first*! Remember that?

Fixing the First Argument

```
(define (fix-first x f)
  (lambda rest
    (apply f (cons x rest))))
```

Macro: Supporting self (`milestone2.rkt`)

The critical piece of the macro:

```
[(hash-has-key? class__dict__ msg)
 (fix-first me (hash-ref class__dict__ msg))]
```

- Only applies to methods, not attributes
- `me` is bound to the current object using `letrec`
- `fix-first` passes `me` as the first parameter to the method

letrec

Allows recursive binding:

```
(letrec ((fac (lambda (n)
                (if (equal? n 0)
                    1
                    (* n (fac (- n 1)))))))
  (fac 5))
```

However, the recursive binding must be used inside of a function definition

Evaluating this expression will produce an error:

- `(letrec ((n (+ n 1))) n)`

Exercise 4 Cookie Challenge Hint

For the cookie challenge, you will need to:

- Use letrec in a smart way
- Change the way environments are represented (wrap in a thunk to delay evaluation)

Streams

Stream

- **Stream**: an abstract model of a (possibly infinite) sequence of values over time
- Implemented as a “lazy list”, a list whose elements are only evaluated when necessary

Racket Lists

In Racket, `cons` is a function that **eagerly-evaluates** its arguments!

```
> (cons 4 (cons 5 (cons (first '()) '())))
; first: contract violation
; expected: (and/c list? (not/c empty?))
```

This means we can't work with infinite lists in Racket.

Haskell Lists

In Haskell, lists are in fact streams!

The cons operator `:` is also a function, but Haskell uses **lazy evaluation**

```
> lst = 4:5:(head []):[]
> head lst
4
> head (tail lst)
5
> head (tail (tail lst))
*** Exception: Prelude.head: empty list
```

Haskell allows us to work with infinite lists.

Streams in Racket?

In order to implement streams in racket we need to be able to **delay evaluation**. Two things we'll need:

1. Thunks
2. Macros

Thunks

- Remember that a **thunk** is a zero-argument function, used to delay evaluation

```
> (thunk (/ 1 0))
#<procedure>
> ((thunk (/ 1 0)))
; /: division by zero
```

Thunks and Streams

- For a **non-empty stream**, we will have a value *wrapped by a thunk*, followed by a stream *also wrapped by a thunk*.
- No element of the stream is evaluated until it is used

```
> (cons (thunk 4)
        (thunk (cons (thunk 5)
                     (thunk (cons (thunk (first '()))
                                  '()))))))
'(#<procedure> . #<procedure>)
```

- We will represent the **empty stream** as a special symbol

car and cdr

- `first` and `rest` work on proper *lists* only
  - `<list> = '() | (cons <expr> <list>)`
- The general version of these functions are called `car` and `cdr`
  - They work on an arbitrary *pair* of elements, whether or not the pair of elements is part of a list

```
> (first (cons 4 (list))) ; ok, this is a list
4
> (first (cons (thunk 4) (thunk (cons (thunk 5) '()))))
; first: contract violation
> (car (cons (thunk 4) (thunk (cons (thunk 5) '()))))
#<procedure>
> (cdr (cons (thunk 4) (thunk (cons (thunk 5) '()))))
#<procedure>
```

Thunks allow us to create infinite lists:

```
> (define (repeat n)
    (cons (thunk n) (thunk (repeat n))))
> (define x (repeat 1))
> ((car x))
1
> ((car ((cdr x))))
1
> ((car ((cdr ((cdr x))))))
1
```

Why Macros?

- Avoid the `(thunk ...)` boilerplate
- Identical public interface to built-in lists. We would like to:
  - Test whether a stream is the empty stream (like `empty?`)
  - "cons" an expression and a stream (like `cons`)
  - Access the first element of the stream (like `first`)
  - Access the rest of the stream (like `rest`)
  - Build a stream from one or more expressions (like `list`)

Empty Streams

The empty list is represented by `'()`. We need an analog for streams!

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))
```

Cons on Streams

The macro `s-cons` takes `<first>` and `<rest>`, wraps them in thunks, and cons's them!

```
(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))
```

Recovering the Elements

```
> (define nums (s-cons 4 (s-cons 5 s-null)))
```

What we hope would work:

```
; retrieve first thunk, and call it to get the 4?
((first nums))
```

Recovering the Elements

```
> (define nums (s-cons 4 (s-cons 5 s-null)))
```

What we hope would work:

```
; retrieve first thunk, and call it to get the 4?
((first nums))
```

Error!

Recovering the Elements, Take 2

```
(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))
```

Convenient Stream-Building

To build a stream, we have to use repeated application of `s-cons`.

```
> (s-cons 1
    (s-cons 2
      (s-cons 3
        (s-cons 4
          (s-cons 5 s-null)))))
```

Convenient Stream-Building

To build a stream, we have to use repeated application of `s-cons`.

```
> (s-cons 1
    (s-cons 2
      (s-cons 3
        (s-cons 4
          (s-cons 5 s-null)))))
```

But building a list is so much easier!

```
> (list 1 2 3 4 5)
```

Convenient Stream-Building...

The macro `make-stream` is like the function `list`: takes one or more expressions and `cons` them all together.

```
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

Infinite Streams in Racket

```
> (define (repeat n) (s-cons n (repeat n)))
> (define x (repeat 1))
> (s-first x)
1
> (s-first (s-rest x))
1
> (s-first (s-rest (s-rest x)))
1
```